Accelerating Stateful Network Packet Processing Using Graphics Hardware

Giorgos Vasiliadis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in Computer Science
in the Graduate Division
of the University of Crete

Heraklion, December 2014

University of Crete
Computer Science Department

Accelerating Stateful Network Packet Processing Using Graphics Hardware

Dissertation submitted by

Giorgos Vasiliadis

in partial fulfilment of the requirements for the Ph.D. degree in Computer Science.

Author: _____
Giorgos Vasiliadis

Committee approvals: _____
Evangelos P. Markatos
Professor, University of Crete, Thesis Supervisor

_____
Sotiris Ioannidis
Principal Researcher, FORTH

_____
Dimitrios S. Nikolopoulos
Professor, Queen's University Belfast

_____
Michalis Polychronakis
Assistant Professor, Stony Brook University

_____
Angelos Bilas
Professor, University of Crete

_____
Kostas Magoutis
Assistant Professor, University of Ioannina

_____
Polyvios Pratikakis
Researcher, FORTH

Departmental approval: _____
Panagiotis Tsakalides
Professor, University of Crete, Chairman of the Department

Heraklion, December 2014

Abstract

The need for differentiated services (such as firewalls, network intrusion detection/prevention systems and traffic classification applications) that lie in the core of the Internet, instead of the end points, constantly increases. These services need to perform complex packet processing operations at upper networking layers, which, unfortunately, are not supported by traditional edge routers. To address this evolution, specialized network appliances (called "middleboxes") are deployed, which typically perform complex packet processing operations—ranging from deep packet inspection operations to packet encryption and redundancy elimination. Packet-processing implemented in software promises to enable the fast deployment of new, sophisticated processing without the need to buy and deploy expensive new equipment. In this thesis, we propose to increase the throughput of packet processing operations by using Graphics Processing Units (GPUs). GPUs have evolved to massively parallel computational devices, containing hundreds of processing cores that can be used for general-purpose computing beyond graphics rendering. GPUs, however, have a different set of constraints and properties that can prevent existing software from obtaining the improved throughput benefits GPUs can provide.

This dissertation analyzes the tradeoffs of using modern graphics processors for stateful packet processing and describes the software techniques needed to improve its performance. First, we present a deep study into accelerating packet processing operations using discrete modern graphics cards. Second, we present a broader multi-parallel stateful packet processing architecture that carefully parallelizes network traffic processing and analysis at three levels, using multi-queue network interfaces (NICs), multiple CPUs, and multiple GPUs. Last, we explore the design of a GPU-based stateful packet processing framework, identifying a modular mech-

anism for writing GPU-based packet processing applications, eliminating excessive data transfers as well as redundant work found in monolithic GPU-assisted applications. Our experimental results demonstrate that properly architecting stateful packet processing software for modern GPU architectures can drastically improve throughput compared to a multi-core CPU implementation.

Supervisor: Evangelos Markatos, Professor

Περίληψη

Η ανάγκη για διαφορετικού τύπου υπηρεσίες στον πυρήνα του Internet (όπως firewalls, συστήματα ανίχνευσης και πρόληψης επιθέσεων, συστήματα κατηγοριοποίησης της κίνησης του δικτύου, κλπ.), συνεχώς αυξάνει. Οι υπηρεσίες αυτές πρέπει να εκτελούν περίπλοκες επεξεργασίες στα πακέτα δεδομένων σε όλα τα επίπεδα της στοίβας δικτύου, οι οποίες, όμως, δεν υποστηρίζονται από τους παραδοσιακούς δρομολογητές. Για το λόγο αυτό, έχουν αναπτυχθεί εξειδικευμένες συσκευές (οι οποίες ονομάζονται 'middleboxes'). Οι συσκευές αυτές εκτελούν διάφορες μορφές επεξεργασίας - οι οποίες περιλαμβάνουν, για παράδειγμα, τη λεπτομερειακή ανάλυση των πακέτων, τη κρυπτογράφηση τους και τη συμπίεση. Επιπλέον, η επεξεργασία των πακέτων που βασίζεται αποκλειστικά σε λογισμικό υπόσχεται την ταχεία ανάπτυξη νέων, εξελιγμένων μορφών επεξεργασίας, χωρίς την ανάγκη για αγορά ακριβού εξοπλισμού. Σε αυτή την διατριβή, προτείνουμε τη χρήση μονάδων επεξεργασίας γραφικών (GPUs) προκειμένου να αυξήσουμε την απόδοση των εφαρμογών επεξεργασίας πακέτων δικτύου. Οι GPUs έχουν εξελιχθεί σε υπολογιστικές συσκευές οι οποίες προσφέρουν μαζικό παραλληλισμό, καθώς περιέχουν εκατοντάδες πυρήνες επεξεργασίας που μπορούν να χρησιμοποιηθούν για υπολογισμούς γενικής χρήσης, πέρα από την χρήση τους σε εφαρμογές γραφικών. Ωστόσο, λόγω των διαφορετικών ιδιοτήτων και περιορισμών των GPUs, δεν είναι πάντα εύκολη υπόθεση να βελτιωθεί η απόδοση του υπάρχοντος λογισμικού.

Η παρούσα διατριβή αναλύει τη χρήση GPUs για την επεξεργασία πακέτων δικτύου (με εποπτεία κατάστασης) και περιγράφει τεχνικές που απαιτούνται ώστε να βελτιωθεί η επίδοση της. Αρχικά, παρουσιάζουμε μια μελέτη σχετικά με την επιτάχυνση της επεξεργασίας πακέτων χρησιμοποιώντας διακριτές κάρτες γραφικών. Δεύτερον, παρουσιάζουμε μια ευρύτερη αρχιτεκτονική επεξεργασίας πακέτων που παραλληλοποιεί προσεκτικά την επεξεργασία και ανάλυση των πακέτων σε τρία επίπεδα, χρησιμοποιώντας (i) διασυνδέσεις δικτύου με πολλαπλές ουρές, (ii) πολλαπλούς επεξεργαστές (CPUs), και (iii) πολλαπλές GPUs. Τέ-

λος, διερευνούμε το σχεδιασμό ενός framework για την επεξεργασία πακέτων βασισμένο σε GPUs, το οποίο προσδιορίζει έναν αρθρωτό (modular) μηχανισμό για τον προγραμματισμό εφαρμογών επεξεργασίας πακέτων, το οποίο περιορίζει τις μεταφορές δεδομένων και τις περιττές επεξεργασίες. Τα πειραματικά αποτελέσματα μας δείχνουν ότι η ορθή υλοποίηση εφαρμογών επεξεργασίας πακέτων βασισμένες σε GPUs, μπορούν να βελτιώσουν συμαντικά την απόδοση από ότι μία πολυ-πύρινη CPU.

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 Sotiris Ioannidis, who gave me the opportunity to work on this subject and whose contribution, advices and support was a key for pushing towards my goals and completing this thesis. I am also indebted to Michalis Polychronakis for his constant and continuous support and the invaluable advices. Also, the remaining members of my committee, Dimitris Nikolopoulos, Angelos Bilas, Kostas Magoutis, and Polyvios Pratikakis, for the valuable comments and useful feedback during my defense. I also feel thankful to Lazaros Koromilas for a great collaboration on projects related to this thesis, and Christos Papachristos for his technical support.

I also want to express (in alphabetic order) my deepest gratitude to Elias Athanasopoulos, Georgios Chinis, Panagiotis Garefalakis, Eleni Gessiou, Alexandros Kapravelos, Georgios Kontaxis, Antonis Krithinakis, Laertis Loutsis, Nick Nikiforakis, Antonis Papadogiannakis, Panagiotis Papadopoulos, Antonis Papaioannou, Vasilis Pappas, Thanasis Petsas, Iasonas Polakis, Giorgos Saloustros, Manolis Stamatogiannakis, Nikos Tsikoudis, Aris Tzermias, Apostolis Zarras and all the other past and present members of the Distributed Systems (DCS) Lab for their help and guidance, their friendship, the drinking, and for making the lab a fun place all these years! My warmest regards to Petros Efstathopoulos and Marc Dacier, for making my internship at Symantec Research Labs so rewarding and an enjoyable experience.

"If everything seems under control, you're just not going fast enough."
— Mario Andretti

Contents

1 Introduction **1**

1.1 Thesis Statement 3
1.2 Contributions 4
1.3 Dissertation Overview 5
1.4 Publications 6

2 Background Concepts and Trends **9**

2.1 Software Packet Processing 9
2.1.1 Stateful Packet Processing 11
2.2 Commodity Hardware 12
2.2.1 Multicore CPUs 12
2.2.2 Multi-queue Network Interfaces 13
2.2.3 Graphics Processors 14
2.3 General-Purpose GPU (GPGPU) 16
2.3.1 Comparison with CPU. 16
2.3.2 CPU versus GPU 18
2.3.3 Programming Considerations 18

3 Accelerating a Single-Threaded Network Intrusion Detection Systems using Graphics Hardware **21**

3.1 Architecture 21
3.1.1 Packet Capturing and Decoding 22
3.1.2 Preprocessing 22
3.1.3 Transferring Packets to the GPU 23
3.1.4 String Matching on the GPU 25
3.1.5 Transferring the Results to the Host 26
3.1.6 Regular Expression Matching on Graphics Processors 27
3.1.7 Execution Flow Overview 30

3.2 Implementation . . . 32
3.2.1 Collecting packets on the CPU . . . 33
3.2.2 GPU-based implementation of String Matching . . . 34
3.2.3 GPU-based Implementation of Regular Expression Matching . . . 35
3.3 Optimizations . . . 38
3.3.1 Exploring Device Memory Hierarchies . . . 38
3.3.2 Optimizing GPU Memory Accesses . . . 39
3.3.3 Packets Layout Transformations . . . 39
3.4 Performance Evaluation . . . 41
3.4.1 Hardware Platform . . . 41
3.4.2 Network Traces . . . 41
3.4.3 String Matching . . . 42
3.4.4 Regular Expression Matching . . . 45
3.4.5 Overall Throughput . . . 46
3.4.6 Worst-case Performance . . . 49
3.5 Discussion . . . 50

4 A Multi-Parallel Network Intrusion Detection Architecture 53
4.1 Design Objectives . . . 53
4.1.1 Inter-flow Parallelism . . . 54
4.1.2 Intra-flow Parallelism . . . 55
4.1.3 Resulting Trade-off . . . 55
4.2 Architecture . . . 56
4.2.1 Packet Capture in 10-Gigabit Ethernet . . . 57
4.2.2 Processing Engine . . . 59
4.3 Performance Optimizations . . . 63
4.4 Experimental Evaluation . . . 63
4.4.1 Experimental Setup . . . 63
4.4.2 Micro-Benchmarks . . . 65
4.4.3 Overall Traffic Processing Throughput . . . 71
4.5 Discussion . . . 74
4.5.1 Price/Performance . . . 74
4.5.2 Limitations . . . 75

5 A Generic Packet Processing Framework for GPUs **77**
5.1 Motivation . . . 77
5.2 Design . . . 78
5.2.1 Processing Modules . . . 79
5.2.2 API . . . 81
5.3 Stateful Protocol Analysis . . . 82
5.3.1 Flow Tracking . . . 82
5.3.2 Parallelizing TCP Stream Reassembly . . . 83
5.3.3 Packet Reordering . . . 84
5.4 Optimizing Performance . . . 86
5.4.1 Inter-Device Data Transfer . . . 86
5.4.2 Packet Decoding . . . 88
5.4.3 Packet Scheduling . . . 88
5.5 Developing with GASPP . . . 90
5.5.1 Firewall . . . 91
5.5.2 L7 Traffic Classification . . . 91
5.5.3 Signature-based Intrusion Detection . . . 91
5.5.4 AES . . . 92
5.6 Performance Evaluation . . . 92
5.6.1 Hardware Setup . . . 92
5.6.2 Data Transfer . . . 92
5.6.3 Raw GPU Processing Throughput . . . 94
5.6.4 End-to-End Performance . . . 99
5.7 Limitations . . . 102

6 Related Work **103**
6.1 Software Packet Processing for IP Forwarding and Routing 103
6.1.1 Implementations on multi-core CPUs . . . 103
6.1.2 GPU-assisted Implementations . . . 106
6.2 Deep Packet Inspection . . . 107
6.2.1 Network Intrusion Detection and Prevention Systems 107
6.2.2 Traffic Classification . . . 112
6.2.3 Load Balancing . . . 113
6.3 Network Packet Processing Frameworks . . . 114

7 Future Work and Conclusion **117**
7.1 Summary of Contributions 117
7.2 Future Work 118
7.3 Conclusion 119

List of Figures

2.1 Network packet processing flow path. 11
2.2 Diagram of a dual-CPU commodity system with integrated memory controllers and point-to-point interconnects between the processors. 12
2.3 The NVIDIA CUDA architecture comprised of multiprocessors, each of which contains a set of stream processors. . . . 15

3.1 Overview of the single-threaded GPU-based network intrusion detection architecture. 22
3.2 Batching different packets to a single buffer. This scheme results to lower memory consumption and also reduces response latency for port groups with low traffic. 23
3.3 Network packet buffer format with fixed-size slots. 24
3.4 DFA matching on the GPU. The algorithm moves over the input data stream one byte at a time and switches the current state according to the state transition table. When a final-state is reached, a match has been found, and the corresponding offset is marked. 25
3.5 Overview of the regular expression matching engine in the GPU. Each requested packet is matched against a different regular expression independently. 29
3.6 Execution flow of the single-threaded GPU-assisted NIDS. GPU communication and computation can overlap with CPU execution. 31
3.7 Matching packets that exceed the MTU size. 34
3.8 The GPU string matching parallelization approach. Each packet is processed by a different stream processor independently of the others. 35

3.9 Impact of word accesses when fetching data from the global device memory. 42
3.10 Memory accesses impact on DFA matching. 44
3.11 Impact of packet transformations on DFA matching. 45
3.12 States (a) and memory requirements (b) for the 11,775 regular expressions contained in the default Snort 2.6 ruleset when compiled to DFAs. The median number of states is 17, and the corresponding memory requirements is 32.2 MB. 47
3.13 Sustained processing throughput for Snort when regular expression matching is executed in the CPU and the GPU respectively. The string matching process is performed on the CPU for both approaches. 48
3.14 Sustained processing throughput for Snort using different network traces. Both string searching and regular expression matching are performed on the GPU. 48
3.15 Sustained throughput for Snort when using only regular expressions. 49

4.1 MIDeA architecture. 56
4.2 State tables of AC-Full and AC-Compact. 61
4.3 Data transfers and GPU execution of different processes can overlap. 64
4.4 Hardware setup. 65
4.5 GPU throughput for AC-Full and AC-Compact. 66
4.6 GPU throughput with an increasing number of CPU processes up to the number of cores. Each process is mapped to a different CPU-core. 68
4.7 Overall sustained throughput for an increasing number of (a) packet sizes and (b) CPU processes (each process is mapped to a different CPU-core). 69
4.8 Breakdown of per-byte processing overhead for different packet sizes. 71
4.9 Observed packet loss for (a) synthetic and (b) real traffic, as a function of the traffic rate. MIDeA can handle real traffic speeds of up to 5.2 Gbit/s without dropping any packets. . 73

5.1 GASPP architecture. 79

5.2 GPU packet processing pipeline. The pipeline is executed by a different thread for every incoming packet. 81
5.3 Ordering sequential TCP packets in parallel. The resulting *next_packet* array contains the next in-order packet, if any (i.e. *next_packet[A] = B*). 83
5.4 Subsequent packets (dashed line) may arrive in-sequence ((a)–(d)) or out of order, creating holes in the reconstructed TCP stream ((e)–(f)). 85
5.5 Normal (a) and zero-copy (b) data transfer between the NIC and the GPU. 86
5.6 Different network packet buffer formats. 87
5.7 The I/O and processing pipeline. 88
5.8 Packet scheduling for eliminating control flow divergences and load imbalances. Packet brightness represents packet size. 90
5.9 Data transfer throughput for different packet sizes when using two dual-port 10GbE NICs. 93
5.10 Average processing throughput sustained by the GPU to (a) decode network packets, (b) maintain flow state and reassemble TCP streams, and (c) perform various network processing operations. 95
5.11 Performance gains on raw GPU execution time when applying packet scheduling (the scheduling cost is included). . . 98
5.12 Sustained traffic forwarding throughput (a) and latency (b) for GASPP-enabled applications. 100
5.13 Sustained throughput for concurrently running applications. 101

List of Tables

2.1 Sustained throughput for transferring data to the Graphics card device, and vice versa (Gbit/s). 18

3.1 Regular expression operations. 28

4.1 Memory requirements of AC-Full and AC-Compact for the default Snort rule set. 67
4.2 `UNI` network trace properties. 74
4.3 Cost of MIDeA components (as of April 2011). 75

5.1 Sustained throughput (Gbit/s) for various packet sizes, when bulk-transferring data to a single GPU. 92
5.2 Time spent (μsec) for traversing the connection table and removing expired connections. 96

Chapter 1

Introduction

Computer networks have been growing in size, complexity, and connection speeds [7, 24]. Especially in access and backbone links, speeds typically reach multi-Gigabit per second rates. At the same time, networking applications become diversified and traffic processing gets more sophisticated, requiring data-plane operations beyond traditional functions of Layer-2 and Layer-3 of the Open System Interconnection (OSI) stack, such as forwarding and routing. Coping with the increasing network capacity and complexity necessitates pushing the performance of network packet processing applications as high as possible.

Typical operations that are required in modern networks and data center networking—such as network intrusion detection and prevention systems (NIDS/NIPS), firewalls, traffic classification and content-centric networking— must ultimately operate at acceptable speeds, which raises the need for a platform that can support fast implementations of these operations, preferably in a highly parallel configuration to allow for efficient scalability. Furthermore, rich programmability is a key requirement for enabling rapid prototyping and robust implementations of these designs.

Previously programmable *special-purpose hardware*, i.e. FPGAs, Network Processors (NPUs), TCAMs, etc., have greatly reduced both the cost and time to develop a network system, and have been successfully used in routers [4, 9] and network intrusion detection systems [44, 86, 127]. These systems offer a scalable method of processing network packets in high-speed environments. However, most implementations require specialized programming, and are usually tied to the underlying device. Moreover, programming these devices is very challenging since low-level hardware

details are exposed to the programmers. For example, the size of each microengine on an Intel IXP 2800 NPU is maximal 8K words and there are 16 engines in total, making difficult to partition code to fit exactly into each microengine. As a consequence, implementations based on special-purpose hardware are very difficult to extend and program, and prohibit them from being widely accepted by the industry.

In contrast, *software implementations* of network packet processing applications, that are based on commodity processors, are low-cost and easily programmable. The emergence of commercial *many*-core architectures (i.e. multicore CPUs [28, 67], modern graphics processors [103, 31], and manycore coprocessors [64, 68]) provide a potential solution for accelerating a vast amount of applications, and have led researchers to deploy them to high-speed environments with high success [106, 79, 51, 60, 72]. The most obvious advantages of software implementations are, to name a few, familiar programming environments and abundant third-party software and tools available for system development. Moreover, the cost of this approach is much lower compared to the dedicate hardware solutions.

Typically, packet processing applications have two inherently features that are suitable for parallelization in many core environments: i) they have naturally separated layers that can be organized into a functional pipeline; and ii) packets of different network flows can be processed in parallel. Nevertheless, it is still very challenging to implement network processing applications on multicore architectures, for several reasons. First, network applications are inherently memory and I/O intensive, hence they may further increase the discrepancy between computing power and memory latencies of multicore architectures. Second, inter-core synchronization and communication must be handled by software, which in general is much slower than the mechanisms employed in dedicated hardware devices [143]. Third, stateful packet processing, needed in complex network applications—such as network intrusion detection and traffic classification systems—require to track the state of network connections, in contrast to low-layer network applications that operate independently on each packet separately; such packet dependencies can significantly reduce parallelism if not handled appropriately.

In this dissertation we examine the parallelization of stateful network packet processing applications—that support complicated processing be-

tween Layer-2 and Layer-7 of the OSI stack—implemented in commodity, off-the-shelf, hardware. First, we show that discrete modern graphics cards can be used to accelerate costly packet processing operations. We offload to the GPU both string searching and regular expression matching—which are among the most intensive packet processing operations—within a popular network intrusion detection system, and we show its improvement in terms of performance. In order to tackle load imbalances across the data-parallel GPU threads and coalesce memory accesses, we propose novel packet transformation techniques that group and transpose the network packets. We then move a step further and improve the performance of other operations that typical network processing applications do rely—such as packet capture and decoding, TCP stream reassembly, and application-level protocol analysis. Particularly, we propose a multi-parallel stateful packet processing architecture that carefully parallelizes network traffic processing and analysis at three levels, using multi-queue network interfaces (NICs), multiple CPUs, and multiple GPUs. The proposed design avoids excessive locking, optimizes data transfers between the different processing units, and speeds up data processing by mapping different operations to the processing units where they are best suited. Finally, we integrate most of the proposed design principles and implementations into a single framework that is tailored for developing stateful packet processing applications on the GPU. Our proposed framework offers, among others, a modular mechanism for writing GPU-based packet processing applications, a zero-copy mechanism that avoids redundant memory copies between the network interface and the GPU, as well as a purely GPU-based implementation of flow state management and TCP stream reconstruction. Especially when consolidating multiple divergent network applications on the same device, our framework employs novel mechanisms for tackling control flow irregularities across GPU threads to allow efficient execution.

1.1 Thesis Statement

Successfully utilizing GPUs requires more than simply switching to a different hardware platform; it requires the careful design and implementation of software techniques optimized for the different execution model, constraints, and memory hierarchies contained in modern GPU architec-

tures. Due to the fact that typical discrete GPUs act as coprocessors, interconnected via the PCIe bus with the base host, more data transfers are required to complete the same amount of work. In addition, GPUs operate on a different, data-parallel, model. Even though network applications can take advantage of packet level parallelism, stateful workloads present further challenges, such as dependencies across packets and significant variations on packet processing. These different properties often prevent existing software from taking full advantage of the computational capabilities that GPUs can provide, however we show that properly architecting packet processing software for modern GPU architectures can improve performance significantly.

This dissertation proves that *it is possible to drastically improve the performance of stateful packet processing systems by employing modern graphics processors. This is achieved by employing careful data movements, scalable execution pipeline techniques and efficient packet scheduling designs.*

1.2 Contributions

This dissertation makes the following contributions:

- We show the implementation of a high-performance single-thread stateful network intrusion detection system, that utilizes the ubiquitous GPU to offload both string searching and regular expression matching operations. Our implementation extends the Snort IDS [111], which is the most popular and widely-used NIDS/NIPS. We have implemented novel packet transformation techniques, that group and transpose the network packets in order to tackle load imbalances across GPU threads and coalesce memory accesses on the GPU. We also characterize extensively the performance of different types of GPU memory hierarchies for signature matching on network packets, and identify the setup that performs best.
- We introduce a novel multi-parallel architecture for high-performance processing and stateful analysis of network traffic. Our architecture is based on inexpensive, off-the-shelf, general-purpose hardware, and combines multi-queue NICs, multi-core CPUs, and multiple GPUs. We present our prototype implementation based on Snort IDS [111], demonstrating that the proposed model is practical, scales well with the number of processing units, and can be adopted by

existing systems.

- We present a novel GPU-based framework, called GASPP, for high-performance passive or inline network traffic processing, which eases the development of GPU-assisted applications that process data at multiple layers of the protocol stack. Our framework employs, among others, (i) the first purely GPU-based implementation of flow state management and TCP stream reconstruction, (ii) novel mechanisms for tackling control flow irregularities across GPU threads, allowing efficient execution when consolidating multiple divergent network applications on the same device, and (iii) a zero-copy mechanism that avoids redundant memory copies between the network interface and the GPU, increasing significantly the throughput of cross-device data transfers.

1.3 Dissertation Overview

The dissertation is organized as follows. In Chapter 2 we provide some background information on software packet processing and describe the recent advancements in commodity, off-the-shelf, hardware. We primarily focus on multicore CPUs, general-purpose graphics processors, and server multi-queue network interfaces.

Chapter 3 shows how to change the execution flow of a complex stateful network processing application in order to offload specific computational tasks, such as the packet payload inspection, to the GPU. We describe the performance achieved by the GPU using different configurations and memory hierarchies and propose packet transformation techniques, that group and transpose the network packets in order to tackle load imbalances across GPU threads and coalesce memory accesses.

Chapter 4 extends this architecture by adding parallelism at other operations of the packet processing execution flow, such as the packet capturing, decoding, flow reassembly and application content normalization. Particularly, the new architecture parallelizes network traffic processing and analysis at three levels, using multi-queue NICs, multiple CPUs, and multiple GPUs. The proposed design avoids excessive locking, optimizes data transfers between the different processing units, and speeds up data processing by mapping different operations to the processing units where they are best suited.

Chapter 5 presents a network packet processing framework tailored to modern graphics processors, called GASPP, for high-performance passive or inline network traffic processing, which eases the development of applications that process data at multiple layers of the network protocol stack. GASPP integrates optimized GPU-based implementations of a broad range of operations commonly used in network traffic processing applications, including the first purely GPU-based implementation of network flow tracking and TCP stream reassembly. GASPP also employs novel mechanisms for tackling control flow irregularities across GPU threads, and sharing memory context between the network interface and the GPU.

Chapter 6 surveys prior work and Chapter 7 summarizes the conclusions of this dissertation and outlines future topics for research.

1.4 Publications

Parts of the work for this dissertation have been published in the following international refereed conferences:

- Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. June 2014, Philadelphia, PA, USA.
- Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Parallelization and Characterization of Pattern Matching using GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. November 2011, Austin, TX, USA.
- Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of the 18th ACM/SIGSAC Computer and Communications Security Conference (CCS)*. October 2011, Chicago, IL, USA.
- Giorgos Vasiliadis and Sotiris Ioannidis. GrAVity: A Massively Parallel Antivirus Engine. In *Proceedings of the 13th International Symposium On Recent Advances In Intrusion Detection (RAID)*. September 2010, Ottawa, Canada.
- Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos and Sotiris Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of*

the 12th International Symposium On Recent Advances In Intrusion Detection (RAID). September 2009, Saint-Malo, France.

- Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium On Recent Advances In Intrusion Detection (RAID)*, September 2008, Boston, MA, USA.

Chapter 2

Background Concepts and Trends

2.1 Software Packet Processing

The enormous expansion of the Internet is largely based on the basic principles on which its architecture is built upon, and that guarantee its resilience and its scalability. Foremost among them is the "end-to-end" design principle [115], which pushes most of the processing to the endpoints. The end-to-end principle suggests that specific application-level functions should not be built into the lower levels of the system—the core of the network. According to this model, intermediate nodes (i.e. switches, routers) are assigned only with the most basic operations, limited to the Layers 2 and 3 of the Open System Interconnection (OSI) stack. For example, a network router does not keep any sort of state neither inspects the contents of the packets. Pushing the responsibilities of guaranteeing a reliable transport to the endpoints helps the Internet scale to 996 million hosts as of July 2013 [12].

Nevertheless, the need for differentiated services (such as firewalls, network intrusion detection and prevention systems, traffic classification applications, content-based networking, etc.) that lie in the core of the Internet, instead of the end points, has challenged the end-to-end principle. These services need to perform complex packet processing operations at upper networking layers, which, unfortunately, does not supported by traditional edge routers. To address this evolution, specialized network appliances (called "middleboxes") are deployed, which typically perform

complex packet processing operations—ranging from deep packet inspection operations to packet encryption and redundancy elimination [121].

Packet-processing implemented in software promises to enable the fast deployment of new, sophisticated processing without the need to buy and deploy expensive new equipment. Software packet processing running on general-purpose platforms allow easy programmability, in contrast to high-end routers, that rely on specialized and closed hardware and software, and are notoriously difficult to extend, program, or otherwise experiment with [1, 16, 15]. In its simplest form, a packet processing architecture forwards packets from one interface to another, as appropriate. More complex packet processing operations include quality of service (QoS) enforcement, encryption of data streams, TCP offloading, and deep packet inspection (DPI). Recent work has demonstrated that software packet processing architectures are able to operate in high-packet-rate environments, while running sophisticated packet-processing applications [50, 51, 53, 60, 98, 148].

One of the most common processing is Deep Packet Inspection (DPI). DPI is a core component for many systems plugged in the network, such as traffic classifiers, packet filters, and network intrusion detection/prevention systems (NIDS/NIPS). Network components use DPI as an essential inspector, applied in different layers of the OSI model. Unlike the early beginnings of packet inspection, where it was applied in packet headers only (e.g., proxies and firewalls, etc.), nowadays, protocol complexity and obfuscation force us to inspect content in all encapsulated layers. For instance, Internet Service Providers (ISP) have been recently relying on DPI systems, which are the most accurate techniques for traffic identification and classification [18]. The use of DPI can result to better QoS, by identifying different styles of content and route them through different quality and/or speed networks. Similary, by examining the contents of the incoming packets, harmful traffic and malware can be identified and removed. As a consequence, DPI has evolved into a fast-growing application area both in terms of technology and market size – it is estimated that the market for DPI within the U.S. Government alone will be worth more than $7 billion over the next five years [5, 14].

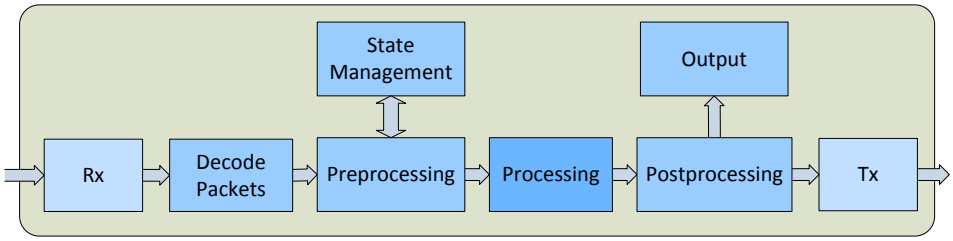


FIGURE 2.1: Network packet processing flow path.

2.1.1 Stateful Packet Processing

Flow tracking and TCP stream reconstruction are mandatory features of a broad range of network packet processing applications. Stateless applications, which treat each network packet in isolation, cannot distinguish if a given packet is part of an existing connection, is trying to establish a new one, or is just a rogue packet. Connection-aware network applications allow a more fine-grained control of network traffic that is needed, for example, to defeat spoofing attacks. Network intrusion detection and traffic classification systems typically inspect the application-layer stream, instead of individual packets, to identify patterns that span multiple packets and thwart evasion attacks [49, 142]. Stateful protocol analysis is also significant for monitoring and analyzing different events within a connection or session. A network intrusion detection sensor can retain valuable events and data during the lifetime of a session, and correlate them in order to identify attacks with multiple components that cannot be detected otherwise. The absence of stateful analysis, restrict the examination to a single packet only, completely isolated from the rest of the session.

The most common approach for stateful processing is to buffer incoming packets, reassemble them, and deliver "chunks" of the reassembled stream to higher-level processing elements [109, 111]. A major drawback of this approach is that it requires extra data copies and significant extra memory space. In Gigabit networks, where packet intervals can be as short as 1.25 μsec (in a 10GbE network, for an MTU of 1.5KB), packet buffering requires large amounts of memory even for very short time windows. To address these challenges, many approaches try to process as many packets as possible on-the-fly, instead of buffering them [49, 43].

