GPU Accelerated Protocol Analysis for Large and Long-term Traffic Traces

Submitted in fulfilment
of the requirements of the degree of

Doctor of Philosophy

of Rhodes University

Alastair Timothy Nottingham

Grahamstown, South Africa
March 2016

Abstract

This thesis describes the design and implementation of GPF+, a complete general packet classification system developed using Nvidia CUDA for Compute Capability 3.5+ GPUs. This system was developed with the aim of accelerating the analysis of arbitrary network protocols within network traffic traces using inexpensive, massively parallel commodity hardware. GPF+ and its supporting components are specifically intended to support the processing of large, long-term network packet traces such as those produced by network telescopes, which are currently difficult and time consuming to analyse.

The GPF+ classifier is based on prior research in the field, which produced a prototype classifier called GPF, targeted at Compute Capability 1.3 GPUs. GPF+ greatly extends the GPF model, improving runtime flexibility and scalability, whilst maintaining high execution efficiency. GPF+ incorporates a compact, lightweight register-based state machine that supports massively-parallel, multi-match filter predicate evaluation, as well as efficient arbitrary field extraction. GPF+ tracks packet composition during execution, and adjusts processing at runtime to avoid redundant memory transactions and unnecessary computation through warp-voting. GPF+ additionally incorporates a 128-bit in-thread cache, accelerated through register shuffling, to accelerate access to packet data in slow GPU global memory. GPF+ uses a high-level DSL to simplify protocol and filter creation, whilst better facilitating protocol reuse. The system is supported by a pipeline of multi-threaded high-performance host components, which communicate asynchronously through 0MQ messaging middleware to buffer, index, and dispatch packet data on the host system.

The system was evaluated using high-end Kepler (Nvidia GTX Titan) and entry-level Maxwell (Nvidia GTX 750) GPUs. The results of this evaluation showed high system performance, limited only by device side IO (600MBps) in all tests. GPF+ maintained high occupancy and device utilisation in all tests, without significant serialisation, and showed improved scaling to more complex filter sets. Results were used to visualise captures of up to 160 GB in seconds, and to extract and pre-filter captures small enough to be easily analysed in applications such as Wireshark.

Acknowledgements

This research would not have been possible without the support provided by my friends and family. I would like to thank my supervisor, Prof. Barry Irwin, who provided invaluable assistance and guidance for over half a decade in shaping and developing this project through its various iterations. I would like to thank my family, who have shown tremendous support, both emotionally and financially, to help me complete this research. In particular, I would like to thank my wife, Dr Spalding Lewis, for her monumental effort in editing this thesis and supporting me throughout the development and writing process. I would also like to thank and my father, Jeremy Nottingham, who provided significant financial support and invaluable proofreading as the document was nearing completion. I would additionally like to thank Caro Watkins, as well as the staff of the Hamilton Building, for all of their help during my many, many years in the department. Finally, I would like to thank Joey Van Vuuren and the Defence Peace Safety and Security (DPSS) unit of the Council for Scientific and Industrial Research (CSIR), and Armscor, for their support, and for generously funding this research.

Contents

List of Figures

List of Tables

List of Code Listings

Glossary

0MQ	Zero Message Queue
ANTLR	ANother Tool For Language Recognition
API	Application Programming Interface
ARP	Address Resolution Protocol
BI	Bitmap Index
BPF	BSD Packet Filter
BSD	Berkeley Software Distribution
CFG	Control Flow Graph
CISC	Complex Instruction Set Computing
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DNS	Domain Name System
DPF	Dynamic Packet Filter
DSL	Domain Specific Language
EBNF	Extended Bachus-Naur Form
FDDI	Fiber Distributed Data Interface

FFPF	Fairly Fast Packet Filter
FPGA	Field Programmable Gate Array
FTP	File Transfer Protocol
GLSL	OpenGL Shading Language
GPF	GPU Packet Filter
GPGPU	General Processing on Graphics Processing Units
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDD	Hard Disk Drive
HTTP	Hyper-Text Transport Protocol
I/O	Input/Output
IBR	Internet Background Information
ILP	Instruction Level Parallelism
IP	Internet Protocol
IPv6	Internet Protocol version 6
JIT	Just-In-Time
LINQ	Language-Integrated Query
MPF	Mach Packet Filter
MTU	Maximum Transmission Unit
NPF	Netgroup Packet Filter
NVCC	The Nvidia CUDA Compiler
OpenCL	Open Compute Language
OpenGL	Open Graphics Library
OpenTK	Open ToolKit

OSI	Open Systems Interconnect
PCIe	PCI (Peripheral Component Interconnect) Express
RISC	Reduced Instruction Set Computing
SDN	Software Defined Networking
SIMD	Single Instruction Multiple Data
SSD	Solid State Drive
SSH	Secure Shell
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol
UI	User Interface
VLAN	Virtual Local Area Network
xPF	Extended Packet Filter

Part I

Introduction

1

Introduction

THIS thesis describes research conducted towards the construction of a fast GPGPU (General Processing on Graphics Processing Units) packet classification and filtering system. The system is intended for the purpose of providing high-speed analysis of arbitrary network protocol headers in large, long-term packet traces. Packet traces are simple static recordings of all traffic intercepted at a network interface, covering intervals ranging from seconds to years. While long-term high-density captures provide a rich source of information for researchers, network administrators and security professionals, large traces are both difficult and tedious to mine, and are thus used sparingly as information sources.

This chapter introduces the conducted research, elaborating on the problems the proposed solution seeks to alleviate, and the methods by which it was derived. The chapter is broken down as follows:

- Section 1.1 introduces the problem domain, and provides a brief overview of both packet classification and GPGPU processing.
- Section 1.2 succinctly describes the problem statement and how this research aims to address it.

- Section 1.3 provides an overview of the research conducted, describing the research scope, methodology and goals.
- Section 1.4 provides an overview of the chapter structure of this thesis.

1.1 Research Context

Packet captures (also known as traces) are important resources in network and security related development, administration and research, as they provide a relatively complete and static record of all network communications arriving at and sent from a network interface over a particular interval. Current mechanisms for capture-based protocol analysis are expensive and time-consuming, however, making the study of long term or high density traffic difficult. This research describes the design and implementation of a functional, programmable general classifier that uses Nvidia CUDA (Compute Unified Device Architecture) and compute capability 3.5+ graphics processors to greatly accelerate the analysis of arbitrary protocols stored in large captures.

This section aims to introduce the primary topics of this research, specifically packet classification and general processing on GPUs (Graphics Processing Units). The section begins with an overview of network telescopes, a notable application of long term traffic captures to the study of network traffic, in order to introduce the problems associated with long term network analysis. The remaining subsections introduce the packet classification and GPU accelerated processing domains in greater detail, and briefly explain the benefits and problems associated with general packet classification on GPU hardware.

1.1.1 IBR and Network Telescopes

Network telescopes [39, 62] are passive, low-interaction traffic collectors that promiscuously monitor an empty network block for incoming packets, and recording them in a capture file for later analysis [62]. Network telescopes passively record detected incoming traffic in packet captures, which are subsequently used to study Internet Background Radiation (IBR): a constant background noise generated by

remote hosts on the Internet, often over long periods. IBR is composed of both non-productive but benign transmissions, and potentially hostile traffic [17, 93, 134]. As network telescopes monitor empty networks that cannot request traffic, all received traffic is known to be unsolicited [62].

The packets which are collected and stored by the network telescope can be divided into one of three general categories: back-scatter, misconfigured transmissions, and malicious or hostile traffic [39]. Back-scatter comprises benign non-productive transmissions resulting from misconfigured or spoofed packets that originated elsewhere [17, 39], while misconfigured transmissions are typically produced by misconfigured or malfunctioning remote hosts. The majority of IBR traffic is aggressive or potentially hostile [39], including remote scans, packets with malicious payloads, and a host of other virus and malware related transmissions.

Packets traces are collected by network telescopes over daily, weekly or monthly intervals, and are stored in capture files for subsequent processing in traffic analysis tools, examples of which include Wireshark, TcpDump and Libtrace [39]. The size of a network telescope has significant implications for the volume of data captured, and depends on the number of unique IP addresses contained in the network block. For instance, assuming standard IPv4 block sizes, a telescope operating on a small /24 network monitors a range of $2^8 = 256$ unique addresses, while telescopes operating on larger /16 and /8 networks monitor ranges of $2^{16} = 65,536$ and $2^{24} = 16,777,216$ unique addresses respectively [39, 62]. As each unique address receives its own share of unsolicited traffic, the volume of traffic collected over a particular time delta scales with respect to the number of unique addresses being monitored. Over long intervals, large network telescopes can produce captures that are extremely time consuming to process in existing protocol analysis software [3].

Capture processing of this nature is impeded by two overarching problems. Firstly, applying a general packet classification program to hundreds of millions of packets is time consuming, and scales poorly to fast interface speeds. Secondly, contemporary consumer-grade interfaces to high-capacity non-volatile storage, such as HDDs (Hard Disk Drives) and SSDs (Solid State Drives), are relatively slow in comparison to modern multi-gigabit network interfaces, which are becoming increasingly common. These two factors often render the analysis of large or long-term traces relatively impractical; in some instances, the time required to perform thorough trace analysis far exceeds the interval over which the traffic was originally collected, impeding the study of IBR, malware propagation, security incidents, and

captured network traffic in general.

1.1.2 Packet Classification

Packet classification (also referred to as packet filtering) refers to processes which identify or categorise network packets. Packet classifiers are ubiquitous components in modern communication networks, and are employed in a variety of contexts including packet demultiplexing, routing, firewalls, intrusion detection, and network analysis. One of the core problems associated with all packet classifiers is their need to scale throughput to accommodate increasing bandwidth. This is particularly true of classifiers deployed on live high-speed networks, as these can easily bottleneck achievable packet throughput if they cannot keep up with incoming traffic.

Network bandwidth has sustained exponential growth over the past three decades, at a growth rate averaging roughly 50% percent each year between 1983 and 2014. This relationship, known as Nielson's Law of Internet Bandwidth [64], results in significant pressure to develop faster and more efficient classification algorithms, which in turn has lead to greater specialisation of classification functionality. Contemporary packet classifiers are extremely varied in construction, and are generally tailored for specific sub-domains according to their individual requirements.

The majority of packet classification research focuses on high-speed and low-latency IP (Internet Protocol) specific classifiers; these algorithms are used extensively in routing, firewalls and intrusion detection systems, and prioritise high speed and low latency to minimise the impact of classification on network throughput [113]. In general, these algorithms categorise packets based on five fields in the IP, TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) headers (source and destination address, source and destination port, and protocol), often referred to as the IP 5-tuple [113]. IP specific algorithms are architecturally diverse, and target a range of sequential and parallel platforms including CPUs (Central Processing Units) [113], FPGAs (Field Programmable Gate Arrays) [40, 41], TCAMs (Ternary Content Addressable Memory) [49, 109] and GPUs [36, 122, 136].

General packet classifiers (or filtering algorithms), in contrast, focus on providing the necessary analytic flexibility to process arbitrary protocols [9, 133]. This requires viewing packet header data at a lower abstraction layer than specific fields

in a predefined header, typically as bits or bytes in a stream [54]. Protocol independent algorithms have historically been designed for serial execution on sequential processors such as CPUs, and rely heavily on decision trees and branching to eliminate redundant computation [23, 54]. The flexibility they afford comes at the expense of speed however, and general algorithms can typically sustain a fraction of the throughput achievable by hardware accelerated 5-tuple algorithms [74].

1.1.3 General Processing on Graphics Processing Units

The application of GPU devices to non-graphical problems is referred to as GP-GPU, and is typically employed to accelerate highly parallelisable, computationally intensive functions [29]. The GPGPU paradigm utilises hundreds or thousands of light-weight cores which concurrently execute thousands of individual threads, organised into small, uniformly-sized SIMD (Single Instruction Multiple Data) groups [87]. These SIMD (Single Instruction Multiple Data) units are referred to as warps in CUDA and wave fronts in OpenCL (Open Compute Language) [45].

Due to the massive parallelism that GPUs expose, a well crafted GPGPU solution to an appropriately phrased parallel problem can potentially improve throughput over a CPU solution by between ten and one hundred fold. GPUs are extremely sensitive to a wide variety of performance criteria, however, which can restrict device utilisation (or occupancy), saturate memory resources, or significantly increase operation latency if not adhered to [86]. GPUs handle heavy thread divergence particularly poorly, serialising rather than parallelising branching code within SIMD thread warps (or wave fronts) [86]. This can create significant difficulties when porting solutions designed for CPUs, as these often employ branching as an optimisation measure to avoid redundant code. Due to the inherent parallelism of GPGPU programs, unaddressed or unavoidable performance bottlenecks can cascade, which can result in throughputs comparable to, or even below, that of a well-optimised CPU implementation in the worst cases [86, 125].

Despite these limitations, GPUs are an appealing platform for general packet classification for a number of reasons:

1. Packet classification is an inherently parallel process, applying a single program or filter set to millions of individual packets [66].

2. GPUs provide abundant and underutilised processing resources that execute concurrently with, but separate from, the host CPU. As packet classification is computationally expensive, offloading classification to a GPU can significantly reduce CPU load, and thereby limit the impact of classification on the performance of external processes and the operating system.

3. The high parallelism of GPU solutions allow them to expand across multiple devices with relative ease, which can be used to significantly improve throughput [87].

4. GPUs are energy efficient, relatively inexpensive, and widely available commodity hardware.

GPU acceleration has been applied numerous times to packet classification in the past few years, particularly in the IP filtering domain where highly parallel algorithms are well established, and classification throughput is of the highest priority [36, 122, 136]. General classification for arbitrary protocols is an exception to this trend however, with little related research conducted publicly in the GPGPU domain. This is primarily due to the processing abstractions employed in general algorithms, which rely heavily on divergence to eliminate redundant operations [54]; this is an efficient and effective strategy on CPUs, but results in heavy serialisation and significantly degraded performance when attempted on GPUs [86]. General algorithms also require more complex programs and internal state structures to handle arbitrary protocols and field types [9, 133], which is difficult to implement efficiently on register-constrained GPUs.

1.2 Problem Statement

Large packet traces provide a wealth of useful information, both in terms of collective metrics and within individual packets [16, 17]. Accessing information contained within large capture files can, however, be difficult, tedious and time consuming using traditional methods, and may in some cases require several hours or even days of processing time to complete. This limits the usefulness of large captures, heavily obfuscates the study of long-term traffic dynamics, and impedes the analysis and exploration of large traces.

This problem was partially addressed in prior research [66, 67, 70, 71], performed by the author, in fulfilment of the degree of Master of Science. This research produced a highly parallel GPU-based approach to general packet classification, called GPF (GPU Packet Filter). This approach incorporated ideas from 5-tuple routing algorithms [113] to construct a general algorithm tailored for parallel execution [66, 68, 71]. This prototype implementation prioritised device-side throughput above all, and avoided all divergent processing and runtime flexibility that might slow execution. Instead, GPF evaluated all defined filters against all packets, regardless of applicability, and relied on compile time optimisation to eliminate redundancies [66].

The approach achieved high throughputs for small sets of filters, but sacrificed substantial runtime flexibility in doing so [72, 73]. In particular, the approach could not handle variable length protocol headers (reducing flexibility, and thus coverage) or adapt processing to packet data at runtime (affecting scalability to complex filter sets) [73]. The solution did not address host-side infrastructure, and while it executed at extremely high rates, packet capture reading consistently bottlenecked performance (at that time facilitated by WinPcap[1]). In addition, no external functions were developed to apply generated outputs in a meaningful or useful way.

This research aims to fully revise and update the approach taken in GPF for modern Nvidia Kepler micro-architecture GPUs[2] [81, 86], sacrificing some of the throughput attained to greatly improve the flexibility, scalability, efficiency and usability. The resultant algorithm, referred to as GPF+, uses an improved, state-based processing abstraction to efficiently adapt to packet contents at runtime, providing sufficient flexibility to handle variable length headers and prune redundant operations. In order to apply this algorithm to the problem of capture analysis, an efficient multi-threaded classification pipeline is implemented alongside the classifier. This pipeline is intended to provide an efficient mechanism to access the contents of expansive packet captures, and produces small, ordered, and non-volatile output files. These files are used by three external post-processors, which perform visualisation, statistical analysis and data transformation functions respectively. These functions have been included as examples to demonstrate the usability of results.

[1] \url{http://www.winpcap.org/}
[2] Note that OpenCL and AMD hardware are not within the scope of this research.

1.3 Research Overview

This section provides an overview of the research performed during the development of this thesis. The section discusses the scope set for this research, the methodology by which the approach was developed into a functional experimental prototype, and the goals the research aims to achieve.

1.3.1 Research Scope

The classification pipeline presented and tested in this thesis is a functional prototype suitable for testing and basic use, focussing on the foundations of a programmable general packet classification algorithm that executes efficiently on massively parallel GPU hardware. This pipeline incorporates aspects from several rich domains into the solution, each offering many possibilities beyond what can be feasibly explored within the implementation of this project.

To maintain a manageable scope, the research focusses predominantly on the classification kernel and capture preprocessing functions, aiming to provide sufficient functionality to perform basic classification on common protocols and their fields at high speed. The implementation was scoped to facilitate sufficient flexibility to handle Ethernet, IP, TCP, UDP and ICMP protocols and their standard fields at a minimum. Mechanisms to parse and process list-based options fields were left out of scope, as this functionality only applies to a small percentage of packets. Direct and efficient support for optional fields and fields larger than 32 bits have additionally been left out of scope, as they can both be handled indirectly through existing mechanisms in the DSL . GPU acceleration is facilitated by the Nvidia CUDA API, and requires Nvidia Geforce 700 series GPUs with a minimum compute capability of 3.5 to function; earlier micro-architectures and other GPGPU platforms such as OpenCL are left out of scope.

The DSL and example applications were considered a lower priority, as these components execute briefly prior to and following the main classification process, and thus have a less significant impact on achievable performance. The DSL was scoped to include only the functionality required to generate programs from reusable high-level protocol definitions, and does not aim to develop or evaluate additional compiler-specific functionality. Refining the DSL has been left to future

work, after the classifier's performance and functionality have been evaluated and adjusted. The example applications are similarly limited to basic functionality, as they are included primarily for the purposes of evaluating and demonstrating classification functionality and classifier usability, and not as critical components in their own right.

1.3.2 Research Methodology

The research presented was developed through extensive and controlled experimentation that focussed on maximising resource efficiency, system scalability and data throughput while retaining as much flexibility as possible. The foundation of the system is based on the approach used in GPF, optimised and expanded through regular testing of functions, techniques and components during the course of development.

In order to manage complexity, the classification system was divided into several relatively distinct components that operate with some independence, and cooperate primarily through asynchronous message passing. These components include: a DSL that simplifies program generation; a preprocessor that efficiently reads, indexes and buffers packet records for classification; and a GPU accelerated classification kernel that processes packets records. The development of each of these components was guided by continuous and often extensive testing of potential architecture alternatives. This was done to help identify the most efficient architectures in terms of processing time and resource efficiency, and to help isolate and exclude poorly performing approaches early on.

In addition to the components involved in the classification pipeline, the chosen implementation also provides a selection of simple but powerful post-classification tools which apply the computed results of classification to simplify the exploration and analysis of large captures. These components were developed to serve two functions; they provide verification for results produced by the system; and demonstrate by example how such results can be applied to accelerate aspects of protocol analysis. The example applications described facilitate capture visualisation, simple field analysis, and the distillation of pre-filtered captures respectively.