Figure 2.1 depicts the processing stages of a typical packet processing application. We note that not all stages are required; depending on the

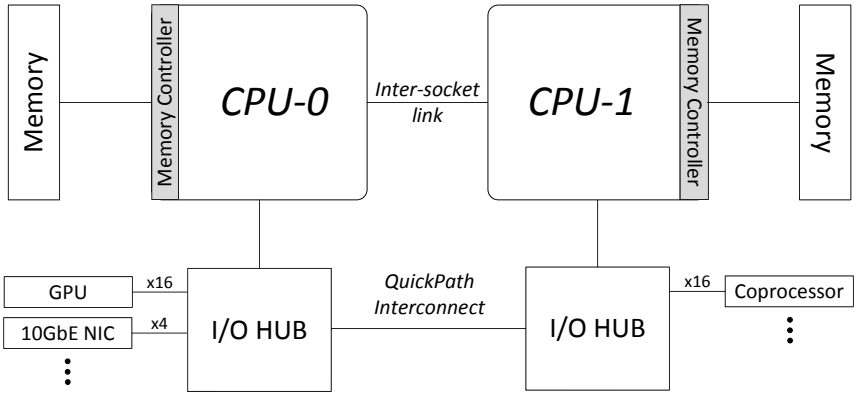


FIGURE 2.2: Diagram of a dual-CPU commodity system with integrated memory controllers and point-to-point interconnects between the processors.

needs of a specific application, several stages can be omitted. For example, a stateless firewall does not need to maintain state for each incoming connection. Initially, packets are captured from the incoming network interface (Rx) and decoded according to their encapsulated protocols. In case the application requires stateful processing, state management and flow reassembly is also performed on the incoming connections. The packets are then processed appropriately, based on the functionality of the packet processing application. Packets of different protocols or flows may be processed differently. Finally, if the application is operating inline, the packets are transmitted to the outgoing network interface. Alternatively, an output event is generated (e.g. an alert that is stored on disk).

2.2 Commodity Hardware

In this section we review some of the most important ideas and systems currently prevalent for parallel computing, which we use in this dissertation. In particular we discuss: multicore CPU issues, general-purpose GPU architecture, and multi-queue network interfaces.

2.2.1 Multicore CPUs

Symmetric multiprocessor (SMP) systems – i.e., equipped with two or more identical CPUs – were being used mostly by high-end computing systems. However, the recent advancements in semiconductor technology made them affordable for commodity computing devices. Nevertheless,

the problem with such commodity multicore architectures is that they do not alleviate the von Neumann bottleneck [33, 144], instead they aggravated it further. This is because in an SMP system, multiple processors are competing for the bandwidth to the same memory banks, hence multiple CPUs may starve for data at the same time. For example, until recently, the most common commodity architecture connected all processors with the memory through a single shared-bus, named the front-side bus (FSB). The memory controller (in this case a single entity within the Northbridge) serializes the accesses amongst the many CPUs that compete for the FSB's bandwidth. Furthermore, since more CPU chips have distinct cache hierarchies, a cache coherency protocol must also be implemented over the FSB, so that all processors have a consistent view of the entire physical memory address space.

Most recently, commodity multicore architectures, such as the Intel Nehalem [66], have adopted a technique to improve CPU-to-memory throughput. In order to avoid the performance penalty when multiple processors access the same memory, each physical processor is equipped with an integrated on-chip memory controller that access a separate memory bank, as shown in Figure 2.2. This technique, called non-uniform memory access (NUMA), has the potential to mitigate the memory contention amongst CPUs, assuming that data have been placed in the nearby physical memory. In case a CPU needs to access memory that is located in the remote CPU, NUMA architectures employ additional hardware to shuttle data between memory banks. Additional hardware also exists for maintaining memory consistency amongst caches, since virtually all NUMA architectures are cache-coherent. Typically, a single mechanism is used both to move data between CPUs and to keep their respective caches consistent; recent commodity implementations (e.g. Intel's QuickPath Interconnect, AMD's HyperTransport) use point-to-point (and packet oriented) links between the distinct cache controllers.

2.2.2 Multi-queue Network Interfaces

The recent advances in processor manufacturing have shown that leading vendors focus on increasing the number of independent CPU cores per silicon chip, rather than increasing the performance of individual processor cores. By contrast, the link speeds have continued to increase at an exponential rate – as a consequence, 10GbE commodity network interfaces

(NICs) are now commonplace. This means that a single core handling traffic at line speed from a single high speed interface has few, if any, cycles to spare. If many CPUs are used to service the same NIC resources, the contention overheads would be prohibitively expensive.

Consequently, multicore scaling has driven vendors to introduce multi-queue NIC hardware. A NIC with multi-queue capabilities can present itself as a virtual set of N individual NICs (where the value of N is equal to the number of CPU cores). The typical multi-queue NIC has the ability to classify the inbound traffic, through Receive-side Scaling (RSS) [94], to determine the corresponding destination receive (Rx) queue for each individual packet (a hashing function typically ensures that packets belonging to the same flow, e.g., TCP, are classified into the same queue). Once a destination Rx-queue is chosen, the NIC transfers the packets via Direct Memory Access (DMA), before issuing message signaled interrupts (MSI/MSI-X) to prompt a receive event solely for the chosen Rx-queue. If the hardware decides which Rx-queue to place the received packets onto, the software is responsible for classifying packets to be placed on transmit (Tx) queues. The kernel uses a driver-specific hash function for classification if one is provided, or the generic simple Tx hash function otherwise. In the latter case, the kernel makes no assumption about the underlying device capabilities, hence the classification may be suboptimal.

2.2.3 Graphics Processors

Modern Graphics Processing Units (GPUs) have evolved to massively parallel computational devices, containing hundreds of processing cores that can be used for general-purpose computing beyond graphics rendering. The architecture of modern GPUs is based on a set of *multiprocessors*, each of which contains a set of *stream processors* operating on SIMT (Single Instruction, Multiple Thread) programs, as shown in Figure 2.3. The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. For this reason, a GPU is ideal for parallel applications requiring high memory bandwidth to access different sets of data. Both NVIDIA and AMD provide convenient programming libraries to use their GPUs as a general purpose processor

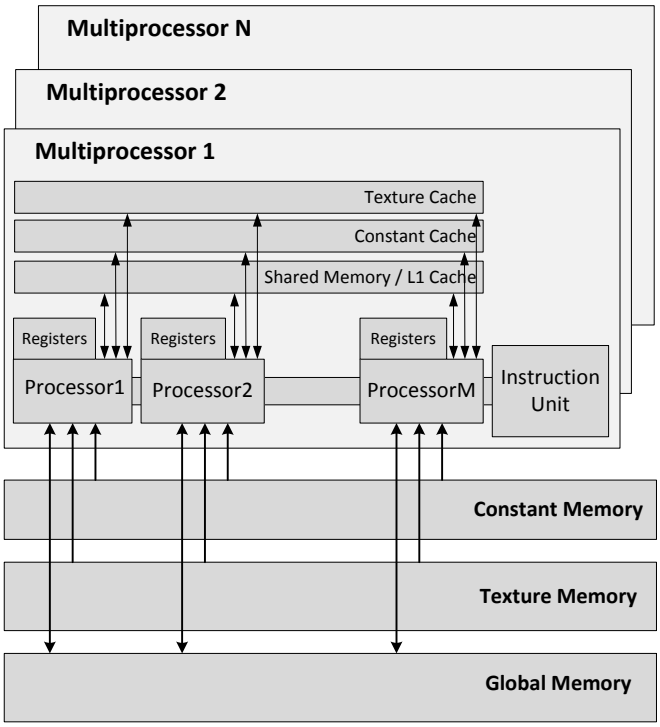


FIGURE 2.3: The NVIDIA CUDA architecture comprised of multiprocessors, each of which contains a set of stream processors.

(GPGPU), 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*. A typical GPU kernel execution takes the following four steps: (i) the DMA controller transfers input data from host memory to GPU device memory; (ii) a host program instructs the GPU to launch the kernel; (iii) the GPU executes threads in parallel; and (iv) the DMA controller transfers the result data back to host memory from device memory. A kernel 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 SIMT 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 [102]. 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.

In this dissertation, we have chosen to work with the NVIDIA architecture, which offers a rich programming environment and flexible abstraction models through the Compute Unified Device Architecture (CUDA) SDK [102]. The CUDA programming model extends the C programming language with directives and libraries that abstract the underlying GPU architecture and make it more suitable for general purpose computing. CUDA also offers highly optimized data transfer operations to and from the GPU. CUDA applications can run either on top of the closed-source NVIDIA CUDA runtime, or on top of the open-source Gdev runtime [76]. The NVIDIA CUDA runtime relies on the closed-source kernel-space NVIDIA driver and a closed-source user-space library. Gdev also supports the NVIDIA driver, as well as the open source Nouveau [17] and PSCNV [20] drivers. Both frameworks support the same APIs: CUDA programs can be written using the runtime API, or the driver API for low-level interaction with the hardware.

2.3 General-Purpose GPU (GPGPU)

Recently, graphics processors have become an important player for accelerating network applications [60, 72, 136, 138]. Their data-parallel model, in regards with the computational capabilities, fits nicely with packet processing operations.

2.3.1 Comparison with CPU.

The major difference between CPUs and GPUs is related to how the transistors are composed in the processor. In GPUs, most of the die area is

devoted to a large number of small ALUs (Arithmetic Logic Units), maximizing thread-level parallelism. In contrast, most CPU resources serve a large cache hierarchy and a control plane that is suitable for sophisticated acceleration of a single thread (i.e, out-of-order execution and branch prediction) [102].

Raw computation capacity. Several traffic processing applications are increasingly demanding for compute-intensive operations, such as encryption and compression. GPU can be an attractive source of extra computation power; for example, NVIDIA GTX480 offers an order of magnitude higher peak performance than X5550 in terms of MIPS (Million Instructions Per Second). Moreover, the recent trend shows that the GPU computing density is improving faster than CPU [104].

Memory access latency. Typical network packet processing applications exhibit large memory requirements that usually do not fit in the CPU cache. Eventhough the memory access latency can be potentially hidden with memory prefetches and out-of-order execution, it is often limited by CPU resources, such as the overall memory bandwidth and the instruction window size. Unlike CPU, GPU can effectively hide device memory access latencies with hundreds (or thousands) of threads. By spawning a sufficient number of threads, memory stalls can be minimized or even eliminated [114].

Memory bandwidth. Network packet processing applications that exhibit random memory access, such as Deterministic Finite Automaton (DFA) matching, exhaust the available memory bandwidth. For example, every 4-byte random memory access consumes 64-byte of memory bandwidth, which is the size of a cache line in x86 architecture. By offloading such memory-intensive operations to the GPU, we can benefit from larger memory bandwidth (177.4 GB/s for NVIDIA GTX480 versus 32 GB/s for Intel Xeon E5520) [60, 139]. Besides, the extra memory bandwidth provided, the GPU also alleviates congestion in the memory bandwidth of the CPU, which is primarily consumed on network I/O.

PCIe Interconnect. Current graphics cards are interconnected over PCIe 2.0 x16 link with the host, providing a theoretical bandwidth of 8 GB/s. In practice though, the effective bandwidth is smaller due to PCIe and DMA overheads, especially for small data transfer units. Table 2.1 shows

Buffer Size	1KB	4KB	64KB	256KB	1MB	16MB
Host to Device	2.04	7.1	34.4	42.1	44.6	45.7
Device to Host	2.03	6.7	21.1	23.8	24.6	24.9

TABLE 2.1: Sustained throughput for transferring data to the Graphics card device, and vice versa (Gbit/s).

the transfer rate to move data to the GPU device, and vice versa. We observe that with a large buffer, the rate for transferring to the device is over 45 Gbit/s, while transferring from device to host decreases to about 25 Gbit/s. The asymmetry in the data transferring throughput from the device, is probably related to the corresponding hardware setup (i.e., the interconnection between the motherboard and the graphics cards), and is speculated that future motherboards will alleviate this asymmetry.

2.3.2 CPU versus GPU

2.3.3 Programming Considerations

The implementation of efficient GPU programs requires the understanding of the underlying GPU architecture. The in-depth knowledge of several hardware characteristics and different memory hierarchies is often necessary to maximize performance. In this section, we discuss general considerations of GPU programming, in the context of network packet processing acceleration [60].

What to offload. GPU architecture focuses on the data processing unit much more than the control flow and caching unit. Therefore, the applications suitable to be run on GPU should be data intensive or have high data to instruction ratio (i.e., process similar operations on different input data). Besides, the offloaded portion to GPU should have non-trivial costs to pay off the extra overhead of transferring the data to and from the GPU. Computation- and memory-intensive algorithms with high regularity typically suit well for GPU acceleration.

How to parallelize. Given the massively parallel architecture of modern GPUs, the most natural way to parallelize packet processing is to map each network packet to a different GPU thread. The parallelism is achieved by buffering incoming packets and transferring them to the graphics card in large batches. Although this buffering scheme adds some latency to the processing path, it pays off in terms of the processing throughput

that can be sustained. In case we want to parallelize within a packet (i.e. intra-packet parallelism), we can map each packet into multiple threads and assign a different portion to each thread; this might be suitable for performing operations that require fine-grained parallelism.

Control flow divergence. Control flow divergence plays a significant role in the performance achieved by the GPU. The reason is that the threads are grouped together into logical units known as warps (in current NVIDIA GPU architectures, 32 threads form a warp) and mapped to SIMT units. For optimal performance, the SIMT architecture of CUDA demands to have minimal code-path divergence caused by data-dependent conditional branches within a warp. In order to avoid warp divergence for differentiated packet processing (e.g., packet inspection with different inspection engines), we can classify and group network packets into separate GPU warps, such that all threads within a warp follow the same execution code path [137].

Data structure usage. Simple data structures such as arrays and hash tables are recommended in GPU. Data structures that require serialized access (e.g., linked lists) or are highly scattered in memory (e.g., tries/trees) would make update the data difficult and degrade the performance due to small caches in GPU and uncoalesced memory access pattern [102].

Chapter 3

Accelerating a Single-Threaded Network Intrusion Detection Systems using Graphics Hardware

In this chapter, we show how to exploit the underutilized power of the graphics processing unit (GPU) to offload specific intensive tasks from a typical packet processing application, i.e. the network intrusion detection system (NIDS). Particularly, we present the design, implementation, and evaluation of string searching and regular expression algorithms engines running on GPUs. We have integrated these implementations in the popular Snort intrusion detection system [22] to offload both string and regular expression matching computation. The significant spare computational power and data parallelism capabilities of modern GPUs permit the efficient matching of multiple inputs at the same time against a large set of fixed string patterns and regular expressions.

3.1 Architecture

The overall architecture of our proposed architecture, called Gnort, is shown in Figure 3.1. We can separate the architecture of our system in several different tasks: packet capturing, preprocesing, the transfer of the network packets to the GPU, the string matching processing on the GPU, and the transfer of the matching results back to the CPU, where all the

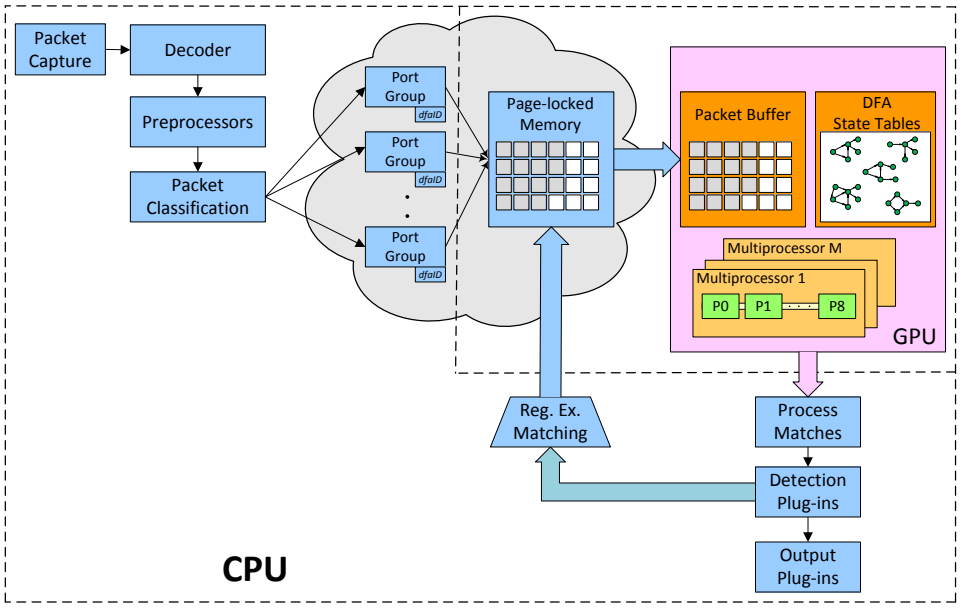


FIGURE 3.1: Overview of the single-threaded GPU-based network intrusion detection architecture.

remaining conditions of the rules are checked. Whenever a packet needs to be scanned against a regular expression, it is subsequently transferred back to the GPU where the actual matching takes place.

3.1.1 Packet Capturing and Decoding

Our system uses a single 1 GbE network interface. Capturing packets at these rates is sufficient with generic packet capturing libraries, such as the pcap library [92]. After network packets have been captured, the decoding stage then validates each packet, detects protocol anomalies, and populates a data structure that contains the tightly-encoded protocol headers and associated information of each packet.

3.1.2 Preprocessing

Preprocessing modules are built on top of the decoding subsystem and preprocessor engines of Snort IDS. The purpose of the decoder is to parse the packet headers according to lower-layer protocols (Ethernet, IP, TCP, and so on). After packets have been decoded, they are sent through a preprocessing stage that includes *flow reassembly* and *protocol analysis*.

TCP packets are reassembled into TCP streams to build the entire ap-

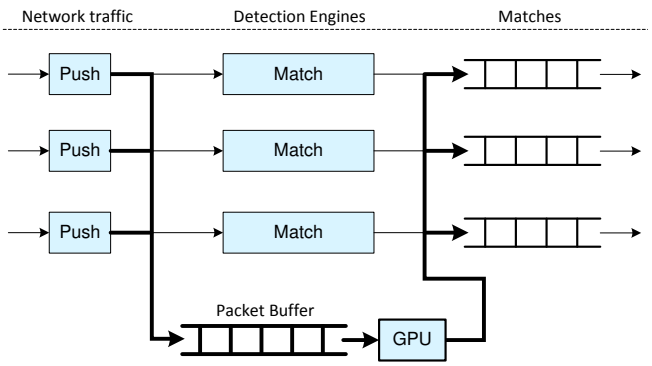


FIGURE 3.2: Batching different packets to a single buffer. This scheme results to lower memory consumption and also reduces response latency for port groups with low traffic.

plication dialog before they are forwarded to the pattern matching engine. Packets that belong to the same direction of a TCP flow, are merged into a single packet by concatenating their payloads according to the TCP protocol. Inspecting the concatenation of several network packets, instead of each network packet separately, enables the handling of overlapping data and other TCP anomalies. This allows the detection engine to match patterns that span multiple packets. Content normalization is also applied for higher-level protocols, such as HTTP and DCE/RPC, to remove potential ambiguities and neutralize evasion tricks.

Once flow reassembly and normalization is complete, the data is forwarded to the detection engine, which performs signature matching on the incoming traffic. As the signature matching is performed on the GPU, the first thing to consider is how to transfer the packets to the memory space of the GPU.

3.1.3 Transferring Packets to the GPU

The payload of the network packets that need to be checked, has to be transferred 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 than making each transfer separately as shown in Table 2.1 (page 18). Thus, we have chosen to copy the packets to the GPU in batches.

Snort organizes the content signatures in groups, based on the source

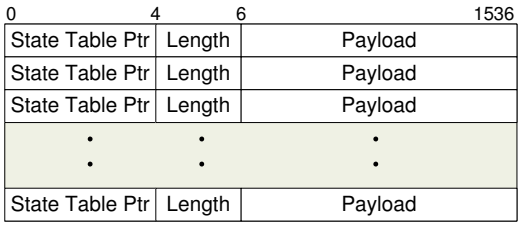


FIGURE 3.3: Network packet buffer format with fixed-size slots.

and destination port numbers of each rule. A separate detection engine instance is used to search for the string patterns of a particular rule group. 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. Consequently, only one buffer is needed, instead of one for each port group, as shown in Figure 3.2. 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, as shown in Figure 3.3. This results to significantly lower memory consumption and reduces response latency for port groups with low traffic. Whenever the buffer gets full, all packets are transferred to the GPU in one operation.

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 [102]. 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 the memory space of the graphics card that can be read later by the GPU through the kernel invocation. While the GPU is processing the packets of the first buffer, the CPU will copy newly arrived packets in the second buffer.

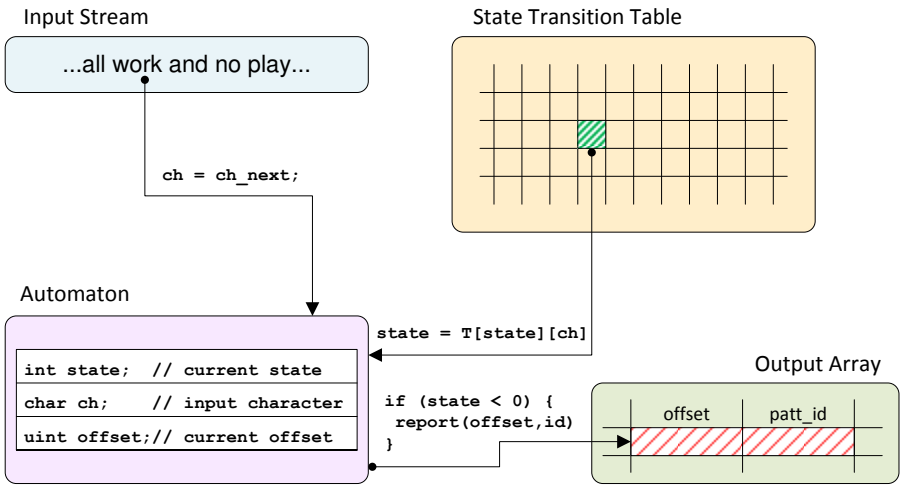


FIGURE 3.4: DFA matching on the GPU. The algorithm moves over the input data stream one byte at a time and switches the current state according to the state transition table. When a final-state is reached, a match has been found, and the corresponding offset is marked.

3.1.4 String Matching on the GPU

Once the packets have been transferred to the GPU, the next step is to perform the string matching operation. We have ported the Aho-Corasick algorithm [27] to run on the graphics card. The Aho-Corasick algorithm seems to be a perfect candidate for the data parallel execution model of the GPU. As shown in Figure 3.4, 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 excess control flow instructions that would probably lead to thread divergence.

In our GPU implementation, the deterministic finite automaton (DFA) of the state machine is represented 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, and each cell consists of four bytes. The first two bytes contain the next state to move, while the other two contain an indication whether the state is a final state or not. In case the state is final, the corresponding cell will contain the unique identification number (*ID*) of the matching string, otherwise zero. The format of the state table allows its easy mapping to the different memory types that mod-

ern GPUs offer. Mapping the state table to each memory yield different performance improvements, as we will see in Section 3.4.3. 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. In Section 4.2.2 we implement more efficient storage structures, like those proposed in [101].

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 the memory space of the GPU. At the searching phase, all state machine tables reside only in GPU memory. As of NVIDIA Kepler architecture and onwards, texture, constant and global memories are all cached. A cache hit consumes only one cycle, instead of several hundreds in case of transfers from the device memory. Since the Aho-Corasick algorithm exhibits strong locality of references [47], the use of cached memory for storing the state machine tables boosts GPU execution time, as shown in Section 3.4.3.

We have implemented a thread-per-packet approach for the Aho-Corasick searching phase; each thread is assigned a whole reassembled packet to search in parallel. 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.

3.1.5 Transferring the Results to the Host

After the string 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 string 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 string 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.

- *String 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" *ID*s 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.
- *Regular expression matching.* In order to describe a wider variety of payload signatures, many rules in Snort contain regular expressions. All network packets that have fulfilled the preliminary criteria of a distinct rule may then forwarded for regular expression scanning, if needed. In the next section we describe how we extended our architecture to make use of the GPU for offloading regular expression matching from the CPU, and decreasing its overall workload.

3.1.6 Regular Expression Matching on Graphics Processors

A regular expression is a very convenient form of representing a set of strings. They are usually used to give a concise description of a set of patterns, without having to list all of them. For example, the expression $(a \mid b) * aa$ represents the infinite set {*"aaa"*, *"baa"*, *"abaa"*, ...}, which is the set of all strings with characters a and b that end in aa. Formally, a regular expression contains at least one of the operations described in Table 3.1.

A deterministic finite automaton (DFA) represents a finite state machine that recognizes a regular expression. A finite automaton is represented by the 5-tuple (Σ, Q, T, q_0, F), where: Σ is the alphabet, Q is the set of states, T is the transition function, q_0 is the initial state, and F is the set of final states. Given an input string $I_0 I_1 ... I_N$, a DFA processes the input as follows: At step 0, the DFA is in state $s_0 = q_0$. At each subsequent step i, the DFA transitions into state $s_i = T(s_{i-1}, I_i)$. To alleviate backtracking at the matching phase, each transition is unique for every state and character combination. A DFA accepts a string or a regular expression if, starting from the initial state and moving from state to state, it reaches a final state.

Name :	*Reg. Expr.*	*Designation*
Epsilon	ϵ	{""}
Character	α	For some character α.
Concatenation	RS	Denoting the set $\{\alpha\beta \mid \alpha$ in R and β in $S\}$. e.g., {"ab"}{"d", "ef"} = {"abd", "abef"}
Alternation	$R\|S$	Denoting the set union of R and S. e.g., {"ab"}|{"ab","d","ef"} = {"ab","d","ef"}.
Kleene star	$A*$	Denoting the smallest super-set of R that contains ϵ and is closed under string concatenation. This is the set of all strings that can be made by concatenating zero or more strings in R. e.g., {"ab", "c"}* = {ϵ,"ab","c","abab","abc","cab","ababab",...}

TABLE 3.1: Regular expression operations.

The transition function can be represented by a two-dimensional table T, which defines the next state $T[s, c]$ for a state s and a character c.

Figure 3.5 depicts the top-level diagram of our regular expression matching engine. Whenever a packet needs to be scanned against a regular expression, it is transferred to the GPU where the actual matching takes place. The data parallel execution of the GPU is ideal for creating multiple instantiations of regular expression state machines that will run on different stream processors and operate on different data.

We use a separate buffer for temporarily storing the packets that need to be matched against a regular expression. Every time the buffer fills up, it is transferred to the GPU for execution. The content of the packet, as well as an identifier of the regular expression that needs to be matched against, are stored in the buffer as shown in Figure 3.3. Since each packet may need to be matched against a different expression, each packet is "marked" so that it can be processed by the appropriate regular expression at the search phase. Therefore, each row of the buffer contains a special field that is used to store a pointer to the state machine of the regular expression the specified packet should be scanned against.

Every time the buffer is filled up, it is processed by all the stream processors of the GPU at once. The matching process is a kernel function capable of scanning the payload of each network packet for a specific ex-

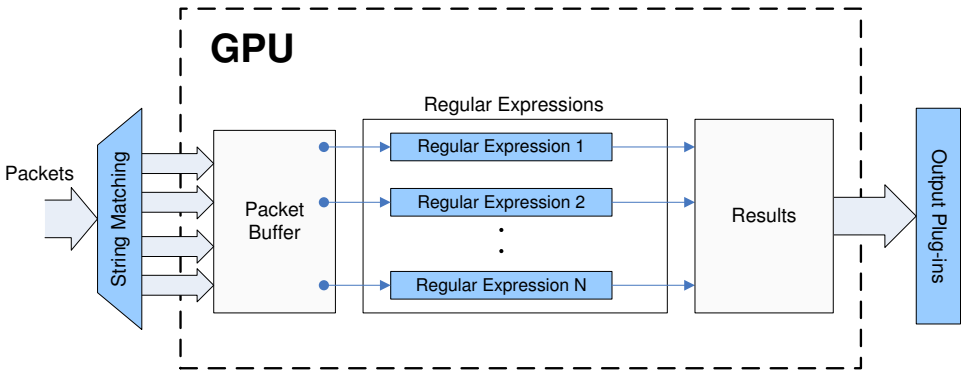


FIGURE 3.5: Overview of the regular expression matching engine in the GPU. Each requested packet is matched against a different regular expression independently.

pression in parallel. The kernel function is executed simultaneously by many threads in parallel. Using the identifier of the regular expression, each thread will scan the whole packet in isolation. The state machines of all regular expressions are stored in the memory space of the graphics processor, thus they can be accessed directly by the stream processors and search the contents of the packets concurrently.

A major design decision for GPU regular expression matching is the type of automaton that will be used for the searching process. DFAs are far more efficient than the corresponding NFAs in terms of speed, thus we base our design of a DFA architecture capable of matching regular expressions on the GPU.

Given the rule set of Snort, all the contained regular expressions are compiled and converted into DFAs that are copied to the memory space of the GPU. The compilation process is performed by the CPU off-line at start-up. Each regular expression is compiled into a separate state machine table that is transferred to the memory space of the GPU. During the searching phase, all state machine tables reside in GPU memory only.

Our regular expression implementation currently does not support a few PCRE keywords related to some look-around expressions and back references [19]. Back references use information about previously captured sub-patterns which is not straightforward to keep track of during searching. Look-around expressions scan the input data without consuming characters. In the current Snort default rule set, less than 2% of the

rules that use regular expressions make use of these features. Therefore our regular expression compiler is able to generate automata for the vast majority of the regular expressions that are currently contained in the Snort rule set. To preserve both accuracy and precision in attack detection, we use a hybrid approach in which all regular expressions that fail to compile into DFAs are matched on the CPU using a corresponding NFA, in the same way the vanilla Snort does. Using the above hybrid approach for the current Snort ruleset, more than 95% of the regular expressions can be converted into DFAs in less than 200 MB of memory, as we will see in Section 3.4.4. The remaining 5% of expressions will be matched on the CPU using NFAs.

Having converted the regular expressions to the corresponding DFAs, the next step is to perform the matching against the requested packets. The simplest approach would be to match each incoming network packet against each DFA individually. Such an approach would be a highly computationally intensive process though. It is obvious that if the rule set consists of m regular expressions, the computational complexity for n packets will be $O(nm)$. Thus, in a massively parallel GPU environment, we could assign a different regular expression to each stream processor and let the matching process execute in parallel for each incoming network packet. Therefore, by assigning a different regular expression on each stream processor, a whole packet will be searched against all of them at once.

However, NIDSes usually contain many more regular expressions that have to be matched against each captured packet, thus each stream processor have to be assigned more than one regular expression, that will search sequentially. As such, even for a multi-processor device, the process of each packet may be repeated several times, until it has been matched against all possible expressions. As we will see in Section 3.4.6, the parallel execution of many regular expressions on the GPU can speed-up the raw packet throughput when compared with the CPU, although fails to scale as the number of regular expressions increases, since the number of processors is fixed.