To evaluate the performance of the implemented system, testing was performed using a small selection of captures (ranging from 2 to 160 GB) and a wide range of

varied filter programs. Testing was divided into two parts, with the first focussing in detail on the classifier's performance in isolation, and the second exploring the performance of the wider system from an end user perspective. This was done to distinguish between the potential performance of the classification kernel in isolation, and the throughput achievable in practice given slow read access to capture files. Testing also briefly measured the achieved throughput of the included example applications in order to evaluate the utility of the classifier's outputs.

1.3.3 Research Goals

This section briefly summarises the goals and intended outcomes of this research. One of the primary goals of the proof-of-concept implementation is to be useful in and of itself, and not merely an interesting but isolated component within a conceptual or hypothetical system. By limiting the scope of the implementation to focus specifically on large capture files, a valuable but underutilised resource, the implementation can provide a potential solution to a real but under-addressed problem in the process of investigating flexible GPU based protocol analysis. The research further aims to develop and evaluate the foundations of an approach suitable for a wider range of network-related applications beyond capture processing (e.g. live network monitoring, intrusion detection, detailed metrics calculation, etc.).

Accomplishing this involves meeting three sub-goals:

1. To extend the functionality and improve (in order of priority) the flexibility, scalability, efficiency and usability of the classification approach originally derived in GPF. Some additional clarification is necessary, as these terms can be ambiguous without sufficient context.

    - Flexibility refers to the range of protocols and field types that the system can process, and the range of functions that the system can perform. To improve upon the flexibility of GPF, which executed as a purely interpretive filtering processor, the GPF+ approach uses a more complex abstraction that maintains internal state and tracks select protocol and packet metadata. This allows for more flexibility with respect to the types of packets that can be processed, and the types of information that can be extracted from them. In particular, the new approach provides

support for handling optional fields and variable length headers, as well as field value extraction, which were not possible in GPF. Flexibility is primarily addressed by the GPF+ classification kernel and it's associated DSL, discussed in Chapters 6 and 7 respectively.

- Scalability refers to the computational efficiency with which the classifier handles larger captures and classification programs. An important goal of the GPF+ classifier is to provide linear or better scaling, in order to handle terabyte-scale captures spanning trillions of packets in reasonable time-frames without saturating memory or compute resources on either the host or the GPU. While the current implementation of GPF+ is intended for deployment on a single GPU, the architecture is compatible with a multi-GPU approach that would allow efficient scaling of throughput based on the number of GPUs utilised. Scalability is discussed in Chapter 5 and Chapter 6.
- Efficiency is in many respects related to scalability, but in this context refers more specifically to the efficient utilisation of GPU compute and memory resources. GPF+ is designed to execute with full GPU multiprocessor occupancy on modern Kepler and Maxwell chipsets, while minimising costly global memory reads and improving coalescing through a local, register based cache. The classifier uses fast, low-latency constant and register memory to accelerate access to runtime constants and variables, and applies efficient warp voting and register shuffling intrinsic functions to facilitate computational pruning and inter-thread communication respectively. The relevant GPGPU concepts are addressed in Chapter 2, while their application within the classifier to maximise efficiency is discussed in Chapter 6.
- Usability refers to the ease with which programs and their component protocols can be constructed and reused. GPF relied on a relatively simple programming abstraction through a low-level DSL, which lacked mechanisms for compartmentalisation, runtime optimisation, and code reuse. GPF+ uses a more robust programming abstraction and DSL that greatly simplifies program construction, facilitates code reuse through protocol definitions, and facilitates automatic runtime optimisation through protocol layering. Usability is primarily addressed in Chapter 7, although many aspects derive from the classification architecture discussed in Chapter 6.

2. To implement efficient parallel host-side architecture to read, index and buffer packet data as quickly as is possible within the limits of storage IO, in order to minimise host-side interference and limit its impact on achievable classification performance. These supporting components are discussed in Chapter 5.

3. To implement a selection of example applications that employ the outputs of the classification system to accelerate aspects of packet analysis, in order to assess the usefulness of results in a real-world context. This goal is addressed through three separate proof-of-concept applications that utilise the outputs of the classifier and its pipeline to analyse, visualise and reduce packet captures. these applications are discussed in Chapter 8.

These goals are evaluated in Chapter 12.

1.4 Document Overview

The chapters in the remainder of this document are divided into four broad parts. In Part II, the background information regarding packet classification and GPGPU is provided. Part III presents a detailed overview of the design and implementation of the GPU accelerated classification pipeline and its various components. The implementation is subsequently evaluated through extensive testing, as described in Part IV. Part V concludes with an overview of implementation and results, and a consideration of extensions and future work. A more detailed breakdown of the chapters in each part follows:

Part II

This part deals with the research background, exploring GPGPU, network traffic and packet classification in some detail.

Chapter 2 provides a brief introduction to GPGPU and CUDA, and details important functions, optimisation strategies and hardware limitations which influenced or shaped the classifier's implementation.

Chapter 3 provides a detailed overview of packets, protocols, and packet capture files. The chapter also introduces pcap and Wireshark, which are commonly used to interface with captures and live traffic.

Chapter 4 discusses packet classification in detail, and summarises the results of previous work in utilising CUDA to perform protocol independent classification. This research forms the foundation for the implementation discussed in Part III. This chapter concludes with a brief overview of related work in the wider field of GPU accelerated packet classification.

Part III

This part of the document describes the implementation of the classification system and its various components.

Chapter 5 introduces the implemented proof-of-concept system, its abstract processing methodology, and the high-level components of which it is comprised. This chapter additionally describes the supporting host side components of the classification pipeline responsible for reading captures, indexing packets, transferring data to and from the GPU device, and writing results to disk.

Chapter 6 introduces the GPU classification approach, and describes both its high level architecture and implementation in detail. In addition to describing the two primary classification processes, the chapter describes the various memory regions used to facilitate classification, and describes the caching mechanism employed to minimise global memory interactions when processing packet data.

Chapter 7 describes the domain specific language used to compile high-level programs into executable instruction streams for the classifier. The chapter introduces the grammar syntax and explains the process by which the language is compiled to an instruction byte-stream. The compilation process is discussed after classification as it is heavily dependent on the architecture of the classifier, and is difficult to describe without first describing the system it supports.

Chapter 8 describes three post-processor applications which use the results of the classification pipeline to significantly accelerate the process of mining large packet traces. These applications include high-level interactive graph-based visualization, capture reduction, and simple field value analysis. These three applications provide examples of how the results of classification may be applied.

Part IV

This part of the document is dedicated to the evaluation of the classification kernel and the the wider classification process.

Chapter 9 provides an overview of the testing configuration and approach, and details the packet captures and filter programs used during evaluation.

Chapter 10 presents and discusses the performance results and gathered metrics for the GPU classification kernel, as applied to three captures of varying size and using nine programs of varying complexity. This chapter ignores host-side processing and file access overhead to specifically focus on the potential throughput of the classification process.

Chapter 11 discusses the performance results collected from the wider system and example applications, inclusive of all GPU and host functions. This chapter aims to illustrate actual achievable capture processing performance of the system from a user perspective.

Part V

Chapter 12 concludes the document with a summary of the findings of each chapter, and an overview of some potential functions, enhancements and extensions that could be incorporated in future work.

Part II

Background

2

General Processing on Graphics Processing Units

NVIDIA CUDA[1] is a platform that enables massively parallel computation on Nvidia GPU hardware [87]. This chapter introduces the fundamental concepts of CUDA programming relevant to this thesis, focussing specifically on Kepler micro-architecture introduced in 2013. The chapter is structured as follows:

- Section 2.1 introduces the GPGPU domain, providing a brief history of the evolution of GPGPU devices and introducing CUDA.
- Section 2.2 briefly summarises the evolution of CUDA micro-architecture, listing the most significant changes between each multi-processor generation.
- Section 2.3 describes the CUDA programming model, introducing the concept of CUDA kernels and how they achieve parallelism through threads and thread warps.

[1] http://www.nvidia.com/object/cuda_home_new.html

- Section 2.4 provides an overview of the Kepler multi-processor [81, 87] and its features.
- Section 2.5 discusses the large but comparatively slow global memory space, housed in GPU DRAM. This section focuses on the various mechanisms available to improve global memory performance.
- Section 2.6 considers on-chip memory resources, including constant, shared and register memory. These memory regions facilitate much higher throughputs than global memory, but have extremely limited capacity.
- Section 2.7 focuses on two hardware accelerated function types that facilitate fast and synchronisation free inter-warp communication.
- Section 2.8 explores several important considerations relating to the efficiency and performance of CUDA programs. These topics include the transmission of data to and from the device, device occupancy, execution streams and concurrent execution, and instruction throughput optimisation.
- Section 2.9 provides a summary of topics covered in the chapter

2.1 Introduction to GPGPU

General Processing on Graphics Processing Units (GPGPU) is a sub-domain of high-performance computing. GPGPU uses the massively parallel capabilities of modern graphics cards to accelerate computationally expensive and data intensive parallel tasks beyond computer graphics. This section provides a concise introduction to the GPGPU paradigm and the CUDA platform, and provides a foundation for the remainder of the chapter.

2.1.1 Brief History of GPGPU

The term *Graphics Processing Unit* was first coined in 1999 when Nvidia introduced the Geforce 256, marketed as “the world’s first GPU” [76]. This slogan aimed to differentiate the Geforce 256’s unified processing chip-set from prior graphics accelerator cards, which were composed of video memory and a range of hardware accelerated special function units.

While the Geforce 256 incorporated transform, lighting, setup and rendering functionality on to a single chip [76], it was the Geforce 3 chip-set introduced in 2001 that provided the first custom programmable vertex and pixel shaders to supplement the previously fixed graphics pipeline [77]. This architectural change provided a relatively simple programmable interface to the graphics hardware, introducing a level of flexibility that allowed researchers to investigate and apply GPUs to highly parallel, non-graphical problems in order to accelerate their performance. This lead to the development of the Brook language specification at Stanford in 2003, an extension to the ANSI C specification designed to easily facilitate data parallelism [14].

In 2006, with the release of Microsoft DirectX10 and the Unified Shading Model, vertex and pixel shaders were combined to form a single unified shading core [11]. This provided greater flexibility, and improved performance in both the well-established graphical domain and the relatively new GPGPU domain. Hardware vendors capitalised on this evolution by introducing their own low-level APIs, which removed the graphical abstraction and provided programmers with more direct access to underlying hardware.

Early GPGPU capable cards were comparatively limited in terms of performance and functionality, and their programs were difficult to both program and debug. Subsequent generations of GPGPU micro-architecture have addressed many of these issues, incrementally scaling processing power whilst relaxing performance constraints, adding functionality, reducing power draw and improving tool chain support [87]. Modern GPUs provide thousands of cores, gigabytes of device memory, and a variety of caches to accelerate GPGPU programs and simplify GPGPU programming.

Currently, several GPGPU-capable APIs are available, including Nvidia's CUDA (Compute Unified Device Architecture) [87], Khronos Group's OpenCL (Open Compute Language) [44], and Microsoft's DirectCompute [78].

2.1.2 Compute Unified Device Architecture (CUDA)

The approach developed in this research is designed for and utilises the Nvidia CUDA API (Application Programming Interface), and employs a small selection

of functions that are currently specific to recent Nvidia GPUs. This subsection provides a basic overview of the API and its history.

CUDA v1.0 was introduced in 2007 [126], and has since become the dominant paradigm for massively parallel scientific computing [52]. The CUDA API is currently in its seventh generation, and is supported by all recent Nvidia GPUs. In addition to desktop and mobile GPUs, the Nvidia maintains the Tesla® processor range specifically for professional GPGPU applications and scientific computing [89]. These devices support additional GPGPU features exclusive to Tesla hardware [75].

All CUDA devices have a Compute Capability (CC) rating which indicates the device's underlying architecture and the features and functions it supports. The CC is divided into a major and minor revision number, delimited by a period. The major revision number indicates the underlying micro-architecture of the device, while the minor revision number indicates incremental improvements and extensions to the major architecture [87]. The application discussed in this thesis was designed for and developed using a Compute Capability 3.5 device. As such, discussion in this research will focus predominantly on compute capability 3.x devices (Kepler micro-architecture) [79, 81], while drawing attention to relevant changes in compute capability 5.x devices (Maxwell micro-architecture) [83, 84].

2.1.3 Benefits and Drawbacks

The GPGPU processing paradigm has numerous benefits and drawbacks. This section summarises a few of these that are relevant to the current research, beginning with a selection of important benefits:

- *Processing Speed* – Perhaps the most highly regarded benefit of the GPGPU paradigm is the computational performance it is capable of achieving with commodity hardware, given an appropriate parallelisable problem and a well crafted solution.
- *Resource Utilisation* – GPU co-processors have significant computational power which is rarely utilised outside of graphics intensive programs. The GPGPU paradigm facilitates the utilisation of this spare processing power, thereby freeing CPU cores and host resources for other tasks.

Table 2.1: Comparison of PCIe aggregate, upstream and downstream transfer bandwidth to GTX Titan device memory bandwidth [80, 96].

	PCIe 2.x	PCIe 3.x	Geforce GTX Titan
Theoretical Bandwidth	16 GB/s (8/8)	$\approx$ 32 GB/s (16/16)	280 GB/s

- *Scalability* – As GPGPU programs are intrinsically highly parallel, they scale with relative ease to multiple devices on a single host, or to distributed hosts on a network or in a cluster. Provided an appropriately scalable algorithm and suitable program infrastructure are available, performance may be increased by adding additional GPU hardware [87].
- *Availability* – GPUs are commodity hardware, and the majority of contemporary hosts thus support some form of GPU acceleration. This differentiates the GPGPU paradigm from other massively parallel computational solutions (such as FPGAs), which rely on specialised non-commodity hardware not found on typical desktop machines.
- *Cost* – GPUs are a cost effective platform for high-performance computation, as they are relatively inexpensive and provide high energy efficiency (or performance per watt) [79, 84].

These benefits are counterbalanced by several draw backs to the paradigm.

- *Transfer Overhead* – One of the more problematic aspects of GPGPU based solutions is the overhead associated with memory transfers between the host and the device. GPUs rely on copies to and from host memory (whether staged or streamed) to both sustain them with data to process, and to eventually return results on completion. In addition to the need to perform an additional memory copy to make data available to the device, these copies are currently performed through the PCIe (Peripheral Component Interconnect Express) bus [96], which is bandwidth limited in comparison to GPU device memory (see Table 2.1). As a result, transferring data to and from the GPU for processing can present a potential bottleneck for data heavy applications [86]. This may be improved upon in future architectures through technologies such as NVLink [15, 25] (see Section 2.2).

- *Problem Applicability* – GPGPU is a paradigm intended specifically for massively parallel computation, and requires a high level of task and instruction level

parallelism to execute efficiently. As such, sequential solutions or those which rely heavily on context-sensitive branching perform poorly on GPGPU hardware [86].

- *Complexity and Accessibility* – GPGPU programming can have a relatively steep initial learning curve due to the differences between it and more traditional sequentially oriented languages. In addition, non-trivial solutions can be much more difficult to conceptualise and debug, due to the vast numbers of independent executing threads. Finally, GPGPU programs have relatively limited on-chip resources and can be extremely sensitive to the structure of program code; inefficiencies can easily slow performance by more than an order of magnitude [86], and can be difficult to isolate. It is worth noting that in these respects GPGPU APIs have improved significantly in recent years, providing greatly relaxed performance restrictions and significantly improved tool chain support to reduce complexity.

2.2 CUDA Micro-architectures

This section briefly outlines the evolution of the architecture of CUDA capable GPUs, discussing in particular the major changes brought by each new generation of chip-set. The GPF algorithm [66] on which this research is based (see Section 4.4) was developed to target Tesla micro-architecture devices [66], while the implementation discussed in Part III of this document was designed to operate on Kepler and Maxwell micro-architectures. This section is intended to summarise the most notable changes between these micro-architecture generations, and to discuss the expected changes to Nvidia micro-architecture in its next iteration.

2.2.1 Tesla (CC 1.x)

The Tesla micro-architecture encapsulates the first CUDA enabled micro-architecture developed and released by Nvidia; a name shared with the Nvidia Tesla [89] brand of high-performance GPUs for scientific and industrial use. The Tesla micro-architecture was introduced with the G80 chip-set (in Geforce 8 series GPUs), initially released in late 2006, and remained in use until the arrival of the Geforce 400 series in early 2010 [28]. Tesla micro-architecture introduced the

basic components and structure of the CUDA API, but provided fewer registers and less shared memory space on chip than all subsequent architectures [87]. In addition, early Tesla cards (those with a compute capability of 1.0 or 1.1) had extremely strict requirements that had to be met in order to efficiently coalesce access to device global memory – a feature which allows a single memory transaction to service multiple threads simultaneously (see Section 2.5.1). These requirements were greatly relaxed in compute capability 1.2 and 1.3 GPUs, but still required care to avoid inefficient access patterns. This problem could be partially mitigated through the use of the read-only texture cache.

2.2.2 Fermi (CC 2.x)

Fermi is the second generation of Nvidia CUDA-enabled micro-architecture, introduced in early 2010, which included devices from the GTX 400 and GTX 500 family of graphics cards [28]. Fermi significantly increased the amount of on chip resources available to threads, doubling the number of registers and quadrupling the maximum amount of shared memory per multiprocessor to 64 KB (this would have to be shared with the L1 cache, which could be configured to consume either 16 KB or 48 KB of shared memory resources) [87]. L2 caching of global memory was additionally introduced, which greatly simplified global memory coalescing requirements and improved the performance of global memory transactions. Fermi also improved the performance of double precision floating point calculations, and raised the maximum number of registers allocatable to a single thread from 32 to 63, among other improvements and optimisations [87].

2.2.3 Kepler (CC 3.x)

Fermi was succeeded in 2012 by the Kepler micro-architecture [79], which further increased available resources on chip, and improved bandwidth between multiprocessors and global memory [81]. Kepler micro-architecture and compute capability 3.0 doubled the number of registers available on each multiprocessor (from 32 thousand to 64 thousand [87]) and introduced the Read-Only (RO) cache [81], among other improvements .

Compute capability 3.5 devices later expanded the maximum number of registers allocatable to a single thread from 63 to 255, improved control of the RO cache, and

introduced register shuffling as a fast alternative to shared memory for inter-warp transfers [81]. Compute capability 3.5 and the GK110 chip-set additionally added support for dynamic parallelism (which allows CUDA kernels to nest calls to other kernels to create additional work at runtime) and Hyper-Q (which improves handling of concurrent streams of execution, allowing for true independence between them).

2.2.4 Maxwell (CC 5.x)

Maxwell micro-architecture [84], introduced in 2014, is the most recent CUDA micro-architecture. In comparison to the Kepler micro-architecture, Maxwell GPUs provide reduced power consumption, expanded shared memory capacity, and increasing overall processing efficiency. Maxwell architecture dissolved the L1 cache, returning the full shared memory capacity to the user (CC 5.0). It also provided faster native shared memory atomic functions, among other efficiency improvements [83]. Second generation Maxwell devices (CC 5.2) expanded the size of shared memory to 96 KB [83], thereby providing twice as much shared memory capacity as Fermi and Kepler architectures per multiprocessor.

2.2.5 Future Architectures

The Pascal micro-architecture [15, 25] is the expected successor to the current Maxwell micro-architecture, and is due for release in 2016 [31]. While Pascal processors are not yet available and do not yet have an assigned compute capability, the feature set of these processors focuses heavily on improving memory bandwidth both on the device and between the GPU and host memory. This has potential implications for GPGPU performance in the near future. The three most significant features with respect to this research problem (3D memory, NVLink and Unified Memory) [15, 25] will be briefly discussed below:

- 3D memory, also referred to as stacked DRAM, is a new memory architecture that stacks multiple memory chips on top of each other, greatly improving memory bandwidth from hundreds to thousands of bits [31]. This is expected to raise maximum throughput between memory and multiprocessors by roughly a factor of three, and increase maximum device memory capacity from 12 GB to 32 GB [15].

- NVLink is a high-speed interface that replaces the PCIe bus as the mechanism by which GPUs communicate with the host and other GPUs. NVLink improves performance over the PCIe bus by between 5x and 12x, allowing GPUs direct access to host memory at similar speeds to that of a CPU [15]. NVLink also increases the number of GPUs supported by a single CPU from four to eight.

- Pascal is expected to provide hardware support for Unified Memory through NVLink and 3D memory, allowing CPUs direct access to GPU device memory (and vice versa) at high speed, potentially eliminating the need to stage expensive copies between host and device memory in data intensive applications [31]. Basic support for unified memory was introduced into the CUDA 6 API [32], but relies on much slower memory architecture and the PCIe bus. Unified memory is thus currently aimed at reducing the complexity of writing CUDA programs by eliminating the memory management component, rather than providing a viable alternative for more intensive or performance-critical applications.

In combination, these features promise dramatically improved memory performance for future CUDA devices and the GPGPU programs executing on them, allowing for greater and more seamless cooperation between CPUs and GPUs. Pascal is expected to be superseded by the Volta micro-architecture in 2018 [15].

2.3 CUDA Programming Model

The CUDA programming model is a programming abstraction designed to facilitate massively parallel general processing in a GPU environment, with many elements derived directly from underlying hardware. CUDA programs, known as kernels, are written using CUDA C syntax (a subset of the C'99 language augmented to facilitate massively parallel execution) and contained within CUDA files (typically identified with the .cu extension). CUDA files may simultaneously contain C and C++ code, as the Nvidia CUDA Compiler (NVCC) automatically separates out host-side code and passes it to the default compiler installed on the system. The CUDA Run-time API and CUDA Driver API facilitate communication and thus interoperability between the host-side process and the CUDA device, achieved through

calls to the CUDA device drivers installed on the system. The host thread schedules data transfers to and from the device, as well as the execution of kernels [87]. The host thread runs concurrently with kernel execution, allowing it to continue processing independently and simultaneously with the device.

2.3.1 CUDA Kernels and Functions

CUDA programs are encapsulated within kernels, which are specially defined functions that execute on GPU hardware but can be called from the host context. In typical cases, kernels pass data to the device through the PCIe bus and write results to a device-side output array. They cannot return data to the host directly (all kernels require a `void` return type), so output must always be written to a device-accessible array and subsequently retrieved by the host thread. Device-side memory persists outside of the context of a kernel's execution, and may be passed to subsequent kernels for additional or elective processing.

Kernels are supplemented by CUDA functions, which are completely interchangeable with host side functions. As long as a function only accesses resources that are available in both host and device contexts, it may be compiled to both CUDA device code and host code for use in both contexts. This one-to-one mapping necessarily implies that CUDA functions may return values and execute other functions, similar to their host side equivalents. In addition to functions, CUDA supports most standard C and C++ elements, including but not limited to: classes, structs, decision operators (if, switch), and iteration operators (for, while, do).

Kernels are declared using the `__global__` keyword, and only support a `void` return type. CUDA functions are declared using the `__device__` keyword, which may be supplemented with the `__host__` keyword if host side execution is also required. All functions in the CUDA file with no prefix are implicitly assigned the `__host__` prefix (which may optionally be specified explicitly) and cannot be called from within GPU functions. Listing 1 shows an example kernel which raises each element of the array `device_in` to the power of `power`, writing results to the array `device_out`. The calculated variable `thread_id` is used to index these arrays such that each thread processes a different element. The built in registers used in the calculation of `thread_id` are explored in the next subsection.

Listing 1 Example CUDA kernel that calculates the n^{th} power of each element in array in, writing results to array out.

```
__device__ int NthPower(int value, int power) {
    int out = 1;
    for (int k = 0; k < power; k++) out *= value;
    return out;
}

__global__ void ExampleKernel(int* device_in, int* device_out, int power){
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
    device_out[thread_id] = NthPower(device_in[thread_id], power);
}
```

Table 2.2: Keywords for thread identification.

Keyword	Components	Description
gridDim	x, y, z	Blocks in each dimension of the grid.
blockDim	x, y, z	Threads in each dimension of the block.
blockIdx	x, y, z	Index of the block in each dimension of the grid.
threadIdx	x, y, z	Index of the thread in each dimension of the block.

2.3.2 Expressing Parallelism

Kernels execute a collection of threads, typically operating over a large region of device memory, with each thread computing a result for a small segment of data [87]. In order to manage thousands of independent threads effectively, kernels are partitioned into thread blocks of up to a maximum of 1024 threads in compute capability 3.x devices or above [87]. Thread blocks are conceptually positioned within a one, two or three dimensional grid, which may contain millions of thread blocks (up to [$2^{31} - 1$, $2^{16} - 1$,$2^{16} - 1$]) [87]. Each thread is aware of both its own position within its block and its block's position within the grid, and can use this knowledge to calculate its global index in the thread pool. This index can then be used to determine which elements of data to operate on, and where to write results when the operations are completed [87]. A list of keywords which support thread identification are provided in Table 2.2.

Each block is executed by a single multiprocessor, which allows all threads within the block to communicate through on-chip shared memory. A single multiprocessor can execute multiple blocks simultaneously, up to a maximum of 16 resident blocks per multiprocessor on Kepler devices, and 32 resident blocks on Maxwell devices [86, 87]. Of course, if n blocks execute on a single multiprocessor, then both the

shared memory capacity and registers available to each block are reduced by a factor of n.

2.3.3 Thread Warps

CUDA kernels use an execution model called SIMT (Single Instruction, Multiple Thread) that allows threads to execute independent and divergent instruction streams [87]. SIMT groups threads into units called warps; sets of 32 adjacent threads which execute concurrently using a shared instruction cache in a SIMD configuration. This correlates closely to GPU hardware, as CUDA cores on a contemporary multiprocessor are divided into SIMD groups, each controlled by a single warp scheduler [42]. While the number of CUDA cores and warp schedulers varies by generation, warp schedulers on all architectures drive at least 32 CUDA cores at a time, typically over the course of two or more clock cycles. This has ramifications for highly divergent code, as divergent instruction streams within a warp must be serialised (see Section 2.8.4).

Thread warp size is independent of hardware architecture, and is consistent across all existing Nvidia GPUs[2] [87]. Thread warps are organised sequentially, such that the first contiguous group of 32 threads in an executing kernel belong to warp 1, while the next group belong to warp 2, and so on [87]. Warp size is an important consideration for all GPU algorithms, as any significant instruction divergence within a warp can dramatically impair performance [86].

2.3.4 Thread Block Configurations

Thread block size can have unexpected effects on performance arising from a number of complex factors. The most significant consideration when selecting thread block size is device occupancy. GPU devices with a compute capability of 3.0 or greater can execute a maximum of 2048 threads per multiprocessor and 16 thread blocks per multiprocessor [87]. In addition, a multi-processor can only execute as many threads as local resources (such as shared memory and registers) allow.

The effect of thread block size on kernel execution is typically not this straight forward, and it is rarely immediately evident what configurations will perform the

[2]Warps are referred to wavefronts and contain 64 threads on AMD hardware [1].

most efficiently. For instance, and rather counter-intuitively, even limiting multiprocessor occupancy (by using blocks of 64 threads) can have net positive performance impact in memory-bound kernels [86]. This can most easily be explained as the result of a memory access pattern that is overutilising global memory bandwidth with two many non-contiguous uncoalesced requests (although this may not be the only possible explanation in all instances). Reducing occupancy reduces the load on the global memory bandwidth, and allows more values to be retrieved or stored per transaction. Test configurations where kernels with only 50% occupancy significantly outperform those with 100% occupancy provide a good indication that memory access is inefficient (see Figure 2.2 for an example).

2.3.5 Preparing and Launching Kernels

Kernels are launched by a host thread after device side memory has been allocated and input data has been copied to the device. This may be done through either blocking or asynchronous API calls. The kernel launch is always asynchronous, and uses a syntax similar to a standard method, extended to allow configuration of grid and block sizes. Kernels may optionally specify shared memory capacity per block, as well as execution stream number (see sections 2.6.3 and 2.8.2 respectively). The syntax is as follows:

```
1 kernelName <<< gridSize, blockSize [, sharedMemoryPerBlock] [,
    executionStream] >>> ( kernel_args ... );
```

A simple host function that uses the example kernel from Listing 1 to raise all values in an array to a specified power is provided in Listing 2. This function illustrates how device memory is allocated and filled, how a kernel is launched, and how results are returned back to the host. Note that the function uses blocking API calls to allocate memory and transfer data, which eliminates the need to synchronise the kernel before reading outputs from the device.

2.4 Kepler Multiprocessor

The application developed as a product of this research is designed for Nvidia GTX 700 series Kepler GPUs or later (see Section 1.3.1), in order to take advantage of

Listing 2 Example host program that manages the execution of the example kernel shown in listing 1.

```
1  #include "cuda_runtime.h"
2  ...
3  void hostFunction(int* array, int array_size, int power, int block_size)
4  {
5      int* dev_in, dev_out; //declare device pointers
6
7      //allocate device memory
8      cudaMalloc((void **) &dev_in, array_size * sizeof(int));
9      cudaMalloc((void **) &dev_out, array_size * sizeof(int));
10
11     //copy input data to device array
12     cudaMemcpy(dev_in, array, array_size * sizeof(int),
           cudaMemcpyHostToDevice);
13
14     //calculate grid size and launch kernel
15     int grid_size = array_size / block_size;
16     ExampleKernel<<<grid_size, block_size>>>(dev_in, dev_out, power);
17
18     //copy results back into array
19     cudaMemcpy(array, dev_out, array_size * sizeof(int),
           cudaMemcpyDeviceToHost);
20 }
```

register shuffle operations and the read-only memory cache. These functions are used to accelerate access to protocol header data at arbitrary offsets in device global memory, arguably the most significant potential bottleneck in the classification kernel. This section provides a brief overview of the Kepler multiprocessor and its on-chip resources. The remaining sections in this chapter assume Kepler microarchitecture unless otherwise specified.

The Kepler Streaming Multiprocessor (SMX) contains 192 CUDA cores that are controlled by four independent warp schedulers. Warp schedulers are responsible for 48 cores each [87], and emit two independent instructions at a time to these cores. Of these cores, 32 are used to process the warps threads, while 16 are used for ILP (Instruction Level Parallelism) [81]. ILP allows mutually exclusive (or independent) operations to execute simultaneously, thereby improving kernel performance [86, 125].

In addition to abundant off-chip device memory, each multiprocessor has access to [87]:

- 64KB of low latency on-chip storage for shared memory and L1 caching.

- 65,536 fast 32-bit registers stored in an on-chip register file.
- 64KB of low latency constant memory which is readable by all multi-processors, but can only be written to by the host thread.
- 48KB read-only data cache for texture, surface and Read-Only memory.
- 32 SFUs (Special Function Units) for single-precision floating-point transcendental functions.

Multiprocessor architecture is not consistent across micro-architectures or device generations. For instance, Maxwell streaming multiprocessors (SM) contain only 128 CUDA cores, while second generation Maxwell devices support up to 96 KB of shared memory [83, 87].

2.5 Global Memory

Global memory is by far the most abundant memory region on CUDA devices, typically providing one or more gigabytes of capacity. This capacity comes at the expense of access latency, with individual uncached requests requiring hundreds of clock cycles to succeed [86]. This introduces a critical bottleneck in kernel execution, which can significantly impoverish the processing throughput in data intensive applications. To mitigate this, multiple memory accesses may be coalesced into fewer, larger transactions within a warp [86], greatly improving warp-level access latency. Coalescing is not always possible, and depends heavily on the physical layout of data in device DRAM.

2.5.1 Coalescing and Caches

Coalescing requirements were extremely strict in early Tesla devices, but have relaxed significantly in subsequent generations due to automatic L2 caching and other improvements [86]. In Kepler and Maxwell devices, memory requests in a warp resulting in an L2 cache hit are coalesced into one or more 128 byte cache lines, each mapping to a 128 byte-aligned segment in global memory [87]. If the requests result in a cache miss, they are instead coalesced into as few 32 byte

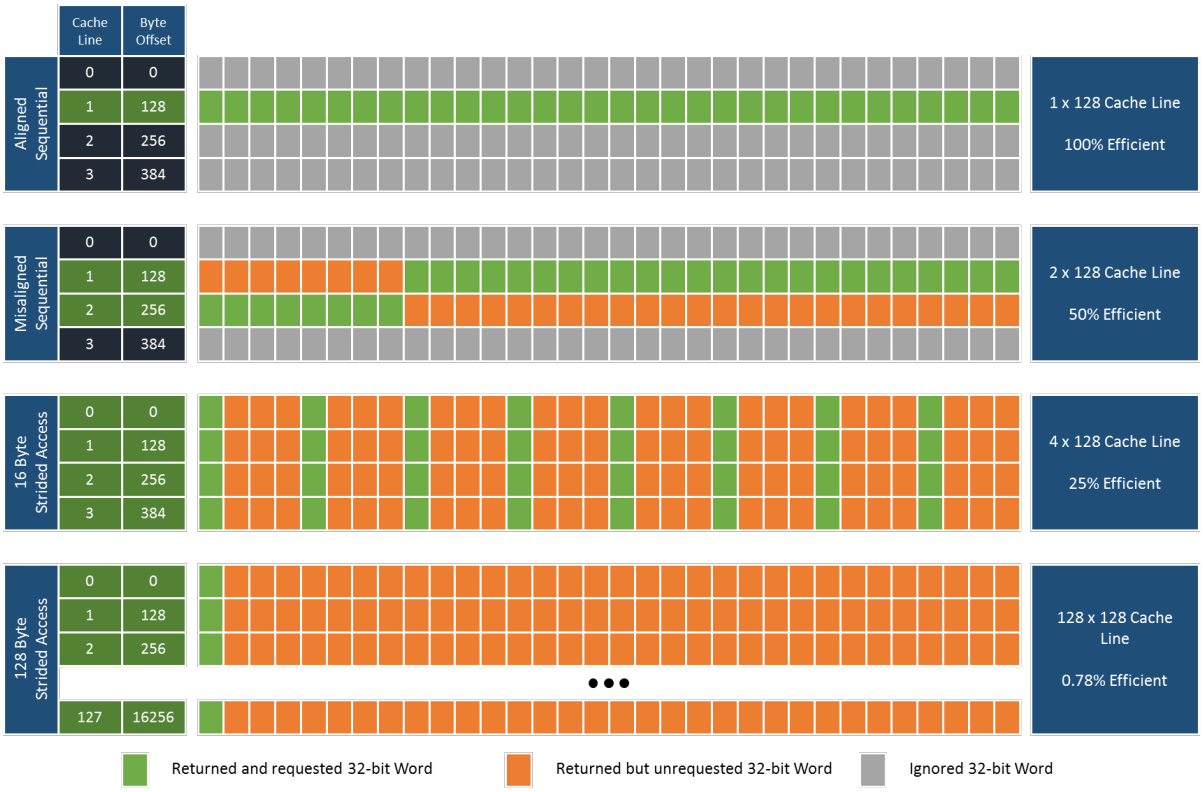


Figure 2.1: Coalescing global memory access for 32-bit words on CC 3.x devices.

segments as possible. This behaviour is different in earlier Fermi devices, which used both L1 and L2 caches to accelerate global memory accesses. Figure 2.1 shows the coalescing results of three different access patterns in Kepler devices, provided as an illustrative example.

Kepler GPUs provide three additional approaches to accelerating access to data contained within the device's global memory space: through texture references and objects, CUDA Arrays, or the recently introduced Read Only cache.

2.5.2 Texture References and Objects

The texture cache is an on-chip cache intended to accelerate texture loads from global memory, and may additionally be used to cache integral and floating point loads in more general CUDA programs. Prior to global caching mechanisms introduced in compute capability 2.0 GPUs, coalescing requirements for global memory loads were significantly more strict, and loading non-contiguous records would often incur a significant performance penalty when reading from uncached global

memory. If it was not possible to coalesce reads, it was often beneficial to perform reads through the texture cache instead. More recent devices support surface memory in addition to texture memory, which additionally support coherent writes during kernel execution.

The texture cache is usually leveraged through globally declared, static texture reference variables that are explicitly bound to a region of GPU linear memory (or as a CUDA Array) prior to the launch of a kernel. Texture references may not be passed as arguments, and incur some additional overhead when a kernel is invoked. Kepler-based devices and above additionally support texture objects, which may be passed as standard objects into and between kernel methods. Texture objects avoid many of the limitations and much of the overhead incurred by globally declared references [114].

All performance measures shown use texture objects, as initial testing showed they outperformed texture references by a noticeable margin in both coalesced and uncoalesced reads.

2.5.3 CUDA Arrays

CUDA arrays are opaque memory layouts that are optimised for texture reading through the texture cache. CUDA arrays can have one, two or three dimensions, with each dimension allowing a maximum of 65,536 elements [87]. While one dimensional CUDA arrays have limited utility due to their small capacity, two dimensional CUDA arrays can contain over 4 billion elements at a time, and thus provide sufficient capacity for processing large collections of data.

2.5.4 Read-Only Cache

The read-only cache is a new cache introduced with the Kepler micro-architecture. It uses the same cache as the texture pipeline to accelerate loads to read only data in global memory, but does not need to bind the data to a texture object or reference, and lacks the size limitations of traditional textures [81, 87]. The read-only cache may only be used explicitly on compute capability 3.5 devices or higher, using the `__ldg()` function. The read-only cache is available on compute capability 3.0 devices, but cannot be explicitly forced; it is used if the compiler determines a

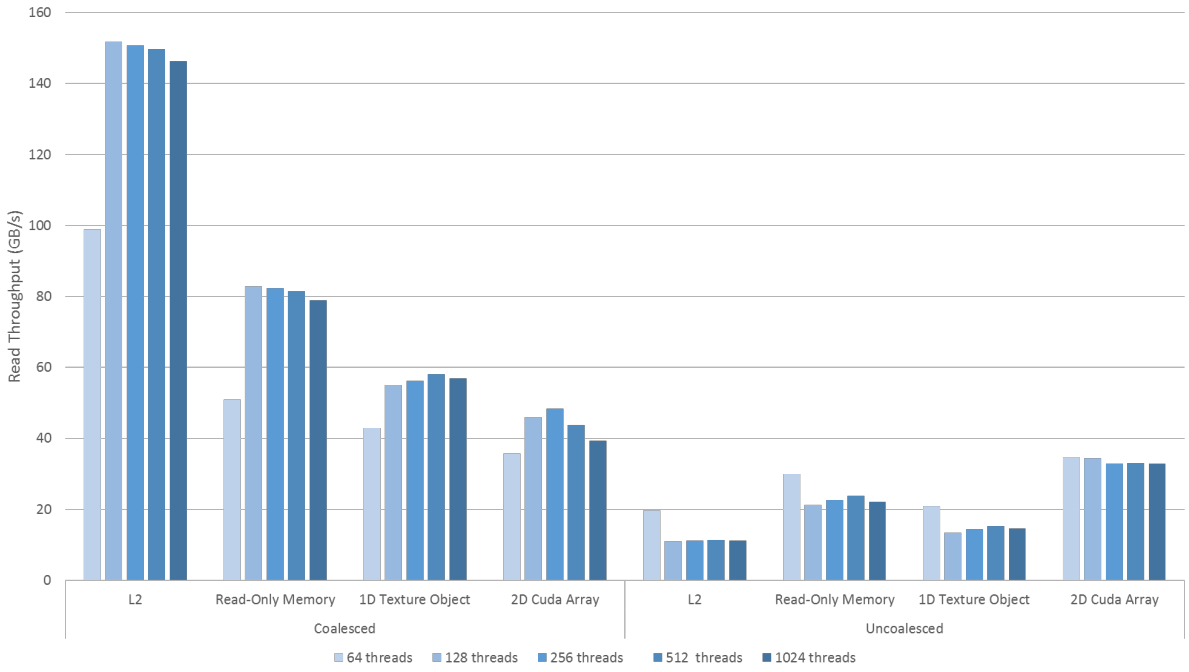


Figure 2.2: Measured read performance by access type, cache memory and thread block size.

particular memory load accesses constant data, which can be hinted at by applying the `__restrict` and `const` modifiers on relevant memory pointers.