3.1.7 Execution Flow Overview

Figure 3.6 shows the pipelined execution of all phases. Each task, represented by a different box, is executed either on the CPU or on the GPU. The arrows show how the two processing units communicate with each

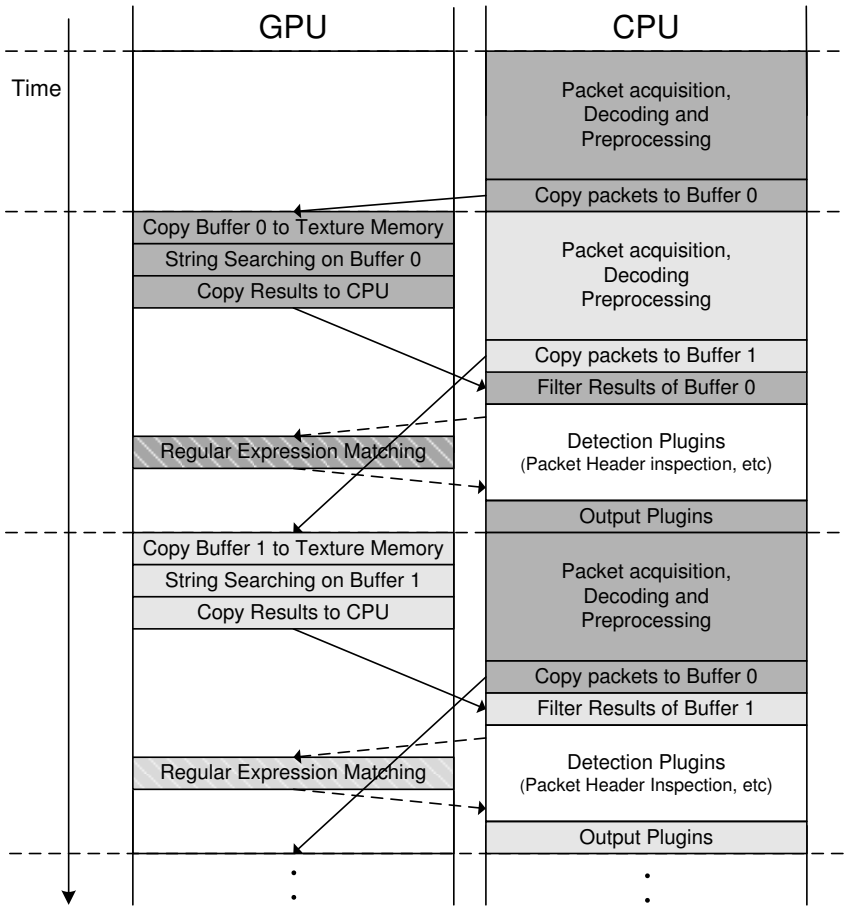


FIGURE 3.6: Execution flow of the single-threaded GPU-assisted NIDS. GPU communication and computation can overlap with CPU execution.

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 gathers packets from the network link using the pcap [92] library. The decoding stage then validates the captured network packets, detects protocol anomalies, and populates a data structure that contains the tightly-encoded protocol headers and associated information of each packet. Any configured preprocessors may then optionally be invoked at this stage.

The payload of the network packets that need to be checked against a detection engine is then copied to the packet buffer. Every time the buffer

gets full, it is copied to the memory space of the GPU where it will be checked for attack signatures by the corresponding detection engines.

The string matching as well as the transfer of the packets to the GPU are performed in an asynchronous fashion. This feature allows packet processing to be done concurrently on the CPU during GPU execution. By using the double-buffering scheme that we described in Section 3.1.3, 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 3.6.

In case the second buffer is full while the GPU is still processing the first buffer, the CPU is busy-waiting until the GPU finishes. As we will see in Section 3.4 though, the computational throughput of the GPU can be increased by providing a sufficient number of network packets for processing every 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 string 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.

Whenever a packet needs to be matched against a regular expression, it is further copied to a separate buffer that is transferred to the GPU for evaluation every time it fills up. The GPU process all network packets contained in the buffer at once, similar to the string searching process, and returns the matching results back to the host.

3.2 Implementation

In this section, we present the details of our implementation, which is based on the NVIDIA CUDA architecture. First, we describe how the gathered network packets are collected and transferred to the memory space of the GPU, a process that is identical for both string searching and regular expression matching. The GPU is not able to directly access the captured packets from the network interface, thus the packets must be copied by the CPU. Next, we describe how patterns—both strings and regular expressions—are compiled and used directly by the graphics pro-

cessor for efficiently inspecting the incoming data stream.

3.2.1 Collecting packets on the CPU

An important performance factor of our architecture is the data transfers to and from the GPU. For that purpose, we use page-locked memory, which is substantially faster than non-page-locked memory, since it can be accessed directly by the GPU through Direct Memory Access (DMA). A limitation of this approach is that page locked memory is of limited size as it cannot be swapped. In practice though this is not a problem since modern PCs can be equipped with ample amounts of physical memory.

Having allocated a buffer for collecting the packets in page-locked memory, every time a packet is classified to be matched against a specific regular expression, it is copied to that buffer and is "marked" for searching against the corresponding finite automaton. We use a double-buffer scheme to permit overlap of computation and communication during data transfers between the GPU and CPU. Whenever the first buffer is transferred to the GPU through DMA, newly arriving packets are copied to the second buffer and vice versa.

A slight complication that must be handled comes from the TCP stream reassembly functionality of modern NIDSs, which reassembles distinct packets into TCP streams to prevent an attacker from evading detection by splitting the attack vector across multiple packets. In Snort, the Stream5 preprocessor aggregates multiple packets from a given direction of a TCP flow and builds a single packet by concatenating their payloads, allowing rules to match patterns that span packet boundaries. This is accomplished by keeping a descriptor for each active TCP session and tracking the state of the session according to the semantics of the TCP protocol. Stream5 also keeps copies of the packet data and periodically "flushes" the stream by reassembling all contents and emitting a large pseudo-packet containing the reassembled data.

Consequently, the size of a pseudo-packet that is created by the Stream5 preprocessor may be up to 65,535 bytes in length, which is the maximum IP packet length. However, assigning the maximum IP packet length as the size of each row of the buffer would result in a huge, sparsely populated array. Copying the whole array to the device would result in high communication costs, limiting overall performance.

A different approach for storing reassembled packets that exceed the

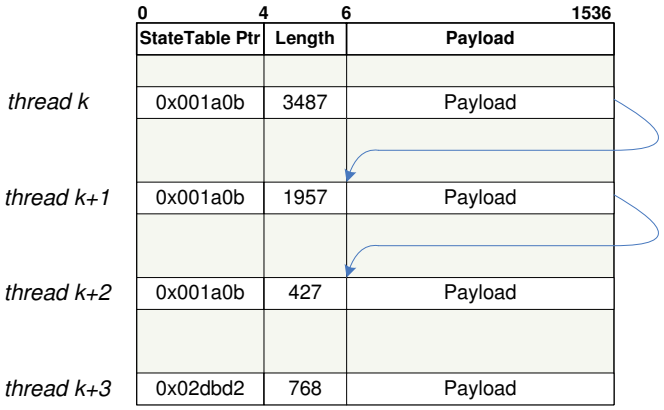


FIGURE 3.7: Matching packets that exceed the MTU size.

Maximum Transmission Unit (MTU) size, without altering the dimensions of the array, is to split them down into several smaller ones. The size of each portion of the split packet will be less or equal to the MTU size and thus can be copied in consecutive rows in the array.

Each portion of the split packet is processed by different threads. To avoid missing matches that span multiple packets, whenever a thread searches a split portion of a packet, it continues the search up to the following row (which contains the consecutive bytes of the packet), until a *final* or a *fail* state is reached, as illustrated in Figure 3.7. While matching a pattern that spans packet boundaries, the state machine will perform regular transitions. However, if the state machine reaches a final or a fail state, then it is obvious that there is no need to process the packet any further, since any consecutive patterns will be matched by the thread that was assigned to search the current portion.

3.2.2 GPU-based implementation of String Matching

We take advantage of all the available streaming processors of the GPU and utilize them by creating multiple data processing threads. An important design decision is how to assign the input data to each thread. The Aho-Corasick algorithm performs multi-pattern search, which means that all patterns of a group are searched concurrently. As shown in Figure 3.8, each packet is processed by a different thread. As we will see in Section 3.4.3, by increasing the number of packets that are processed at once, the multithreaded capabilities of the GPU at hiding memory laten-

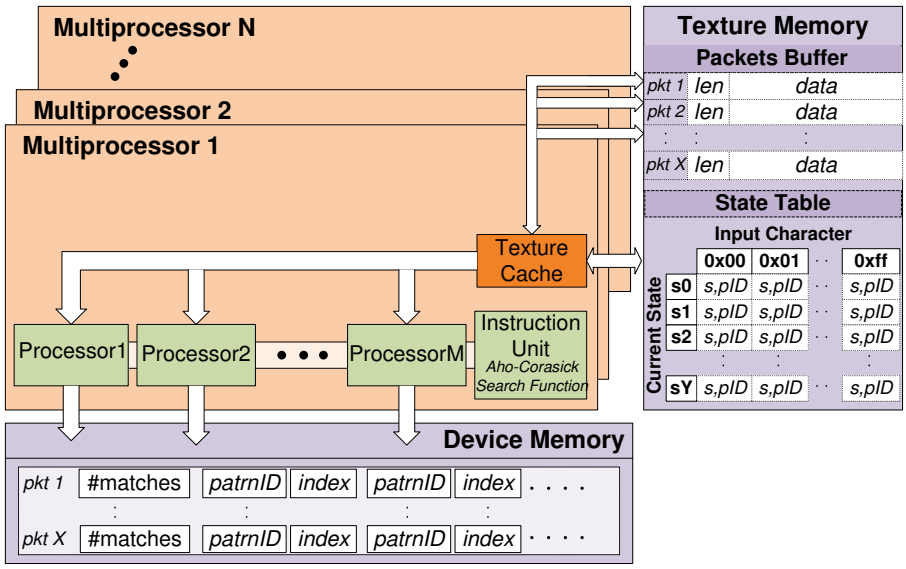


FIGURE 3.8: The GPU string matching parallelization approach. Each packet is processed by a different stream processor independently of the others.

cies results to an increase in the throughput sustained by the GPU. 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 Section 3.3.3 we propose optimizations to tackle such load imbalances.

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.

3.2.3 GPU-based Implementation of Regular Expression Matching

In this section we describe how regular expressions are compiled and used directly by the graphics processor. First, we describe how regular expressions are transformed to state machines to efficiently utilize the memory

space of modern graphics cards. Then, we analyze the inspection process of the incoming network packets that incur in parallel.

Compiling PCRE Regular Expressions to DFA state tables

Many existing tools that use regular expressions have support for converting regular expressions into DFAs [37, 19]. The most common approach is to first compile them into NFAs, and then convert them into DFAs. We follow the same approach, and first convert each regular expression into an NFA using the Thompson algorithm [132]. The generated NFA is then converted to an equivalent DFA incrementally, using the *Subset Construction* algorithm [108]. The basic idea of subset construction is to define a DFA in which each state is a set of states of the corresponding NFA. Each state in the DFA represents a set of active states in which the corresponding NFA can be in after some transition. The resulting DFA achieves $O(1)$ computational cost for each incoming character during the matching phase.

A major concern when converting regular expressions into DFAs is the *state-space explosion* that may occur during compilation [38]. To distinguish among the states, a different DFA state may be required for all possible NFA states. It is obvious that this may cause exponential growth to the total memory required. This is primarily caused by wildcards, e.g. `.*`, and repetition expressions, e.g. `a(x,y)`. A theoretical worst case study shows that a single regular expression of length n can be expressed as a DFA of up to $O(\Sigma^n)$ states, where Σ is the size of the alphabet, i.e. 2^8 symbols for the extended ASCII character set [62]. Due to state explosion, it is possible that certain regular expressions may consume large amounts of memory when compiled to DFAs.

To prevent greedy memory consumption caused by some regular expressions, we use a hybrid approach and convert only the regular expressions that do not exceed a certain threshold of states; the remaining regular expressions will be matched on the CPU using NFAs. We track of the total number of states during the incremental conversion from the NFA to the DFA and stop when a certain threshold is reached. As shown in Section 3.4.4, setting an upper bound of 5000 states per expression, more than 97% of the total regular expressions can be converted to DFAs. The remaining expressions will be processed by the CPU using an NFA schema, just like the stock implementation of Snort.

Each constructed DFA is a two-dimensional state table array that is mapped linearly on the memory space of the GPU. 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 contains the next state to move to, as well as an indication of whether the state is a final state or not. Since transition numbers may be positive integers only, we represent final states as negative numbers. Whenever the state machine reaches into a state that is represented by a negative number, it considers it as a final state and reports a match at the current input offset. The state table array is mapped on the memory space of the GPU, as we describe in the following section.

Regular Expression Matching

We have investigated storing the DFA state table both as textures in the texture memory space, as well as on the linear global memory of the graphics card. A straightforward way to store the DFA of each regular expression would be to dynamically allocate global device memory every time. However, texture memory can be accessed in a random fashion for reading, in contrast to global memory, in which the access patterns must be coalesced [102]. This feature can be very useful for algorithms like DFA matching, which exhibit irregular access patterns across large datasets. Furthermore, texture fetches are cached, increasing the performance when read operations preserve locality. However, CUDA does not support dynamic binding of memory to texture references. Therefore, it is not feasible to dynamically allocate memory for each state table individually and later bind it to a texture reference. To overcome this limitation, we pre-allocate a large amount of linear memory that is statically bound to a texture reference. All constructed state tables are stored sequentially in this texture memory segment.

During the searching phase, each thread searches a different network packet in isolation. Whenever a thread matches a regular expression on an incoming packet, it reports it by writing the event to a single-dimension array allocated in the global device memory. The size of the array is equal to the number of packets that are processed by the GPU at once, while each cell of the array contains the position within the packet where the match occurred.

3.3 Optimizations

3.3.1 Exploring Device Memory Hierarchies

Both our string searching and regular expression matching operations are implemented using DFAs. The two major tasks of DFA matching, as depicted in Figure 3.4, is reading the input data and fetching the next state from the device memory. These memory transfers can take up to several hundreds of nanoseconds, depending on the stream conditions and congestion.

In general, memory latencies can be hidden by running many threads in parallel. Multiple threads can improve the utilization of the memory subsystem by overlapping data transfer with computation. To obtain the highest level of performance, we performed several tests to determine how the computational throughput is affected by the number of threads. In Section 3.4.3 we discuss how the memory subsystem is utilized when there is a large number of threads running in parallel.

Moreover, we have investigated storing the network packets and the DFA state table both in the global memory space, as well as in the texture memory space of the graphics card. The texture memory can be accessed in a random fashion for reading, without the need to follow any coalescence rules. Furthermore, texture fetches are cached, increasing the performance when read operations preserve locality. In addition, texture cache is optimized for 2D spatial locality; to that end, we have investigated the use of both 1D and 2D textures. A programming limitation when dealing with 2D textures, is that the maximum y-dimension is equal to 65,536. Therefore, in order to map large state tables, we split the initial table into several smaller (each of which contains $64K$ states at most) and align them sequentially. In order to find the transitions of a given state at the matching phase, it is first divided with 65,536 in order to find the subtable that resides.

When using 1D linear memory, the maximum transition table that can be mapped to texture memory is 512 MB (totalling 524,288 states, since each state holds 1024 bytes for transitions). The theoretical dimensions of the maximum 2D texture are equal to 64K $*$ 32K elements, which are by far greater than the total amounts of memory that a modern GPU holds (i.e., up to 3 GB currently). Therefore, for cases that a single 1D state table

is greater than 512 MB, we bind only the initial part of the table to the texture memory, leaving the rest in the global device memory.

3.3.2 Optimizing GPU Memory Accesses

One important optimization is related to the way the input data is loaded from the device memory. Since the input symbols belong to the ASCII alphabet, they are represented with 8 bits. However, the minimum size for every device memory transaction is 32 bytes. Thus, by reading the input stream one byte at a time, the overall memory throughput would be reduced by a factor of up to 32. To utilize the memory more efficiently, we redesigned the input reading process such that each thread is fetching multiple bytes at a time instead of one. We have explored fetching 4 or 16 bytes at a time using the `char4` and `int4` built-in data types, respectively. The `int4` data type is the largest data type that can be used for texture alignment, allowing the utilization of about 50% of the memory bandwidth.

Finally, we tried to stage some data on the on-chip shared memory, but there was not any improvement due to the following reasons. First, the tradeoff of copying the data to the shared memory is worse than the benefit that the shared memory can provide, since each byte of the input is accessed only once. Second, by not using the shared memory, the performance of global memory accesses is boosted, since shared memory and L1-cache are implemented using the same physical memory. Therefore, in our implementation we do not take advantage of the shared memory. Nevertheless, input data is transferred in four 32-bit registers using the `int4` built-in data type, and are accessed byte-by-byte, through the `.x`, `.y`, `.z` and `.w` fields.

3.3.3 Packets Layout Transformations

As the type of received network packets is typically very mixed, it is important to find an appropriate mapping between threads and network packets, such that: (i) the workload will be equally distributed among threads (i.e., the corresponding packet lengths within a thread warp will have small fluctuations), in order to maintain symmetrical processing effort, and (ii) the memory transactions when reading the network packets from the global device memory will be coalesced. We have realized two ways to achieve the above goals at runtime: packet grouping and packet

transposition.

Packet Grouping

Packet grouping takes as input the packet index that is shown in Figure 5.6 and creates a new index redirection array, in which the packets are sorted based on their packet lengths. After the transformation, the computation and data accesses remain the same as before, although computations are executed by different threads. As such, the transformation is applicable and does produce the same output as the original program does. The thread-packet remapping operation occurs dynamically at runtime, hence it is crucial that the overhead of the mapping process would be small; otherwise it would jeopardize the overall performance.

To create the new index redirection array, we use the MGPU Sort library [120], which is based on Radix key-value sort [107]. Specifically, we assign a separate keys for each packet, that is its length. The keys are calculated on the GPU, using a separate thread for each packet, and are used by the radix sort algorithm to group the packets. This results to a new index redirection array, in which packets are grouped based on their length.

Packet Transposition

The thread-packet mapping, that described previously, produces a balanced distribution across the GPU thread warps. Still, in GPU, memory reference patterns strongly affect the effective memory bandwidth. For instance, if the words accessed by a warp fall into n different segments of global memory (a segment contains 32, 64, or 128 consecutive bytes), the GPU needs to conduct n memory transactions for those accesses; when the threads in a warp access memory locations in a small range, all the references by that warp may take only one transaction; such references are termed *coalesced memory accesses*.

To coalesce memory accesses, the corresponding packets are first copied to sequential locations, and opportunely padded in order to maintain the same dimensionality in the packet buffer array. The packet buffer is then transposed directly on the GPU through an already optimized kernel which exploits the shared memory as temporary storage to read data, perform the transposition and write the transposed data back in memory all with coalesced accesses, similar to [133]. The new, transformed, packet

buffer layout allows coalesced memory accesses.

3.4 Performance Evaluation

The performance evaluation of our proposed architecture is divided as follows: first, we evaluate our string searching and regular expression matching implementations in isolation. For each implementation, we measure the raw computational performance achieved in the GPU, as well as the memory requirements and the communication overheads introduced. We also examine how our matching implementations affect the overall performance of Snort in a realistic scenario. Finally, in Section 3.4.5 we measure the overall performance achieved by our prototype implementation, when both string searching and regular expression matching computations are offloaded to the GPU.

3.4.1 Hardware Platform

For our experiments, we used an GeForce GTX480 card, which consists of 480 cores, organized in 15 multiprocessors, and 1.5 GB of GDDR5 memory. Our base system has two processor sockets, each with one Intel Xeon E5520 Quad-core CPU at 2.27 GHz and 8192 KB of L3-cache. The sockets are connected to each other and to the I/O hub via dedicated high-speed point-to-point links.

We used both synthetic signatures as well as the default rule set released with Snort 2.6. The set consists of 7,179 rules that contain a total of 11,775 `pcre` regular expressions. All preprocessors were enabled, except the HTTP inspect preprocessor, in order to force all web traffic to be matched against corresponding rules regardless of protocol semantics.

3.4.2 Network Traces

For our experiments, we use both synthetic network traces of fixed-size packets, as well as the following real network traces:

- *U-Web:* A full payload trace of anonymized HTTP traffic captured in a university campus. The trace totals 194 MB, 280,088 packets, and 4,711 flows.
- *SCH-Web:* A full payload trace of anonymized HTTP traffic captured at the access link that connects an educational network with thousands of hosts to the Internet. The trace contains 365,538 packets in 14,585 different flows, resulting to about 164 MB of data.

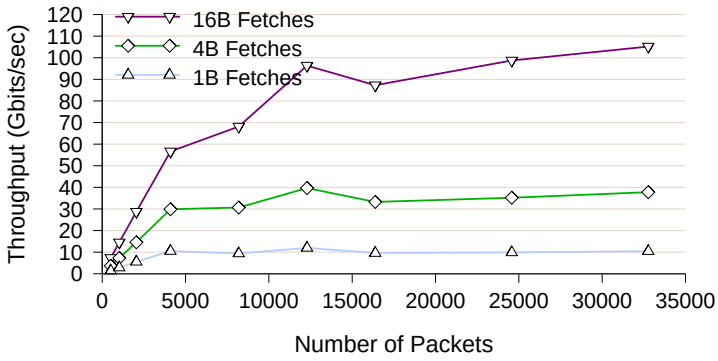


FIGURE 3.9: Impact of word accesses when fetching data from the global device memory.

- *LLI:* A full payload trace from the 1998-1999 DARPA intrusion detection evaluation set of MIT Lincoln Lab [26]. The trace is a simulation of a large military network and generated specifically for IDS testing. It contains a collection of ordinary-looking traffic mixed with attacks that were known at the time. The whole trace is about 382 MB and consists of 1,753,464 packets and 86,954 flows.
- *Equinix:* A trace-driven workload based on a headers-only packet trace captured by captured by CAIDA's *equinix-sanjose* monitor [25].

3.4.3 String Matching

Fetching the packets from the device memory

In our first experiment, we evaluate the performance of reading the packets from the memory of the GPU, using different sized word accesses. Figure 3.9 shows the performance achieved when fetching 1, 4, and 16 bytes at a time. The horizontal axis corresponds to the number of packets processed at once. Each packet is processed by one thread, hence the number of packets is equal to the number of threads. The network packets (1500-bytes long) reside on the global device memory. Each byte in the packet requires two memory accesses: one access for fetching the contents of the packet, and one access to the state machine, in order to find the next state to traverse.

As we increase the number of threads, the multithreaded capabilities of the GPU at hiding memory latencies results to an increase in the throughput sustained by the GPU. An interesting observation is that performance

levels-off proportionally to the size of the word accesses. For example, when fetching the input data one byte at a time, we observe that the throughput sustained by the GPU remains constant after processing 4,096 packets at a time. Moreover, using the `char4` and `int4` data types for loading the data from the device memory has a positive impact to overall performance. When reading the data one-byte at a time, unused bytes are transferred for every device memory transaction, since the minimum size per transaction is 32 bytes. The `char4` type uses four bytes per transaction and boosts the performance up to four times. With 16 bytes per transaction using the `int4` type, an additional performance boost of 300% is achieved, while the plateau starts when using 24,576 threads.

It is clear that reducing the number of memory transactions from the device memory, results in a significant increase of the processing throughput. Finally, we observe a performance degradation, when moving from 12288 to 16384 threads, which we speculate that is related to the internal GPU thread scheduler.

Evaluating Memory Hierarchies

Figure 3.10 shows the raw processing throughput obtained for different types of memories. The horizontal axis corresponds to the number of packets processed at once. Each thread process a different 1500-byte packet in isolation, fetching 16-bytes at once, which performs better as we have shown in the previous experiment. By storing the network packets and the state table to different types of memory, we measure how each type affects the processing throughput.

Storing the state table as a 2D texture significantly decreases the overall throughput. We speculate that state table accesses exhibits bad 2D spatial locality, hence the 2D optimized textured cache reduces the performance. In contrast, when accessing the network packets, the 2D textures sustain the best performance. All threads achieve coalesced reads when accessing packet data, in contrast to DFA matching that exhibits irregular memory accesses. This irregularity might lead to cache thrashing, and it results in very poor performance.

Regarding packet accesses, we observe that 1D texture memory improves about 20% the performance, while 2D textures provide a 50% improvement over global device memory. On the other hand, global device memory and 1D texture differ slightly for state table accesses, with

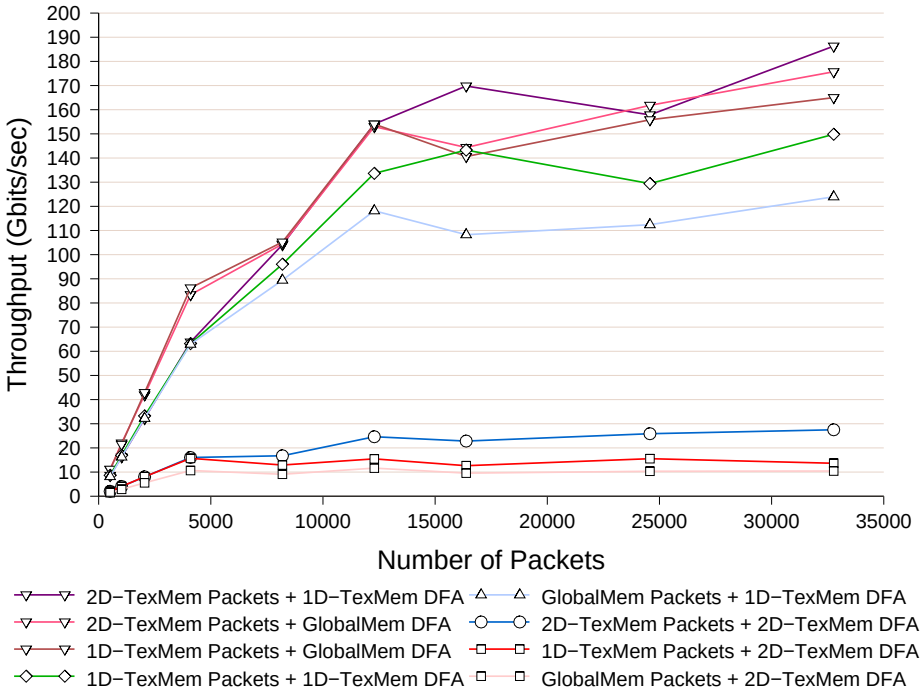


FIGURE 3.10: Memory accesses impact on DFA matching.

global memory providing about 10% better performance. The GPU contains caches for both types of memory—a 12 KB L1-cache per multiprocessor for texture memory, and a 16 KB L1-cache per multiprocessor for global memory—hence the performance is almost the same for state table accesses. When there is a cache hit, the latency for a fetch is only a few cycles, against the hundreds of cycles required to access the global memory.

An interesting observation, is that texture memory seems to fit better for packet accesses, in contrast to state table accesses that performs better on global device memory. Although texture memory does not require to follow any coalescence patterns, it seems that is expose better cache performance.

Packets Layout Transformations

Figure 3.11 shows the raw GPU processing achieved when (i) grouping, and (ii) transposing the network packets in the global GPU memory. We also show how each transformation performs when using the `char4` and

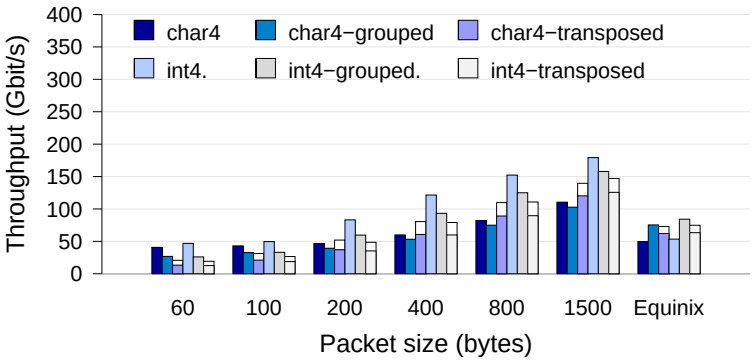


FIGURE 3.11: Impact of packet transformations on DFA matching.

`int4` data types. Specifically, we observe that packet grouping makes sense only when the input traffic contains a mixed type of packet sizes and protocols ("Equinix" case). When synthetically generating fixed packets of the same length, packet grouping has a negative effect, as the overhead to group the packets does not amortized by the resulting accorded SIMT execution—the workload is distributed equally to each GPU thread anyway. In contrast, when using real network traffic ("Equinix"), in which the packet sizes vary significantly, grouping offers about 58% increase in the computational throughput.

The transposition of the memory packets in the global device memory gives better performance only when the `char4` built-in data type is used for reading, and only for relatively large packet sizes (i.e., 400 bytes or more). When using `int4` though, it is better to access the packets without performing the transposition. Even when the transposition cost is excluded (white bar), performance still remains lower. The reason is that the texture memory that we used tolerates memory pattern changes better than global memory for its use of cache.

3.4.4 Regular Expression Matching

In this section we explore the performance of our regular expression matching implementation. First, we measure the overheads in terms of memory consumption and communication costs. We then examine the raw processing throughput achieved on the GPU, taking into account the different memory spaces of modern graphics cards. Final, we examine the overall performance of Snort using our GPU-assisted regular expression matching

implementation.

Memory Requirements

In our first experiment, we measured the memory requirements of our system. Modern graphics cards are equipped with enough and fast memory, ranging from 512 MB DDR up to 6 GB GDDR3 SDRAM. However, the compilation of several regular expression to DFAs may lead to state explosion and consume large amounts of memory.

Figure 3.12(a) shows the cumulative fraction of the DFA states for the regular expressions of the Snort rule set. It appears that only a few expressions are prone to the state-space explosion effect. By setting an upper bound of 5000 states per regular expression, it is feasible to convert more than 97% of the regular expressions to DFAs, consuming less than 200 MB of memory, as shown in Figure 3.12(b).

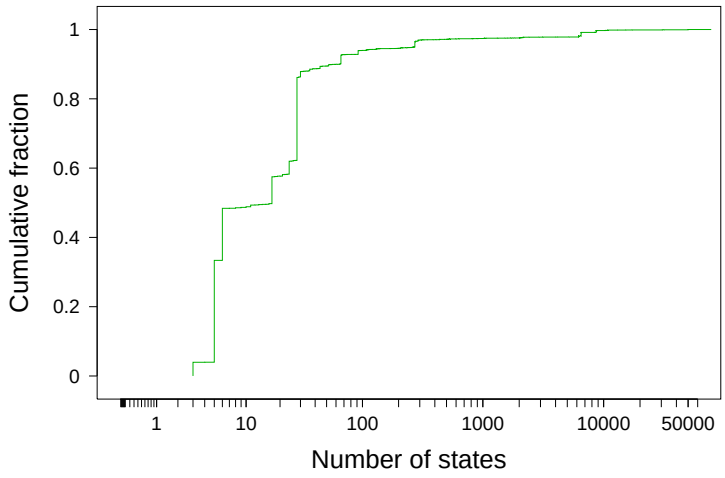
GPU-based Regular Expression Matching in Snort

In our next experiment we evaluated the overall performance of the Snort IDS using our GPU-assisted regular expression matching implementation. We ran Snort using the full-payload network traces described in Section 3.4.2. Figure 3.13 shows the achieved throughput for each network trace, when regular expressions are executed in CPU and GPU, respectively. In both cases, all `content` rules are executed by the CPU. We can see that even when `pcre` matching is disabled, the overall throughput is still limited. This is because `content` rules are executed on the CPU, which limits the overall throughput.

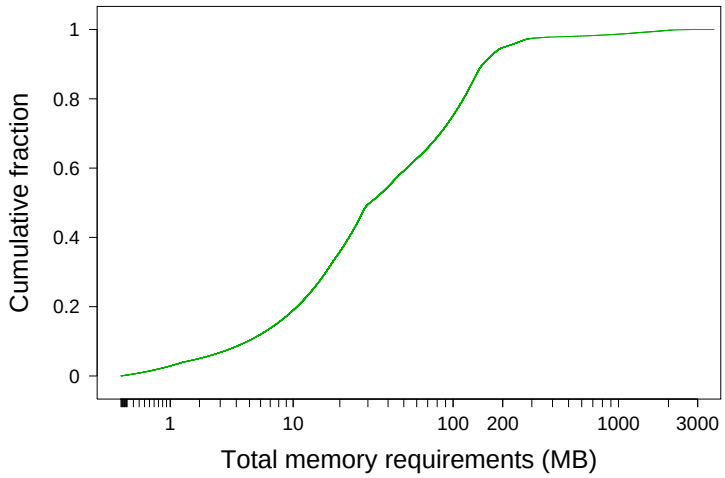
3.4.5 Overall Throughput

In our final experiment we evaluated the overall performance of the Snort IDS using our GPU-assisted string searching and regular expression matching implementations. We ran Snort using the full-payload network traces described in Section 3.4.2. Figure 3.14 shows the achieved throughput when both `content` and `pcre` patterns are matched on the GPU.

As we can see, the overall throughput exceeds 800 Mbit/s, which is an eight times increase over the default Snort implementation. The performance for the *LLI* trace is still limited, primarily due to the extra overhead spent for reassembling the large amount of different flows that are contained in the trace.