2.5.5 Performance Comparison

This section compares the performance results of each of the above mentioned global memory access approaches when utilised to read 128 bytes of fully coalesced and fully uncoalesced synthetic data, using 16 bytes of register memory as a rolling cache. These memory layouts represent the performance extremes of ideal case and worst case respectively, and thus give a decent approximation of the performance range of each type. The performance of coalesced and uncoalesced memory access, collected by the researcher using Nvidia Nsight 4.2 [85], are summarised in Figure 2.2. These results show the effective throughput achieved when summing sets of 32 integers (or 128 bytes) per thread, using coalesced and uncoalesced read patterns.

This figure shows that while L2 caching significantly outperforms all other methods when reading fully coalesced data, it performs least efficiently when accessing fully uncoalesced data. In contrast, the performance of 2D CUDA arrays vary the least overall, providing the best uncoalesced performance and the worst coalesced

read performance. Of the remaining three, read-only memory consistently outperformed texture objects, which in turn out performed texture references (not shown). These results will be revisited in Section 2.8.3 to explain why block sizes of 64 threads typically perform far better than other block configurations when memory access is not coalesced, and how this can be an indicator of bandwidth saturation.

2.6 Other Memory Regions

CUDA GPUs provide additional memory regions which support higher throughputs and lower latency than global memory, at the expense of highly limited storage capacity. Effective management and application of these limited memory resources is critical to achieving high execution efficiency, as they reduce dependence on slow global memory. These regions include register memory, constant memory and shared memory.

2.6.1 Registers

Registers are contained within a register file on each multiprocessor [87], and provide fast thread-local storage during kernel execution. Kepler and Maxwell architectures provide 65,536 registers per file, which are evenly divided between all active thread blocks executing on the multiprocessor [87]. Registers are typically accessed with zero added clock cycle overhead if register operations do not cause register bank conflicts, read-after-write dependencies issues, or register spills resulting from high register pressure [86].

Bank conflicts cause increased latency and cannot be directly avoided, as executing threads have no direct control over register allocation. The chances of avoiding bank conflicts can be improved, however, by ensuring that the thread block size is a multiple of 64 [86]. Read-after-write dependencies occur if a thread tries to access the value stored in a register within 24 clock-cycles of it being written (the register's update latency). Read-after-write dependencies increase register access latency, but can be partially or completely hidden if the warp scheduler can context switch between other warps in the block while waiting for the register update to complete [86].

Register spilling refers to the transparent use of high-latency local memory to supplement register storage. Under certain circumstances, a local variable may be stored in an automatic variable in global memory rather than in the register file. This occurs when the Nvidia CUDA compiler determines that there is insufficient register space to contain the variable, which may occur to save register space when storing a large array or data structure, or if the register file is fully exhausted [86]. While automatic variables are considered local memory, they are stored off-chip in device DRAM and thus incur the same access penalties as global memory. Kepler GPUs use the L1 cache exclusively for accelerating local memory accesses, and may substitute unused shared memory capacity for an increased L1 cache size to improve local memory performance [87].

2.6.2 Constant Memory

Constant memory is a small read-only region of globally accessible memory which resides in device DRAM [87]. In contrast to global memory, constant memory has only 64KB of storage capacity, but benefits from an 8KB on-chip cache which greatly reduces access latency [86]. While a cache-miss is as costly as a global memory read, a cache-hit reduces access time to that of a local register, costing no additional clock cycles at all as long as all active threads access the same memory index [86]. If active threads in a warp access different constant memory indexes, these requests are serialised, negatively impacting total performance [86]. The limited size of constant memory unfortunately prohibits its utilisation as a medium for storing large data collections such as packet sets, but it is well suited to storing device pointers, program directives, constant data structures and run-time constant variables.

2.6.3 Shared Memory

Shared memory is a multiprocessor local memory region that facilitates cooperation between multiple threads in an executing thread block [87]. Kepler devices provide a total of 64 KB of shared memory per multiprocessor and support three shared memory configurations: 16 KB, 32 KB and 48 KB [87]. The remainder of shared memory on Kepler multiprocessors is used for L1 caching. Maxwell devices do not use L1 caching; instead, the L2 cache stores both local and global

accesses, allowing the full capacity of shared memory to be used by executing kernels. Shared memory is divided evenly between all blocks executing on a particular multiprocessor, and as such is a limited resource [86].

Shared memory performance can be adversely affected by bank conflicts. These can occur if two threads in a warp access different 64-bit words from the same shared memory bank, and ultimately results in serialised access to the bank [86]. Kepler's shared memory is partitioned into 32 separate banks (each supplying 64 bits of bandwidth), organised such that successive 32-bit (or 64-bit) words map to successive banks. If multiple threads in a warp read values stored in the same 64-bit word, that 64-bit word is broadcast, avoiding serialisation. If multiple threads write to the same shared memory location, the write is only performed by one of the threads.

Shared memory was, with the exception of warp voting (see Section 2.7.1), the only facilitator of fast on-chip communication between threads in Tesla and Fermi devices, and the only method besides global memory to transfer integral and floating point values between threads in a block [87]. Kepler supports a faster mechanism, warp shuffling, that provides a more direct and efficient means of passing values between threads if they are contained within the same warp (see Section 2.7.2).

2.7 Inter-warp Communication

In addition to shared memory, which provides a shared memory space for threads executing in a single block, Kepler micro-architecture provides two mechanisms for inter-thread communication at the warp level. These include warp voting, which communicates predicate results between warp threads, and warp shuffling, which passes integral and floating point values. Warp voting and warp shuffling are implicitly synchronised, as threads in a warp cannot diverge.

2.7.1 Warp Vote Functions

Kepler provides three warp voting functions: `__all()`, `__any()` and `__ballot()`. The `__all()` and `__any()` functions were introduced with CC 1.2 devices, while

the __ballot() function was added to Fermi architecture. All functions take a 32-bit integer as input from each thread in the warp; from the perspective of these functions, zero values are taken as false while non-zero values are considered true.

The __all() function performs a logical conjunction between all integer inputs, producing a non-zero value if the resultant proposition is true, and a zero value if it is false. Similarly, the __any() function returns the result of a logical disjunction of input truth values. The __ballot() function, in contrast, provides more detailed results by returning the value supplied by each thread in the warp as a single bit in a 32-bit integer, thereby allowing any thread to review the results of any other thread in the warp.

2.7.2 Warp Shuffle Functions

Warp shuffle functions, introduced in Kepler micro-architecture, provide a low-latency alternative to shared memory for exchanging integral and floating-point values between threads (referred to as lanes) in a thread warp. Kepler includes four distinct source-lane addressing modes: __shfl(), __shfl_up(), __shfl_down(), and __shfl_xor(). Shuffle functions take only a single clock-cycle to service an entire warp [87], provide more direct transfers than shared memory, and consume no shared memory capacity. In addition, they provide a safe and future-proof means of synchronising threads, previously only achievable with expensive block-level synchronisation using __syncthreads() intrinsic function [81].

All shuffle functions take a value v to be transmitted and a width w (where w is a power of two and no greater than the warp size) as arguments, as well as one other argument that is specific to each addressing mode. Setting a width $w < 32$ creates $\frac{32}{w}$ distinct sub-grouping of w contiguous threads, where each sub-grouping is restricted to communications with other threads in its sub-group. Width is an optional argument, and defaults to 32 if omitted.

The basic shuffle function __shfl() takes an integral source lane (or thread) value as its third argument. When the __shfl() function is performed by a warp, each thread receives the value v contained in the thread specified by the source lane value. The __shfl_up() and __shfl_down() functions operate similarly, but take a delta argument rather than a source lane. These functions, respectively,

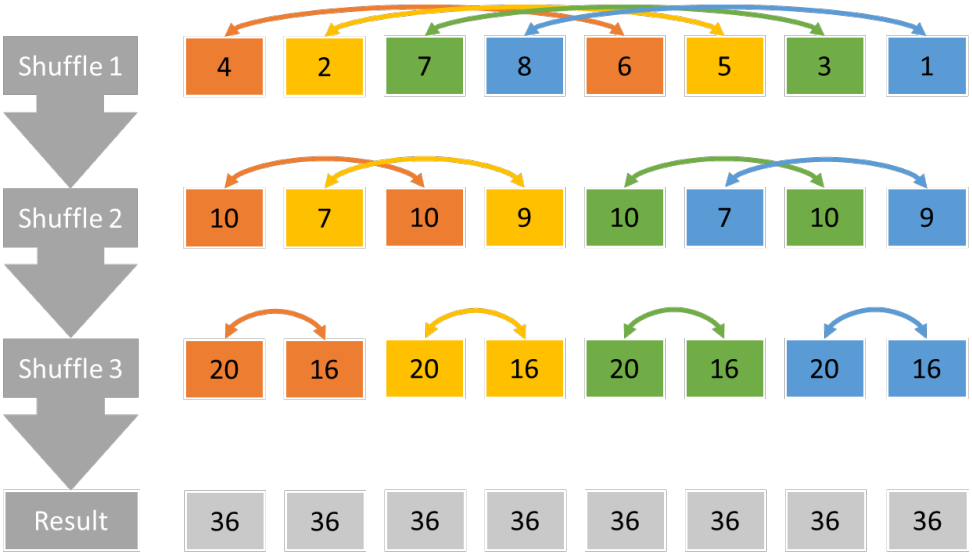


Figure 2.3: Summing eight elements with a butterfly reduction.

shuffle thread-specific v values up and down by delta lanes. Values shuffled out of the range of the thread warp (or subgroup) are ignored, while threads receiving shuffled registers from threads located outside the warp or subgroup simply receive their own value for v.

The final shuffle function, `__shfl_xor()`, takes a bitwise lane mask as a third argument, which is combined using a bitwise exclusive OR with the thread's lane ID to produce the source lane from which to copy v. The primary purpose of the `__shfl_xor()` function is to facilitate efficient parallel butterfly reduction using shared memory, which can be used to sum, multiply or otherwise combine register values across a warp [87]. Figure 2.3 outlines the basic concept behind butterfly reduction when shuffling a subgroup with a width of eight.

In the first iteration of the butterfly reduction, the warp (or group) containing 2^n elements is conceptually divided in two and summed together, leaving two identical sets of 2^{n-1} elements. These sets are then subdivided again and summed, each producing two identical sets of 2^{n-2} elements. This process continues recursively until the single set of 2^n elements is transformed into 2^n copies of a single element.

The `__shfl_xor()` function achieves butterfly reduction through a sequence of lane masks that effectively replicate this process. Each lane mask has only a single bit set so that when the lane mask is applied to the lane ID, the matching bit in the lane ID is effectively flipped. Inverting the n^{th} bit of a positive integer results in either (a) subtracting 2^n from the value if the bit was initially set, or (b) adding 2^n

Listing 3 Sum via butterfly reduction for a full warp using `__shfl_xor()`.

```
1 int sum = values[blockDim.x * blockIdx.x + threadIdx.x];
2
3 sum += __shfl_xor(sum, 16); //0x10000
4 sum += __shfl_xor(sum, 8); //0x01000
5 sum += __shfl_xor(sum, 4); //0x00100
6 sum += __shfl_xor(sum, 2); //0x00010
7 sum += __shfl_xor(sum, 1); //0x00001
8
9 int result = sum;
```

to the value if the bit was initially cleared. Using the example in Figure 2.3, which comprises eight (2^3) lanes, the first pass applies a lane mask of 0x100 (2^2) which sums elements four indexes apart. The second pass applies a lane mask of 0x10 (2^1) to the four resultant (and replicated) elements, summing elements two elements apart. After the third iteration of this procedure, every participating thread contains a copy of the final sum. The number of iterations scales logarithmically with respect to the number of lanes, so reducing a full warp of 32 threads requires only five passes. Listing 3 shows example CUDA code for summing across a full warp of threads using the `__shfl_xor()` function.

2.8 Performance Considerations

CUDA Kernels are highly sensitive to a wide array of factors which negatively impact efficiency. This section focuses on factors relevant to this research, and indicates how they may be avoided or capitalised upon.

2.8.1 Transferring Data

Memory transfer speed is limited by the bandwidth of the PCIe bus, which provides a total of either sixteen (PCIe 2) or thirty-two (PCIe 3) 1GBps channels. These are divided evenly between dedicated upstream and dedicated downstream channels, allowing for a maximum of 8GBps or 16GBps transfer in each direction respectively [86]. In comparison, the Kepler-based GTX Titan provides 288.4 GB/s bandwidth between device memory and device multiprocessors [80], over 17x more than the PCIe 3 bus. This vast difference in performance can result in a critical bottleneck if the device depends on and / or produces large quantities of data.

CUDA provides several mechanisms to mitigate the impact of memory transfers for applications which rely on large volumes of data, at the expense of host side memory resources. While host memory is typically allocated as pageable and may be cycled to disk to free up host memory, optimised transfers require page-locked memory allocations which must remain in host memory. Page-locked memory provides significantly higher bandwidth than pageable memory, and facilitates a number of additional optimizations, such as asynchronous concurrent execution, and write-combining memory. Excessive use of page-locked memory is not encouraged, however, as it is a scarce system resource; overuse can negatively impact host performance and application stability [86].

The data transfer rate to the device may be improved further by allocating page-locked memory as write-combining memory [86]. Page-locked memory is typically allocated such that it is cacheable by the CPU, which consumes system cache resources to accelerate host side access to the data. Memory allocated as write-combining is by contrast not cacheable, which frees L1 and L2 resources for use by other parts of the application. In addition, it prevents data snooping by the PCIe bus, which can improve transfers by up to 40% [86]. This comes at the expense of host-side read performance, which suffers due to the lack of caching. Write-combining memory is thus well suited for transfers to the device, as it is unlikely that the host would need to read this data after it has been written. It is less useful when transferring data from the device, as it is much more likely this data will be read by the CPU, either within subsequent calculations, or when being copied from host memory to a file.

Page-locked memory also supports mapped memory. Mapped memory eliminates the need for explicit memory copies between the host and device by allowing kernels to access host memory directly, typically through the PCIe bus [86]. Mapped memory is particularly useful on integrated GPUs which use the same physical memory as the host system, as mapped memory can be accessed directly rather than through the PCIe bus.

2.8.2 Streams and Concurrency

CUDA allows for both synchronous and asynchronous copies between host and device memory. Memory copies between host and device memory block until completion when transferred synchronously, but return control immediately after

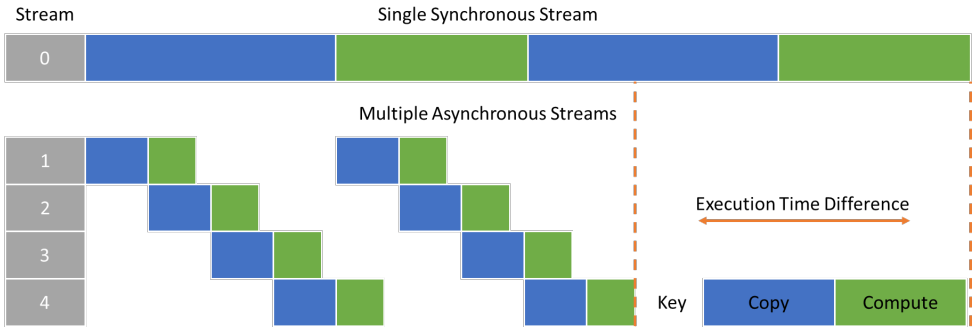


Figure 2.4: Synchronous execution versus asynchronous execution.

scheduling the copy transaction when performed asynchronously [87]. Asynchronous copies can be capitalised on to perform additional host side processing during the copy period, allowing kernels to be scheduled for execution prior to transfer completion. As contemporary CUDA devices support concurrent copy and compute, which allows kernel execution to overlap with data transfers, it is possible to begin executing a kernel on a subset of input data prior to the completion of the entire transfer process [86, 87].

CUDA applications facilitate concurrency by partitioning input data into separate and independent instruction streams. By default, CUDA schedules all events (including memory transfers and kernel calls) in the default stream, which process these events in series. In this configuration, memory transfers must fully complete before kernel execution can begin, as there exists no other context in which the kernel can execute. Creating additional asynchronous streams that each transfer and process a subset of data solves this, making concurrent copy and compute possible. These streams behave similarly to the default stream, with one significant exception: operations performed in the default stream are always scheduled in all streams, while operations performed in other streams are only visible within their specific stream. Each stream transfers a subset of data and then executes the kernel with stream specific input parameters [86], and can be scheduled such that while one stream is executing a kernel, another is transferring data. This can improve overall device utilisation and significantly reduce the time necessary to evaluate a kernel. Concurrent copy and compute is illustrated in Figure 2.4.

2.8.3 Occupancy

Multiprocessor occupancy is a measure of GPU resource utilisation; it is the ratio of active warps on the multiprocessor to the maximum number of possible active warps. When occupancy is high, the multiprocessor is able to context switch to ready warps to hide the access latency in the executing warp. If occupancy is too low, the multiprocessor is not able context switch to other warps and must remain idle until the operation completes or the resource returns. Occupancy is affected by the thread block size, in combination with register allocation and shared memory utilisation.

While multiple thread blocks may execute concurrently on a single multiprocessor, multiple multiprocessors cannot divide a single thread block between them. This derives from the requirement that on-chip shared memory be accessible to all threads executing in the current block, which would not be possible if a block's shared data was distributed between multiple multiprocessors. With respect to Kepler architecture, if the requested thread block size is not a factor 2048 (the maximum supported thread count per multiprocessor of this compute capability), then a proportion of the multiprocessor's processing resources must be left idle. Kepler supports block sizes that are a multiple of the warp size, from 32 threads (1 warp) up to a maximum of 1024 threads (64 warps). Kepler multiprocessors are limited to a maximum of 16 active blocks at a time however, and thus block sizes below 128 threads also result in lost occupancy, regardless of other factors. By comparison, Maxwell supports 32 active blocks per multiprocessor, and thus can process block sizes as low as 64 threads without losing occupancy. Figure 2.5 expresses this diagrammatically.

Block size is not the only kernel attribute which impacts occupancy, with shared memory and register utilisation being of similar importance. Each multiprocessor contains a finite amount of on-chip shared memory and register storage, which must be shared between all active blocks running on the multiprocessor. Kepler multiprocessors support up to 48KB of shared memory and 65,536 registers on each multiprocessor. If each block in a hypothetical kernel utilised 16KB of shared memory, only three blocks could be active at once regardless of block cardinality. Similarly, if each block utilised 16,384 registers, then only four blocks could be active on each multiprocessor. To achieve full occupancy, with all 64 warps active, each thread must consume at most 32 registers and 24 bytes of shared memory.