(a)



(b)

FIGURE 3.12: States (a) and memory requirements (b) for the 11,775 regular expressions contained in the default Snort 2.6 ruleset when compiled to DFAs. The median number of states is 17, and the corresponding memory requirements is 32.2 MB.

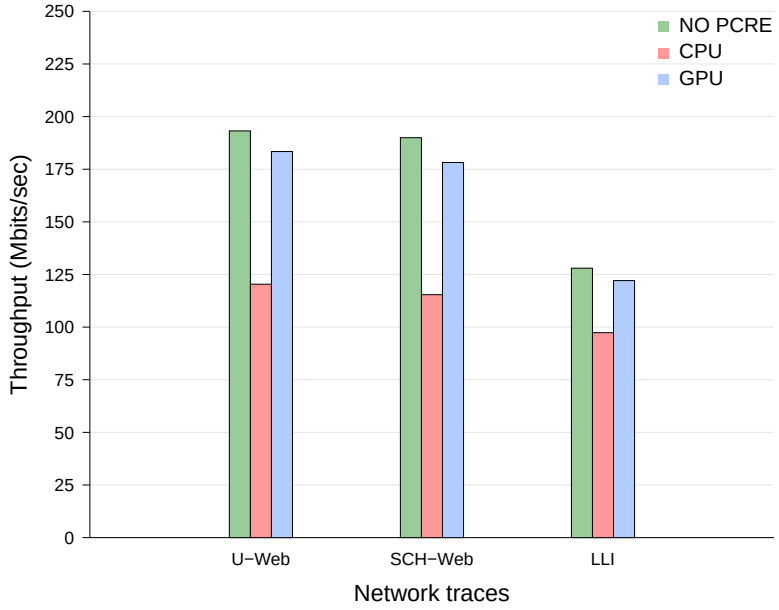


FIGURE 3.13: Sustained processing throughput for Snort when regular expression matching is executed in the CPU and the GPU respectively. The string matching process is performed on the CPU for both approaches.

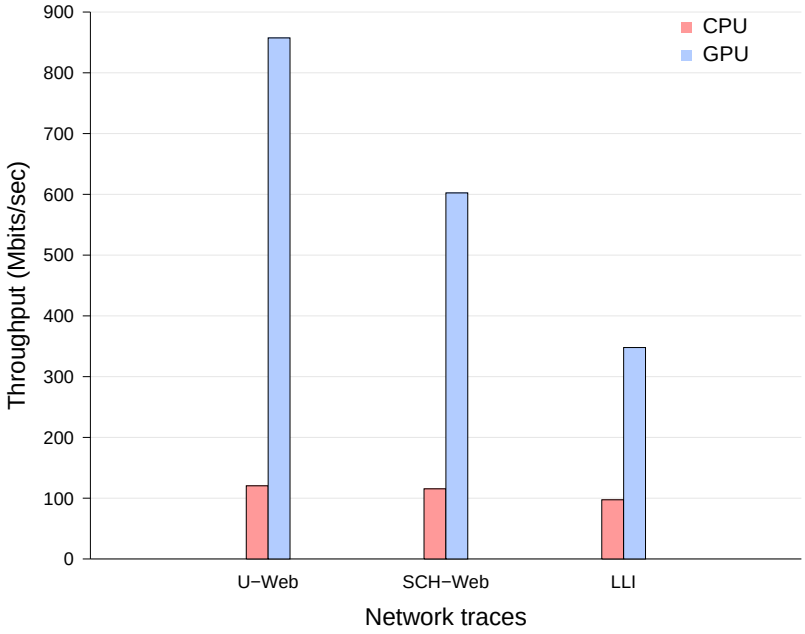


FIGURE 3.14: Sustained processing throughput for Snort using different network traces. Both string searching and regular expression matching are performed on the GPU.

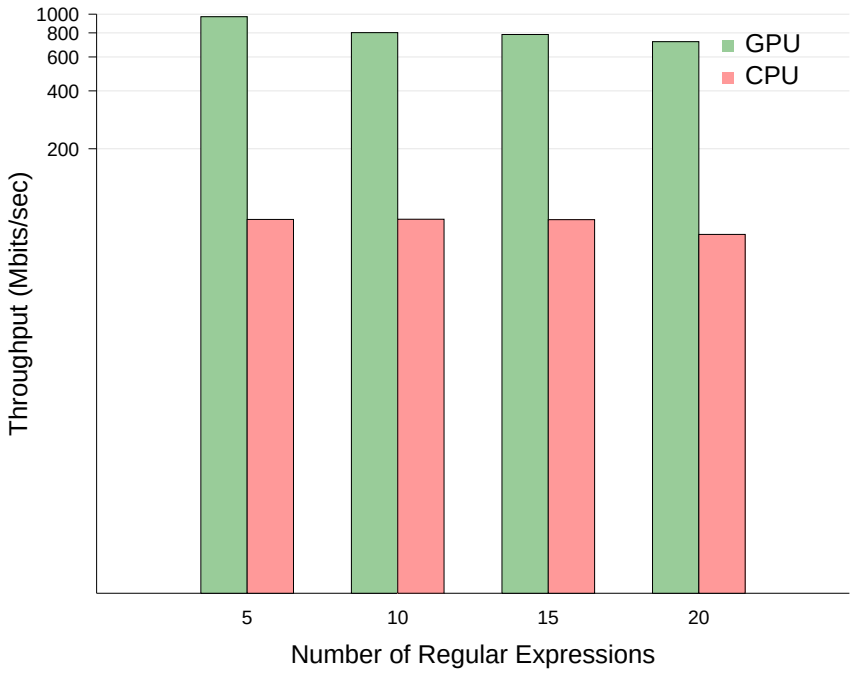


FIGURE 3.15: Sustained throughput for Snort when using only regular expressions.

3.4.6 Worst-case Performance

In this section, we evaluate the performance of Snort for the worst-case scenario in which each captured packet has to be matched against several regular expressions independently. By sending crafted traffic, an attacker may trigger worst-case backtracking behavior that forces a packet to be matched against more than one regular expressions [122].

We synthetically create worst-case conditions, in which each and every packet has to be matched against a number of regular expressions, by removing all `content` and `uricontent` keywords from all Snort rules. Therefore, Snort's pre-filtering pattern matching engine is rendered completely ineffective, forcing all captured packets to be evaluated against each `pcre` pattern individually.

Figure 3.15 shows how the CPU and the GPU implementations scale as the number of regular expressions increases. We vary the number of `pcre` web rules from 5 to 20, while Snort was operating on the *U-Web* trace. In each run, each packet of the network trace is matched against all regular

expressions. Even if the attacker succeeds in causing every packet to be matched against 20 different regular expressions, the overall throughput of Snort remains over 700 Mbit/s when regular expression matching is performed on the GPU. Furthermore, in all cases the sustained throughput of the GPU implementation was 9 to 10 times faster than the throughput on the CPU implementation.

3.5 Discussion

An alternative approach for regular expression matching, not studied in this dissertation, is to combine many regular expressions into a single large one. The combination can be performed by concatenating all individual expressions using the logical union operator [125]. However, the compilation of the resulting single expression may exponentially increase the total number of states of the resulting deterministic automaton [82, 123]. The exponential increase, mainly referred as *state-space explosion* in the literature, occurs primarily due to the inability of the DFA to follow multiple partial matches with a single state of execution [81].

To prevent state-space explosion, the set of regular expressions can be partitioned into multiple groups, which can dramatically reduce the required memory space [147, 82]. However, multiple DFAs require the traversal of input data multiple times, which reduces the overall throughput. Recent approaches attempt to reduce the space requirements of the automaton by reducing the number of transitions [82] or using extra scratch memory per state [123]. The resulting automaton is compacted into a structure that consists of a reasonable number of states that are feasible to store in low-memory systems.

Although most of these approaches have succeed in combining all regular expressions contained in current network intrusion detection systems into a small number of automata, it is not straightforward how current intrusion detection systems (like Snort) can adopt these techniques. This is because most of the regular expressions used in attack signatures have been designed such that each one is scanned in isolation for each packet. For example, many expressions in Snort are of the form `/^.{27}/` or `/.{1024}/`, where `.` is the wild card for *any* character followed by the number of repetitions. Such expressions are used for matching the presence of fixed size segments in packets that seem suspicious. Therefore,

even one regular expression of the form `/.{N}/` will cause the relevant automaton to generate a huge number of matches in the input stream that need to be checked against in isolation.

Moreover, the combination of regular expressions into a single one prohibits the use of specific modifiers for each regular expression. For example, a regular expression in a Snort rule may use internal information, like the matching position of the previous pattern in the same rule. In contrast, our proposed approach has been implemented directly in the current Snort architecture and boost its overall performance in a straightforward way. In our future work we plan to explore how a single-automaton approach could be implemented on the GPU.

Finally, an important issue in network intrusion detection systems is traffic normalization. However, this is not a problem for our proposed architecture since traffic normalization is performed by the Snort preprocessors. For example, the URI preprocessor normalizes all URL instances in web traffic, so that URLs like "`GET /%43md.exe HTTP/1.1`" become `GET /cmd.exe HTTP/1.1`. Furthermore, traffic normalization can be expressed as a regular expression matching process [112], which can also take advantage of GPU regular expression matching.

Chapter 4

A Multi-Parallel Network Intrusion Detection Architecture

In the previous chapter we show an approach that takes advantage of a single GPU to accelerate the performance of a single-threaded packet processing application. In practice, however, the performance of complex packet processing system, such as NIDSs, depends on several other operations, including packet capture and decoding, TCP stream reassembly, and application-level protocol analysis. A scalable architecture need to extract parallelism at each stage of the pipeline that is shown in Figure 2.1, otherwise Amdahl's Law will fundamentally limit the performance that the hardware can provide [105]. This chapter presents a multi-parallel intrusion detection architecture that parallelizes network traffic processing and analysis at three levels, using *multi-queue NICs*, *multiple CPUs*, and *multiple GPUs*. The proposed design avoids excessive locking, optimizes data transfers between the different processing units, and speeds up data processing by mapping different operations to the processing units where they are best suited.

4.1 Design Objectives

We begin by discussing the design principles and practical challenges of mapping the different functional components of a signature-based network intrusion detection system to a multi-parallel system architecture.

4.1.1 Inter-flow Parallelism

Our aim is to design a NIDS architecture that scales with the number of available processing units, enabling it to operate at line-rate without packet loss. The primary role of a NIDS is to passively capture the network packets through the network interface (NIC), process them, and report any suspicious events. Therefore, the main tasks of the NIDS can be summarized as: (i) packet capturing, usually at multi-Gigabit rates, and (ii) packet processing, including TCP stream reassembly, application-level protocol parsing, and pattern matching.

In current *hardware NIDS platforms* [6, 126], packet processing operates at line-rates, handling a single input port; therefore, the platform must inspect input traffic at several Gigabit per second. Existing *software-based NIDS*, in contrast, typically follow a multi-core approach and split the traffic at the flow-level to N slices, where N is the number of processing nodes available to the system [118, 135]. Flow-based partitioning achieves an almost even processing load at all processing nodes, without requiring any intra-node communication for processing operations that are limited in scope to a single flow. Traffic is distributed using either an external traffic splitter—which is quite a costly solution—or a software-based load-balancing scheme, where a simple hash function is applied on each captured packet, based on which it is assigned to the appropriate node for processing.

Unfortunately, having many different cores receiving traffic from the same network interface or a shared packet queue, increases contention to the shared resource, which incurs additional delay in packet capturing [69, 70]. This observation leads us to our first design principle: *traffic has to be separated at the network flow level using existing, commodity solutions, without incurring any serialization on the processing path*. In Section 4.2, we show how our system takes advantage of recent load-balancing technologies such as Receive-Side Scaling (RSS) [21], which allows different cores to receive portions of the monitored traffic directly. This inherently leads us to a multi-core architecture, in which each core runs a separate instance of the inspection engine, processing only a subset of the network flows.

4.1.2 Intra-flow Parallelism

Distributing the monitored traffic to different CPU cores offers significant performance benefits. Recent studies [70, 118] have shown a close-to-linear speedup in the number of cores. However, the CPU is still saturated by the large number of diverse and computationally heavy operations it needs to perform: network flow tracking, TCP stream reassembly, protocol parsing, string searching, regular expression matching, and so on. The problem then is how to further parallelize content inspection on each core, enabling a further increase in the overall traffic processing throughput, *without* incurring any packet loss.

This leads us to our second design principle: *per-flow traffic processing should be parallelized beyond simple per-flow load balancing across different CPU cores*. To enable such "intra-flow" parallelism, network packets from the same flow have to be processed in parallel, while also maintaining flow-state dependencies. In Section 4.2.2, we discuss how our system can take advantage of multiple graphics processors to inspect high-volume traffic concurrently with the CPU cores. Intra-flow parallelism is achieved by buffering incoming packets and transferring them to the graphics card in large batches. Although this buffering scheme adds some latency to the processing path, it pays off in terms of the processing throughput that can be sustained.

By parallelizing both packet pre-processing and content inspection across multiple CPUs and GPUs, the proposed multi-parallel NIDS architecture can operate at line rate in multi-Gigabit networks using solely commodity components. Our parallelization scheme also leads to an architecture that is incrementally extensible in terms of hardware resources. We demonstrate that the overall processing throughput of the system can be increased simply by adding more processing elements.

4.1.3 Resulting Trade-off

A potential issue of our design is the data transfer operations that must take place between the memory address spaces of each device. Specifically, network packets are transferred from the NIC to the main memory of the host, and from there to the device memory of the GPU. However, the extra data transfers between the CPU and the GPU over the PCIe bus can be worth the computational gain offered by the GPU. To further mitigate

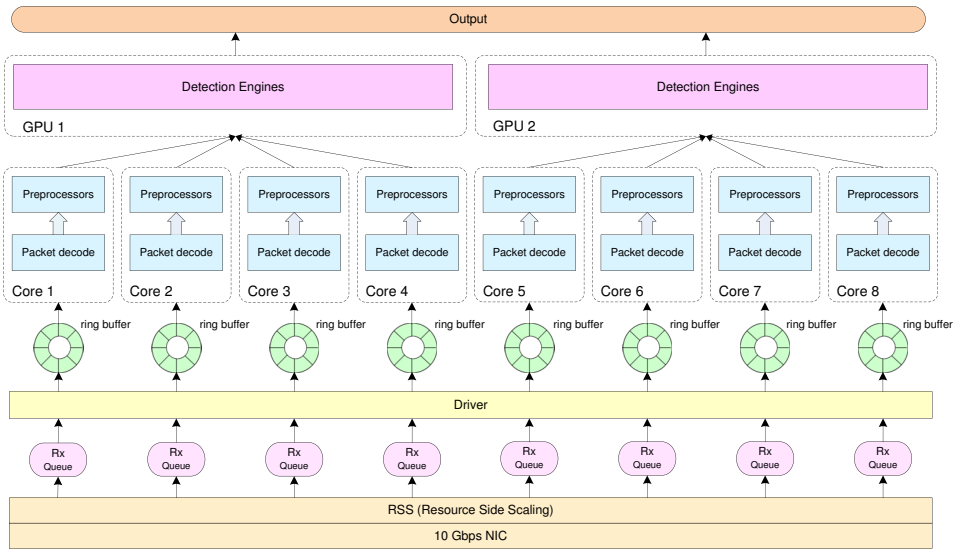


FIGURE 4.1: MIDeA architecture.

this data transfer overhead, we have implemented a pipelining scheme that allows CPU and GPU execution to overlap, and consequently hides the added latencies. Although the raw computational power of the GPU offers enough performance benefits even when considering all data transferring overheads, the pipelining scheme that we introduce, depicted in Figure 4.3, offers an additional level of parallelism to the overall execution path. In Section 4.3, we discuss in detail how these optimizations have been implemented in our system.

4.2 Architecture

In this section, we describe the overall design of our multi-parallel network intrusion detection architecture. The key factors for achieving good performance are: (i) load balancing between processing units, and (ii) linear performance scalability with the addition of more processing units. Additionally, for high-performance packet capturing we consider the use of only inexpensive commodity NICs.

As shown in Figure 4.1, the NIDS application is mapped to the different processing units using both *task* and *data* parallelism across the incoming network flows. In particular, the network interface distributes the captured packets to the CPU-cores, ensuring flow-pinning and equal workload across the cores. Each CPU-core reassembles and normalizes

the captured traffic before offloading it to the GPU for pattern matching. Any matching results are logged by the corresponding CPU-core using the specified logging mechanism, such as a file or database.

This design has a number of benefits: First, it does not require any synchronization or lock mechanisms since different cores process different data in isolation. Second, having several smaller data structures (such as the TCP reassembly tables) instead of sharing a few large ones, not only reduces the number of table look-ups required to find a matching element, but also reduces the size of the working set in each cache, increasing overall cache efficiency.

4.2.1 Packet Capture in 10-Gigabit Ethernet

Our system uses 10GbE NICs, which are currently the state-of-the-art general-purpose network interfaces. Capturing packets at these rates is non-trivial and requires the coordinated effort of the network controller and the multi-core CPUs.

Multiqueue NICs

In order to avoid contention when multiple cores access the same 10GbE port, modern network cards can partition incoming traffic into several Rx-queues [94]. This allows each CPU core to access its own hardware queue independently, while the NIC controller is responsible for classifying incoming network packets and distributing them to the appropriate queue. The Rx-queues are not shared between the CPU cores, eliminating the need of synchronization. Each Rx-queue is dedicated to a specific user-level process that is mapped to a different core, as shown in Figure 4.1. Each user-level process fetches packets from a single queue and forwards them to the next processing module. The controller can set up a number of Rx-queues equal to the number of available CPU cores (the Intel 82599EB Ethernet controller [10] that we used in our implementation supports up to 128 Rx-queues).

To avoid costly packet copies and context switches between user and kernel space, we use the `PF_RING` network socket [48]. The most efficient way to integrate a `PF_RING` socket with a multi-queue NIC is to dedicate a separate ring buffer for each available Rx-queue [56]. Network packets of each Rx-queue are stored into a separate ring buffer and are pulled by the user-level process through DMA, without going through the kernel's

network stack.

We also take into consideration the interrupt handling of each queue. In Linux, interrupts are handled automatically by the kernel through the `irqbalance` daemon. This daemon is responsible for evenly distributing interrupts from each Rx-queue to CPU cores, in a round-robin fashion. Unfortunately, this is not the optimal solution for multi-core systems, because distributing the handling of interrupts from a single Rx-queue to multiple cores results in cache invalidation and performance degradation [85]. This means that `irqbalance` does not guarantee that the interrupt of the next packet of the same flow will be handled by the same core. Therefore, we bind the interrupt handling of each Rx-queue to a specific CPU core by setting the corresponding `/proc/irq /X/smp_affinity` entry (where `X` is the IRQ number of each Rx-queue, which can be obtained from `/proc/interrupts`).

Load Balancing

A major implication when partitioning the incoming traffic to multiple instances is to guarantee that all packets of a specific flow will be processed by the same user-level process. It is also important to distribute the load equally to the different processing cores. Modern NICs [21] support *hash-based* (or *flow-based*), and *address-based* classification schemes. In hash-based schemes, such as Receive-Side Scaling (RSS), a hash function is applied to the protocol headers of the incoming packets in order to assign them to one of the Rx-queues. In address-based schemes, such as Virtual Machine Device Queues (VMDQ), each Rx-queue is assigned a different Ethernet address, to provide an abstraction of a dedicated interface to guest virtual machines.

For our purposes, we choose the hash-based method. The hash function, computed on the typical 5-tuple `<ip_src, ip_dst, port_src, port_dst, protocol>` achieves good distribution among the different queues. The RSS specification [94] allows the explicit parameterization of the tuple fields that will be used to compute the hash. Unfortunately, current RSS-enabled network interfaces (such as the Intel 82599EB that we used) use a fixed hashing type, which only ensures that the packets of the uni-directional streams of a connection will result to the same hash value. This means that the client-to-server stream of the flow will may end up to one Rx-queue, and the server-to-client stream to a different one.

In order to insure that packets of both directions end up into the same ring buffer, a symmetric hashing is further applied on the 5-tuple fields of each packet header. Eventually, all packets of the same flow will always be placed in the same ring buffer, and will be processed by the same user-level process. In addition, we bind the process that reads from each ring buffer to the same core using the CPU affinity of the Linux scheduler (see `sched_setaffinity(2)`), in order to increase cache locality. We assume that the monitored traffic consists of many different concurrent flows (at least as many as the available CPU cores), hence all processes are fed with data. This is not an issue even in small networks, since even a single host usually has tens of concurrent active connections.

4.2.2 Processing Engine

Incoming traffic is forwarded to the processing engines for analysis. Each processing engine is implemented as a single process and is mapped to a certain CPU core to avoid costs due to process scheduling. The basic functionality of each processing engine is to retrieve the network packets from its assigned hardware queue, decode them and apply higher-level protocol analysis, and finally transfer them to the GPU for content inspection.

Preprocessing

Any configured preprocessors may optionally be invoked at this stage, as we have already described in Section 3.1.2. These include flow reassembly and normalization. Once preprocessing is complete, the data is forwarded to the detection engine, which performs signature matching on the incoming traffic. As we have already described in Section 3.1.3, the detection signatures are organized in *port groups*, based on the source and destination port numbers of each rule. Additionally, a separate detection engine instance is used to search for the string patterns of a particular rule group. To achieve *intra-flow* parallelization, MIDeA takes advantage of the data-parallel capabilities of modern graphics processors.

Incoming traffic is transferred to the memory space of the GPU in batches. As we discuss in Section 4.4.2, small transfers results to significant PCIe throughput degradation, hence we batch lots of data together to reduce the PCIe transaction overhead. Also, instead of allocating a different buffer for each port group, we simply mark each packet so that it will be processed by the appropriate detection engine in the searching phase.

Consequently, only one buffer is needed per process, instead of one for each port group. This results to significantly lower memory consumption and reduces response latency for port groups with low traffic. Whenever the buffer gets full, all packets are transferred to the GPU in one operation.

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 area results to higher data transfer throughput between the host and the GPU device, because the GPU driver knows the location of the data in RAM and does not have to locate it—neither swap it from disk, nor copy it to a non-pageable buffer—before transferring it to the GPU. Data transfers between page-locked memory and the GPU are performed through DMA, without occupying the CPU.

Parallel Multi-Pattern Engine

A major design criterion for matching large data streams against many different patterns, is the choice of an efficient pattern matching algorithm. The majority of network intrusion detection systems use a flavor of the Aho-Corasick algorithm [27] for string searching, which uses a transition function to match input data. The transition function gives the next state $T[state, ch]$ for a given *state* and a character *ch*. A pattern is matched when starting from the *start state* and moving from state to state, the algorithm reaches a *final state*. The memory and performance requirements of Aho-Corasick depend on the way the transition function is represented. In the full representation, each transition is represented with 256 elements, one for each 8-bit character. Each element contains the next state to move to, hence given an input character, the next state can be found in $O(1)$ steps. This gives a linear complexity over the input data, independently on the number of patterns, which is very efficient in terms of performance.

In the full state representation, hereinafter *AC-Full*, every possible input byte leads to *at most* one new state, which ensures high performance. Unfortunately, a full state representation requires large amounts of memory, even for small signature sets. When compiling the whole rule set of Snort, the size of the compiled state table can reach up to several hundreds Megabytes of memory. On most modern graphics cards, available memory is not a constraint any more, since they are usually equipped with ample

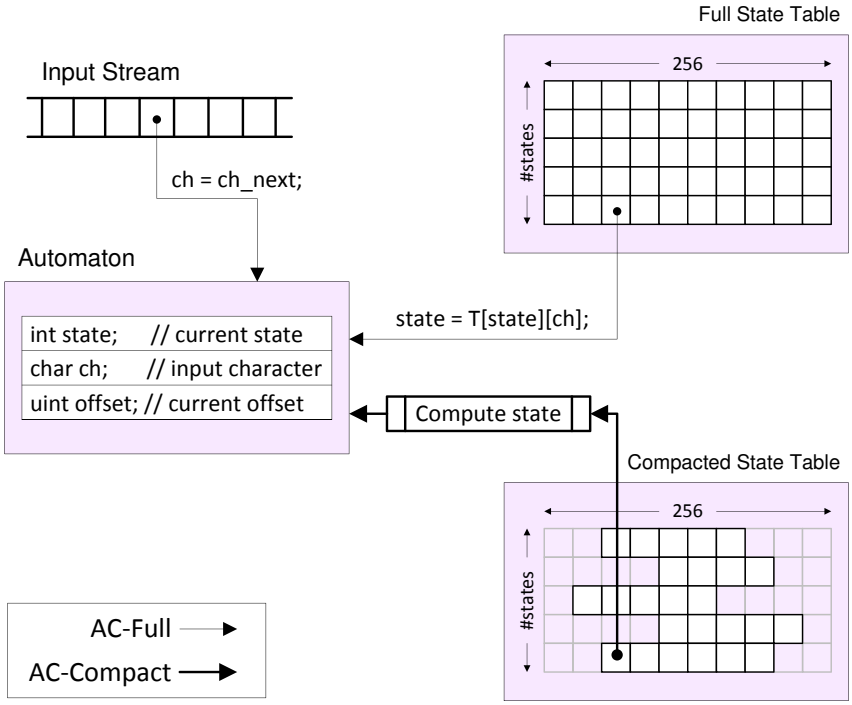


FIGURE 4.2: State tables of AC-Full and AC-Compact.

amounts of memory—a GeForce GTX480 comes with 1.5 GB of memory at a reasonable price. Unfortunately, in the CUDA runtime system [102], on which MIDeA is based, each CPU thread is executed in its own CUDA context. In other words, a different memory space has to be allocated in the GPU for each process, since they cannot share memory on the GPU device. As we discuss in Section 4.4.2, when using the AC-Full algorithm, only the detection engines of a single Snort instance can fit in the memory space of the GPU. That means that only one Snort instance can fully utilize the GPU at a time.

To overcome the memory sharing limitation of CUDA and maintain scalability, it is important to keep the memory requirements low. Instead of creating a full state table, we use a *compacted state table* structure for representing the compiled patterns [101]. The compacted state table is represented in a banded-row format, where only the elements from the first non-zero value to the last non-zero value of the table are actually stored. The number of the stored elements is known as the *bandwidth* of the sparse table. In our new implementation, *AC-Compact*, the next state is not

directly accessible while matching input bytes, but it has to be computed, as shown in Figure 4.2. This computation adds a small overhead at the searching phase, which is amortized by the significantly lower memory consumption.

Moreover, it is common that many patterns are case-insensitive, or share the same final state in the transition table. Instead of inserting every different combination of lowercase and capital letters for the pattern, we simply insert only one combination (i.e., all characters are converted to lowercase), and mark that pattern in the pattern list as case-insensitive. In case the pattern is matched in a packet, an extra case-insensitive search should be made at the index where the pattern was found. If two patterns share the same final list (i.e., the match list contains more than one pointers to patterns), the patterns contained in the list have to be verified for finding the actual match.

Each packet is processed by a different GPU thread. Packets are stored into an array, which dimensions are equal to the number of the packets that are processed at once and the Maximum Transmission Unit (MTU). Packets that exceed MTU (which is 1500 bytes in Ethernet) are splitted down into several smaller ones, and are copied in consecutive rows in the array. To detect attacks that span multiple rows, each thread continues its search to the following portions of the packet (if any) iteratively, until a *final* or *fail state* is reached.

Multi-GPU Support

A key feature of MIDeA is its support for pattern matching using several GPUs at a data-parallel level. Modern motherboards, such as the one we used in our evaluation, support multiple GPUs on the PCI Express bus. MIDeA utilizes the different GPUs by dividing the incoming flows equally and performing the signature matching in parallel across all devices.

By default, MIDeA utilizes as many GPUs as it can find in the system; however, this can be controlled by defining the number of GPUs it should try to use in the configuration file. In the CUDA runtime system, on which MIDeA is based, each CPU process is bound to one device. To make multi-GPU computation possible, several host processes must then be created, with at least one process per device. A static GPU assignment is used for each process. Each process receives a uniform amount of flows, due to the load balancing scheme described in Section 4.2.1, and thus flows are

equally distributed to the different GPUs.

4.3 Performance Optimizations

Having described our architecture, we now go into a couple of optimizations that improve CPU and GPU execution through pipelining.

Our core idea for hiding the pattern matching computation time on the GPU is double buffering. Our architecture improves the achieved parallelism by pipelining the execution of CPU cores and the GPUs. For each process, when the first buffer becomes full, it is copied to a texture bounded array that can be later read by the GPU through the kernel invocation. While the GPU is performing pattern matching on the flows of the first buffer, the CPU processes newly arrived packets, as shown in Figure 3.6.

Moreover, on recent CUDA-enabled devices, it is possible to overlap kernel execution on the device with data transfers between the host and the device, even for different processes. The dedicated DMA engine of NVIDIA GPUs[1] allows the concurrent execution of a CUDA kernel along with data transfers over the PCIe bus. For example, while one process transfers the data to the GPU, another process can execute the pattern matching operations. This allows better GPU utilization, as depicted in Figure 4.3. As we discuss in Section 4.4.2, the performance improvement due to overlapping execution in the GPU is up to 330%.

4.4 Experimental Evaluation

We evaluate the performance of our system under a variety of workloads. We first describe the experimental testbed (Section 4.4.1), and then analyze the performance of MIDeA under different scenarios using micro-benchmarks (Section 4.4.2), as well as high-level end-to-end performance measurements (Section 4.4.3).

4.4.1 Experimental Setup

Hardware Setup. The overall architecture of our test system is shown in Figure 4.4. Our base system has two processor sockets, each with one Intel Xeon E5520 Quad-core CPU at 2.27 GHz and 8192 KB of L3-cache. Each socket has an integrated memory controller, connected to memory via a memory bus; this offers parallelism in memory accesses and, therefore,

[1] For devices with Compute Capability 1.1 or greater.

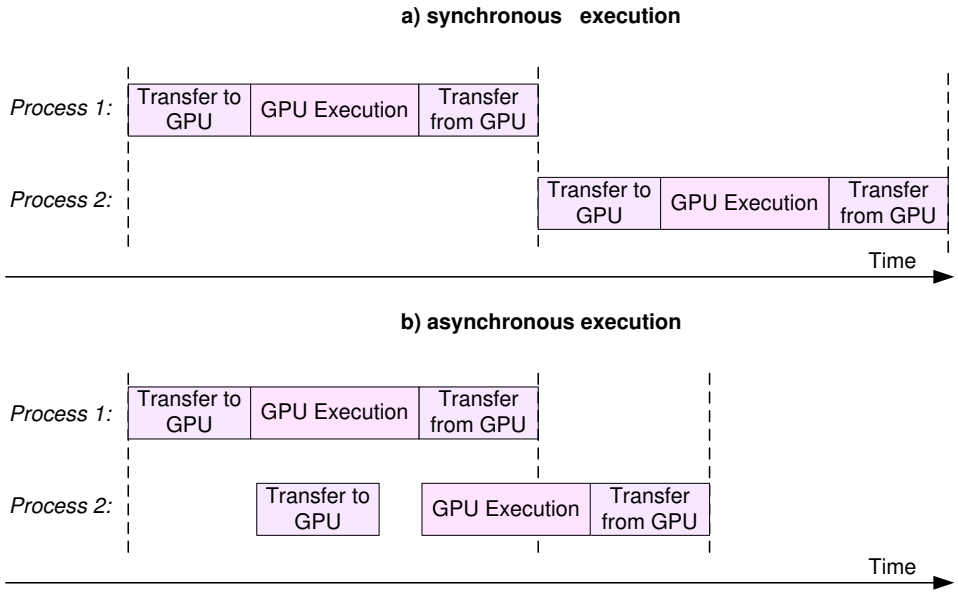


FIGURE 4.3: Data transfers and GPU execution of different processes can overlap.

to higher aggregate and per-CPU bandwidth, as previous studies have shown [51]. The sockets are connected to each other and to the I/O hub via dedicated high-speed point-to-point links. The I/O hub is connected to the GPUs and the NIC via a set of PCIe buses: two PCIe 2.0 x16 slots, which we populated with two GeForce GTX480 graphics cards, and one PCIe 2.0 x8 slot holding one Intel 82599EB 10GbE NIC. To cover the needs for PCIe lanes, we acquired a motherboard with a dual I/O hub and a total of 72 lanes. Each NVIDIA GeForce GTX 480 is equipped with 480 cores, organized in 15 multiprocessors, and 1.5 GB of GDDR5 memory.