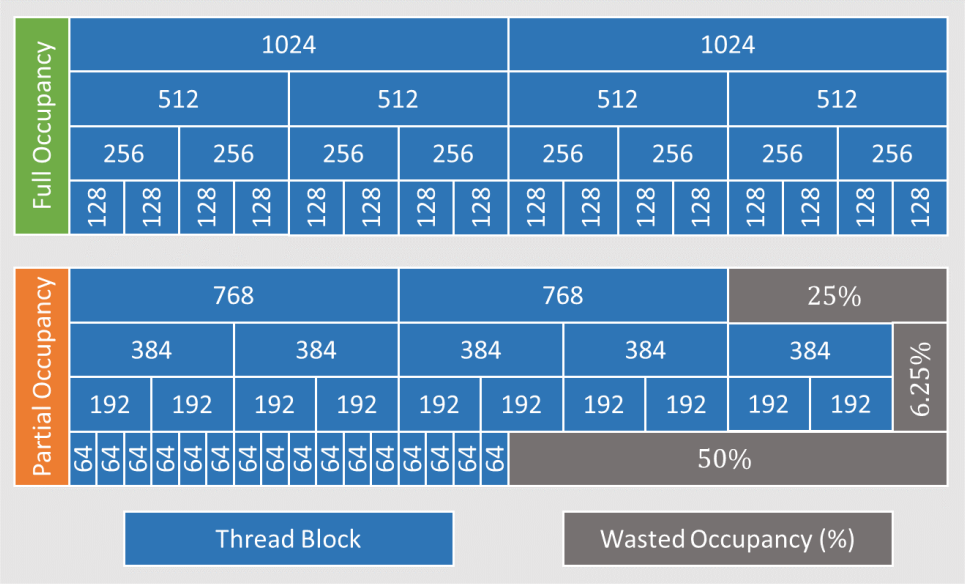


Figure 2.5: Affect of thread block sizes on Kepler multiprocessor occupancy.

While maintaining high occupancy is highly advisable, it must be balanced with memory bandwidth and other factors to ensure the best possible performance. This is illustrated by Figure 2.2 (provided in Section 2.5.5) which shows that when device memory access is not coalesced, blocks with a cardinality of 64 (which result in only 50% occupancy) consistently outperform their full occupancy counterparts by a significant margin. This is a symptom of device memory bandwidth oversaturation. Specifically, the thread block containing 64 threads requires half as much bandwidth to service all executing warps. This reduces contention for device memory resources and bandwidth congestion, resulting in better performance.

2.8.4 Instruction Throughput

GPU performance is sensitive to control flow instructions (`if`, `switch`, `for`, `do`, `while`) and certain arithmetic functions (division and modulus), and can be significantly impeded if these instructions are used excessively or inefficiently.

Decision based operations (`if`, `switch`) work efficiently when all threads in a warp agree on the branch condition, but may cause severe divergence and serialisation if threads follow a wide variety of distinct and non-trivial branches [86, 125]. When this happens, each divergent path is processed sequentially while threads following other paths remain inactive. As a result, substantial divergence between threads

can significantly impair GPGPU parallel processing performance [86]. When decisions which may potentially cause divergence are unavoidable, branches should be as short and infrequent as possible to minimise serialisation overhead and increase the possibility that they will be replaced with branch prediction predicates by the CUDA compiler [87].

Control flow functions that facilitate iteration are similarly expensive on GPUs due to branching and control overhead. Where possible, it is often desirable to fully or partially unroll loops to eliminate this unnecessary overhead and improve efficiency. Unrolling may be performed automatically by the compiler (if possible) or manually by the kernel designer. While often effective, unrolling loops is not guaranteed to improve efficiency as it often trades control logic for higher register utilisation, increasing register pressure (see Section 2.6.1) [87].

Finally, integer division and modulo operations are significantly more expensive to compute than other arithmetic functions [86], but can be replaced by far more efficient bit-shift and bitwise operations if the divisor or modulus is a power of two. Specifically, if $k = 2^n$ where $n \geq 1$, then x/k is equivalent to $x \gg log_2 k$, and $x \,\%\, k$ is equivalent to $x \,\&\, (k-1)$ [86].

2.8.5 Device Memory Allocation

A final performance consideration is how device memory is allocated. The CUDA 6 API supports dynamic memory allocation from within a kernel, allowing individual threads to allocate and deallocate their own memory containers within global memory at runtime, providing an alternative to pre-allocation of device buffers on the host [87]. Dynamic allocation is expensive however, and can quickly cripple the performance of a kernel if not used sparingly. Dynamic memory is thus most useful in combination with dynamic parallelism, a feature which allows executing kernels to launch additional kernels and thereby create more work for the GPU without dependence on the host process [87]. Dynamic memory allocation and kernel execution are not within the scope of this research, but could be considered and incorporated at a later stage.

2.9 Summary

This chapter discussed GPGPU domain, Nvidia GPU micro-architectures and the Nvidia CUDA API, focussing specifically on the features of Kepler and Maxwell micro-architectures used during the course of this research. The GPGPU domain was introduced in Section 2.1, which summarised the history of GPUs and the GPGPU paradigm, introduced the CUDA API, and listed some of the benefits and drawbacks associated with the domain.

Section 2.2 provided a brief summary of CUDA micro-architectures, from the Tesla micro-architecture originally introduced in late 2006, to the as-yet unreleased Pascal micro-architecture expected to arrive in 2016. This section highlighted the major changes that occurred between early and contemporary micro-architectures, and explored three significant new features (3D memory, NVLink, and fast unified memory) included in Pascal micro-architecture. Collectively these features are expected to improve device memory throughput, accelerate host-device communication, and eliminate the need to stage explicit copies between host and device memory.

Section 2.3 explored the CUDA programming model. The section introduced the concept of kernels and device side functions, and explained how threads are organised into grids, thread blocks and warps to express parallelism. The section concluded with an overview of how kernels are launched from a host thread.

Section 2.4 introduced Kepler micro-architecture, and provided an overview of the Kepler SMX multiprocessor.

Section 2.5 addressed the bottleneck presented by global memory, and introduced both coalescing and the various caching mechanisms available to improve device memory throughput.

Section 2.6 described the on-chip memory shared, constant and register memory regions, which provide fast, low-latency storage but limited capacity.

Section 2.7 described two alternative mechanisms for fast inter-warp communication: warp voting and register shuffling. Warp vote functions allow threads in a warp to communicate and make decisions based on their collective states, while register shuffle operations allow threads to pass registers directly and efficiently to other threads in a warp.

Section 2.8 concluded the chapter by elaborating on several performance considerations which can heavily impact achievable performance. This section first described current mechanisms for accelerating transfers between the host and device, and how transfers and computation can be overlapped through asynchronous and concurrent execution. The section also discussed device occupancy and its relationship to achievable performance, and briefly summarised more efficient workarounds for inefficient division and modulo operations.

The following chapter addresses background relevant to the networking component of the performed research, in preparation for a more detailed discussion of packet classifiers in Chapter 4.

3

Packets and Captures

THIS chapter introduces background information on network packets, protocols, captures and capture processing. These topics are foundational to the performed research, and provide context for the discussion of packet classification in Chapter 4. The following summarises the sections in this chapter:

- Section 3.1 provides a brief high-level introduction to packets and protocol headers.
- Section 3.2 introduces the TCP/IP (Transmission Control Protocol/Internet Protocol) model for network communication, and describes how the model is used in the transmission of payloads across networks.
- Section 3.3 introduces packet captures, particularly focussing on the structure and limitations of the pcap capture format typically used to store network traces. These files form the input medium for the classification system discussed in Part III, and are thus important to understand.

- Section 3.4 discusses the PcapNg (Pcap Next Generation) capture format which aims to address the limitations of the original pcap format. This section is included primarily for the sake of completeness, as the PcapNg format is currently a work in progress and is not yet fully supported by capture processing applications [132].
- Section 3.5 discusses the pcap APIs (Libpcap and WinPcap) which facilitate the creation, filtering and parsing of capture files on *nix and Windows systems respectively. The section discusses the history of these APIs and how they are used to interface with capture files.
- Section 3.6 provides an overview of the capture analysis functionality of the protocol analyser Wireshark, and discusses its performance with respect to large capture analysis.
- Section 3.7 concludes the chapter with a summary.

3.1 Packets

Digital communication networks connect local and/or geographically distributed hosts, which interact with one another via discreet message frames called packets. Packets encapsulate a context (or application) specific payload – an HTTP (HyperText Transport Protocol) [24] web page, DNS (Domain Name System) [60] query or ARP (Address Resolution Protocol) [97] request, for example – within one or more protocol headers. These headers are used to guide the transmission of the packet within and across network boundaries. Packets are relatively short segments of binary data and can be recorded with relative ease, making it possible to capture all dynamic communication to and from a host (or network) over an particular time delta into a static file – known as a capture or trace file – for detailed and repeatable analysis.

3.1.1 Protocol headers

A packet comprises a set of ordered, application-specific transmission directives called protocol headers, followed by a segment of application-specific transmission data [112]. Each successive protocol header is contained within the payload section

of the previous protocol header [112], with most packets comprising at least four distinct protocol layers.

Within the context of modern networking, a protocol is an established set of rules and specifications for communicating and receiving information associated with a particular function, application or service. Networks utilise a wide variety of system and application level protocols of varying complexity to service network transactions, each tailored for a specific purpose. Common protocols include the Ethernet protocol, the Internet Protocol (IP) [101], the Internet Protocol Version 6 (IPv6) [105], the Transmission Control Protocol (TCP) [102], the Internet Control Message Protocol (ICMP) [98], the User Datagram Protocol (UDP) [100], the File Transfer Protocol (FTP) [99], among many others.

Conceptually, packet headers can be thought of as a hierarchy or stack. Each level in the stack is associated with a different type of service, and the stack is organised such that each layer receives services from the layer directly below it and provides services to the layer directly above it [112]. The TCP/IP model, for instance, is represented as four broad layers [13], whereas the Open Systems Interconnection (OSI) model [59] is divided into seven layers. These models are discussed in the following section.

3.1.2 Packet Size and Fragmentation

The volume of data contained in a particular packet depends on the purpose of the packet, and may vary greatly from simple host name resolution requests to large segments of multimedia data. As a result, packet lengths may vary significantly, ranging from tens to thousands of bytes. Packet sizes are ultimately limited by the transportation mediums Maximum Transmission Unit (MTU) that prescribes the maximum size a particular packet type may be [61, 101]. For example, the Ethernet II protocol specifies an MTU of 1500 bytes, while the FDDI (Fiber Distributed Data Interface) protocol specifies an MTU of 4352 bytes [61]. Protocols such as IP [101] allow payloads which exceed the MTU to be divided over multiple packets, termed fragments. Fragments may be reconstituted into a single payload by the receiving host on arrival, achieved through inspection of fragment-related information contained within each protocol header [112].

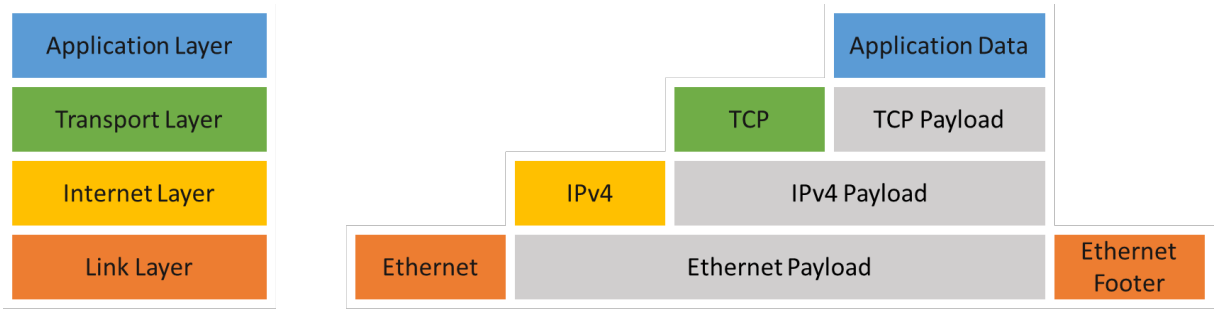


Figure 3.1: TCP/IP decomposed into its abstract layers.

3.2 The TCP/IP Model

TCP/IP, also known as the Internet Protocol Suite [13, 112], is a network communication model which organises protocols based on the services they provide. The TCP/IP model is leveraged in the transmission of the majority of contemporary network traffic and has been pivotal to the success of the Internet. Although not an explicit design choice, TCP/IP may be viewed as a four layer stack consisting of the Link Layer, Internet Layer, Transport Layer and Application Layer [13, 112]. A high-level overview of the structure of a TCP/IP packet mapped onto these four layers is provided in Figure 3.1.

The majority of this section briefly describes each layer of the TCP/IP stack in greater detail, establishing their roles in facilitating payload transmission. The section concludes by briefly discussing the OSI model, an alternative communication model which employs seven layers instead of four [59]. This model is described through comparison to the TCP/IP model in order to highlight their similarities and differences.

3.2.1 Link Layer

The link layer is responsible for preparing packets for dispatch and for the physical transmission of packets to a remote host or the next-hop router [13]. This layer is only responsible for delivering a packet to the next router or host in the chain, and it is up to the receiving interface to direct the packet on to a router or host closer to the transmission end-point. This process is repeated by each node in the chain, until such time as the packet arrives at its destination. To achieve this, a frame header is added to the packet, containing information relevant to the delivery of the

packet to the target host or the next-hop router over the specified network medium. This header is updated by each node in the path as the packet is transmitted to its destination. The Link Layer is associated with protocols which support this physical transmission, such as Ethernet II or WiFi (802.11 a/b/g/n etc.) [13].

3.2.2 Internet Layer

The Internet layer, located directly above the Link Layer in the TCP/IP stack, is responsible for the delivery of packets between end-points in a transmission [13, 112]. The Internet Layer's functionality is contained within an IP or IPv6 protocol header [101, 105]; facilitating logical, hierarchical end-point addressing through IP addresses, and enabling packet routing by specifying the terminal node in the transmission. The Link Layer uses the address information encapsulated in IP, in conjunction with routing tables, to derive the physical address of the next network interface between the sending and receiving host. In this way, the Link Layer provides a service to the Internet Layer by determining the delivery route a packet navigates to arrive at its remote destination, and transmitting it along that route. IP has two widely used implementations, namely IP version 4 (IPv4), which supports just over four billion 32-bit addresses [101], and IP version 6 (IPv6), which uses 128-bit addresses that provide roughly 3.4×10^{38} unique address values [105].

3.2.3 Transport Layer

The transport layer is entirely independent of the underlying network [13, 112], and is responsible for ensuring that packets are delivered to the correct application through service ports. Transport layer protocols in the TCP/IP suite include TCP and UDP [13, 100, 102].

TCP is a connection-orientated protocol that addresses transmission reliability concerns [102, 112]. It includes functionality to:

1. Discard duplicate packets.
2. Ensure lost or dropped packets are resent.
3. Ensure packet sequence is maintained.

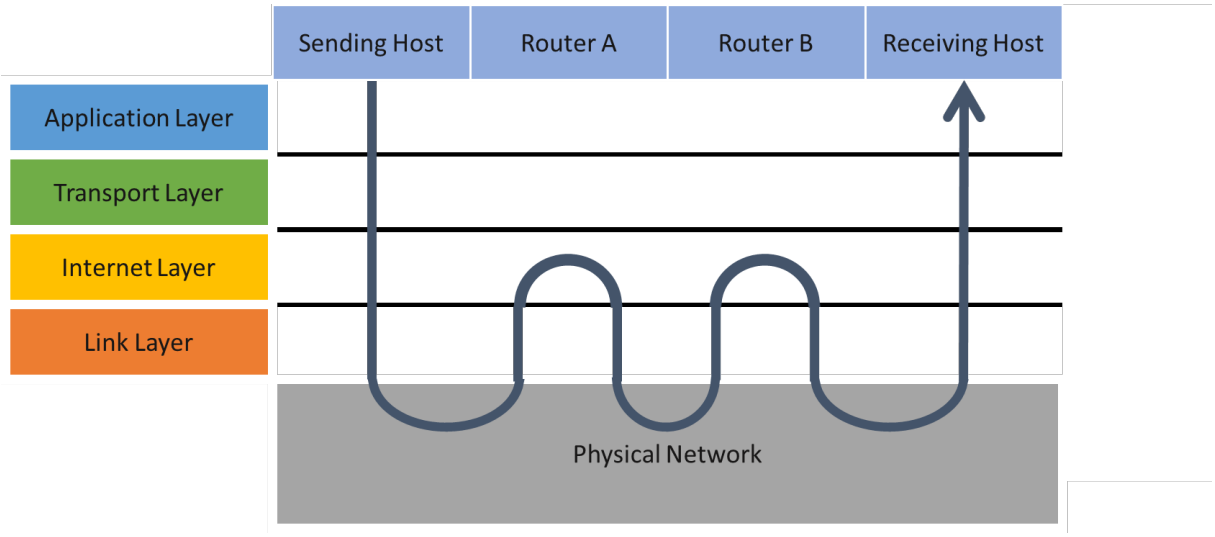


Figure 3.2: Stack level traversal when transmitting a packet to a remote host using the TCP/IP model.

4. Check for correctness and corruption through a 16 bit check-sum.

In contrast, UDP is a connectionless protocol which provides only best-effort delivery and weak error checking [100]. Unlike TCP, UDP sacrifices reliability for efficiency, making it ideal for applications such as DNS look-ups, where the overhead necessary for maintaining a connection is disproportionate to the task itself [112].

Both TCP and UDP define two 16-bit service ports, namely Source Port and Destination Port, which are used to determine which application a particular packet should be delivered to. As has been noted, both TCP and UDP are network agnostic, and leave network related functionality to lower layers in the protocol stack [100, 102].

3.2.4 Application Layer

The top-most layer in the TCP/IP stack is the application layer, which simply encapsulates the data to be delivered to the waiting application. This data may itself contain further application specific headers, which are handled by the receiving process. The packet is terminated by the frame footer, associated with the link layer, which delimits the packet and provides error checking functionality.

Application payloads dispatched to remote hosts conceptually descend the TCP/IP stack, which applies relevant headers to the payload at each level [13]. Figure 3.2 illustrates this process for a payload transmitted from a sending host to a distant receiving host via two routers.

The application layer header is applied to the data payload first, followed by the transport layer header, Internet layer header, and finally link layer header. Once all headers have been applied, the packet is transmitted to the next-hop router, Router A. Router A receives the packet and, using information contained in the Internet layer header and routing tables, determines the shortest path to the receiving host. It then re-sends the packet with a new link layer header, destined for Router B. Router B repeats this process, delivering the packet to the Receiving Host. The payload is then extracted and delivered to the waiting application by ascending the stack, removing headers at each layer.

3.2.5 The OSI Model

The OSI model, a product of the International Organisation for Standardisation, is a seven layer standard model for network communication that provides an alternative to TCP/IP.

Layering in the OSI model is both explicit and an integral part of the model's design, with each abstract layer tasked with providing specific services [59]. As a result, the OSI model can be generally applied to the design of arbitrary protocols. This differs from the TCP/IP model, where layers map to specific protocol groups in the TCP/IP suite. In practice, the OSI model's seven explicit layers are functionally quite similar to the four general layers in the TCP/IP model, and provide the same basic services. Due to this inherent similarity, it is possible to outline the OSI model in terms of the TCP/IP model.