Software. Our prototype runs on Linux 2.6.32 with the `ioatdma` and `dca` modules loaded. The `ioatdma` driver is required for supporting QuickPath Interconnect architecture of recent Intel processors. DCA (Direct Cache Access) is a NIC technology that directly places incoming packets into the cache of the CPU core for immediate access by the application. In all of our experiments we used the default rule set of Snort 2.8.6, which consists of 8,192 rules, comprising about 193,000 substrings for string searching. All default preprocessors, including `frag3`, `stream5`, `rpc_decode`, `ftp_telnet`, `smtp`, `dns`, and `http_inspect`, were en-

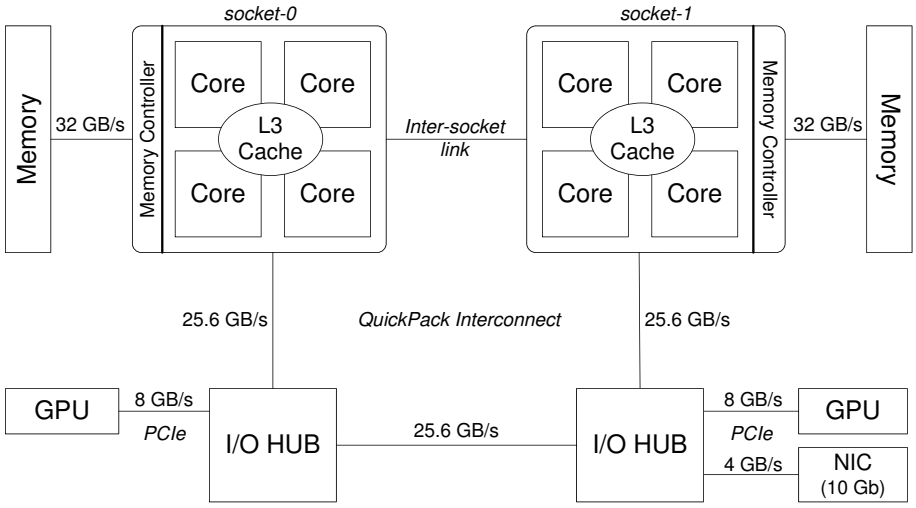


FIGURE 4.4: Hardware setup.

abled.

Traffic Generation. We used two servers for traffic generation to overcome the poor performance of the Linux kernel's network stack when sending small packets. The traffic generation servers and MIDeA are connected through a 10GbE switch. The test traffic, consisting of both generated synthetic traffic as well as real traffic traces, is sent using `tcpreplay` [23].

4.4.2 Micro-Benchmarks

We begin our evaluation by measuring the computational throughput of MIDeA using a varying number of CPU processes and GPU devices. Each process runs on a different CPU core, therefore we can utilize all cores by creating eight processes.

For the input data stream we used synthetic network traces of varying length with random payload. The data stream was carefully created to exercise most code branches, as well as different parameters of our implementation. To simulate the multi-queue capabilities of the NIC, we loaded the network packets of the trace file into separate queues (one for each core) using a simplified version of the Toeplitz hash function, which is used for RSS in modern NICs [94]. This is the "ideal NIC" case, where no overhead is added due to the transferring of the packets from the network interface to the host's main memory. All network packets are stored in memory, thus no blocks were transferred from disk when reading pack-

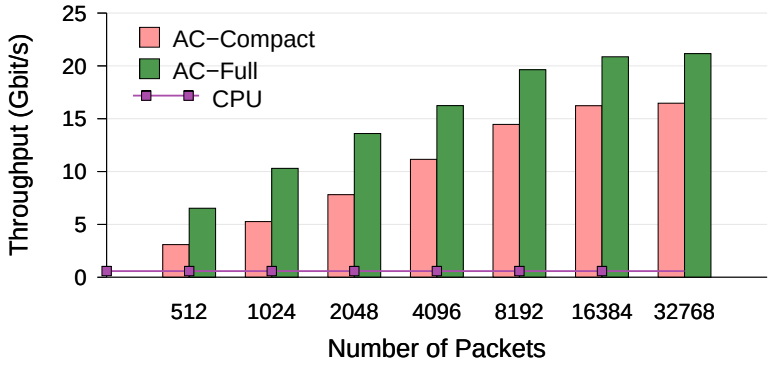


FIGURE 4.5: GPU throughput for AC-Full and AC-Compact.

ets. We have verified the absence of I/O latencies using the `iostat(1)` tool.

GPU Performance

Data Transfer. Flows are transferred to each GPU device over the shared PCIe x16 bus. PCIe uses point-to-point serial links, allowing more than one pair of devices to communicate with each other at the same time.

Table 2.1 (page 18) shows the transfer rate of one process for moving data to a single GPU device, and vice versa, for different buffer sizes. We observe that with a large buffer, the rate of data transfer to the device is over 45 Gbit/s, while the transfer rate from the device to the host decreases to about 25 Gbit/s. This asymmetry in the data transfer throughput is probably related to the chosen hardware setup (i.e., the interconnection between the motherboard and the graphics cards), and has been also observed by other researchers [60]. We speculate that future motherboards will alleviate this asymmetry.

Computational Throughput. Having examined the data transfer costs, we now measure the GPU performance of the AC-Compact and AC-Full algorithms, described in Section 4.2.2.

Figure 4.5 shows the sustained throughput for pattern matching on a single GTX480. We fix the packet length to 1500 bytes and vary the number of packets that are processed at once from 512 to 32,768. Our AC-Full and AC-Compact implementations achieve a peak performance of 21.1 Gbit/s and 16.4 Gbit/s, respectively, including the data transferring costs to and from the device. The CPU achieves a performance of 0.6 Gbit/s for the AC-

#Rules	#Patterns	#States	AC-Full	AC-Compact
8,192	193,167	1,703,023	890.46 MB	24.18 MB

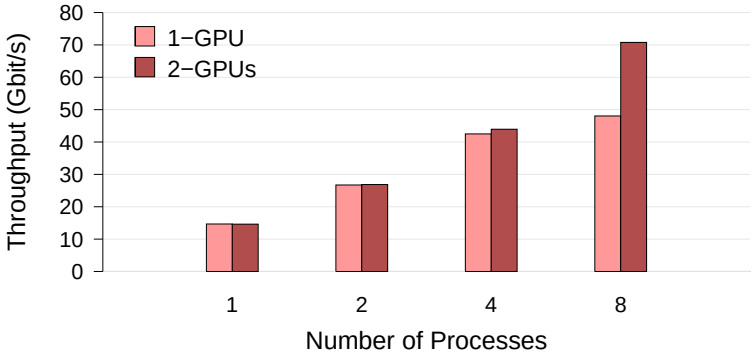
TABLE 4.1: Memory requirements of AC-Full and AC-Compact for the default Snort rule set.

Full implementation, and thus a single GPU instance corresponds to 36.2 and 28.1 CPU cores for the AC-Full and AC-Compact implementations, respectively.

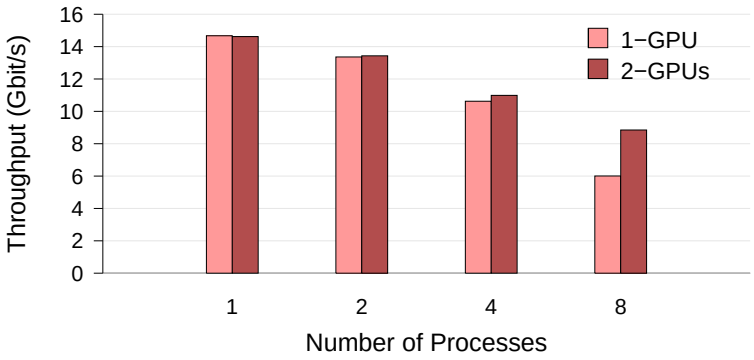
As expected, AC-Full outperforms AC-Compact in all cases. The added overhead of the extra computation that AC-Compact performs in every transition decreases its performance about 30%. The main advantage of AC-Compact is that it has significantly lower memory consumption than AC-Full. Table 4.1 shows the corresponding memory requirements for storing the detection engines of a single Snort instance. AC-Compact utilizes up to 36 times less memory, which makes it a better fit for a multi-CPU environment, due to CUDA's limitation of allocating a separate memory context for each host thread. Using AC-Compact, a single GTX480 card can store the detection engines of about 50 Snort instances ($50 \times 24.18MB \approx 1.2GB$). The remaining memory is used for storing the contents of network packets. If AC-Full is used, only one instance can fit in device memory. In all subsequent experiments we use the AC-Compact algorithm.

Utilization. We investigate the performance of the AC-Compact algorithm further, by varying the number of CPU processes that feed the GPU devices with data.

Figure 4.6(a) shows the aggregate data processing throughput of the GPU(s) for an increasing number of CPU processes. Figure 4.6(b) plots the same data normalized by the number of processes. It is clear that multiple processes offer an improvement even when utilizing only one GPU device. Currently, GPUs support multitasking through the use of non-preemptive execution: each program receives a time slice of the GPU resources and cannot be suspended. When many processes use the same GPU device, data transfers and GPU execution may overlap, offering better GPU utilization. GPU executions have short run times, ranging from 100–300ns per packet, and hence, the GPU device can be effectively timesliced among the CPU processes.



(a) Aggregate



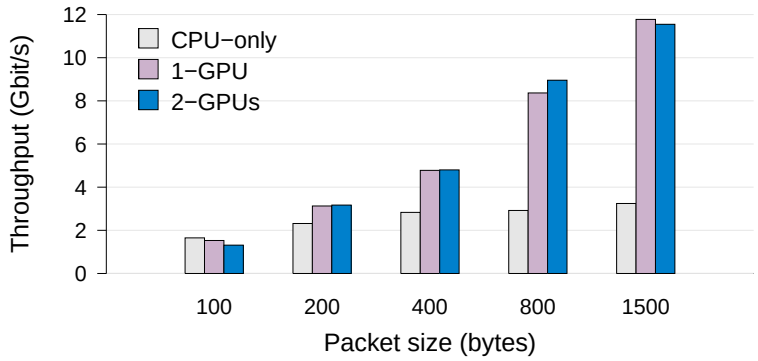
(b) Per-process

FIGURE 4.6: GPU throughput with an increasing number of CPU processes up to the number of cores. Each process is mapped to a different CPU-core.

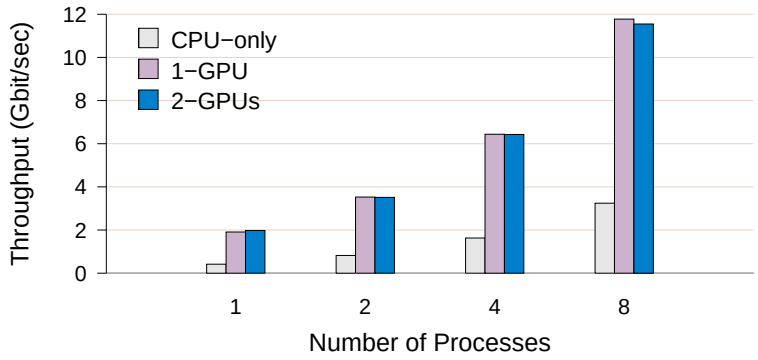
We observe that with two spawned processes, the overhead of the AC-Compact implementation increases, since the 25 Gbit/s throughput achieved is greater than the 21.1 Gbit/s achieved by the AC-Full algorithm, as shown in Figure 4.5. Increasing to eight processes, a single GPU reaches a maximum of 48 Gbit/s throughput. The PCIe bus saturation, which was shown in Table 2.1, is the main reason for this upper bound. However, since the PCIe bus is a point-to-point link, adding one more GPU device to the system increases the aggregate GPU throughput to over 70 Gbit/s.

Overall Performance

Throughput. In our next experiment, we measured the overall processing throughput achieved by our multi-parallel implementation. Figure 4.7(a) shows the sustained throughput for different packet sizes. We observe



(a) Different packet sizes.



(b) Different numbers of CPU processes.

FIGURE 4.7: Overall sustained throughput for an increasing number of (a) packet sizes and (b) CPU processes (each process is mapped to a different CPU-core).

that for very small packet sizes, the GPU-assisted design exhibits a slightly worse performance compared to the multi-core approach alone. The main reason for this is that the buffering overheads for very small packets are greater than the corresponding pattern matching costs, as shown in more detail in the following experiment. Therefore, it is better in terms of performance to match very small packets on the CPU, rather than transferring them to the GPU.

As a consequence, we adopted a simple opportunistic offloading scheme, in which pattern matching of very small packets is performed on the CPU instead of the GPU. Thus, only packets that exceed a minimum size threshold are copied to the buffer that is transferred to the GPU for pattern matching. The packets contain already the TCP reassembled stream of a

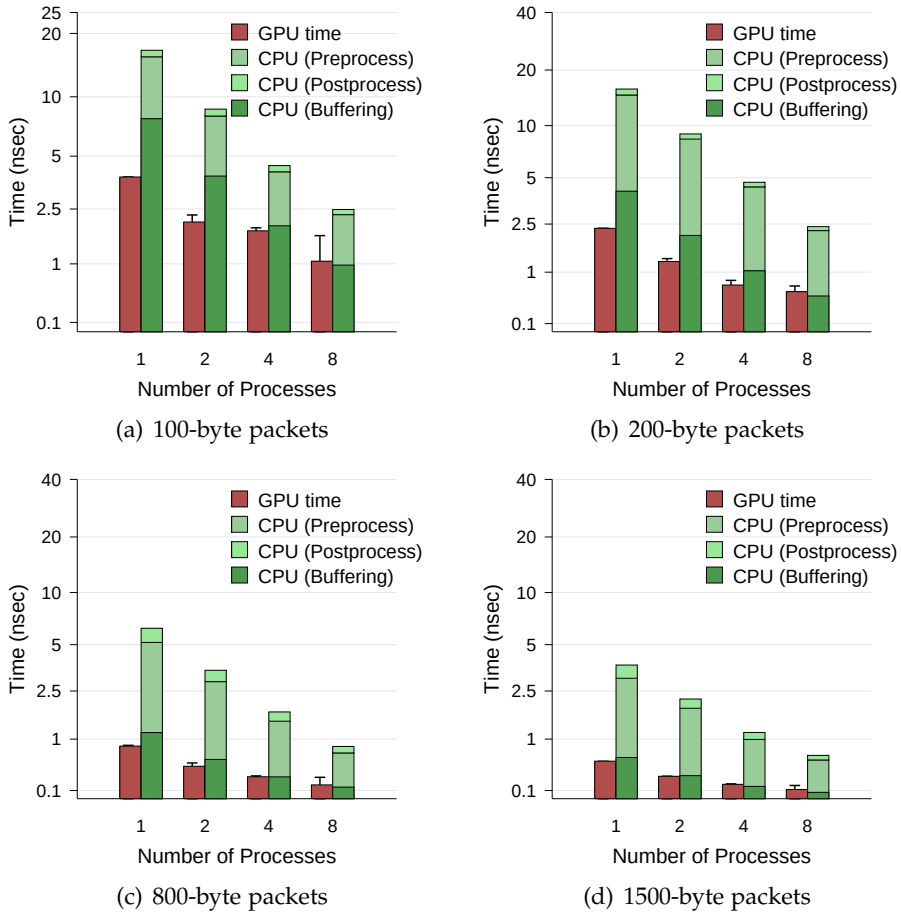
given direction, hence no state needs to be shared between the CPU and the GPU. The minimum threshold can be inferred off line, using a simple profiling measurement, or automatically at runtime. For simplicity we currently use the former method, although we plan to implement an automated solution in the future.

Figure 4.7(b) shows the sustained throughput for a different number of CPU processes, using 1500-byte packets. We observe that as the number of processes increases, the sustained throughput also increases linearly. When pattern matching is offloaded on the GPU, the throughput of the legacy multi-core implementation is increased 3.5–4.5 times, depending on the number of processes. The maximum throughput achieved by our base system reaches about 11.7 Gbit/s when utilizing all resources—eight CPU cores and two GPUs.

Finally, we observe that switching from one to two GPUs does not offer significant improvements to the overall performance. This can be explained by the fact that GPU communication and computation costs are completely hidden by the overlapped CPU computation, as discussed in the following experiment.

Timing breakdown. We proceed and examine in greater detail the overall performance achieved by profiling each device separately. In Figures 4.8(a)–4.8(d) we plot the individual execution times for various packet lengths. We show the times of each device with different bars, since execution is performed in parallel. CPU and GPU execution is pipelined, hence the CPU can continue unaffected while GPU execution is in progress. Each bar represents the execution time of the two GPU devices, while the thin line on each bar represents the corresponding time when utilizing one GPU. We observe that even when a single GPU is used, the cost for the data transfers and the pattern matching on the GPU is completely hidden by the overlapped CPU workload, for all packet sizes.

The extra cost for packet buffering before transferring them to the GPU depends highly on the packet size. Small packets incur higher cost per-byte, due to the start-up overhead of the `memcpy(3)` function. 100-byte packets or smaller induce a prohibitively large overhead, in comparison with the pattern matching cost. We tried to optimize the copies using a byte-by-byte procedure instead of calling the `memcpy(3)`, however the overhead was still higher. Thereupon, we avoid the small-packets penalty



(a) 100-byte packets

(b) 200-byte packets

(c) 800-byte packets

(d) 1500-byte packets

FIGURE 4.8: Breakdown of per-byte processing overhead for different packet sizes.

by opportunistically offloading pattern matching computation on the GPU depending on the packet length.

Finally, we notice that GPU execution times for small packets also increase. The main reason for this is that the dimensions of the buffer that is used for transferring the packets to the GPU are fixed, hence it is populated sparsely for small packets.

4.4.3 Overall Traffic Processing Throughput

In this section, we measure the end-to-end performance of our prototype implementation under realistic conditions.

Synthetic Traffic

Figure 4.9(a) shows the packet loss ratio for different packet sizes, when replaying traffic at varying rates. We plot values up to the maximum achieved replay rate, hence the smaller the packet size, the lower the replay rate reached. For example, for 200-byte packets, we managed to replay traffic at maximum rate of 1.86 Gbit/s, while for 1500-byte packets we achieved a rate of 7.67 Gbit/s.

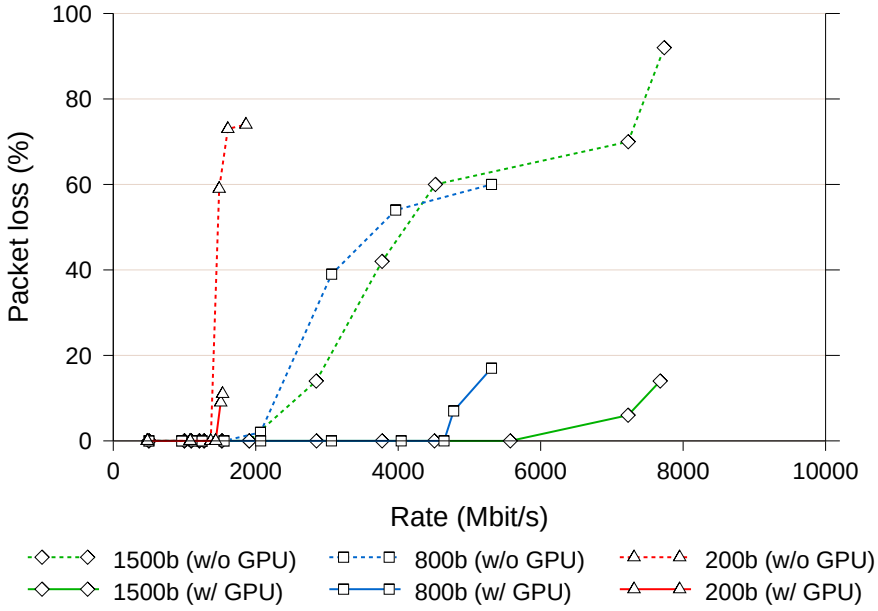
Given these traffic replay rates, our prototype system begins to drop packets at 7.22 Gbit/s for 1500-byte packets, which is a 253% improvement over the traditional multi-core implementation. When processing smaller packets, the performance falls to 1.5 Gbit/s, which is slightly higher than the traditional multi-core implementation, although the drop rate is about 6.6 times lower.

Comparing the achieved throughput with the "ideal NIC" case in Figure 4.7(a), we observe that the NIC adds a variable overhead that depends on the size of the captured packets. It is clear that small packets add more latency to the capturing process than larger ones. For the traditional multi-core approach, we observe an extra overhead of 55% for 200-byte packets, that falls to 18% for 800-byte packets, and 13% for 1500-byte packets. Similarly, the extra overhead for the GPU-accelerated implementation is 110% for 200-byte packets, about 87% for 800-byte packets, and 52% for 1500-byte packets. We observe that the NIC overhead is larger in the GPU-accelerated implementation, and we speculate that this is an issue related to congestion in the PCIe controller.

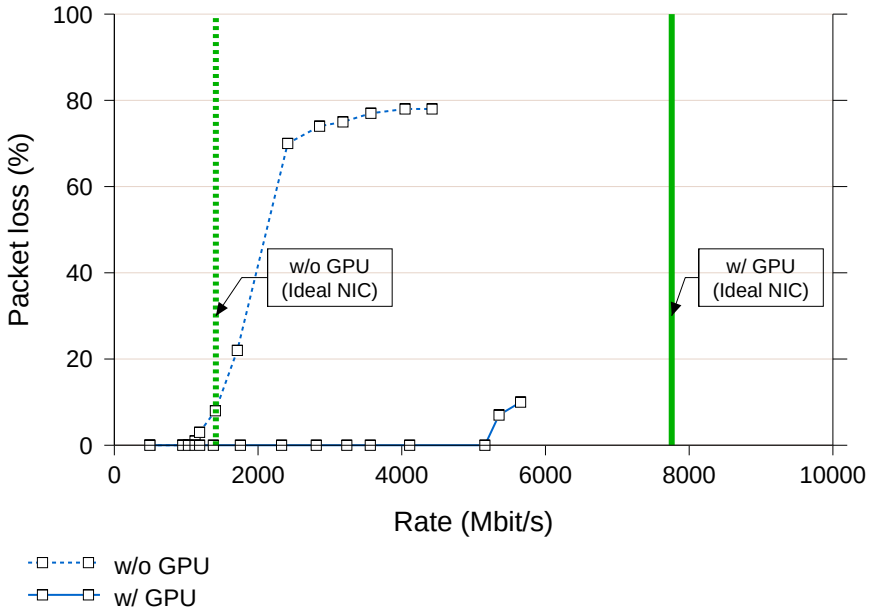
Real Traffic

In our final experiment, we evaluate MIDeA in a scenario using real traffic. We used a trace of anonymized network traffic (referred to as `UNI`), captured at the gateway of a large university campus with several thousands of users. Specifically, the trace spans 74 minutes, and includes all packets and their payloads, totalling 46 GB. Table 4.2 summarizes the most important properties of the trace.

To replay the captured trace at high-speed, it has to reside in the main memory of the host to avoid disk accesses. Unfortunately, the main memory of our two traffic generator machines is only 4 GB, hence it is impossible to load the whole trace in memory. To overcome this issue, we split



(a) Synthetic traffic



(b) Real traffic

FIGURE 4.9: Observed packet loss for (a) synthetic and (b) real traffic, as a function of the traffic rate. MIDeA can handle real traffic speeds of up to 5.2 Gbit/s without dropping any packets.

Packets	73,162,723
Packet size (min/max/avg)	60/1,514/ 679.57
IP Fragments	88,411
TCP sessions	185,642
UDP sessions	174,442
Triggered Snort Alerts	183,050

TABLE 4.2: UNI network trace properties.

the trace to several 2 GB parts. While one part is replayed, the other part is loaded into main memory. Since reading from disk is much slower, each part is replayed several times, up until the next part is fully loaded into memory. Using the above pre-fetching scheme, we successfully managed to replay the captured trace with speeds of up to 5.7 Gbit/s.

Figure 4.9(b) shows the dropped packets when increasing the traffic rate. We also annotate the throughput achieved when reading the network packets directly from main memory instead of the NIC. The traditional multi-core implementation starts to drop packets at 1.1 Gbit/s, while the ideal throughput is near 1.4 Gbit/s. When GPU acceleration is enabled, we did not observe any packet loss for speeds of *up to 5.2 Gbit/s*. For comparison, the ideal throughput is 7.8 Gbit/s.

4.5 Discussion

So far in this chapter we went over a detailed description of the design aspects, trade-offs, and performance issues of our proposed architecture. Even though we focused on the parallelization of an intrusion detection system, we strongly believe that the proposed model can benefit a variety of other network monitoring applications, such as traffic classification, content-aware firewalls, spam filtering, and other network traffic analysis systems. With this in mind, we could easily augment a router with multi-parallel network processing capabilities, expanding its functionality without affecting its normal packet routing operations [51].

4.5.1 Price/Performance

For our hardware setup, we have selected relatively low-end devices: two Intel Xeon E5520 processors, two NVIDIA GeForce GTX 480 graphics cards, and an Intel 82599EB 10GbE NIC. Table 4.3 shows the approximate cost of each component, as of April 2011. The total cost of our base system

Model	Qty	Unit price
NIC: Intel 82599EB	1	$687
CPU: Intel Xeon E5520	2	$336
GPU: NVIDIA GTX480	2	$340

TABLE 4.3: Cost of MIDeA components (as of April 2011).

is about $2739, achieving a throughput per dollar cost of 1.8 Mbps/$.

4.5.2 Limitations

In favor of programming simplicity, we chose to use processes instead of threads for parallelizing the CPU part of MIDeA. We believe that a multi-threaded implementation would further increase the complexity of the design without a significant increase in the overall throughput.

In our flow-based partitioning scheme, we avoid any communication between the cores. Traditional Snort-style signature matching does not require any communication for analysis outside the scope of a single flow. In case a network analysis system needs this functionality, e.g., for detecting DDoS attacks or malware propagation, a lightweight communication scheme needs to be integrated for coordinating the different cores [135].

The buffering of network packets, described in section 4.2.2, introduces an extra copy operation. This is mandatory for our design, considering that most packets have to be processed before matching them against signatures, and that transferring a single packet each time significantly reduces the PCIe throughput.

Finally, each process allocates a different memory space on the GPU, due to the restriction of the CUDA driver for preventing sharing of GPU memory between different processes. Although the same policy applies to threads, we believe that future releases of the CUDA driver will support device memory sharing. In that case, we could easily migrate to the faster AC-Full algorithm. We also believe that a shared GPU memory space would exhibit higher locality and increase the computational throughput.

Chapter 5

A Generic Packet Processing Framework for GPUs

In this chapter we present a network traffic processing framework tailored to modern graphics processors, called GASPP. GASPP integrates optimized GPU-based implementations of a broad range of operations commonly used in network traffic processing applications, including the first purely GPU-based implementation of network flow tracking and TCP stream reassembly. GASPP also employs novel mechanisms for tackling control flow irregularities across SIMT threads, and sharing memory context between the network interface and the GPU. GASPP can achieve multi-gigabit traffic forwarding rates even for computationally intensive and complex network operations such as stateful traffic classification, intrusion detection, and packet encryption. Especially when consolidating multiple network applications on the same device, GASPP achieves up to 16.2x speedup compared to standalone GPU-based implementations of the same applications.

5.1 Motivation

The Need for Modularity. The rise of general-purpose computing on GPUs (GPGPU) and related frameworks, such as CUDA and OpenCL, has made the implementation of GPU-accelerated applications easier than ever. Unfortunately, the majority of GPU-assisted network applications follow a monolithic design, lacking both modularity and flexibility. As a result, building, maintaining, and extending such systems eventually

becomes a real burden. In addition, the absence of libraries for network processing operations—even for simple tasks like packet decoding or filtering—increases development costs even further. GASPP integrates a broad range of operations that different types of network applications rely on, with all the advantages of a GPU-powered implementation, into a single application development platform. This allows developers to focus on core application logic, alleviating the low-level technical challenges of data transfer to and from the GPU, packet batching, asynchronous execution, synchronization issues, connection state management, and so on.

The Need for Stateful Processing. Flow tracking and TCP stream reconstruction are mandatory features of a broad range of network applications. Intrusion detection and traffic classification systems typically inspect the application-layer stream to identify patterns that span multiple packets and thwart evasion attacks [49, 142]. Existing GPU-assisted network processing applications, however, just offload to the GPU certain data-parallel tasks, and are saturated by the many computationally heavy operations that are still being carried out on the CPU, such as network flow tracking, TCP stream reassembly, and protocol parsing [139, 71].

The most common approach for stateful processing is to buffer incoming packets, reassemble them, and deliver "chunks" of the reassembled stream to higher-level processing elements [109, 22]. A major drawback of this approach is that it requires several data copies and significant extra memory space. In Gigabit networks, where packet intervals can be as short as 1.25 μsec (in a 10GbE network, for a MTU of 1.5KB), packet buffering requires large amounts of memory even for very short time windows. To address these challenges, the primary objectives of our GPU-based stateful processing implementation are: (i) process as many packets as possible *on-the-fly* (instead of buffering them), and (ii) ensure that packets of the same connection are processed *in-order*.

5.2 Design

The high-level design of GASPP is shown in Figure 5.1. Packets are transferred from the NICs to the memory space of the GPU in batches. The captured packets are then classified according to their protocol and are processed in parallel by the GPU. For stateful protocols, connection state management and TCP stream reconstruction are supported for delivering

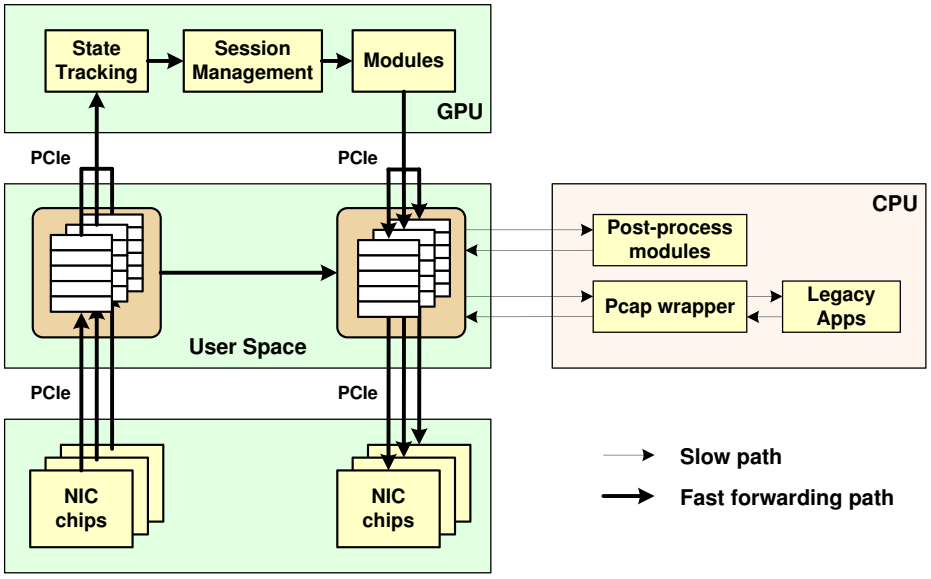


FIGURE 5.1: GASPP architecture.

a consistent application-layer byte stream.

GASPP applications consist of *modules* that control all aspects of the traffic processing flow. Modules are represented as GPU device functions, and take as input a network packet or stream chunk. Internally, each module is executed in parallel on a batch of packets. After processing is completed, the packets are transferred back to the memory space of the host, and depending on the application, to the appropriate output NIC.

5.2.1 Processing Modules

A central concept of NVIDIA's CUDA [102] that has influenced the design of GASPP is the organization of GPU programs into *kernels*, which in essence are functions that are executed by groups of threads. GASPP allows users to specify processing tasks on the incoming traffic by writing GASPP *modules*, applicable on different protocol layers, which are then mapped into GPU kernel functions. Modules can be implemented according to the following prototypes:

```
__device__ uint processEth(unsigned pktid,
    ethhdr *eth, uint cxtkey);
__device__ uint processIP(unsigned pktid,
    ethhdr *eth, iphdr *ip, uint cxtkey);
```

```
__device__ uint processUDP(unsigned pktid,
    ethhdr *eth, iphdr *ip, udphdr *udp, uint cxtkey);
__device__ uint processTCP(unsigned pktid,
    ethhdr *eth, iphdr *ip, tcphdr *tcp, uint cxtkey);
__device__ uint processStream(unsigned pktid,
    ethhdr *eth, iphdr *ip, tcphdr *tcp, uchar *chunk,
    unsigned chunklen, uint cxtkey);
```

The framework is responsible for decoding incoming packets and executing all registered `process*()` modules by passing the appropriate parameters. Packet decoding and stream reassembly is performed by the underlying system, eliminating any extra effort from the side of the developer. Each module is executed at the corresponding layer, with pointer arguments to the encapsulated protocol headers. Arguments also include a unique identifier for each packet and a user-defined key that denotes the packet's class (described in more detail in Section 5.4.3). Currently, GASPP supports the most common network protocols, such as Ethernet, IP, TCP and UDP. Other protocols can easily be handled by explicitly parsing raw packets. Modules are executed per-packet in a data-parallel fashion. If more than one modules have been registered, they are executed back-to-back in a packet processing pipeline, resulting in GPU module chains, as shown in Figure 5.2.

The `processStream()` modules are executed whenever a new normalized TCP chunk of data is available. These modules are responsible for keeping internally the state between consecutive chunks—or, alternatively, for storing chunks in global memory for future use—and continuing the processing from the last state of the previous chunk. For example, a pattern matching application can match the contents of the current chunk and keep the state of its matching algorithm to a global variable; on the arrival of the next chunk, the matching process will continue from the previously stored state.

As modules are simple to write, we expect that users will write new ones as needed using the function prototypes described above. In fact, the complete implementation of a module that simply passes packets from an input to an output interface takes only a few lines of code. More complex network applications, such as NIDS, L7 traffic classification, and packet encryption, require a few dozen lines of code, as described in Section 5.5.

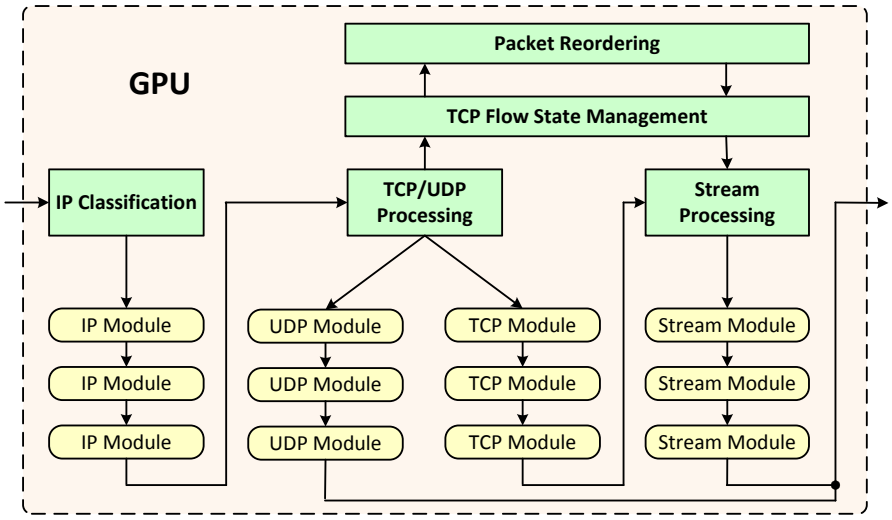


FIGURE 5.2: GPU packet processing pipeline. The pipeline is executed by a different thread for every incoming packet.

5.2.2 API

To cover the needs of a broad range of network traffic processing applications, GASPP offers a rich GPU API with data structures and algorithms for processing network packets.

Shared Hash Table. GASPP enables applications to access the processed data through a global hash table. Data stored in an instance of the hash table is persistent across GPU kernel invocations, and is shared between the host and the device. Internally, data objects are hashed and mapped to a given bucket. To enable GPU threads to add or remove nodes from the table in parallel, we associate an atomic lock with each bucket, so that only a single thread can make changes to a given bucket at a time.

Pattern Matching. Our framework provides a GPU-based API for matching fixed strings and regular expressions. We have ported a variant of the Aho-Corasick algorithm for string searching, and use a DFA-based implementation for regular expression matching. Both implementations have linear complexity over the input data, independent of the number of patterns to be searched. To utilize efficiently the GPU memory subsystem, packet payloads are accessed 16-bytes at a time, using an `int4` variable [140].

Cipher Operations. Currently, GASPP provides AES (128-bit to 512-bit key sizes) and RSA (1024-bit and 2048-bit key sizes) functions for encryption and decryption, and supports all modes of AES (ECB, CTR, CFB and OFB). Again, packet contents are read and written 16-bytes at a time, as this substantially improves GPU performance. The encryption and decryption process happens in-place and as packet lengths may be modified, the checksums for IP and TCP/UDP packets are recomputed to be consistent. In cases where the NIC controller supports checksum computation offloading, GASPP simply forwards the altered packets to the NIC.

Network Packet Manipulation Functions. GASPP provides special functions for dropping network packets (`Drop()`), ignoring any subsequent registered user-defined modules (`Ignore()`), passing packets to the host for further processing (`ToLinux()`), or writing their contents to a dump file (`ToDump()`). Each function updates accordingly the packet index array, which holds the offsets where each packet is stored in the packet buffer, and a separate "metadata" array.

5.3 Stateful Protocol Analysis

The stateful protocol analysis component of GASPP is designed with minimal complexity so as to maximize processing speed. This component is responsible for maintaining the state of TCP connections, and reconstructing the application-level byte stream by merging packet payloads and reordering out-of-order packets.

5.3.1 Flow Tracking

GASPP uses a connection table array stored in the global device memory of the GPU for keeping the state of TCP connections. Each record is 17-byte long. A 4-byte hash of the source and destination IP addresses and TCP ports is used to handle collisions in the flow classifier. Connection state is stored in a single-byte variable. The sequence numbers of the most recently received client and server segments are stored in two 4-byte fields, and are updated every time the next in-order segment arrives. Hash table collisions are handled using a locking chained hash table with linked lists (described in detail in Section 5.2.2). A 4-byte pointer points to the next record (if any).

The connection table can easily fill up with adversarial partially established connections, benign connections that stay idle for a long time, or

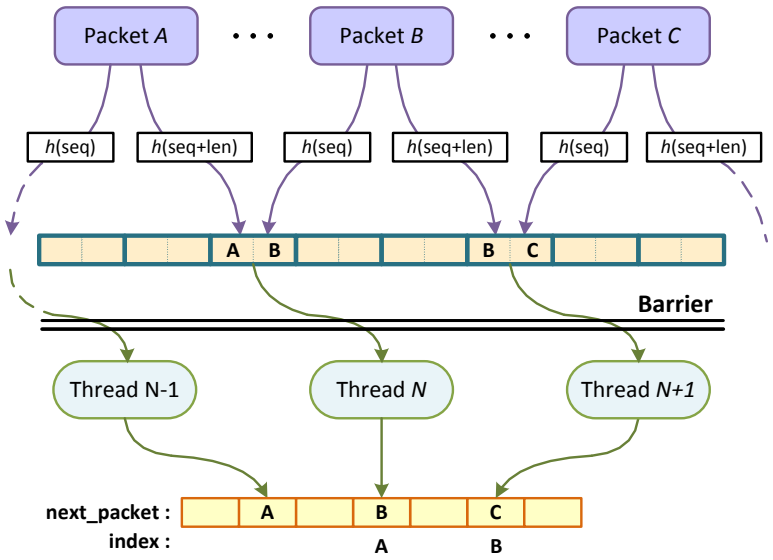


FIGURE 5.3: Ordering sequential TCP packets in parallel. The resulting *next_packet* array contains the next in-order packet, if any (i.e. *next_packet[A]* = *B*).

connections that failed to terminate properly. For this reason, connection records that have been idle for more than a certain timeout, set to 60 seconds by default, are periodically removed. As current GPU devices do not provide support for measuring real-world time, we resort to a separate GPU kernel that is initiated periodically according to the timeout value. Its task is to simply mark each connection record by setting the first bit of the state variable. If a connection record is already marked, it is removed from the table. A marked record is unmarked when a new packet for this connection is received before the timeout expires.

5.3.2 Parallelizing TCP Stream Reassembly

Maintaining the state of incoming connections is simple as long as the packets that are processed in parallel by the GPU belong to different connections. Typically, however, a batch of packets usually contains several packets of the same connection. It is thus important to ensure that the order of connection updates will be correct when processing packets of the same connection in parallel.

TCP reconstruction threads are synchronized through a separate array used for pairing threads that must process consecutive packets. When a new batch is received, each thread hashes its packet twice: once using

hash(addr_s, addr_d, port_s, port_d, seq), and a second time using *hash(addr_s, addr_d, port_s, port_d, seq + len)*, as shown in Figure 5.3. A memory barrier is used to guarantee that all threads have finished hashing their packets. Using this scheme, two packets x and y are consecutive if: $hash_x(\text{4-}tuple, seq + len) = hash_y(\text{4-}tuple, seq)$. The hash function is unidirectional to ensure that each stream direction is reconstructed separately. The SYN and SYN-ACK packets are paired by hashing the sequence and acknowledge numbers correspondingly. If both the SYN and SYN-ACK packets are present, the state of the connection is changed to `ESTABLISHED`, otherwise if only the SYN packet is present, the state is set to `SYN_RECEIVED`.

Having hashed all pairs of consecutive packets in the hash table, the next step is to create the proper packet ordering for each TCP stream using the `next_packet` array, as shown in Figure 5.3. Each packet is uniquely identified by an `id`, which corresponds to the index where the packet is stored in the packet index array. The `next_packet` array is set at the beginning of the current batch, and its cells contain the `id` of the next in-order packet (or `-1` if it does not exist in the current batch), e.g., if `x` is the `id` of the current packet, the `id` of the next in-order packet will be `y = next_packet[x]`. Finally, the connection table is updated with the sequence number of the last packet of each flow direction, i.e., the packet `x` that does not have a next packet in the current batch.

5.3.3 Packet Reordering

Although batch processing handles out-of-order packets that are included in the same batch, it does not solve the problem in the general case. A potential solution for in-line applications would be to just drop out-of-sequence packets, forcing the host to retransmit them. Whenever an expected packet would be missing, subsequent packets would be actively dropped until the missing packet arrives. Although this approach would ensure an in-order packet flow, it has several disadvantages. First, in situations where the percentage of out-of-order packets is high, performance will degrade. Second, if the endpoints are using selective retransmission and there is a high rate of data loss in the network, connections would be rendered unusable due to excessive packet drops.

To deal with TCP sequence hole scenarios, GASPP only processes packets with sequence numbers less than or equal to the connection's current sequence number (Figure 5.4(a)–(d)). Received packets with no preceding

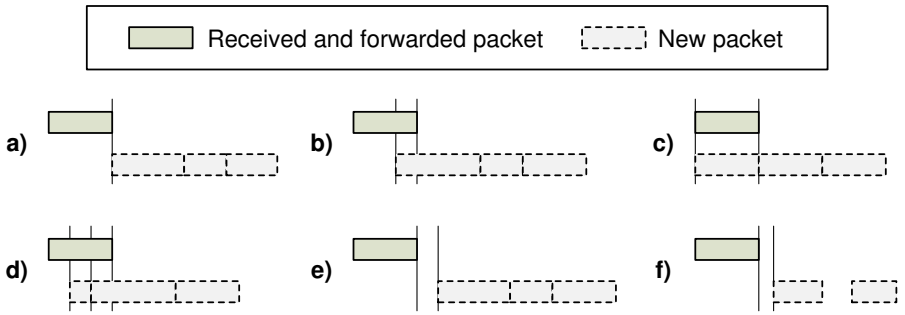


FIGURE 5.4: Subsequent packets (dashed line) may arrive in-sequence ((a)–(d)) or out of order, creating holes in the reconstructed TCP stream ((e)–(f)).

packets in the current batch and with sequence numbers larger than the ones stored in the connection table imply sequence holes (Figure 5.4(e)–(f)), and are copied in a separate buffer in global device memory. If a thread encounters an out-of-order packet (i.e., a packet with a sequence number larger than the sequence number stored in the connection table, with no preceding packet in the current batch after the hashing calculations of Section 5.3.2), it traverses the `next_packet` array and marks as out-of-order all subsequent packets of the same flow contained in the current batch (if any). This allows the system to identify sequences of out-of-order packets, as the ones shown in the examples of Figure 5.4(e)–(f). The buffer size is configurable and can be up to several hundred MBs, depending on the network needs. If the buffer contains any out-of-order packets, these are processed right after a new batch of incoming packets is processed.

Although packets are copied using the very fast device-to-device copy mechanism, with a memory bandwidth of about 145 GB/s, an increased number of out-of-order packets can have a major effect on overall performance. For this reason, by default we limit the number of out-of-order packets that can be buffered to be equal to the available slots in a batch of packets. This size is enough under normal conditions, where out-of-order packets are quite rare [49], and it can be configured as needed for other environments. If the percentage of out-of-order packets exceeds this limit, our system starts to drop out-of-order packets, causing the corresponding host to retransmit them.

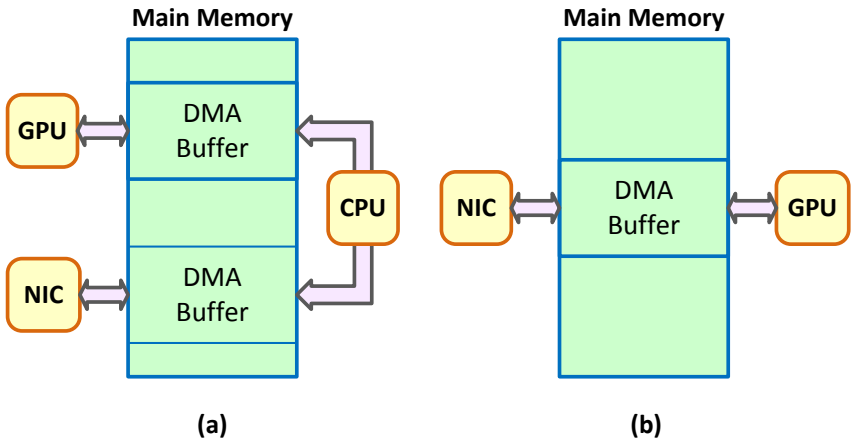


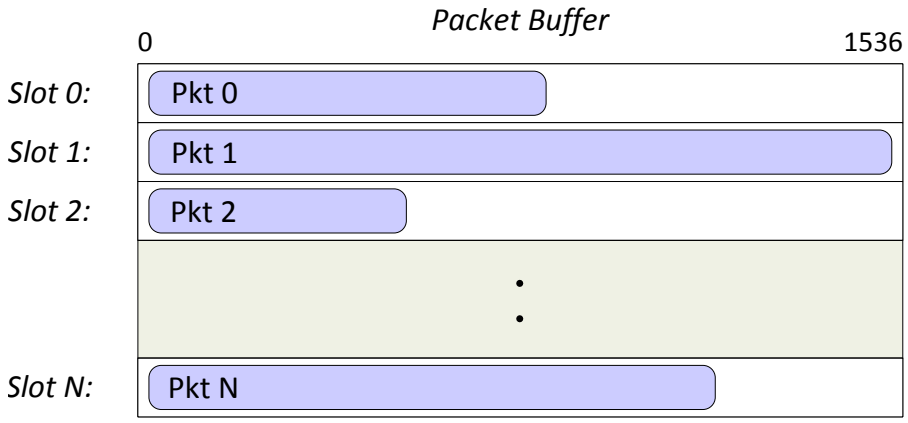
FIGURE 5.5: Normal (a) and zero-copy (b) data transfer between the NIC and the GPU.

5.4 Optimizing Performance

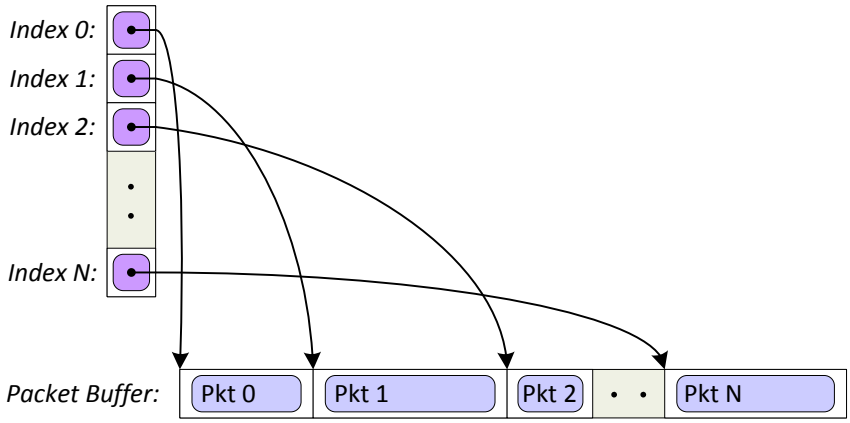
5.4.1 Inter-Device Data Transfer

The problem of data transfers between the CPU and the GPU is well-known in the GPGPU community, as it results in redundant cross-device communication. The traditional approach is to exchange data using DMA between the memory regions assigned by the OS to each device. As shown in Figure 5.5(a), network packets are transferred to the page-locked memory of the NIC, then copied to the page-locked memory of the GPU, and from there, they are finally transferred to the GPU.

To avoid costly packet copies and context switches, GASPP uses a single buffer for efficient data sharing between the NIC and the GPU, as shown in Figure 5.5(b), by adjusting the `netmap` module [110]. The shared buffer is added to the internal tracking mechanism of the CUDA driver to automatically accelerate calls to functions, as it can be accessed directly by the GPU. The buffer is managed by GASPP through the specification of a policy based on time and size constraints. This enables real-time applications to process incoming packets whenever a timeout is triggered, instead of waiting for buffers to fill up over a specified threshold. Per-packet buffer allocation overheads are reduced by transferring several packets at a time. Buffers consist of fixed-size slots, with each slot corresponding to one packet in the hardware queue. Slots are reused whenever the circular hardware queue wraps around. The size of each slot is 1,536 bytes, which



(a) Packets are stored to distinct buckets.



(b) Packets are stored sequentially and indexed by a separate directory.

FIGURE 5.6: Different network packet buffer formats.

is consistent with the NIC's alignment requirements, and enough for the typical 1,518-byte maximum Ethernet frame size.

Although making the NIC's packet queue directly accessible to the GPU eliminates redundant copies, this does not always lead to better performance. As previous studies have shown [60, 139] (we verify their results in Section 5.6.2), contrary to NICs, current GPU implementations suffer from poor performance for small data transfers. To improve PCIe throughput, we batch several packets and transfer them at once. However, the fixed-size partitioning of the NIC's queue leads to redundant data transfers for traffic with many small packets. For example, a 64-byte packet consumes only 1/24th of the available space in its slot. This introduces

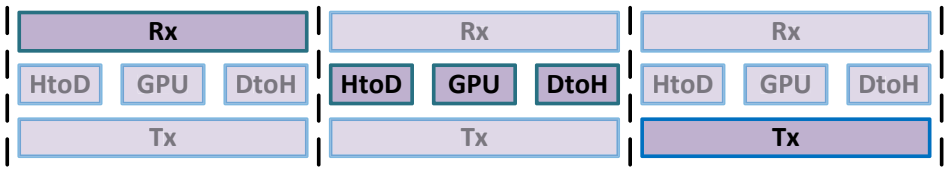


FIGURE 5.7: The I/O and processing pipeline.

an interesting trade-off, and as we show in Section 5.6.2, occasionally it is better to copy packets back-to-back into a second buffer, along with an array of offsets for each packet as shown in Figure 5.6(b), and transferring it to the GPU. In order to avoid an extra transaction over the PCIe bus, the index array is stored in the beginning of the packet buffer. The packet buffer and the indices are transferred to the GPU at once, adding a minor transfer cost, since the size of the index array is quite small in regards to the size of the packet buffer. GASPP dynamically switches to the optimal approach by monitoring the actual utilization of the slots.

The forwarding path requires the transmission of network packets after processing is completed, and this is achieved using a triple-pipeline solution, as shown in Figure 5.7. Packet reception, GPU data transfers and execution, and packet transmission are executed asynchronously in a multiplexed manner.

5.4.2 Packet Decoding

Memory alignment is a major factor that affects the packet decoding process, as GPU execution constrains memory accesses to be aligned for all data types. For example, `int` variables should be stored to addresses that are a multiple of `sizeof(int)`. Due to the layered nature of network protocols, however, several fields of encapsulated protocols are not aligned when transferred to the memory space of the GPU. To overcome this issue, GASPP reads the packet headers from global memory, parses them using bitwise logic and shifting operations, and stores them in appropriately aligned structures. To optimize memory usage, input data is accessed in units of 16 bytes (using an `int4` variable).

5.4.3 Packet Scheduling

Registered modules are scheduled on the GPU, per protocol, in a serial fashion. Whenever a new batch of packets is available, it is processed in parallel using a number of threads equal to the number of packets in the

batch (each thread processes a different packet). As shown in Figure 5.2, all registered modules for a certain protocol are executed serially on decoded packets in a lockstep way.

Network packets are processed by different threads, grouped together into logical units known as *warps* (in current NVIDIA GPU architectures, 32 threads form a warp) and mapped to SIMT units. As threads within the same warp have to execute the same instructions, load imbalance and code flow divergence within a warp can cause inefficiencies. This may occur under the following primary conditions: (i) when processing different transport-layer protocols (i.e., TCP and UDP) in the same warp, (ii) in full-packet processing applications when packet lengths within a warp differ significantly, and (iii) when different packets follow different processing paths, i.e., threads of the same warp execute different user-defined modules.

As the received traffic mix is typically very dynamic, it is essential to find an appropriate mapping between threads and network packets at runtime. It is also crucial that the overhead of the mapping process is low, so as to not jeopardize overall performance. To that end, our basic strategy is to group the packets of a batch according to their encapsulated transport-layer protocol and their length. In addition, module developers can specify *context keys* to describe packets that belong to the same class, which should follow the same module execution pipeline. A context key is a value returned by a user-defined module and is passed (as the final parameter) to the next registered module. GASPP uses these context keys to further pack packets of the same class together and map them to threads of the same warp after each module execution. This gives developers the flexibility to build complex packet processing pipelines that will be mapped efficiently to the underlying GPU architecture at runtime.

To group a batch of packets on the GPU, we have adapted a GPU-based radix sort implementation [8]. Specifically, we assign a separate weight for each packet consisting of the byte concatenation of the `ip_proto` field of its IP header, the value of the context key returned by the previously executed module, and its length. Weights are calculated on the GPU after each module execution using a separate thread for each packet, and are used by the radix sort algorithm to group the packets. Moreover, instead of copying each packet to the appropriate (i.e., sorted) position, we simply

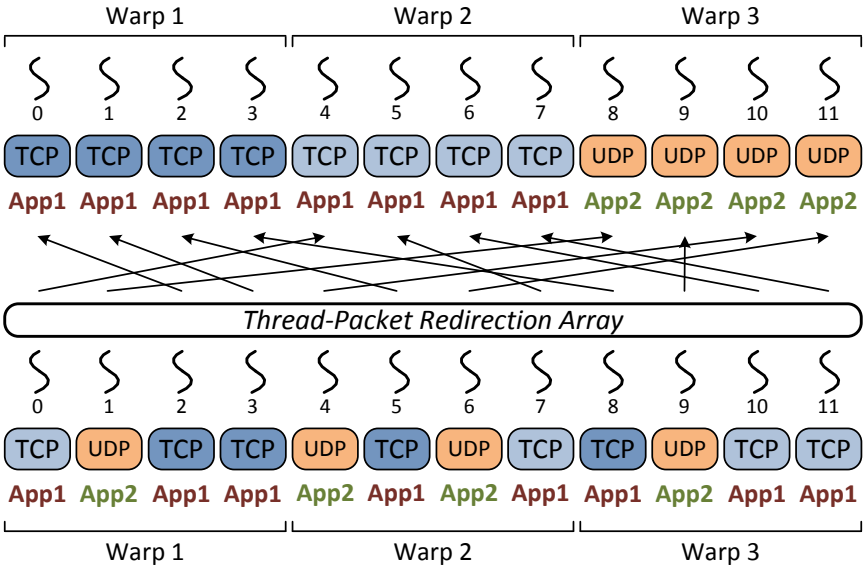


FIGURE 5.8: Packet scheduling for eliminating control flow divergences and load imbalances. Packet brightness represents packet size.

change their order in the packet index array. We also attempted to relocate packets by transposing the packet array on the GPU device memory, in order to benefit from memory coalescing [102]. Unfortunately, the overall cost of the corresponding data movements was not amortized by the resulting memory coalescing gains.

Using the above procedure, GASPP assigns dynamically to the same warp any similar-sized packets meant to be processed by the same module, as shown in Figure 5.8. Packets that were discarded earlier or of which the processing pipeline has been completed are grouped and mapped to warps that contain only idle threads—otherwise warps would contain both idle and active threads, degrading the utilization of the SIMT processors. To prevent packet reordering from taking place during packet forwarding, we also preserve the initial (pre-sorted) packet index array. In Section 5.6.3 we analyze in detail how control flow divergence affects the performance of the GPU, and show how our packet scheduling mechanisms tackle the irregular code execution at a fixed cost.

5.5 Developing with GASPP

In this section we present simple examples of representative applications built using the GASPP framework.

5.5.1 Firewall

Firewalls operate at the network layer (port-based) or the application layer (content-based). For our purposes, we have built a GASPP module that can drop traffic based on Layer-3 and Layer-4 rules. An incoming packet is filtered if the corresponding IP addresses and port numbers are found in the hash table; otherwise the packet is forwarded.

5.5.2 L7 Traffic Classification

We have implemented a L7 traffic classification tool (similar to the L7-filter tool [2]) on top of GASPP. The tool dynamically loads the pattern set of the L7-filter tool, in which each application-level protocol (HTTP, SMTP, etc.) is represented by a different regular expression. At runtime, each incoming flow is matched against each regular expression independently. In order to match patterns that cross TCP segment boundaries that lie on the same batch, each thread continues the processing to the next TCP segment (obtained from the `next_packet` array). The processing of the next TCP segment continues until a final or a fail DFA-state is reached, as suggested in [138]. In addition, the DFA-state of the last TCP segment of the current batch is stored in a global variable, so that on the arrival of the next stream chunk, the matching process continues from the previously stored state. This allows the detection of regular expressions that span (potentially deliberately) not only multiple packets, but also two or more stream chunks.

5.5.3 Signature-based Intrusion Detection

Modern NIDS, such as Snort [22], use a large number of regular expressions to determine whether a packet stream contains an attack vector or not. To reduce the number of packets that need to be matched against a regular expression, typical NIDS take advantage of the string matching engine and use it as a first-level filtering mechanism before proceeding to regular expression matching. We have implemented the same functionality on top of GASPP, using a different module for scanning each incoming traffic stream against all the fixed strings in a signature set. Patterns that cross TCP segments are handled similarly to the L7 Traffic Classification module. Only the matching streams are further processed against the corresponding regular expressions set.

Packet size (bytes)	64	128	256	512	1024	1518
Copy back-to-back	13.76	18.21	20.53	19.21	19.24	20.04
Zero-copy	2.06	4.03	8.07	16.13	32.26	47.83

TABLE 5.1: Sustained throughput (Gbit/s) for various packet sizes, when bulk-transferring data to a single GPU.

5.5.4 AES

Encryption is used by protocols and services, such as SSL, VPN, and IPsec, for securing communications by authenticating and encrypting the IP packets of a communication session. While stock protocol suites that are used to secure communications, such as IPsec, actually use connectionless integrity and data origin authentication, for simplicity, we only encrypt all incoming packets using the AES-CBC algorithm and a different 128-bit key for each connection.

5.6 Performance Evaluation

5.6.1 Hardware Setup

Our base system is equipped with two Intel Xeon E5520 Quad-core CPUs at 2.27 GHz and 12 GB of RAM (6 GB per NUMA domain). Each CPU is connected to peripherals via a separate I/O hub, linked to several PCIe slots. Each I/O hub is connected to an NVIDIA GTX480 graphics card via a PCIe v2.0 x16 slot, and one Intel 82599EB with two 10 GbE ports, via a PCIe v2.0 8× slot. The system runs Linux 3.5 with CUDA v5.0 installed. After experimentation, we have found that the best placement is to have a GPU and a NIC on each NUMA node. We also place the GPU and NIC buffers in the same memory domain, as local memory accesses sustain lower latency and more bandwidth compared to remote accesses.

For traffic generation we use a custom packet generator built on top of `netmap` [110]. Test traffic consists of both synthetic traffic, as well as real traffic traces.

5.6.2 Data Transfer

We evaluate the zero-copy mechanism by taking into account the size of the transferred packets. The system can efficiently deliver all incoming packets to user space, regardless of the packet size, by mapping the NIC's DMA packet buffer. However, small data transfers to the GPU incur

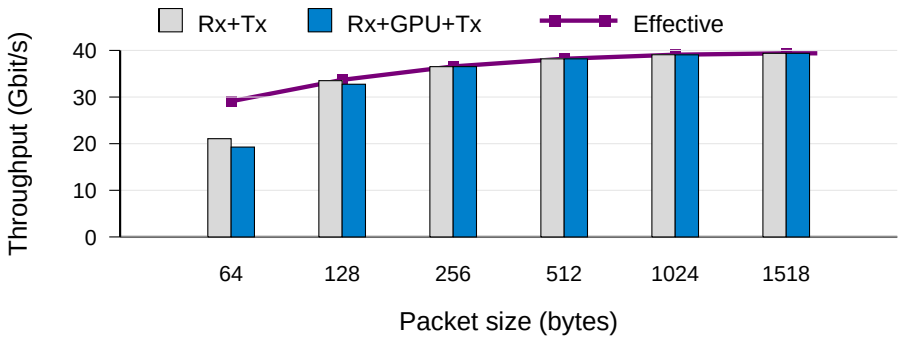


FIGURE 5.9: Data transfer throughput for different packet sizes when using two dual-port 10GbE NICs.

significant penalties. Table 2.1 (page 18) shows that for transfers of less than 4 KB, the PCIe throughput falls below 7 Gbit/s. With a large buffer though, the transfer rate to the GPU exceeds 45 Gbit/s, while the transfer rate from the GPU to the host decreases to about 25 Gbit/s.

To overcome the low PCIe throughput, GASPP transfers batches of network packets to the GPU, instead of one at a time. However, as packets are placed in fixed-sized slots, transferring many slots at once results in redundant data transfers when the slots are not fully occupied. As we can see in Table 5.1, when traffic consists of small packets, the actual PCIe throughput drops drastically. Thus, it is better to copy small network packets sequentially into another buffer, rather than transfer the corresponding slots directly. Direct transfer pays off only for packet sizes over 512 bytes (when buffer occupancy is over $512/1536 = 33.3\%$), achieving 47.8 Gbit/s for 1518-byte packets (a 2.3× speedup).