The seven layers defined by the OSI model, from lowest to highest, are the Physical Layer, Data-Link Layer, Network Layer, Transport Layer, Session Layer and Application Layer [59]. The physical and data-link layers in the OSI model are essentially encapsulated by the link layer of the TCP/IP stack, breaking it down into two distinct processes; physical transmission and packet framing. The OSI network layer is roughly equivalent to the Internet layer of the TCP/IP model, although there is some overlap with the TCP/IP link layer. Similarly, the OSI transport layer, and a small subset of the session layer, are contained within the TCP/IP

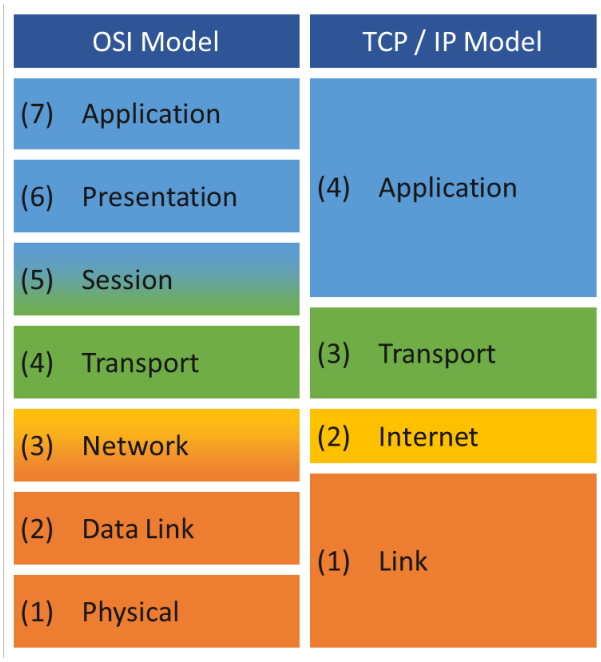


Figure 3.3: Layer comparison between the OSI and TCP/IP models.

transport layer. The remainder of the OSI session layer, as well as the presentation and application layers, are encapsulated by the TCP/IP application layer. An illustration of this breakdown is provided in Figure 3.3.

While not as prevalent as the TCP/IP model, the OSI model is widely used outside of the TCP/IP suite. In particular, the OSI model provides the foundation for the IEEE 802 standards for local and metropolitan networks, including IEEE 802.3 Ethernet and 802.11 Wireless LAN (Local Area Network) protocols.

As the scope of this research is restricted to protocols in the IP suite, future discussion refers specifically to the TCP/IP model. Due to their similarities however, discussion referring to the TCP/IP model applies generally to the OSI model as well.

3.3 Pcap Capture Format

Packet captures, or traces, are a static record of otherwise transient packet data that are typically captured at a specific network interface. They may span from a few seconds to several years of collected traffic, and can range in size from a few

kilobytes to hundreds of gigabytes or more depending on the capture's composition. In general, capture size is primarily affected by three factors, which may be generalised to the following:

1. Average packet size, in bytes.
2. Average packet arrival rate, in packets per second.
3. Capture period, in seconds.

The size of a capture is determined by the product of these three metrics, which may differ greatly from capture to capture depending on a variety of factors, including the network interface, network environment and application context.

Packet captures come in a variety of open and proprietary formats, with the pcap dumpfile format being the most common general and open format in use [51, 131]. Most capture types follow a similar storage model to that of pcap, encoding packets as a sequence of variable length records with static length headers. The pcap format is still in wide use, but lacks many desirable features that limit its utility in certain domains. For instance, it lacks the ability to store packets from multiple interfaces, store meta data, or break large collections of packets into smaller subsections. A newer candidate format intended to address these limitations – PcapNg – has been in development since 2004 [47], but has yet to be completely finalised. PcapNg has recently risen in popularity due to its extended feature set, and is currently used as the default capture format in recent builds of the Wireshark protocol analyser [132] (see Section 3.6), although support for the format is only partial, and the specification still lacks many key features [132]. The PcapNg capture format is described in Section 3.4.

The pcap dumpfile format originated as part of the Libpcap[1] *nix library, and stores packets as byte-aligned arrays which are concatenated to form an inline linked-list, delimited by header records [51]. This configuration supports arbitrary packet sizes without the need for padding, and allows new packets to be appended to capture files without needing to update existing records. The pcap file structure, illustrated in Figure 3.4, begins with a 24 byte global header detailing information specific to the entire capture, which is followed by a list of packet records.

[1] `http://www.tcpdump.org/`

Table 3.1: PCAP dumpfile global header fields [51].

Header Field	Byte Offset	Size (bytes)	Signedness
Magic Number	0	4	Unsigned
Version Major	4	2	Unsigned
Version Minor	6	2	Unsigned
This Zone	8	4	Signed
Significant Figures	12	4	Unsigned
Snapshot Length	16	4	Unsigned
Network	20	4	Unsigned

3.3.1 Global Header

The global header is a 24 byte segment at the beginning of the capture file. It contains seven integral values that identify the format of the capture and provide global context information to each of its records. A list of fields in the global header is shown in Table 3.1 and summarised below.

Magic Number A constant 32-bit integer value – `0xA1B2C3D4` or `0xA1B23C4D` in hexadecimal notation for millisecond and nanosecond resolution files respectively – that both identifies the file as a pcap capture and indicates the byte order of the encoded data. If the byte order of the value is reversed when read by an application (i.e. `0xD4C3B2A1` or `0x4D3CB2A1` respectively), it indicates to the application that the capture is encoded using a different byte order than the host system, and all subsequent global and record header fields must be swapped as well [51].

Version Major/Minor Short integer values that specify the major and minor version numbers of the file format. The current pcap file format (version 2.4) has remained unchanged since 1998 and thus this field may typically be ignored [51].

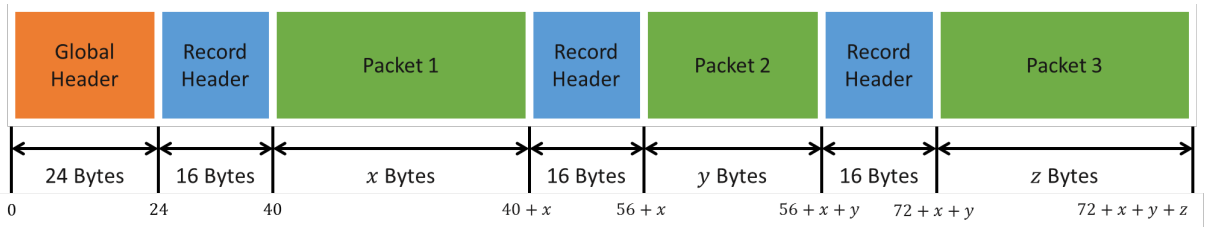


Figure 3.4: Structure of a pcap capture file.

This Zone A signed integral value provided to indicate the adjustment (in seconds) from Greenwich Mean Time (GMT) to the local time zone where the capture was created. This field is not used in practice as timestamps are always in GMT format [51].

Significant Figure An unsigned integer intended to indicate the accuracy of timestamps. The field is ignored in practice [51].

Snapshot Length An unsigned integer field (also referred to as the snap length) that specifies the maximum amount of data (in bytes) that may can be stored in each record body [51]. Intercepted packets that exceed this length are cropped to this size on storage. This value is set to 65535 or greater by default (well above typical MTU values) to ensure that all packets are captured in their entirety [51]. Smaller values may be used to reduce storage and I/O overhead, or as a simple data privacy measure to remove or partially erase sensitive payloads.

Network An integer identifier that indicates the physical (or link) layer protocol of each packet in the capture [51]. This value is included in the global header rather than the record header as all packets are assumed to arrive at a single interface, and thus the link layer protocol is assumed to be the same for all packets in the capture. There are over 80 official link layer protocols defined, as well as 15 values reserved for private use. A complete list of official link header values may be found at [116]. This value is set to 1 for Ethernet interfaces, and to 105 for IEEE 802.11 Wireless LAN interfaces [116].

3.3.2 Record Header

The global header is followed by an arbitrary number of individual packet records. Each packet record contains a 16 byte record header followed by the raw packet byte array. A list of fields in the global header is shown in Table 3.2 and summarised below.

Timestamp Two 32-bit integers that indicate the arrival time of the packet as, respectively, second and sub-second components [51]. The magic number in the global header is used to determine the sub-second component, and is stored in microseconds or nanoseconds.

Table 3.2: Pcap dumpfile record header fields.

Header Field	Byte Offset	Size (bytes)	Signedness
Timestamp (s)	0	4	Unsigned
Timestamp (μs/ns)	4	4	Unsigned
Included Length	8	4	Unsigned
Original Length	12	4	Unsigned

Included Length The number of bytes of packet data actually included in the record data [51]. This value should always be less than or equal to the snapshot length defined in the global header. This field is of critical importance to record navigation as the location of record n in the capture file is found by offsetting the location of record $n-1$ by $IncludedLength(n-1)+16$ bytes.

Original Length A value indicating the original size of the packet (in bytes) when it was originally received [51]. This field may be used in conjunction with the included length field to determine if a packet has been partially cropped.

While this record format simplifies the process of appending records to capture, it provides no mechanism for random record access and thus requires serially parsing the capture one record at a time for navigation. Specifically, access to the record of the n^{th} packet requires sequentially reading and parsing all $n-1$ prior record headers. Sequential access has little impact when processing small files, but the process can become a time-consuming, resource intensive process when attempting to access later records in captures spanning hundreds of millions (or billions) of packets.

3.3.3 Format Limitations

Due in part to its structural simplicity and open nature, the pcap dump file format has remained in widespread use without alteration for well over a decade. In particular, the format is compact, easy to parse and efficient to write. In spite of this, it has a number of general and application-specific limitations when applied in a modern context. The following list enumerates four of the more significant limitations:

1. Serial access to records scales poorly to large captures.

2. Captures can only store packets for a single network interface [47].
3. The format does not facilitate storage of additional types of relevant network environment information, such as resolved host names [47].
4. There is no mechanisms to annotate captures or individual packets with additional derived or context specific information, such as packet drop counts, runtime metrics or comments [47].

These key limitations motivated the need for a modern alternative to the format to address the complexity of modern network environments. This resulted in the development of the PcapNg format, which is described separately in Section 3.4.

3.3.4 Storage Considerations

One of the most problematic aspects of accessing packet records stored in large capture files is the bandwidth limitations of low-cost, non-volatile and high capacity storage such as HDDs and SSDs. For instance, typical large (TB+) 7200RPM HDDs can typically sustain throughputs between 100 MB/s and 130 MB/s, roughly comparable to a 1 Gbps network interface. SATA3 SSDs provide higher throughputs of between 450 MB/s and 550 MB/s, but are significantly more expensive per MB of storage. Applications processing captures stored on and accessed from such hardware cannot exceed this throughput, regardless of other infrastructure.

One of the most commonly utilised approaches to mitigate the slow performance of large disk drives is to connect multiple drives into a RAID (Redundant Array of Independent Disks) configuration [3]. A RAID array consists of multiple drives connected to a hardware or software based RAID controller, which interleaves (or stripes) data and recovery information across each drive [19]. RAID arrays have two primary benefits relating to speed and data security; they improve read and write throughput by distributing work across multiple drives, and also provide fault tolerance and data recovery through recovery sectors and data redundancy, which may be used to reconstruct the data if a component drive fails[19].

Different RAID configurations support different recovery and performance benefits. The simplest of these are RAID 0 and RAID 1; RAID 0 only includes fine-grained striping, and is used primarily to improve performance [19], while RAID 1

mirrors its contents completely over two or more disks to provide redundancy [95]. RAID 0 provides greater capacity and performance (n times that of the smallest and slowest drive, for n drives respectively), but provides no means of recovery if a drive fails. RAID 1, on the other hand, provides no performance benefit and assumes the capacity of the smallest drive, but is fully recoverable as long as at least one drive remains. There are many other standard and non-standard RAID configurations which provide different balances of performance, reliability, recoverability and capacity; these are, however, outside the scope of this research, and will not be explored in this document.

3.3.5 Indexing

Indexes are supporting files generated through inspection of captures, and are used to improve searching and analysis efficiency at the expense of additional storage space. Capture indexes store the offsets of each packet in the capture, as well as more specific information such as the contents of common fields, so as to avoid re-parsing information from inefficiently organised raw packet records [26, 46, 74].

These records are often compressed and stored in efficient data structures, allowing the contents of packets to be searched and analysed with far greater speed and efficiency than can be achieved unaided. In some cases, and with enough fields indexed, index files can effectively replace the raw capture as the primary information source for high-level analysis and searching. Indexing approaches use memory efficient data structures and encoding to compress generated files for storage, in order to minimise the potentially substantial costs of archiving a wide assortment of field values [26]. Compression requires some measure of processing to achieve, and thus inefficient compression can limit indexing throughput. Similarly, compressed data formats which are expensive to extract limit the achievable throughput for processes which apply indexed data. Capture indexers thus focus on highly-compressed data formats that are computationally inexpensive to read and write.

Examples of packet indexers include pcapIndex [26] and PcapWT (Pcap Wavelet Tree) [46]. PcapIndex is an extension to Libpcap which uses compressed Bitmap Indexes (BIs) to archive field data, providing fast insertion rates and compact indexes [26]. Later work used the CUDA Thrust API[2] to generate BIs using GPU

[2] `http://code.google.com/p/thrust/`

hardware, achieving a 20 fold improvement in indexing throughput over its CPU counterpart [27]. PcapWT provides another alternative approach, encoding index data using wavelet trees [46]; versatile data structures based on binary trees which facilitate efficient representation and querying of sequences [63].

3.4 PcapNg Format

The Pcap Next Generation Dumpfile Format (or PcapNg) is an attempt to address many of the limitations of the original pcap format, particularly those listed in Section 3.3.3. The PcapNg format is a modular approach to encoding packet records; it greatly improves flexibility, and provides mechanisms for extensibility at the expense of simplicity. This added complexity has had a dramatic impact on widespread adoption of the format; while the original draft specification for PcapNg was made available in 2004, it remains a somewhat experimental work in progress in 2015 [47, 132]. The format has however become more prevalent since its adoption in 2012 as the default capture format in Wireshark[3] and TShark 1.8.0 [132], and should eventually replace pcap as the de facto open format for storing captures.

3.4.1 Blocks

PcapNg files are composed of a series of blocks that partition information relevant to a specific aspect of the capture. PcapNg currently defines six standard block types, and identifies seven additional experimental block types that have yet to be formally defined [47]. In addition, the format supports arbitrary user-defined block types that provide extensibility beyond the basic types defined in the specification. All blocks share the same basic structure; two header fields and a footer field encapsulating a type-specific body, followed by zero or more options. Each block begins with two 32-bit header fields containing the block's type identifier and length (in bytes), and ends with a reiteration of the block length. This allows for fast forward and backward navigation between blocks, which in turn helps to accelerate access to records not positioned near the beginning of the capture. This allows irrelevant or unsupported blocks to be skipped by individual applications [47].

[3] www.wireshark.org

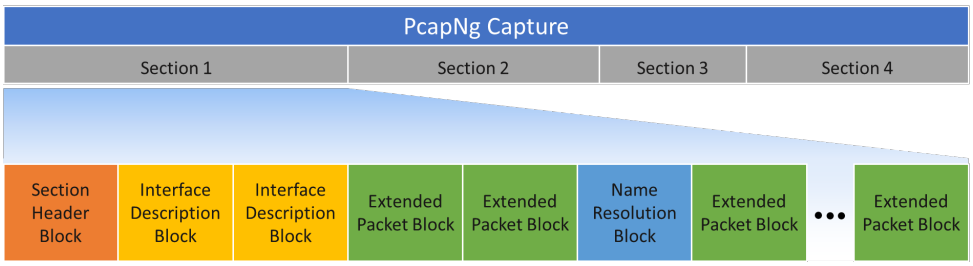


Figure 3.5: PcapNg file structure.

All blocks can include any number of options, which are specified at the end of the block's body. Options consist of a 16-bit Option Code and and 16-bit Option Length fields, followed by an option specific body. Most blocks define a set of block specific optional fields, but all blocks support Comment and EndOfOptions types. The former allows user comments to be appended to any block, while the latter indicates that no further options are included. Options can be skipped by moving to the next block, as they are included within the Block Length field. Figure 3.5 shows an overview of the PcapNg file structure, with blocks organised into multiple sections.

3.4.2 Mandatory Blocks

Blocks are arranged into one or more sections, each headed by a Section Header block [47]. PcapNg's Section Header block serves a similar purpose to pcap's Global Header, providing the major and minor version numbers and the magic number so that the file can be parsed correctly. Additional Section Headers can be added to a capture to form multiple discreet and self-contained sections. Each Section header includes a Section Length field to allow fast navigation between sections.

The remaining fields in pcap's Global Header have for the most part been relocated to the Interface Description block type, facilitating multiple interfaces per section by decoupling interface information from the header. This allows multiple interfaces to be defined within a single section. The Interface Description block contains a 16-bit Link Type field (equivalent to the Network field in pcap's Global Header) and a 32-bit SnapLen field; additionally, it supports 14 different optional field types, including fields for timestamp resolution and time zone correction previously contained in the Global Header. As with Section Header blocks, Interface Description blocks are mandatory and each Section must define at least one, as re-

Bit Offset	Section Header Block	
0	Block Type = 0x0A0D0D0A	
32	Block Length	
64	Magic Number	
96	Major Version	Minor Version
128	Block Type	
160		
192	Options (variable)	
224+	Block Length	

Bit Offset	Interface Description Block	
0	Block Type = 0x00000001	
32	Block Length	
64	LinkType	Reserved
96	SnapLen	
128	Options (variable)	
160+	Block Length	

Figure 3.6: PcapNg Mandatory Blocks

cords cannot be interpreted without the information they contain. The remainder of block types are optional.

The basic structure of the section header block and interface description block are shown in Figure 3.6.

3.4.3 Optional Blocks

PcapNg, at present, officially defines four optional block types that are not required to appear in a capture file; these include Simple and Enhanced Packet blocks, Name Resolution blocks and Interface Statistics blocks. Simple Packet blocks are designed for efficient reading and writing, and are minimalistic as a result; they include only a Packet Length field and the packet data, with no block specific optional fields defined. Unlike pcap Record Headers, time stamp information is not captured in Simple Packet blocks, as they are not required by all applications and can be expensive to collect [47].

The Enhanced Packet block is more robust than the Simple Packet block. In addition to the packet's original length, the block stores the Interface ID corresponding to a particular Interface Description block, a 64-bit timestamp field, and a captured length field to identify cropped packets. Enhanced Packet records also support three block specific optional fields for storing link-layer flags, hashes of packet

Bit Offset	Simple Packet Block
0	Block Type = 0x00000003
32	Block Length
64	Packet Length
96	Packet Data (variable)
128+	Block Length

Bit Offset	Enhanced Packet Block
0	Block Type = 0x0000006
32	Block Length
64	Interface ID
96	Timestamp (High)
128	Timestamp (Low)
160	Captured Length
192	Packet Length
224	Packet Data (variable)
	Options (variable)
256+	Block Length

Figure 3.7: PcapNg Packet Container Blocks

data, and the number of dropped packets between the current and previously recorded packets. Only enhanced packet blocks can be present in sections with multiple Interface Definition blocks, since they provide an Interface Identifier [47].

The remaining two fully defined optional types store meta-information rather than traffic itself. The Name Resolution block records one or more resolved host names for each of a range of IPv4 and IPv6 addresses. The Name Resolution block provides a means to store which named hosts a particular address corresponded to at the time the packet was captured; this is useful as IP address allocations for a particular host may change over time [47]. Similarly, the Interface Statistics block stores time-stamped global statistics for a particular interface, such as total packets received, dropped (in general and due to lack of resources), and delivered [47].

The specification also enumerates several experimental block types. Experimental blocks are not properly defined at this time, and have not been integrated into the specification. These include:

- Alternative types of packet blocks
- Compression and encryption blocks
- Traffic Statistics and Monitoring Blocks
- Fixed Length and Directory (Packet Index) Block
- Event/Security Blocks

Application, vendor or domain specific block types or block options may be defined and added, allowing the format to support highly specialised applications beyond the scope of the original specification. These blocks are simply skipped over by any application which does not support them.

3.4.4 Comparison to Pcap Format

The PcapNg format addresses many of the limitations of the original pcap format, including all those listed in Section 3.3.3. The format accelerates access to arbitrary packets through skippable sections, facilitates multiple interfaces, provides a greater diversity of record types, and is easily extensible [47]. The primary drawback of the format is its complexity, which makes it more difficult to parse and encode data efficiently and accurately. A direct result of this complexity is a lack of widespread support for the format, with only partial support in a small selection of applications [132]. This means that while the format provides the means to encode traffic with greater accessibility and in far greater detail, it is still quite difficult to effectively access and utilise that detail, due to limited support in existing tools.

The provision of Simple Packet Block mitigates some of these limitations, but introduces two of its own. Firstly, simple packet blocks are designed for efficiency and do not store captured length or timestamp information, primarily to improve performance and reduce disk space utilisation [132]. This limits their utility in some aspects of traffic analysis, as packet size and arrival time cannot easily be determined. Secondly, simple packet blocks do not support multiple interfaces, and thus cannot benefit from one of the dominant features that separates PcapNg from from traditional pcap.

As the pcap format is typically sufficient (although not necessarily ideal) for most applications and is both widely supported and relatively simple to handle, it is still often used in preference of the PcapNg format. This should change, however, as the format matures.

3.5 Libpcap and WinPcap

Pcap (short for packet capture) is the de facto API for traffic capture and generalised filtering on most contemporary systems. Pcap was first distributed as Libpcap

on Unix-like systems, and was later adapted independently to the Windows operating systems as WinPcap.

Libpcap was developed – along with the Berkeley Packet Filter (BPF) [9, 54] (see Sections 4.2.1 and 4.2.5) which pcap uses to filter traffic – at Lawrence Berkeley Laboratory, and was applied as an integral part of TcpDump [115], a command-line network analyser which allows users to intelligently filter and query the contents of recorded network traces. TcpDump uses the Libpcap API to record, retrieve and filter traffic, the latter of which Libpcap achieves with the aide of a BPF-based filtering virtual machine executing in kernel space [115].

WinPcap is implemented to closely replicate Libpcap, but substitutes BPF for its Windows equivalent, NPF (Netgroup Packet Filter) [128]. As this research was conducted on the Windows 7 Operating System, related testing performed during the course of this research used the WinPcap API.

Pcap has been in use for over two decades and remains popular to the point of ubiquity today. In addition to TcpDump (and its Windows variant WinDump), pcap is used in many open source and commercial applications, including but not limited to traffic sniffers, protocol analysers, network monitors and intrusion detection systems. Most significantly, pcap is used by the IDS Snort [118], the port scanner Nmap [65], and the protocol analyser Wireshark [51]. Wireshark's application of the pcap API is discussed in Section 3.6.

3.5.1 Interfacing with Capture Files

This subsection discusses how pcap is used to interface with captures. Pcap is a low-level C API that provides methods for filtering and/or extracting raw packets from either a live or offline packet stream [128]. Filtering and extraction are performed separately and sequentially, with the former handled internally in kernel space and the latter largely offloaded to the application developer in user space. Both filtering and manual parsing are omittable; packet streams may be accessed unfiltered and returned packet contents may be ignored, but applications typically employ at least one of these functions in all but trivial packet counters.

Packets in a capture are accessed iteratively within a loop, with each iteration returning the next valid packet in the stream [128]. If a filter is applied, any

packets that fail to match the specified filter are transparently dropped by the filtering virtual machine and are thus never passed to the packet iterator or the user. Filtering is performed by either BPF or NPF depending on the host operating system, which are discussed in Section 4.3.

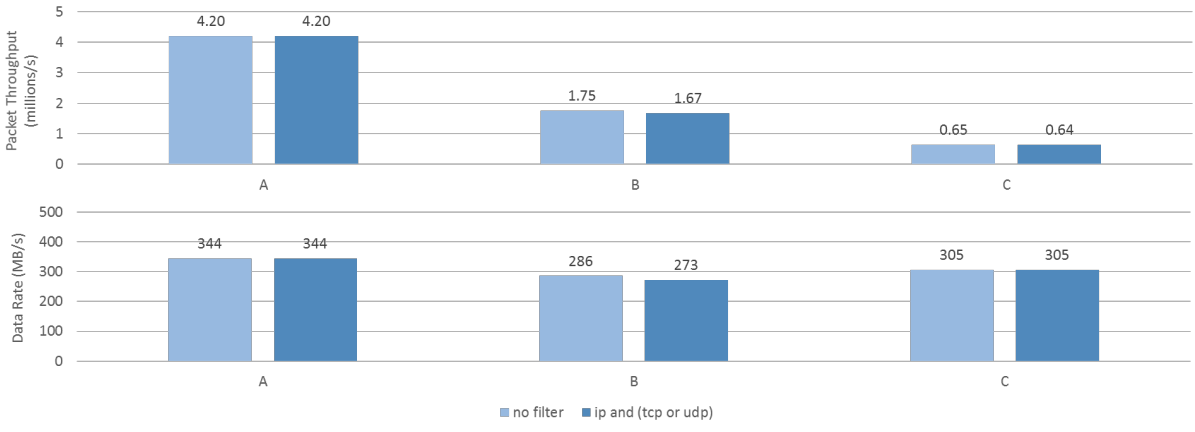
The packet iterator returns records in the raw pcap record format, which consists of a 16 byte packet header segment (comprising time stamps and length information), and a packet data segment containing the raw packet byte stream. At this point, it is largely up to the application developer to navigate the byte stream and extract information from the returned and filtered packets.

3.5.2 WinPcap Performance

As a low level API for filtering and extracting raw packet byte streams on CPUs, WinPcap performs admirably well. This is due at least in part to the continued active development of the platform in both the Libpcap and WinPcap APIs as well as its continued widespread use, which has resulted in improvements and optimisations to the architecture. It is also a partial product of the API's scope: pcap provides fast and efficient transparent filtering and buffering of packets from an incoming packet stream, but leaves the actual parsing and interpretation of packet data to the encapsulating application.

This approach ensures generality and flexibility within the user processes (limited only by the programmer's ability to efficiently extract and utilise the raw packets byte stream) without interfering with or bloating the filtering process. Figure 3.8 is included to provide an indication of the performance of WinPcap, showing the time taken to read and optionally filter three large captures (see Table C.4) from a software RAID array containing two SATA III SSDs (≈600 MB/s). Measurements were taken using WinPcap 4.1.3, running in Windows 7 on an Intel Core i7-3930K processor with 32 GB RAM (see Table C.1 in the appendices). The test program used is a trivial iterator written in C; it does not interact with any individual packet records, dropping all returned packets immediately. Each configuration was executed eleven times, with performance calculated by averaging the last ten iterations. The captures used are discussed in greater detail in Section 9.4 in Part IV of this document.

These performance figures show that pcap was able to process captures at up to 350 MB/s, with packet throughput scaling down as packet size increases. These



	A	B	C
Capture Size	2 GB	41 GB	156 GB
Average Packet Size	70 B	155 B	480 B

Figure 3.8: Pcap packet throughput and data rate for three captures.

processes were memory efficient, with each iteration utilising roughly 5 MB of system memory during the course of execution. These results cover only the collection, filtering and delivery of packet records however, and each returned packet must be processed within an external context to extract packet specific information. This can have a severe impact on performance, as the user code segment is often significantly more time consuming than WinPcap's own filtering and collection processes, particularly when performing detailed analysis.

3.6 Wireshark

Wireshark is a powerful and widely used cross-platform network protocol analysis tool which employs the pcap to interface with and analyse packets within an encapsulating network trace [50]. Network traces may be recorded live from pcap's network tap, or loaded from packet captures encoded in a variety of open and vendor formats (including pcap and PcapNg as discussed above) from long term storage. Wireshark is the de facto tool for investigating the contents of network packet traces, and provides numerous useful and varied filtering options, alongside statistical and analytic functions. Wireshark is quite similar in function to TcpDump, differing in that it supports wider functionality and better overall classification performance, in addition to providing an interactive GUI. A command-line version of Wireshark, called TShark, is also supplied within the standard distribution [50].

3.6.1 Capture Analysis Process

In essence, Wireshark identifies packets iteratively (optionally filtering input using pcap's filtering engine), adding a summary of each packet as a separate row in a scrollable list. By default, Wireshark shows the packet's arrival number, arrival time, source and destination (typically either an IP or MAC address), leaf protocol and a brief summary of the leaf protocol's contents. Selecting a row provides a detailed breakdown of the associated packet's headers, as well as the complete packet as a raw byte stream. The list may then be filtered using Wireshark's display filter syntax on protocol header contents [129] to focus on a specific protocol, stream or combination of header field values. Wireshark's native display filters support over 1000 protocols and 174,000 field types [129] allowing for complex dissection of individual packets.

Packet traces are loaded (and possibly filtered) using pcap, which provides Wireshark with a stream of pcap packet records [51]. Each record received from pcap is processed to extract timestamp and detailed protocol information from the pcap header and raw packet byte stream. The resultant record, containing both the extracted details and raw packet data, is appended to the list of processed packets and ultimately displayed as a row in the GUI. This process is performed sequentially, and continues until either the packet stream or system memory is exhausted. Filtering may be applied directly during packet acquisition (referred to as capture filtering by Wireshark) through pcap's native filtering engine using standard pcap syntax. As pcap's filtering engine is highly optimised and runs in kernel space, it provides faster filtering at the expense of depth and functionality. As a result, it acts primarily as an optional fast pre-filter to reduce load on display filters, which are significantly more computationally expensive and memory intensive by comparison.

Wireshark provides a wealth of functionality to support packet analysis and is highly configurable, providing facilities including but not limited to colouring packet records based on record contents, following specific protocol streams, extracting statistics and exporting specific packets. It supports a variety of formats including partial support for PcapNg, which allows it to capture traffic from multiple interfaces simultaneously.

3.6.2 Scaling to Large Captures

The Wireshark platform is highly effective when employed to analyse live traffic or short-term low-volume traces. Unfortunately its current architecture scales poorly with capture size, rendering it impractical for long term analysis and inefficient for medium term analysis. This section will elaborate on three significant barriers to large capture analysis, in descending order of severity. Measurements were taken using Wireshark 1.8.3 64-bit, running in Windows 7 on an Intel Core i7-3930K processor with 32 GB RAM, reading from a 600 MB/s SATA III RAID array.

3.6.2.1 Memory Utilisation

Excessive memory utilisation is one of Wireshark's more significant architectural problems with respect to large captures. Wireshark stores packet details and raw packet byte streams in system memory for fast retrieval, which has a significant impact on memory utilisation, as each record is guaranteed to require more storage than the original pcap record it encapsulates. This typically requires significantly more system memory than the total capture size to adequately contain all packet records. As a result, memory requirements can often bloat to up to between 1.5x and 10x the original capture size (depending on the composition of the capture). Applying subsequent display filters may further bloat memory utilisation the first time they are applied. For example, Figure 3.9 shows a comparison between actual capture size and Wireshark memory utilisation for the three captures previously shown in Figure 3.8 (due to limited host memory resources, the memory utilisations listed for captures B and C in the figure are estimates extrapolated from partial results).

These measurements show that Wireshark is extremely wasteful of host memory, and scales consumption with capture size. As system memory is a scarce resource with finite capacity, the maximum capture size supported by Wireshark is determinate on the executing host's RAM.

3.6.2.2 Load and Filter Performance

Wireshark's display filters and packet dissection processes are detailed and powerful, but come at high computational expense. This translates into extreme capture processing times, multiple orders of magnitude longer than the pcap API. In

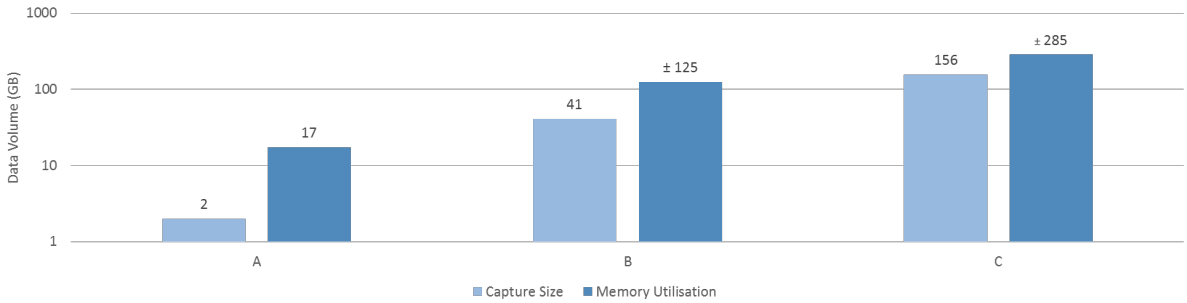


Figure 3.9: Capture size versus Wireshark memory utilisation while opening the captures listed in Figure 3.8.

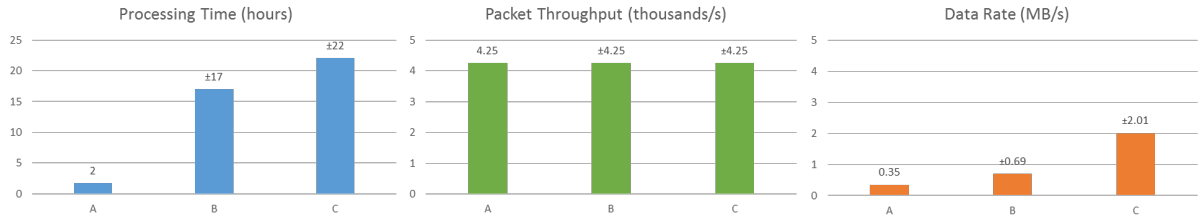


Figure 3.10: Average packet throughput and a data rate achieved while opening the captures listed in Figure 3.8.

general, and among other factors, packet rate remains relatively constant, while performance tends to scale inversely to capture density (i.e. average number of packets contained per MB); higher density captures with small packets require longer processing times than low density captures with large packets. To illustrate this, Figure 3.10 shows the packet throughput and data rate achieved when processing captures A, B and C as previously listed in Figure 3.8. As noted above, the measurements reported for captures B and C are extrapolated from partial results due to host memory limitations, and are thus only estimates.

The results presented show that Wireshark's throughput is between one and three orders of magnitude slower than pcap, depending on capture composition, achieving a throughput of only a few MB per second at best. Capture processing is performed during both the initial loading phase and also each time a display filter is applied to the loaded data; additional passes take roughly the same amount of time to execute, and further increase the pressure on host memory. Wireshark's low performance is largely due to the complexity of parsing and categorising packets into hundreds of potential protocols. Wireshark is designed to facilitate detailed analysis rather than filtering, and must parse and process every protocol defined in every packet. This differs from pcap, which only differentiates packets into pass

and fail categories, avoiding unnecessary complexity and increasing the achievable throughput of packet stream filtering.

3.6.2.3 User Interface

Wireshark presents results as individual record rows in a large scrollable list, and applies colouring rules to aid in identifying various protocol types. Again, this design works well for relatively small captures, but breaks down entirely when dealing with millions of records.

Wireshark presents a significant amount of detail regarding each packet within the packet list, displaying pages of between 50 and 60 records at a time at a resolution of 1920x1200. Displaying a million packet records in this way requires over 16,000 pages worth of complex record lines, presenting far more information than can reasonably be considered by a user. While filtering and analysis functions can reduce the number of records displayed, or find records of interest, these processes are time consuming and computationally expensive, and may fail to reduce the records displayed to a manageable number.

3.7 Summary

This chapter introduced and described how network packets are transmitted and received by distributed hosts, how the constituent protocols of packets are typically organised and put into operation, the formats used for recording traffic on live networks, and the applications used to process recorded capture files.

Section 3.1 began with a brief introduction to packets as containers for data transmission. Packets were described as variable length byte arrays, with payload data encapsulated within several layers of service specific protocol headers. Each protocol header is responsible for a specific aspect of transmission, and each packet uses multiple protocols to facilitate a particular communication.

Section 3.2 introduced the TCP/IP model for network communication, which organises protocols in the TCP/IP protocol suite into a four layer hierarchy, or stack. Each protocol receives services from protocols at lower layers in the stack, and provides services to protocols at higher layers in the stack. These layers include the Link

layer, Internet layer, Transport layer and Application layer. The section concluded by introducing the 7 layer OSI model and describing it through comparison to the TCP/IP model.

Section 3.3 described the pcap capture format for archiving packets. The section described how the location of each record in the capture is determined from the offset and length of the previous record, which enforces serial and sequential access to records and necessarily complicates the navigation of large captures. The section also discussed RAID arrays and capture indexes, which may be used to accelerate access to packet records and field data.

Section 3.4 described the PcapNg format, which attempts to address many of the limitations of the pcap format by incorporating section headers to accelerate capture traversal, and a selection of additional block types to store a wider variety of information. This section also discussed practicalities surrounding use of PcapNg, including the fact that the PcapNg format is not yet finalised, and is therefore only partially supported by a limited number of applications.

Section 3.5 provides an overview of the Libpcap and WinPcap APIs, which facilitate the creation, filtering and parsing of pcap captures for *nix and Windows systems respectively. The section discussed how these APIs are used to iteratively retrieve a stream of packet records from packet captures, optionally removing (or filtering) unwanted records. The section concluded by demonstrating the performance of the WinPcap API, showing that WinPcap achieved an average throughput of between 280 MB/s and 350 MB/s, depending on the capture processed.

Section 3.6 introduced the Wireshark Protocol Analyser and detailed its capture processing functionality. The majority of the section discussed three significant problems associated with processing large captures in Wireshark. These relate to excessive memory utilisation (which limits scalability), low throughput (which limits performance), and dense result sets (which are difficult to navigate and apply).

This chapter has described how packets are composed and transmitted, how intercepted packets are stored for long term analysis, and how the resultant packet captures are subsequently accessed and processed. The next chapter focusses on efficient approaches to classifying or filtering packet records, with a specific emphasis on general algorithms capable of classifying arbitrary protocols.

4

Packet Classification

PACKET classification refers to the process of identifying or categorising the raw binary data contained in network packets, so that they may be handled appropriately by the receiving host, application or service. Packet classifiers are ubiquitous in modern networks, and are critical components in many networking domains, including but not limited to endpoint demultiplexing [135], packet routing [113], firewalls [35, 53], intrusion detection[118], protocol analysis [103, 129], traffic monitoring, and network administration and management. This chapter introduces the fundamentals of packet classification, and briefly outlines four general approaches to maximising classification efficiency.

- Section 4.1 provides some context for packet classification and introduces the abstract classification process. The section concludes by introducing two subdomains of classification (referred to as filtering and routing) to provide context for later sections.
- Section 4.2 describes a selection of algorithms aimed at filtering packet streams. These algorithms are designed for flexibility and generality, and can handle

arbitrary protocols and fields. They are heavily dependent on divergence however, and are thus difficult to parallelise in a GPU context.