Consequently, we adopted a simple *selective offloading* scheme, whereby packets in the shared buffer are copied to another buffer sequentially (in 16-byte aligned boundaries) if the overall occupancy of the shared buffer is sparse. Otherwise, the shared buffer is transferred directly to the GPU. Occupancy is computed—without any additional overhead—by simply counting the number of bytes of the newly arrived packets every time a new interrupt is generated by the NIC.

Figure 5.9 shows the throughput for forwarding packets with all data transfers included, but without any GPU computations. We observe that the forwarding performance for 64-byte packets reaches 21 Gbit/s, out of the maximum 29.09 Gbit/s, while for large packets it reaches the maxi-

mum full line rate. We also observe that the GPU transfers of large packets are completely hidden on the Rx+GPU+Tx path, as they are performed in parallel using the pipeline shown in Figure 5.7, and thus they do not affect overall performance. Unfortunately, this is not the case for small packets (less than 128-bytes), which suffer an additional 2–9% hit due to memory contention.

5.6.3 Raw GPU Processing Throughput

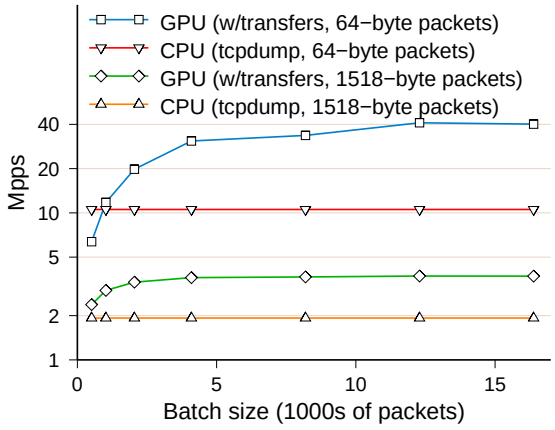
Having examined data transfer costs, we now evaluate the computational performance of a single GPU—exluding all network I/O transfers—for packet decoding, connection state management, TCP stream reassembly, and some representative traffic processing applications.

Packet Decoding

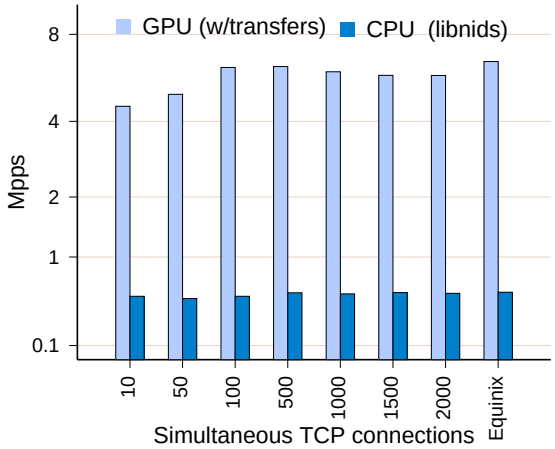
Decoding a packet according to its protocols is one of the most basic packet processing operations, and thus we use it as a base cost of our framework. Figure 5.10(a) shows the GPU performance for fully decoding incoming UDP packets into appropriately aligned structures, as described in Section 5.4.2 (throughput is very similar for TCP). As expected, the throughput increases as the number of packets processed in parallel increases. When decoding 64-byte packets, the GPU performance with PCIe transfers included, reaches 48 Mpps, which is about 4.5 times faster than the computational throughput of the `tcpdump` decoding process sustained by a single CPU core, when packets are read from memory. For 1518-byte packets, the GPU sustains about 3.8 Mpps and matches the performance of 1.92 CPU cores.

Connection State Management and TCP Stream Reassembly

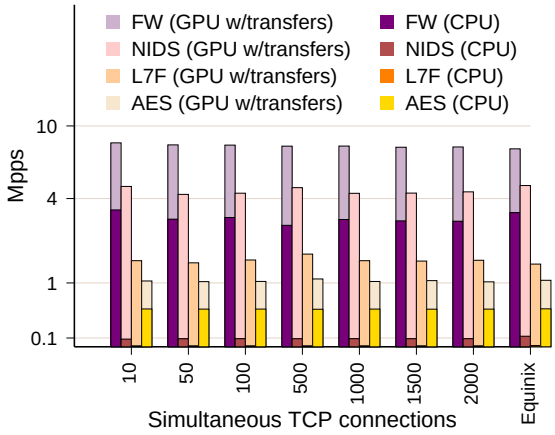
In this experiment we measure the performance of maintaining connection state on the GPU, and the performance of reassembling the packets of TCP flows into application-level streams. Figure 5.10(b) shows the packets processed per second for both operations. Test traffic consists of real HTTP connections with random IP addresses and TCP ports. Each connection fetches about 800 KB from a server, and comprises about 870 packets (320 minimum-size ACKs, and 550 full-size data packets). We also use a trace-driven workload ("Equinix") based on a trace captured by CAIDA's *equinix-sanjose* monitor [25], in which the average and median packet length is 606.2 and 81 bytes respectively.



(a) Packet decode.



(b) TCP state management and reassembly



(c) Applications.

FIGURE 5.10: Average processing throughput sustained by the GPU to (a) decode network packets, (b) maintain flow state and reassemble TCP streams, and (c) perform various network processing operations.

Elements	1M buckets	8M buckets	16M buckets
0.1M	463	3,595	7,166
1M	463	3,588	7,173
2M	934	3,593	7,181
4M	1,924	3,593	7,177
8M	3,935	3,597	7,171
16M	7,991	7,430	7,173
32M	16,060	15,344	14,851

TABLE 5.2: Time spent (μsec) for traversing the connection table and removing expired connections.

Keeping state and reassembling streams requires several hashtable lookups and updates, which result to marginal overhead for a sufficient number of simultaneous TCP connections and the Equinix trace; about 20–25% on the raw GPU performance sustained for packet decoding, that increases to 45–50% when the number of concurrent connections is low. The reason is that smaller numbers of concurrent connections result to lower parallelism. To compare with a CPU implementation, we measure the equivalent functionality of the Libnids TCP reassembly library [109], when packets are read from memory. Although Libnids implements more specific cases of the TCP stack processing, compared to GASPP, the network traces that we used for the evaluation enforce exactly the same functionality to be exercised. We can see that the throughput of a single CPU core is 0.55 Mpps, about 10$\times$ lower than the GPU version with all PCIe data transfers included.

Removing Expired Connections

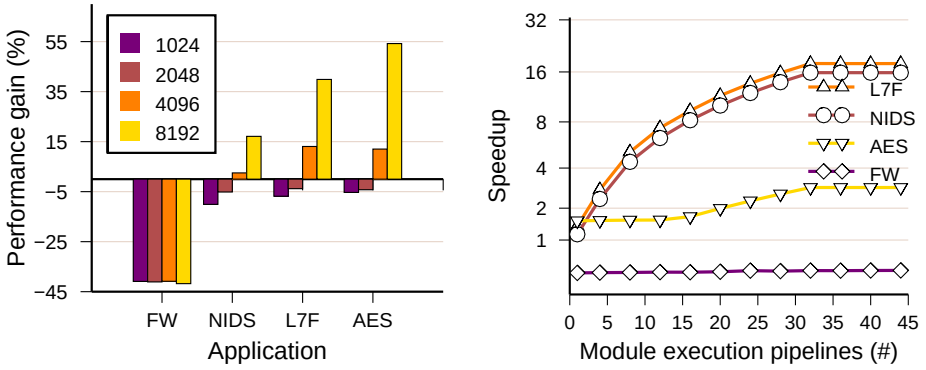
Removal of expired connections is very important for preventing the connection table from becoming full with stale adversarial connections, idle benign connections, or connections that failed to terminate cleanly [142]. Table 5.2 shows the GPU time spent for connection expiration. The time spent to traverse the table is constant when occupancy is lower than 100%, and analogous to the number of buckets; for larger values it increases due to the extra overhead of iterating the chain lists. Having a small hash table with a large load factor is better than a large but sparsely populated table. For example, the time to traverse a 1M-bucket table that contains up to 1M elements is about 20$\times$ lower than a 16M-bucket table with the same

number of elements. If the occupancy is higher than 100% though, it is slightly better to use a 16M-bucket table.

Packet Processing Applications

In this experiment we measure the computational throughput of the GPU for the applications presented in Section 5.5. The NIDS is configured to use all the `content` patterns (about 10,000 strings) of the latest Snort distribution [22], combined into a single Aho-Corasick state machine, and their corresponding `pcre` regular expressions compiled into individual DFA state machines. The application-layer filter application (L7F) uses the "best-quality" patterns (12 regular expressions for identifying common services such as HTTP and SSH) of L7-filter [2], compiled into 12 different DFA state machines. The Firewall (FW) application uses 10,000 randomly generated rules for blocking incoming and outgoing traffic based on certain TCP/UDP port numbers and IP addresses. The test traffic consists of the HTTP-based traffic and the trace-driven Equinix workload described earlier. Note that the increased asymmetry in packet lengths and network protocols in the above traces is a stress-test workload for our data-parallel applications, given the SIMT architecture of GPUs [102].

Figure 5.10(c) shows the GPU throughput sustained by each application, including PCIe transfers, when packets are read from host memory. FW, as expected, has the highest throughput of about 8 Mpps—about 2.3 times higher than the equivalent single-core CPU execution. The throughput for NIDS is about 4.2–5.7 Mpps, and for L7F is about 1.45–1.73 Mpps. The large difference between the two applications is due to the fact that the NIDS shares the same Aho-Corasick state machine to initially search all packets (as we described in Section 5.5). In the common case, each packet will be matched only once against a single DFA. In contrast, the L7F requires each packet to be explicitly matched against each of the 12 different regular expression DFAs for both CPU and GPU implementations. The corresponding single-core CPU implementation of NIDS reaches about 0.1 Mpps, while L7F reaches 0.01 Mpps. We also note that both applications are explicitly forced to match all packets of all flows, even after they have been successfully classified (worst-case analysis). Finally, AES has a throughput of about 1.1 Mpps, as it is more computationally intensive. The corresponding CPU implementation using the AES-NI [11]



(a) Load imbalance (b) Divergence

FIGURE 5.11: Performance gains on raw GPU execution time when applying packet scheduling (the scheduling cost is included).

instruction set on a single core reaches about 0.51 Mpps.[1]

Packet Scheduling

In this experiment we measure how the packet scheduling technique, described in Section 5.4.3, affects the performance of different network applications. For test traffic we used the trace-driven Equinix workload. Figure 5.11(a) shows the performance gain of each application for different packet batch sizes. We note that although the actual work of the modules is the same every time (i.e., the same processing will be applied on each packet), it is executed by different code blocks, thus execution is forced to diverge.

We observe that packet scheduling boosts the performance of full-packet processing applications, up to 55% for computationally intensive workloads like AES. Memory-intensive applications, such as NIDS, have a lower (about 15%) benefit. We also observe that gains increase as the batch size increases. With larger batch sizes, there is a greater range of packet sizes and protocols, hence more opportunities for better grouping. In contrast, packet scheduling has a negative effect on lightweight processing (as in FW, which only processes a few bytes of each packet), because the sorting overhead is not amortized by the resulting SIMT execution. As we cannot know at runtime if processing will be heavyweight or not, it is not feasible to predict if packet scheduling is worth applying. As a re-

[1]The CPU performance of AES was measured on an Intel Xeon E5620 at 2.40GHz, because the Intel Xeon E5520 of our base system does not support AES-NI.

sult, quite lightweight workloads (as in FW) will perform worse, although this lower performance will be hidden most of the time by data transfer overlap (Figure 5.7).

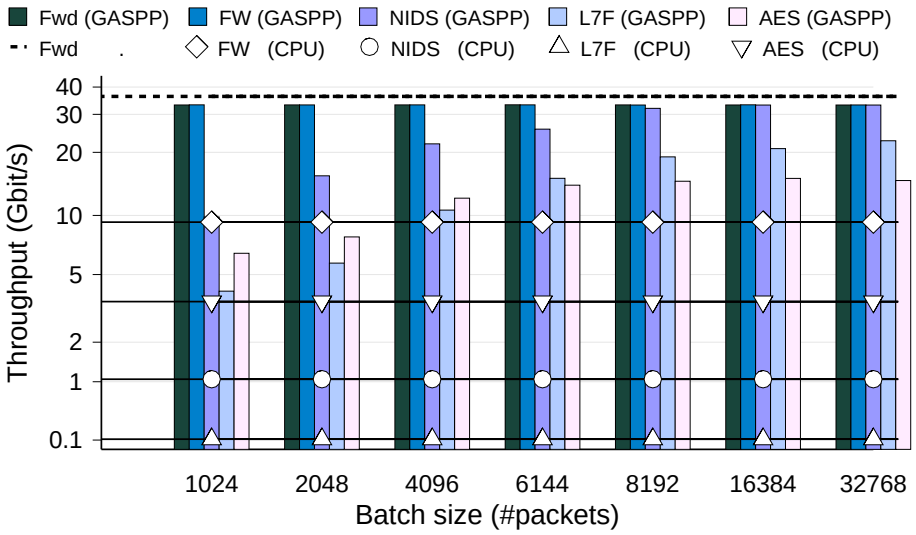
Another important aspect is how control flow divergence affects performance, e.g., when packets follow different module execution pipelines. To achieve this, we explicitly enforce different packets of the same batch to be processed by different modules. Figure 5.11(b) shows the achieved speedup when applying packet scheduling over the baseline case of mapping packets to thread warps without any reordering (network order). We see that as the number of different modules increases, our packet scheduling technique achieves a significant speedup. The speedup stabilizes after the number of modules exceeds 32, as only 32 threads (warp size) can run in a SIMT manner any given time. In general, code divergence within warps plays a significant role in GPU performance. The thread remapping achieved through our packet scheduling technique tolerates the irregular code execution at a fixed cost.

5.6.4 End-to-End Performance

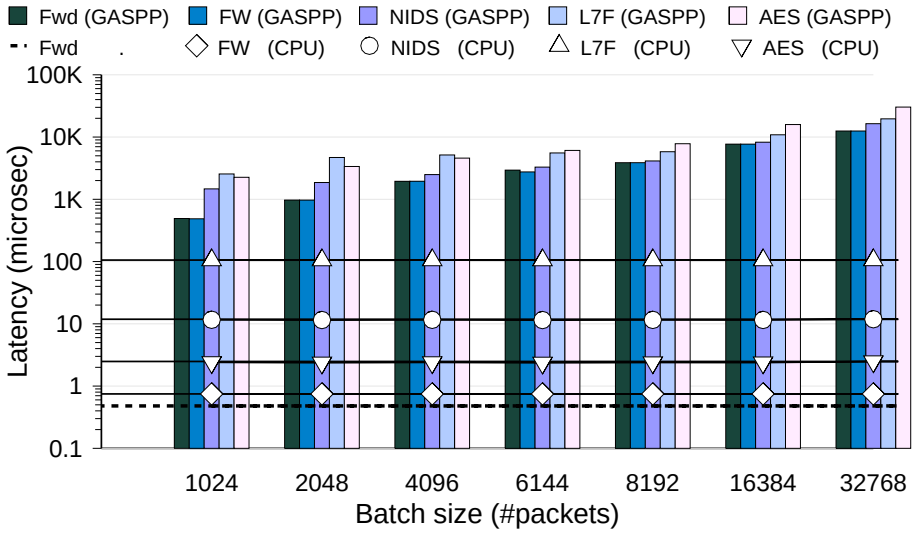
Individual Applications

Figure 5.12 shows the sustained end-to-end forwarding throughput and latency of individual GASPP-enabled applications for different batch sizes. We use four different traffic generators, equal to the number of available 10 GbE ports in our system. To prevent synchronization effects between the generators, the test workload consists of the HTTP-based traffic described earlier. For comparison, we also evaluate the corresponding CPU-based implementations running on a single core, on top of `netmap`.

The FW application can process all traffic delivered to the GPU, even for small batch sizes. NIDS, L7F, and AES, on the other hand, require larger batch sizes. The NIDS application requires batches of 8,192 packets to reach similar performance. Equivalent performance would be achieved (assuming ideal parallelization) by 28.4 CPU cores. More computationally intensive applications, however, such as L7F and AES, cannot process all traffic. L7F reaches 19 Gbit/s a batch size of 8,192 packets, and converges to 22.6 Gbit/s for larger sizes—about 205.1 times faster than a single CPU core. AES converges to about 15.8 Gbit/s, and matches the performance of 4.4 CPU cores with AES-NI support. As expected, latency increases



(a) Throughput.



(b) Latency.

FIGURE 5.12: Sustained traffic forwarding throughput (a) and latency (b) for GASPP-enabled applications.

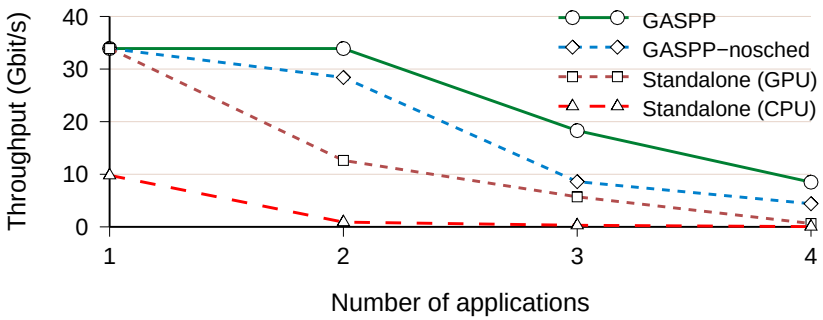


FIGURE 5.13: Sustained throughput for concurrently running applications.

linearly with the batch size, and for certain applications and large batch sizes it can reach tens of milliseconds (Figure 5.12(b)). Fortunately, a batch size of 8,192 packets allows for a reasonable latency for all applications, while it sufficiently utilizes the PCIe bus and the parallel capabilities of the GTX480 card (Figure 5.12(a)). For instance, NIDS, L7F, and FW have a latency of 3–5 ms, while AES, which suffers from an extra GPU-to-host data transfer, has a latency of 7.8 ms.

Consolidated Applications

Consolidating multiple applications has the benefit of distributing the overhead of data transfer, packet decoding, state management, and stream reassembly across all applications, as all these operations are performed only once. Moreover, through the use of context keys,GASPP optimizes SIMT execution when packets of the same batch are processed by different applications. Figure 5.13 shows the sustained throughput when running multiple GASPP applications. Applications are added in the following order: FW, NIDS, L7F, AES (increasing overhead). We also enforce packets of different connections to follow different application processing paths. Specifically, we use the hash of the each packet's 5-tuple for deciding the order of execution. For example, a class of packets will be processed by application 1 and then application 2, while others will be processed by application 2 and then by application 1; eventually, all packets will be processed by all registered applications. For comparison, we also plot the performance of GASPP when packet scheduling is disabled (GASPP-nosched), and the performance of having multiple standalone applications running on the GPU and the CPU.

We see that the throughput for GASPP converges to the throughput

of the most intensive application. When combining the first two applications, the throughput remains at 33.9 Gbit/s. When adding the L7F (x=3), performance degrades to 18.3 Gbit/s. L7F alone reaches about 20 Gbit/s (Figure 5.12(a)). When adding AES (x=4), performance drops to 8.5 Gbit/s, which is about 1.93× faster than GASPP-nosched. The achieved throughput when running multiple standalone GPU-based implementations is about 16.25× lower than GASPP, due to excessive data transfers.

5.7 Limitations

Typically, a GASPP developer will prefer to port functionality that is parallelizable, and thus benefit from the GPU execution model. However, there may be parts of data processing operations that do not necessarily fit well on the GPU. In particular, middlebox functionality with complex conditional processing and frequent branching may require extra effort.

The packet scheduling mechanisms described in Section 5.4.3 help accommodate such cases by forming groups of packets that will follow the same execution path and will not affect GPU execution. Still, (i) divergent workloads that perform quite lightweight processing (e.g., which process only a few bytes from each packet, such as the FW application), or (ii) workloads where it is not easy to know which packet will follow which execution path, may not be parallelized efficiently on top of GASPP. The reason is that in these cases the cost of grouping is much higher than the resulting benefits, while GASPP cannot predict if packet scheduling is worth the case at runtime. To overcome this, GASPP allows applications to selectively pass network packets and their metadata to the host CPU for further post-processing, as shown in Figure 5.1. As such, for workloads that are hard to build on top of GASPP, the correct way is to implement them by offloading them to the CPU. A limitation of this approach is that any subsequent processing that might be required also has to be carried out by the CPU, as the cost of transferring the data back to the GPU would be prohibitive.

Another limitation of the current GASPP implementation is its relatively high packet processing latency. Due to the batch processing nature of GPUs, GASPP may not be suitable for protocols with hard real-time per-packet processing constraints.

Chapter 6

Related Work

6.1 Software Packet Processing for IP Forwarding and Routing

Software routers promise to enable the fast deployment of new, sophisticated kinds of packet processing without the need to buy and deploy expensive new equipment. Packet-processing in software running on general-purpose platforms allow easy programmability, in contrast to high-end routers which rely on specialized and closed hardware and software. Recent work has demonstrated that software routers are capable for high-packet-rate environments, while running sophisticated packet-processing applications [50, 51, 53, 60, 98, 148].

6.1.1 Implementations on multi-core CPUs

Egi et al. [53] study the performance limitations when building software routers in multiple multi-core CPUs and high-speed system interconnects. Their system achieves forwarding rates of 7 million minimum-sized packets per second, while the bottleneck lies in the shared bus (i.e. the "front-side bus" or FSB) that connects the CPUs to the memory subsystem. Bolla et al. [39] also report that the forwarding performance does not scale to the number of CPU cores due to FSB clogging — FSB clogging happens due to cache coherency snoops in the multi-core architecture [141]. In line with the above works, Argyraki et al. [30] also identifies memory as the main system bottleneck on modern commodity hardware.

By taking advantage of the ellimination of FSB in modern CPU architectures [66], RouteBricks reports 2–3x performance improvement [51].

RouteBricks is a software router architecture that parallelizes router functionality both across multiple servers and across multiple CPU-cores within a single server [51]. In the case of minimal-packet forwarding, the small number of instructions per packet shows that the software architecture of Click [97] is efficient. In other words, a poor software architecture is not the problem; performance is limited by the lack of CPU cycles. Hence, software routers stand to benefit from the expectation that the number of cores will scale with Mooreb••s law. By carefully exploiting parallelism at every opportunity, the authors demonstrate a 35 Gbps parallel router prototype that scales linearly through the use of additional servers. In addition, the router is fully programmable using the familiar Click/Linux environment and is built entirely from off-the-shelf, general-purpose server hardware.

In contrast to RouteBricks—that demonstrates high-performance only for conventional simple workloads (like packet forwarding and IP routing)—Dobrescu et al. [50] show that a software router can achieve high performance for a range of sophisticated packet processing applications without losing programmability. The high-level goal of this work is to automatically parallelize a packet processing application, in order to maximize performance. To that end, the authors define the parallelization options as *pipelining* versus *cloning*. Then, they identify three key factors that determine which parallelization approach is desirable: inter-socket synchronization costs, cache contention for data-structures, and the "imbalance" in a data flow. Finally, they propose a strawman optimization framework that takes as input a profile of server resources and a data flow element's resource consumption and outputs an optimal mapping of elements to cores based on weighting the relative impact of the above factors. Although their concept is in very naive stage yet, the results so far indicate that, at least for shared-memory architectures like Nehalem, pure cloning achieves the highest throughput, except from, not so realistic, cases of high frequent memory accesses, contention-sensitive data structures and nearly perfectly balanced pipeline stages. Flowstream [57] also provides a new class of system architectures for building network flow processing platforms, where more advanced, higher-layer processing can be done. Unfortunately, the system is not yet implemented, hence there is no performance evaluation available yet.

Packet Scheduling and Load Balancing

The typical approach to scale network processing is to distribute packets, or rather flows of packets, across multiple cores in order to be processed in parallel. However, the traffic that arrives on a single network port of a router is inherently serial. As such, a mechanism is needed, in order to appropriately demultiplex the traffic between a serial link and a set of cores.

Static Load Balancing. Recent efforts point to modern server network interface cards as offering the required mux/demux capability through new hardware classiffication features [51, 53, 60, 91]. These NICs offer the option of classifying packets that arrive at a port into one of multiple hardware queues on the NIC and this set of queues can then be mapped to different cores; each core thus processes the subset of traffic that arrives at its assigned queue(s). Effectively, this NIC-level classiffication capability serves to parallelize the path from a NIC to the cores, and vice versa. Building on this observation, recent efforts leverage multi-queue NICs in their prototype systems [51, 60, 91] and show that enabling NIC-level classification signifficantly improves a server's packet processing capability (for instance, a 3.3x increase in forwarding throughput through NIC-level classiffication has been reported [51]).

In [91], the authors evaluate the latest generation of widely-used server NICs and experimentally compare its performance characteristics to that of an ideal "parallel" NIC and a "serial only" NIC. They conclude that even though commodity NICs do improve on serial-only NICs (with 3x higher throughput on typical workloads) they lag an ideal parallel NIC (achieving 30% lower throughput).

Dynamic Load-balancing. In [145], the authors present a distributed algorithm that can load-balance packet processing workloads on homogeneous many-core processors. The distributed run-time system manages processing resources dynamically and allows balanced distribution of task across the entire chip. The distributed task offloading algorithm is capable of making local offloading decisions in parallel and chip-level offloading decisions within $O(nlogn)$ time complexity.

6.1.2 GPU-assisted Implementations

In order to address the CPU bottleneck in current software routers, GPU acceleration has been also proposed [60, 98, 148].

In [98], the authors implemented a series of key router applications, including IP routing table lookup and pattern matching for network intrusion detection. To perform the routing table lookup procedure in the GPU, the authors extend a radix tree based LPM algorithm, which is taken from RouterBench [89]. The routing table is organized as a tree structure in which a node represents a given state in the search process and edges correspond to values of bits in the destination IP address. A matching process is actually a traversal of a given path in the tree structure. The route table is calculated and managed on CPU. Everytime the route table is constructed or updated, it will be transferred to the GPU memory. The parallel organization is then trivial: one thread will process the IP address of a single packet. The results prove that GPU can accelerate IP routing up to 6 times faster than a CPU.

PacketShader [60] is a high-performance software router framework for general packet processing with GPU acceleration. PacketShader present an optimized I/O software architecture that signio••cantly improves the packet-processing capability of a single server and offers forwarding rates of up to 10Gbps per core. In addition, it offloads core packet processing operations – such as IP table lookup and IPsec encryption – to GPUs and scale packet processing with massive parallelism. The results suggest that, GPUs bring significant performance improvement for both memory and compute-intensive applications.

Besides the parallelism presented in the previous works, Zhao et al. [148] propose an IP lookup scheme that could achieve O(1) time complexity for each IP lookup. This is achieved by translating IP addresses into memory addresses directly, using a large routing table on GPU memory. With proper design, there is the potential for signifficant improvement, e.g., 6x faster than trie-based implementation. In addition, route-update operations are also mapped to the GPU by leveraging graphics processing facilities, like Z-buffer and Stencil buffer.

6.2 Deep Packet Inspection

Deep Packet Inspection (DPI) is a core component for many systems plugged in the network, including application layer packet classifiers, content-aware firewalls, and network intrusion detection/prevention systems (NIDS/NIPS). Network components use DPI as an essential inspector, applied in different layers of the OSI model. Unlike the early beginnings of packet inspection, where it was applied in packet headers only (e.g., proxies and firewalls, etc.), nowadays, protocol complexity and obfuscation force us to inspect content in all encapsulated layers. For instance, Internet Service Providers (ISPs) have been recently relying on DPI systems, in order to identify and classify network traffic accurately. The use of DPI can result to better QoS, by identifying different types of content and route them through different quality and/or speed networks. Similary, the detection of harmful traffic and malware relies on the inspection of packets content. As a consequence, DPI has evolved into a fast-growing application area both in terms of technology and market size – it is estimated that the market for DPI within the U.S. Government alone will be worth more than $7 billion over the next five years [5, 14].

6.2.1 Network Intrusion Detection and Prevention Systems

Network Intrusion Detection Systems (NIDSs) serve a critical role in computer networks security. Fortunately, the requirements for more complex traffic inspection, as well as the constant increase in network speeds, have motivated numerous works for improving the performance of NIDSs.

A number of approaches have been proposed to address the problem of matching stateful signatures in high-speed networks, both *hardware* and *software* based. Hardware-based implementations offer a scalable method of inspecting packets in high-speed environments [32, 34, 44, 45, 84, 86, 93, 95, 127]. These systems usually consist of special-purpose hardware, such as FPGAs, CAMs, and ASICs, that is used to process network packets in parallel. Many FPGA-based NIDS architectures have been also implemented to accelerate pattern matching. For instance, Baker and Prasanna implement efficient pattern matching [34], while Attig and Lockwood implement a framework for packet header processing combined with payload content scanning [32]. Clark et al. [44] use network processors to pipeline the processing stages related to each hardware resource. Re-

cently, Meiners et al. [93] propose a custom regular expression matching approach based on TCAMs, that achieves a throughput of up to 18.6 Gbit/s.

Although the use of specialized hardware achieves high processing rates, most implementations require specialized programming, and are usually tied to the underlying device. As a consequence, they are very difficult to extend and program. Additionally, most of these approaches focus on the raw inspection of the network packets alone, without implementing crucial functionalities of modern NIDSs, such as protocol analysis and application-level parsing. In practice however, the performance of modern NIDSs depends on several operations, including packet capture and decoding, TCP stream reassembly, and application-level protocol analysis. A scalable parallel architecture should exploit parallelism for each operation individually, otherwise Amdahl's Law will fundamentally limit the performance that the hardware can provide [105].

Offloading Processing to the Network Interface

Many work has been made towards pushing much of the packet analysis and processing into the network interface. In [44], the authors use the multithreaded microengines of an Intel IXP1200-based platform to reassemble incoming TCP streams, that are then feed to an FPGA-based string content matcher. Bos and Huang also propose a system that is based on an IXP1200, and is capable of performing both stream reassembly and string content matching on the microengines [40]. SafeCard [47] is a network intrusion prevention system, built entirely on the network card, that is capable of reconstructing and scanning TCP streams at Gigabit rates, in order to detect polymorphic buffer-overflow attacks. Similarilly, LineSnort [117] parallelizes Snort using concurrency across TCP sessions and executes these parallel tasks on multiple low-frequency pipelined RISC processors embedded in future NICs.

Distributing Workload to Multiple Hosts

To cope with high traffic volumes, many approaches propose to distribute the load across multiple hosts, instead of using a single computer [106]. Kruegel and Valeur [80] propose a partitioning approach for NIDS using a cluster of PCs as individual sensors. A slicing mechanism divides the traffic into subsets, which are assigned to sensors in a way that each subset

contains all the necessary evidence to detect a specific attack without any need for communication between the detectors. The rules that specify the traffic splitting are formed by modelling the attacks. Spanids [116] utilizes a specially-designed FPGA-based switch, that consider flow information and the load of each server when redirecting network packets to the back-end servers. Xinidis et al. [146] present an active splitter architecture that provides early filtering to reduce the load of the back-end sensors. Vallentin et al. [135] present another NIDS cluster based on commodity PCs. Some front-end nodes are responsible to distribute the traffic across the back-end nodes of the cluster. Several traffic distribution schemes are discussed, that focus on minimizing the communication between the sensors and keeping them simple enough to be implemented effectively in the front-end nodes. For the first property, flow based schemes are considered more promising, while stateless distribution schemes are prefered for simplicity. Thus, hashing a flow identifier gives the right choice. In order to keep the computational costs low, an additive hash is proposed. Furthermore, 2-tuple hashing is used (instead of 5-tuple or 4-tuple) to decrease computations, and also to be portable across transport protocols that do not use port numbers.

Migrating to Multicore CPUs

The advent of multicore processors has allow the development of parallel NIDS, in a single-box. In Supra-linear Packet Processing [69], a single thread is responsible for packet gathering and dispatching, while many other threads are processing incoming flows in parallel. Thus, each processing thread process a specific flow in isolation. Unfortunately, a lot of time is spent on context switches between threads, most likely due to the high level of locking contention to the shared packet queue. To eliminate the excessive contention rates around packet queue access in the packet receive/processing architecture, flow-pinning can also be used (i.e. all packets of a flow are "pinned" to be processed by a specific thread) [70]. This approach requires slightly more data storage to keep the incoming packets to separate queues. However, it allows most of the threads to work independently, which is a key characteristic of a good multi-threaded algorithm.

Schuff et al. [118, 119] evaluate two different approaches for parallelizing the Snort NIDS: one conservative and one optimistic. The conserva-

tive scheme, called flow-concurrent parallelization, exploits concurrency by parallelizing rule-set processing on a flow-by-flow basis. A separate thread is responsible to receive the incoming packets and steer them to the processing thread that has been assigned to process the flow to which each packet belongs. Since each given flow is only processed by one thread at any given time, it is easily to maintain the dependencies required for proper stream reassembly and flow tracking. This scheme works well if there are enough independent flows, but provides no benefits if all packets are from the same flow. Even though the latter case is a rare situation for a high-bandwidth edge NIDS, still it does represent a limitation of this scheme. The alternative parallelization is an optimistic variant on flow concurrency. It is based on flow-concurrent parallelization, which has also the ability to dynamically reassign a flow to a different thread even while earlier packets of the flow are still being processed, potentially exploiting parallelism even with just one flow. This optimistic version relies on two key observations. First, TCP stream reassembly will still take place even if a stream is broken at some arbitrary point; reassembly is triggered by various flush conditions, one of which is a timeout. It is also easy to force additional flushes if needed for correctness. Consequently, any unprocessed earlier packets will still go through stream reassembly at their thread even though later packets are being reassembled and processed in another thread. Second, most packets do not match rules that use flowbits tracking, so enforcing ordering across all packets in a flow just to deal with a few problematic rules may be too restrictive. To precisely deal with the rules that do use flowbits, the optimistic system stalls processing in any packet that sets or checks flowbits unless it is the oldest packet in its flow. This condition is checked by adding per-flow reorder buffers. This system is optimistic in the sense that it reassigns threads under the assumption that the actual use of flowbits is uncommon, but is still conservative in maintaining correct ruleset processing without requiring rollbacks and redundant processing.

GPU-based Implementations

Recently, graphics processors have been used to boost computationally intensive tasks of intrusion detection systems, such as pattern matching and regular expression operations [42, 63, 124, 136, 138]. Pattern matching is commonly performed by converting patterns into finite state automata.

Two kinds of automata are typically used in the literature, deterministic (DFA) and non-deterministic (NFA). NFA traversal requires, by definition, non-deterministic choices that are hard to emulate on actual, deterministic processors; on the other hand DFAs, while fast to execute, are much more inefficient in terms of space, requiring very large amounts of memory to store certain peculiar patterns (an effect known as *state space explosion*) [35]. In contrast, software-based NFA implementations suffer from a higher per-byte traversal cost when compared with DFAs: intuitively, this is because multiple NFA states can be active at any given step, while only a single one must be considered when processing DFAs.

In order to speed-up the pattern matching computation on the GPU, Tumeo et al. [134], redesigned the packet reading process, such that each thread is fetching four bytes at a time, instead of one. Since the input symbols belong to the ASCII alphabet, they are represented with 8 bits. However, the minimum size for every device memory transaction is 32 bytes. Thus, by reading the input stream one byte at a time, the overall memory throughput may be reduced by a factor of up to 32. To overcome this limitation, the authors use the `char4` built-in data type (4-bytes size), to read the content of each packet. Unfortunately, they do not experiment with larger word accesses (e.g. 16-bytes or 32-bytes).

Gnort [136, 138] was the first attempt that sufficiently utilized the graphics processor for pattern matching and regular expression operations. For performance issues, Gnort utilize a DFA for pattern matching, at the cost of high memory space utilization. Other approaches adopt non-deterministic automata, allowing the compilation of very large and complex rule sets that are otherwise hard to treat [42]. Furthermore, Gnort takes advantage of DMA and the asynchronous execution of modern GPUs, to overcome the data transfer costs from the host memory to the device memory, and vice versa. Therefore, Gnort impose concurrency between the operations handled by the CPU and the GPU. Many other approaches follow the above scheme [63, 124], without significant differences in the architecture and the performance benefits. Finally, Kargus [71] incorporates several system optimizations to the architecture of MIDeA [139], such as function batching. It also decreases the packet copies by mapping the packet buffers of the network interface to the user-space.

6.2.2 Traffic Classification

Traffic classification is an important QoS (Quality of Service) component at central network traffic ingress points, as well as at end-host computers. Unfortunately, application protocols are becoming more-and-more complex, using random ports, and they are usually employ payload encryption and other obfuscation techniques. In particular, several network applications, such as peer-to-peer and multimedia applications, have started to masquerade and obfuscate their traffic in order to avoid detection. Many classification techniques are based on traffic and flow statistics, like packet and flow sizes, interarrival times, and aggregated statistics [65, 74, 75]. Recently, DPI-based programs are gaining more publicity in both academic studies [83, 58, 59, 143] and industrial products [3, 13]. Unfortunately, DPI requires more complex analysis and reduces significantly the packet processing throughput. In high speed networks, it will also lead to decreased performance and lower accuracy, when the application is overloaded.

The majority of the approaches are *flow based classification* techniques that try to classify flows instead of packets, according to the application that creates them. The flow based classification techniques have a significant performance benefit when only the first N packets of a flow are inspected (or the first B bytes respectively). Indeed, after inspecting the first N packets of a flow with no success, it is useless to continue searching for applications signatures, since the most possible case is that there is no signature for this type of traffic, and further processing will be overhead to the system; it is preferably to mark this flow as 'unknown traffic'. Similarly, upon a packet matches an application signature and is identified that belongs to a known application, its flow will be also considered that belongs to this application. Thus, the next packets of this flow will not be inspected, but only used for accounting the application's bandwidth usage. Therefore, flow based traffic classification can significantly reduce the per-packet processing cost.

Guo et al. [59] propose a highly scalable parallelized L7-filter system architecture with affinity-based scheduling in a multicore server. Similar to Receive Side Scaling (RSS) in the NIC, the authors develop a model to explore the connection level parallelism in L7-filter and propose a cache affinity scheduler to optimize system scalability. Performance results show that the optimized L7-filter has superior scalability over the naive mul-

tithreaded version, improving system performance by about 50% when all the cores are deployed. In addition, their model also ameliorates the performance degradation in a virtualized environment, due to the direct mapping mechanism.

Recently, Wang et al. [143] present a parallelization of a complete Layer-2 to Layer-7 (L2-L7) network processing system, including a TCP/IP stack based on `libnids` (L2-L4), and a port-independent protocol identification engine based on deep packet inspection (L7+). The experience gains suggest that (i) fine-grained pipelining can be a good software solution for parallelizing network applications on multicore architectures if connection-affinity and lock-free design principles are used; (ii) a delicate partitioning scheme is required to map pipelined structures onto specific multicore architecture. In particular, the authors propose an automatic parallelization approach that can work if domain knowledge is considered in the parallelizing process. The performance evaluation shows that an Intel 2 Quad processor achieve 6 Gbps processing speed for large packet sizes, and 2 Gbps speed for smaller packets. Guo et al. [58] also implement a multi-threaded L7-filter on a Sun Niagara 2-based eigh-core workstation, which achieves less than 2 Gbps processing speed.

6.2.3 Load Balancing

The key issue in multicore scheduling is to balance the workload across available processing resources. Previously proposed works [52, 55, 130] mainly achieve balanced workload based on real time thread migration. The advantage of research in this domain is that the locality of running threads can be adjusted to shorten the blocking delay incurred by uneven workload distribution based on real time statistics. The downside of migration-based load balance algorithms is cache thrashing, i.e., old cache data might be replaced by new data for the recently migrated thread.

In order to dispatch network packets among CPU cores, a load-balancing approach is needed. Due to the nature of fine-grained parallelism, static load-balancing is preferred by trading the imbalance of workload with the much less runtime overhead [143]. Many approaches use a packet-classifying hash function (*symetric* or *asymmetric*) to dispatch packets among CPU cores. Even though asymmetric hash function distributes packets more evenly among CPU cores, it cannot guarantee connection-affinity. Therefore, the majority of approaches use a symmetric hash function to

distribute packets among CPU cores, since on multicore platforms cache locality is more important than load balance [100].

Kencl and Le Boudec [77] present a feedback system, implemented in a network processor simulator, for the traditional Highest Random Weight (HRW) hash [131]. The original HRW hash guarantees the connection locality but only balances the workload over the number of different connections. In order to provide run-time load balancing, the hardware counters were polled at a per-packet basis.

In [58], the authors propose an adaptive hash-based multilayer scheduler, based on HRW, for a Sun Niagara 2 server, instead of a network processor. The original single-layer HRW is enhanced into a hierarchical "hash tree" scheduler, in order to balance the connection workload in accordance with the hierarchical processors architecture. The scheduler maintains connection locality and adaptively adjusts the scheduling to balance the real time workload. In addition, a low overhead feedback metric is chosen, i.e. the length of the runqueue, to provide better load balance rather than to poll values from the hardware counter, which is infeasible to do at a per-packet basis in a high speed network. The scheduler is shown to increase the system throuhgput by 59.2% compared to the previously proposed connection locality optimization.

6.3 Network Packet Processing Frameworks

Spalink et al. [128] design and implement a software-based router that uses the IXP1200 network processor. The network processor handles simple forwarding, while more complex operations are offloaded to the host processor. SwitchBlade [29] provides a model that allows packet processing modules to be swapped in and out of reconfigurable hardware without the need to resynthesize the hardware. Orphal [96] and ServerSwitch [87] provide a common API for proprietary switching hardware, and leverages the programmability of commodity Ethernet switching chips for packet forwarding. ServerSwitch also leverages the resources of the server CPU to provide extra programmability. Chimpp [113] and the G framework [99] also use modules for packet processing on the NetFPGA platform. The main disadvantage of these systems is that their cost is usually very high, and more importantly, most implementations require specialized programming and are usually tied to the underlying architecture. Finally, similar to

GASPP, Snap [129] is a programmable network traffic processing framework that manage to simplify the development of GPU-accelerated network traffic processing applications.

Chapter 7

Future Work and Conclusion

Modern graphics processors have the potential to provide significant throughput improvements to stateful packet processing applications when offloading their corresponding operations. This dissertation demonstrates that achieving this potential involves careful data movements and software techniques optimized for the different execution model, constraints, and memory hierarchies contained in modern GPU architectures. In this section we summarize the major contributions of our work, discuss the future direction left by this dissertation, and conclude with a brief summary.

7.1 Summary of Contributions

The main contributions of this dissertation are as follows:

- The implementation of a high-performance single-thread stateful network intrusion detection system, that utilizes the ubiquitous GPU to offload both string searching and regular expression matching operations. Our implementation extends the Snort IDS [111], which is the most popular and widely used open-source NIDS. We have implemented novel packet transformation techniques, that group and transpose the network packets in order to tackle load imbalances across SIMT threads and coalesce memory accesses on the GPU. We also characterize extensively the performance of different types of GPU memory hierarchies for signature matching on network packets, and identify the setup that performs best.
- We introduce a novel multi-parallel architecture for high-performance processing and stateful analysis of network traffic. Our architecture is based on inexpensive, off-the-shelf, general-purpose hard-

ware, and combines multi-queue NICs, multi-core CPUs, and multiple GPUs. We present our prototype implementation based on Snort [111], demonstrating that the proposed model is practical, scales well with the number of processing units, and can be adopted by existing systems.

- We present a novel GPU-based framework, namely GASPP, for high-performance passive or inline network traffic processing, which eases the development of GPU-assisted applications that process data at multiple layers of the protocol stack. Our framework employs, among others, (i) the first (to the best of our knowledge) purely GPU-based implementation of flow state management and TCP stream reconstruction, (ii) novel mechanisms for tackling control flow irregularities across SIMT threads, allowing efficient execution when consolidating multiple divergent network applications on the same device, and (iii) a zero-copy mechanism that avoids redundant memory copies between the network interface and the GPU, increasing significantly the throughput of cross-device data transfers.

7.2 Future Work

Broadly categorized, there are several directions of future work that this dissertation leaves open.

More Workload Studies. This dissertation studied the GPU-assisted stateful packet processing acceleration with an emphasis primarily on network intrusion detection and prevention systems. Performing an in-depth study on other network monitoring applications—such as traffic classification, content-aware firewalls, spam filtering, and other network traffic analysis systems—will always remain an area of future work.

Integrated Graphics Processors. Integrated graphics processors utilize a portion of the host memory rather than dedicated graphics memory. They can be either integrated into the motherboard or integrated with the CPU die. Recent integrated graphics processors, such as AMD Accelerated Processing Unit [73] and Intel HD Graphics [54] have improved performance, although they are still far less capable than current generation dedicated GPUs. Still, the advantage of this approach is that the integrated GPU and CPU share the same physical memory address space, which allows in-place data processing. This results to fewer data transfers, lower pro-

cessing latency and lower power consumption, all of which are quite appealing for typical packet processing workloads.

Energy Consumption Characterization. Besides the great performance potential of modern GPUs, consumptions in terms of energy may be very high [61]. As such, there may be cases where it is not beneficial to utilize the GPU (e.g. when traffic rates are low and hence can be handled by processors with lower consumption characteristics, such as the CPU). To make matters worse, current GPUs do not have performance states, hence power adjustment is currently not possible. Motivated by this deficiency, it is very interesting to explore the practicability of building an energy-efficient network packet processing system.

Heterogeneous Systems. Heterogeneous, multi-computational-device systems typically offer system designers different optimization opportunities that offer inherent trade-offs between energy consumption and various performance metrics. The challenge to fully tap the heterogeneous systems, is to effectively map computations to processing devices, and do so as automated as possible. Recent work has attempted to solve this problem by developing load-balancing frameworks that automatically partition the workload across the devices [78, 88, 41]. These approaches either assume that all devices provide equal performance [78] or require a series of small execution trials to determine their relative performance [88, 41]. The disadvantage of such approaches is that they have been designed for applications that take as input constant streaming data, and as a consequence, they are slow to adapt when the input data stream varies. This makes them extremely difficult to apply to network infrastructure, where traffic variability [90, 36] and overloads [46] significantly affect the utilization and performance of network applications.

7.3 Conclusion

This dissertation proposed stateful packet processing using modern graphics processors, and analyzed the tradeoffs and software techniques needed to improve its performance. Our evaluation demonstrated that GPUs can accelerate stateful packet processing applications but requires revisiting several layers of the execution flow to account for the different constraints and properties of the GPU execution model. At the application layer, we developed a signature-based network intrusion detection system that used

multiple GPUs paired with multiple CPU-cores and multi-queue network interfaces, identifying a combination of techniques to effectively parallelize its architecture at multiple layers.

We then explored the design of a GPU-based stateful packet processing framework, identifying and eliminating bottlenecks in the existing network I/O stack, as well as redundant work found in monolithic GPU-assisted applications. We generalized the design, identifying a modular mechanism for writing GPU-based packet processing applications. Last, we demonstrated novel mechanisms for tackling control flow irregularities across SIMT threads, allowing efficient execution when consolidating multiple divergent network applications on the same device.

In summary, this dissertation demonstrated that novel, efficient software techniques are needed to take advantage of modern graphics processors, in order to improve the throughput of stateful packet processing applications. We hope the design principles and practices provided by this work will apply when developing flexible and faster network equipment based on commodity hardware.

Bibliography

[1] 7950 Extensible Routing System. http://www.alcatel-lucent.com/products/7950-extensible-routing-system-0.

[2] Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net/.

[3] Cisco IOS Netflow. http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[4] Cisco taps processor array architecture for NPU. http://www.eetimes.com/General/DisplayPrintViewContent?contentItemId=4049718.

[5] Deep packet inspection soon to be $1.5 billion business. http://arstechnica.com/tech-policy/news/2010/06/deep-packet-inspection-soon-to-be-15-billion-business.ars.

[6] Endace Security Manager. http://www.endace.com/endace-security-manager2.html.

[7] Global Internet usage. http://en.wikipedia.org/wiki/Global_Internet_usage.

[8] High performance GPU radix sorting in CUDA. http://code.google.com/p/back40computing/wiki/RadixSorting.

[9] Huawei Launches NetEngine80 Core Router At Networld+Interop 2001 Exhibition in US. http://www.huawei.com/en/about-huawei/newsroom/product_launch/hw-090830-productlaunch.htm.

[10] Intel 82599EB 10 Gigabit Ethernet Controller. http://ark.intel.com/Product.aspx?id=32207.

[11] Intel Advanced Encryption Standard Instructions (AES-NI). http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/.

[12] Isc domain survey — internet systems consortium. https://www.isc.org/services/survey/.

[13] MSCG Hierarchical DPI Solution. http://www.huawei.com/products/datacomm/catalog.do?id=1219.

[14] Network Processor: Deep Packet Inspection Market Estimated in Billions Over Next Five Years. http://www.tmcnet.com/channels/network-processor/articles/136968-network-processor-deep-packet-inspection-market-estimated-billions.htm.

[15] Network Routers - Mobile Routers - Cisco Routers - Cisco. http://www.cisco.com/c/en/us/products/routers/index.html.

[16] Network Routers - Secure Network Router Solution - Juniper Networks. http://www.juniper.net/us/en/products-services/routing/.

[17] Nouveau: Accelerated Open Source driver for nVidia cards. http://nouveau.freedesktop.org/.

[18] NSS Ipoque DPI Deep Packet Inspection Bandwidth Management. https://www.nss.gr/en/onsite-products/networking/ipoque.html.

[19] PCRE: Perl Compatible Regular Expressions. http://www.pcre.org.

[20] pscnv - PathScale NVIDIA graphics driver. https://github.com/pathscale/pscnv.

[21] Receive side scaling on Intel Network Adapters. http://www.intel.com/support/network/adapter/pro100/sb/cs-027574.htm.

[22] Snort Network Intrusion Prevention and Detection System (IDS/IPS). http://www.snort.org.

[23] Tcpreplay. http://tcpreplay.synfin.net/.

[24] The Akamai State of the Internet Report. http://www.akamai.com/stateoftheinternet/.

[25] The CAIDA Anonymized Internet Traces 2011 Dataset. http://www.caida.org/data/passive/passive_2011_dataset.xml.

[26] Testing Intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory. *ACM Trans. Inf. Syst. Secur.*, 3(4):262–294, 2000.

[27] 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.

[28] AMD. Multi-Core Processing with AMD. http://www.amd.com/us/products/technologies/multi-core-processing/pages/multi-core-processing.aspx.

[29] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM, 2010.

[30] K. Argyraki, S. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedevschi, and S. Ratnasamy. Can software routers scale? In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, PRESTO, 2008.

[31] ATI. OpenCL: The GPU Open Compute Language. http://www.fireprographics.com/ws/tech/opencl/index.asp.

[32] 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, 2005.

[33] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. 21:613–641, August 1978.

[34] Z. K. Baker and V. K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of the 12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA, 2004.

[35] M. Becchi and P. Crowley. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS, 2008.

[36] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. *SIGCOMM Computer Communication Review*, 40(1), Jan. 2010.

[37] E. Berk and C. Ananian. Jlex: A lexical analyzer generator for java. Online: www.cs.princeton.edu/ appel/modern/java/JLex/.

[38] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theor. Comput. Sci.*, 48(1):117–126, 1986.

[39] R. Bolla and R. Bruschi. PC-based Software Routers: High Performance and Application Service Support. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, PRESTO, 2008.

[40] H. Bos and K. Huang. Towards Software-based Signature Detection for Intrusion Prevention on the Network Card. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, RAID, September 2005.

[41] M. Boyer, K. Skadron, S. Che, and N. Jayasena. Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF, 2013.

[42] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review*, 40:20–26.

[43] X. Chen, K. Ge, Z. Chen, and J. Li. AC-Suffix-Tree: Buffer Free String Matching on Out-of-Sequence Packets. In *Proceedings of the 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS, 2011.

[44] C. R. Clark, W. Lee, D. E. Schimmel, D. Contis, M. Koné, and A. Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In *Proceedings of the 3rd Workshop on Network Processors and Applications*, NP3, 2005.

[45] C. R. Clark and D. E. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, FPL, 2003.

[46] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM, 2003.

[47] 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 the 9th International Symposium on Recent Advances in Intrusion Detection*, RAID, September 2006.

[48] L. Deri. Improving Passive Packet Capture: Beyond Device Polling. In *Proceedings of 4th International System Administration and Network Engineering Conference*, SANE, 2004.

[49] S. Dharmapurikar and V. Paxson. Robust TCP stream reassembly in the presence of adversaries. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, USENIX Security, 2005.

[50] M. Dobrescu, K. Argyraki, G. Iannaccone, M. Manesh, and S. Ratnasamy. Controlling parallelism in a multicore software router. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO, 2010.

[51] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, SOSP, 2009.

[52] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS, 1996.

[53] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high performance virtual routers on commodity hardware. In *Proceedings of the ACM International Conference on emerging Networking Experiments and Technologies Conference*, CoNEXT, 2008.

[54] Eliseo Hernandez. Accelerate Performance Using OpenCL with Intel HD Graphics. `http://software.intel.com/en-us/articles/accelerate-performance-using-opencl-with-intel-hd-graphics`.

[55] M. S. A. Fedorova. Cache-fair thread scheduling for multicore processors. Technical report, Division of Engineering and Applied Sciences, Harvard University, 10 2006.

[56] F. Fusco and L. Deri. High Speed Network Traffic Analysis with Commodity Multi-core Systems. In *Proceedings of the 10th Internet Measurement Conference*, IMC, 2010.

[57] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy. Flow processing and the rise of commodity network hardware. *ACM SIGCOMM Computer Communication Review*, 39:20–26, March 2009.

[58] D. Guo, G. Liao, L. N. Bhuyan, and B. Liu. An adaptive hash-based multilayer scheduler for L7-filter on a highly threaded hierarchical multi-core server. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS, 2009.

[59] D. Guo, G. Liao, L. N. Bhuyan, B. Liu, and J. J. Ding. A Scalable Multithreaded L7-filter Design for Multi-core Servers. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS, 2008.

[60] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM, August 2010.

[61] S. Hong and H. Kim. An integrated GPU power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA, 2010.

[62] J. E. Hopcroft and J. D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[63] N.-F. Huang, H.-W. Hung, S.-H. Lai, Y.-M. Chu, and W.-Y. Tsai. A GPU-Based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems. In *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications - Workshops (AINAW)*, 2008.

[64] IBM. Cell Broadband Engine. `https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine`.

[65] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese. Network monitoring using traffic dispersion graphs (tdgs). In *Proceedings of the 7th Internet Measurement Conference (IMC '07)*, 2007.

[66] Intel. First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem). `http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf`.

[67] Intel. Intel Multi-Core Processors Making the Move to Quad-Core and Beyond. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[68] Intel. Intel Xeon Phi Product Family. `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[69] Intel Corporation. Supra-linear Packet Processing Performance with Intel Multi-core Processors, 2006.

[70] Intel Corporation. Removing System Bottlenecks in Multi-threaded Applications, 2008.

[71] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a highly-scalable software-based intrusion detection system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, 2012.

[72] K. Jang, S. Han, S. Han, K. Park, and S. Moon. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, March 2011.

[73] Jon Stokes. AMD reveals Fusion CPU+GPU, to challenge Intel in laptops. http://arstechnica.com/business/2010/02/amd-reveals-fusion-cpugpu-to-challege-intel-in-laptops/.

[74] S. Kandula, R. Chandra, and D. Katabi. What's going on?: learning communication rules in edge networks. *ACM SIGCOMM Computer Communication Review*, 38:87–98, August 2008.

[75] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM, 2005.

[76] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *Proceedings of the 2012 USENIX Annual Technical conference*, USENIX ATC, 2012.

[77] L. Kencl and J.-Y. Le Boudec. Adaptive load sharing for network processors. *IEEE/ACM Trans. Netw.*, 16:293–306, April 2008.

[78] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, 2011.

[79] M. E. Kounavis, X. Kang, K. Grewal, M. Eszenyi, S. Gueron, and D. Durham. Encrypting the Internet. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM, pages 135–146, 2010.

[80] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy*, S&P 2002, May 2002.

[81] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS, 2007.

[82] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In *Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2006.

[83] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS, 2006.

[84] J. Lee, S. H. Hwang, N. Park, S.-W. Lee, S. Jun, and Y. S. Kim. A High Performance NIDS Using FPGA-based Regular Expression Matching. In *Proceedings of the 22nd ACM Symposium on Applied Computing*, SAC, 2007.

[85] B. H. Leitao. Tuning 10Gb Network Cards on Linux. In *Proceedings of the 2009 Linux Symposium*, July 2009.

[86] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao. A Fast String-matching Algorithm for Network Processor-based Intrusion Detection System. *ACM Transactions on Embedded Computing Systems*, 3(3):614–633, 2004.

[87] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: a programmable and high performance platform for data center networks. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI, 2011.

[88] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2009.

[89] Y. Luo, L. N. Bhuyan, and X. Chen. Shared Memory Multiprocessor Architectures for Software IP Routers. *IEEE Transactions on Parallel and Distributed Systems*, 14:1240–1249, 2003.

[90] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On dominant characteristics of residential broadband internet traffic. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC, 2009.

[91] M. Manesh, K. Argyraki, M. Dobrescu, N. Egi, K. Fall, G. Iannaccone, E. Kohler, and S. Ratnasamy. Evaluating the Suitability of Server Network Cards for Software Routers. In *Proceedings of the ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO, November 2010.

[92] S. McCanne, C. Leres, and V. Jacobson. libpcap. Lawrence Berkeley Laboratory, Berkeley, CA. (software available from http://www.tcpdump.org/).

[93] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast Regular Expression Matching Using Small TCAMs for Network Intrusion Detection and Prevention Systems. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[94] Microsoft Corporation. Scalable Networking: Eliminating the Receive Processing Bottleneck - Introducing RSS, 2005.

[95] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS, 2007.

[96] J. C. Mogul, P. Yalag, J. Tourrilhes, R. Mcgeer, S. Banerjee, T. Connors, and P. Sharma. API Design Challenges for Open Router Platforms on Proprietary Hardware. In *Proceedings of the ACM Workshop on Hot Topics in Networks*, HotNets-VII, 2008.

[97] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, SOSP, 1999.

[98] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang. IP Routing Processing with Graphic Processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE, 2010.

[99] C. E. Neely, G. J. Brebner, and W. Shang. ShapeUp: A High-Level Design Approach to Simplify Module Interconnection on FPGAs. In *Proceedings of the 18th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM, 2010.

[100] T. Nelms and M. Ahamad. Packet scheduling for deep packet inspection on multi-core architectures. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, pages 21:1–21:11, 2010.

[101] M. Norton. Optimizing Pattern Matching for Intrusion Detection, July 2004.

[102] NVIDIA. CUDA Programming Guide, v5.0. `http://http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[103] NVIDIA. Parallel Programming and Computing Platform — CUDA — NVIDIA. `http://www.nvidia.com/object/cuda_home_new.html`.

[104] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrG'Oger, A. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[105] V. Paxson, K. Asanović, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking Hardware Support for Network Analysis and Intrusion Prevention. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Security*, HotSec, 2006.

[106] V. Paxson, R. Sommer, and N. Weaver. An Architecture for Exploiting Multi-core Processors to Parallelize Network Intrusion Prevention. In *Proceedings of the 30th IEEE Sarnoff Symposium*, May 2007.

[107] Pierre Terdiman. Radix Sort Revisited.

[108] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, Apr. 1959.

[109] Rafal Wojtczuk. Libnids - NIDS E-component library. `http://libnids.sourceforge.net/`.

[110] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX conference on USENIX annual technical conference*, USENIX ATC, 2012.

[111] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proceedings of the 1999 USENIX Large Installation System Administration Conference*, LISA, 1999.

[112] S. Rubin, S. Jha, and B. Miller. Protomatching Network Traffic for High Throughput Network Intrusion Detection. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS, 2006.

[113] E. Rubow, R. McGeer, J. C. Mogul, and A. Vahdat. Chimpp: a click-based programming and simulation environment for reconfigurable networking hardware. In *Proceedings of the 6th ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS, 2010.

[114] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. 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*, PPoPP, 2008.

[115] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.

[116] L. Schaelicke, K. Wheeler, and C. Freeland. SPANIDS: A Scalable Network Intrusion Detection Loadbalancer. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF, 2005.

[117] D. Schuff and V. Pai. Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface. In *Proceedings of the 21th IEEE International Parallel & Distributed Processing Symposium*, IPDPS, march 2007.

[118] D. L. Schuff, Y. R. Choe, and V. S. Pai. Conservative vs. Optimistic Parallelization of Stateful Network Intrusion Detection. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2007.

[119] D. L. Schuff, Y. R. Choe, and V. S. Pai. Conservative vs. optimistic parallelization of stateful network intrusion detection. In *Proceedings of the 2008 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 32–43, 2008.

[120] Sean Baxter. MGPU Sort Library.

[121] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI, 2012.

[122] R. Smith, C. Estan, and S. Jha. Backtracking Algorithmic Complexity Attacks against a NIDS. In *Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, ACSAC, 2006.

[123] R. Smith, C. Estan, and S. Jha. XFA: Faster Signature Matching with Extended Automata. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, IEEE S&P, 2008.

[124] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for Network Packet Signature Matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS, 2009.

[125] R. Sommer and V. Paxson. Enhancing Byte-level Network Intrusion Detection Signatures with Context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS, 2003.

[126] Sourcefire. Sourcefire 3D System. `http://www.sourcefire.com/security-technologies/cyber-security-products/3d-system`.

[127] I. Sourdis and D. Pnevmatikatos. Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. In *Proceedings of the 12th Annual IEEE Symposium on Field Programmable Custom Computing Machines*, FCCM, 2004.

[128] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, SOSP, 2001.

[129] W. Sun and R. Ricci. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In *Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS, 2013.

[130] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2007 EuroSys Conference*, EuroSys, 2007.

[131] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6:1–14, February 1998.

[132] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[133] A. Tumeo, S. Secchi, and O. Villa. Experiences with string matching on the fermi architecture. In *Proceedings of the 24th International Conference on Architecture of Computing Systems*, ARCS, 2011.

[134] A. Tumeo, O. Villa, and D. Sciuto. Efficient pattern matching on GPUs for intrusion detection systems. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF, 2010.

[135] 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, 2007.

[136] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID, 2008.

[137] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC, 2014.

[138] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID, 2009.

[139] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, CCS, 2011.

[140] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC, 2011.

[141] B. Veal and A. Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS, 2007.

[142] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and Robust TCP Stream Normalization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, IEEE S&P, 2008.

[143] J. Wang, H. Cheng, B. Hua, and X. Tang. Practice of parallelizing network applications on multi-core architectures. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS, 2009.

[144] E. W.Dijkstra. A review of the 1977 turing award lecture by john backus. http://userweb.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD692.html.

[145] Q. Wu, D. J. Mampilly, and T. Wolf. Distributed runtime load-balancing for software routers on homogeneous many-core processors. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO, 2010.

[146] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, and E. P. Markatos. An Active Splitter Architecture for Intrusion Detection and Prevention. *IEEE Transactions on Dependable and Secure Computing*, 3:31–44, January 2006.

[147] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the 2nd ACM/IEEE symposium on Architecture for Networking and Communications Systems*, ANCS, 2006.

[148] J. Zhao, X. Zhang, X. Wang, and X. Xue. Achieving O(1) IP Lookup on GPU-based Software Routers. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM, pages 429–430, 2010.