- Section 4.3 describes a selection of algorithms primarily aimed at routing and switching. These algorithms are designed for high throughput and low latency execution, and support specific fields in select protocols. Routing algorithms use a wide variety of sequential and parallel processing abstractions, and scale better to large filter sets than filtering algorithms.
- Section 4.4 describes the architecture and primary limitations of the GPF algorithm, which used elements from both filtering and routing algorithms to facilitate scalable parallel classification of multiple concurrent filters on GPU hardware. This algorithm forms the primary basis for research described in the following chapters.
- Section 4.5 briefly summarises related work in GPU accelerated packet classification. These approaches occupy different domains to the performed research, but have been included for completeness.
- Section 4.6 provides a summary of the chapter.

4.1 Classification Overview

This section provides a basic overview of packet classification as it relates to this research. Depending on how it is defined, packet classification can cover a broad range of processes, from filtering and routing algorithms, to application demultiplexing and deep-packet inspection algorithms. This research focusses on classifiers which target protocol headers and their fields, addressing in particular both protocol-independent filtering algorithms and high-performance routing algorithms (see Section 1.1.2).

4.1.1 Process

All packet header classifiers perform the same basic function: they compare information contained in a packet's protocol header against a set of static classification rules in order to categorise the packet as a particular type [66, 113]. This

information is then used to determine if and how to respond to the packet. Complications arise due to the layered nature of packet headers (see Section 3.1); the position of a field in a header is dependent on both the layout of its protocol, and the header sizes of all preceding protocols in the protocol stack. The packet classification process must therefore either parse and process header fields sequentially [7, 23], or abstract away this complexity by limiting classification to specific protocol fields [113].

Classifiers are typically optimised for specific contexts and implementation mediums, and vary greatly with respect to processing abstraction, functionality, performance and scope. The following section briefly introduces two such types which inform this research.

4.1.2 Algorithm Types

The approach described in Part III of this document is influenced by two relatively distinct types of packet header classifiers, referred to as routing and filtering algorithms for simplicity. This section briefly describes these types to contextualise later discussion of particular algorithms.

Routing algorithms are highly abstracted classifiers designed for high throughput and low latency execution, often utilising high-performance hardware. These properties are essential in performance-critical networking applications and services such as routing, firewalls and intrusion detection. Routing algorithms differentiate packets based on the values contained in specific fields of known protocols, and commonly target the five fields of the IP 5-tuple [113]. Some recent approaches, however, target larger sets of field values to support the OpenFlow 12-tuple used in virtual switching for Software Defined Networking (SDN) [36, 90, 122], a modern networking paradigm that decouples network management from traffic forwarding to provide open, programmable, centralised, and vendor-independent network management [91]. Routing algorithms are extremely diverse, scale to very large rule sets, and have been adapted to highly parallel contexts, including FGPAs and GPUs [40, 136]. They do not, however, generalise to arbitrary fields and protocols easily, and have limited field coverage.

Filtering algorithms are designed for flexibility and generality [54, 135], and are typically implemented in software targeting CPU processors. Packet filters are constructed as virtual filtering machines, which execute one or more filter programs

to guide the processing of each packet's byte stream. These programs essentially encode a boolean-valued filter predicate composed of one or more field comparisons to either accept or reject incoming packets [7, 9, 54]. In contrast to routing algorithms, which target a limited set of fixed fields in known protocols, packet filtering algorithms allow classification based on any arbitrary field or protocol header. This makes them an attractive alternative in security related domains, such as protocol analysis, where accuracy and coverage is a priority. They are however substantially slower than routing algorithms due to their complexity, and typically scale poorly to large filter sets [9, 135]. In addition, filtering algorithms universally rely on sequential decision-tree based abstractions, and are therefore difficult to parallelise.

While filtering algorithms are poorly suited to GPU processing due to their inherent dependence on branching control flow to evaluate filter sets, they do provide the flexibility to classify against arbitrary protocols and fields. Routing algorithms, in contrast, lack flexibility but are more architecturally diverse, with many designed specifically for highly parallel environments. The GPF algorithm was derived based on observations from both routing and filtering domains in order to adapt protocol independent classification to a highly parallel context. The remainder of this chapter discusses algorithms from both the filtering and routing domains, before addressing the GPF algorithm which forms the basis for this research. An overview of the algorithms discussed is provided in Figure 4.1.

4.2 Filtering Algorithms

Packet filters form a sub-domain of protocol header classification which facilitate the filtering of packet streams based on the contents of arbitrary protocol headers. Packet filters are conceptualised as virtual machines which processes incoming packet data against a low-level filter program. Most filtering algorithms share similar underlying architecture, and all employ abstractions similar to decision-trees that rely on branching control flow to efficiently evaluate filters. This makes it difficult to efficiently translate filtering algorithms to a GPGPU context, where divergence typically results in serialisation (see Section 2.3.3). This section discusses a selection of packet filtering approaches, focussing primarily on early works which developed the foundation of the domain. More recent algorithms are discussed

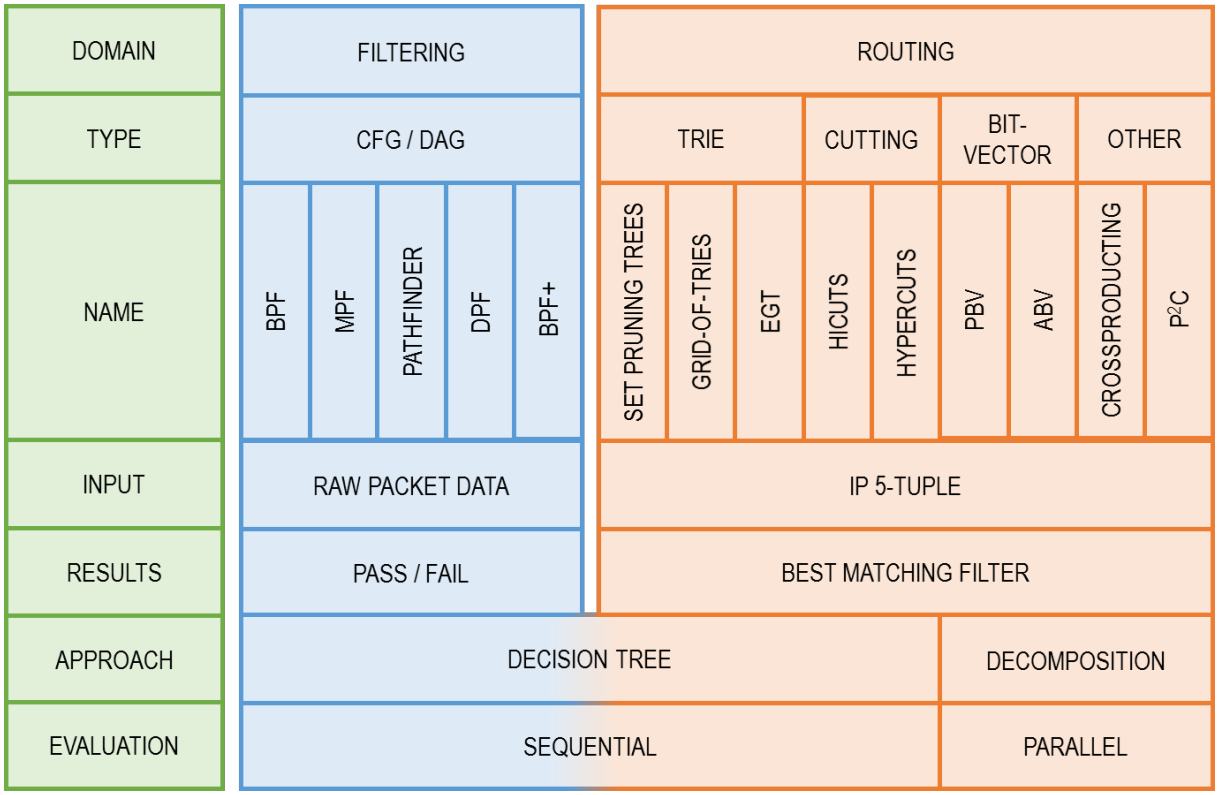


Figure 4.1: Table of Algorithms

briefly, but most do not alter the processing paradigm outside of domain specific extensions and optimisations that are beyond the scope of this research.

4.2.1 BPF

BSD Packet Filter (BPF) is an early protocol-independent packet demultiplexing filter tailored for efficient execution on register-based CPUs [54]. It uses a RISC (Reduced Instruction Set Computing) virtual machine, residing in kernel space, to rapidly execute arbitrary low-level filter programs over incoming packet data [54]. BPF was developed in parallel with Libpcap (see Section 3.5) to provide the API's filtering functionality, a role it has fulfilled ever since.

BPF filters treat packet data as an array of bytes, and are specified via a program written in an assembly language that maps to the virtual machine's instruction set. The approach does not rely on explicit internal protocol definitions to locate and process fields, and can be used to process any arbitrary protocol [54]. The BPF instruction set includes various load, store, jump, comparison and arithmetic

Listing 4 Example BPF program matching TCP packets with a source port of 1234. Adapted from [54].

```
1      ldh [12] //load ethertype half word
2      jeq #0x800, L1, L5 //if IP jump to L1, else fail
3  L1: ldb [23] //load IP protocol byte
4      jeq #6, L2, L5 //if TCP jump to L2, else fail
5  L3: ldx 4*([14]&0xf) //extract ip length into x register
6      ldh [x+14] //load half word at IP len + eth length
7      jeq #1234, L4, L5 //if correct port pass, else fail
8  L4: ret #TRUE
9  L5: ret #0
```

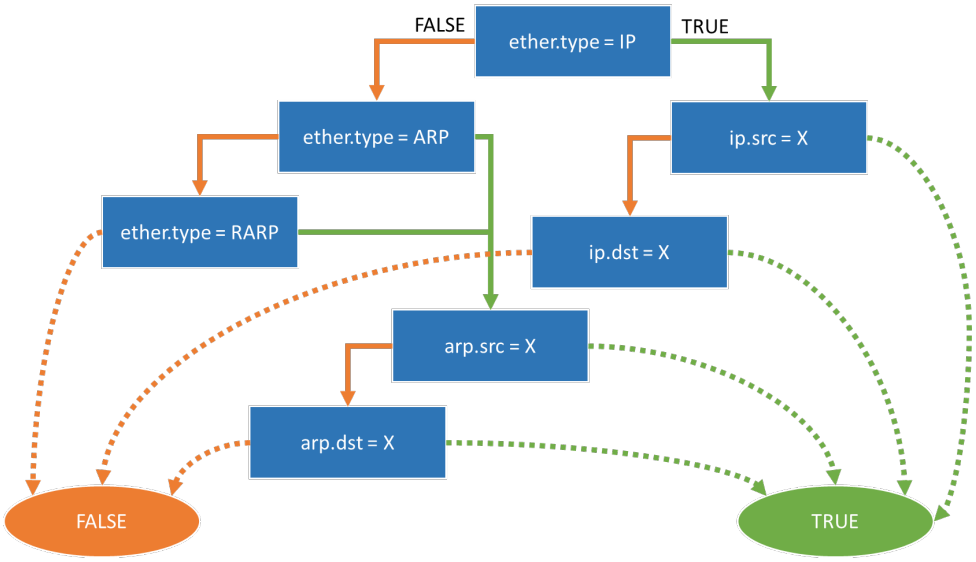


Figure 4.2: Example high-level Control Flow Graph checking for a reference to a host “X”. Adapted from [54].

operations, as well as a return statement which specifies the total number of bytes to be saved by the filter [54]. An example BPF program that matches TCP packets with a port of 1234 is shown in Listing 4.

BPF filters may be viewed conceptually as a Control Flow Graph (CFG), where each node in the tree contains any necessary pre-comparison operations (such as loading values into a register) followed by a branch comparison which directs computation to one of two possible successor nodes. Any number of paths may lead to acceptance or rejection, allowing for significant flexibility. An example high-level CFG which matches IP and ARP packets with a source or destination address X is shown in Figure 4.2.

4.2.2 Mach Packet Filter

MPF (Mach Packet Filter) was developed to provide efficient packet demultiplexing for the Mach micro-kernel architecture, using a process and filter language similar to that of BPF [135]. MPF added additional functionality to handle packet fragments efficiently, and incorporated a scalable function to support protocol demultiplexing by quickly dispatching packets to multiple end-point applications rather than simply passing or failing [135].

MPF uses the observation that many filter programs share a common structure, and vary only in the specific field values (such as port or address) matched by each filter. MPF thus divides filters into two sections, comprising a common and unique part. When multiple filters are added, MPF searches for filters with equivalent common parts, and merges these such that the common part is only evaluated once. To process the unique part of each filter, MPF introduced the `ret_match_imm` function, which matches n field values stored in array `M` against n keys. The keys of each unique match condition are added to a hash table, which is probed during execution of the filter using the field values extracted by the common part. If a match is found, the packet is sent to the corresponding receive port. If no match exists, the packet is rejected by the filter set and passed to the next filter in the chain. The primary benefit of this approach is improved scaling to multiple filters, which allows for more efficient end-point demultiplexing of packet streams.

An example of an MPF program which matches TCP packets with a source port of 1234 and a destination port of 5678 is shown in Listing 5,

4.2.3 Pathfinder

Pathfinder was released at roughly the same time as MPF, and utilised pattern matching techniques to facilitate both software and hardware based implementations, targeting CPUs and FPGAs respectively [7]. The software version employs DAGs (Directed Acyclic Graphs), which prevent circular graph node traversal. This section only considers the software version, as the hardware version is somewhat limited in comparison [7].

In Pathfinder, pattern matching is facilitated through cells and lines [7]. A cell is defined as the 4-tuple (*offset*, *length*, *mask*, *value*), which is used to classify a

Listing 5 Example BPF program matching TCP packets with a source port of 1234 and a destination port of 5678. Adapted from [135].

```
1      ldh P[12] //load ethertype half word
2      jeq #0x800, L1, FAIL //if IP jump to L1, else fail
3
4  L1: ldb P[23] //load IP protocol byte
5      jeq #6, L2, FAIL  //if TCP jump to L2, else fail
6
7  L2: ldxb 4*(P[14]&0xf) //extract ip length into x register
8      ldh P[x+14] //load half word at srcport offset
9      st M[0] //store srcport value in memory
10     ldh P[x+16] //load half word at destport offset
11     st M[1] //store destport value in memory
12
13     ret_match_imm #2, #ALL
14     key #1234
15     key #5678
16 FAIL:
17     ret #0
```

packet header field — located *offset* bytes from the start of the protocol header and spanning *length* bytes — against a target specified by *value*. As header fields typically span bits rather than bytes, the *mask* is used to remove unwanted bits from the classification. A line is composed of one or more cells, and a packet is said to have matched a line if all specified cell comparisons return true.

Patterns are specified as a protocol specific header declaration, which indicates the total length of the protocol header, in combination with a set of one or more lines. Patterns are organised hierarchically as a DAG, where the results of each pattern determine the next pattern to apply. If a pattern specifies multiple lines, the next pattern is determined by the best matching line. The global offset of a field in a packet header is calculated by summing all previous matching pattern's specified header lengths, and adding the local offset for the cell being matched in the current pattern. Because offsets are propagated, and not statically defined, Pathfinder only requires a single definition for each protocol which may succeed multiple variable or fixed length protocol patterns. Pathfinder additionally merges similar patterns to improve scaling to larger filter sets in a similar manner to MPF, relying on fast hash functions to differentiate between patterns which only differ in the field values tested. Other features include packet fragment handling and mechanisms to manage out-of-order packet delivery.

Listing 6 Example DPF program matching TCP packets with a source port of 1234.

```
1 (
2     (12:16 == 0x800) && # Check protocol is IP
3     (SHIFT(14)) && # Shift to IP header
4     (9:8 == 6) && # Check protocol is TCP
5     (SHIFT((4:8 & 0xf) << 2)) && # Shift to TCP header using IP length
6     (0:16 == 1234) # Check source port == 1234
7 )
```

4.2.4 Dynamic Packet Filter

DPF (Dynamic Packet Filter) exploits run-time information to optimise filter programs through dynamic code generation [23]. DPF treats filters as chains of atoms that specify bit comparisons and index shifts. Atoms are converted into filters and merged into a trie data structure (see Section 4.3.2) to minimise prefix match redundancy of common fields [23]. Other optimisations included dynamically converting run-time variables into immediate values, while optimising disjunctions at run-time to improve efficiency. An example DPF program consisting of five atoms and matching TCP packets with a source port of 1234 is shown in Listing 6.

DPF also introduces atom coalescing and alignment estimation. Atom coalescing combines adjacent atoms operating on consecutive bytes into a single atom in order to reduce instruction overhead. For instance, adjacent atoms testing 16-bit TCP source and destination ports may be coalesced into a single 32-bit atomic comparison. Alignment estimation (or alignment information propagation) involves recording the effect of each individual shift of the index register in order to predict word alignment. Repetitive shift operations may also be avoided by dynamically propagating this information to subsequent atoms in the classification chain [23].

4.2.5 BPF+

BPF+ is a revision of the BPF virtual machine, which applies a range of local and global optimisations to improve overall filtering performance. BPF+ translates high-level filter code into an acyclic CFG using an SSA (Static Single Assignment) intermediate form. SSA is a compiler optimisation technique that ensures each register is written to exactly once, allowing BPF+ to take advantage of numerous global data-flow optimisations [9]. Both local and global optimisations are applied

Listing 7 Example high-level BPF+ expression matching TCP packets with a source port of 1234. This is compiled into a form similar to Listing 4.

```
1 tcp srcport 1234
2     OR
3 tcp[0:2] = 1234
```

to the intermediate control flow graph, resulting in the optimised BPF+ byte code (equivalent to BPF filter code).

An assortment of control flow graph reduction techniques are used to reduce the length of the intermediate CFG. These optimisations include partial redundancy elimination, predicate assertion propagation and static predicate prediction, as well as peep-hole optimisations [9]. Partial redundancy elimination removes unnecessary instructions in a particular path, such as duplicate loads or comparison predicates. Similarly, predicate assertion propagation and static predicate prediction are used to eliminate predicates which can be determined from previous comparisons [9]. For instance, if a CFG node n contains some comparison $x = y$, and a subsequent node m in the same path as n contains the comparison $x \neq y$, the result of m may be statically determined from the result of n and may therefore be omitted. If m is a descendent of $n = true$, then m will always be $false$, and vice-versa. Peep-hole optimisations find inefficient and redundant instructions, replacing or removing them. Partial redundancy elimination, predicate assertion propagation and static predicate prediction optimisations are repeated until such time as there are no new changes, with peep-hole optimisations applied after each iteration.

Once the filter is delivered to its target environment for execution, a safety verifier ensures its integrity before passing the filter to a JIT (Just-In-Time) assembler. JIT compilation translates the optimised byte-code assembly into native machine code, and may optionally perform machine specific optimisations when executed on hardware rather than within an interpreted software environment [9]. An example BPF+ high-level filter that matches TCP packets with a source port of 1234 is shown in Listing 7.

4.2.6 Recent Algorithms

This section provides a brief overview of more recent general classification approaches that have adapted, extended, and improved upon the BPF model in vari-

ous ways. While these algorithms provide improved performance in certain contexts and supply additional functionality, they do not differ significantly from BPF or other classifiers in terms of processing abstraction.

- NPF (Netgroup Packet Filter) implements a BPF+ based filtering engine optimised for the Windows operating systems specifically to support WinPcap (See Section 3.5). NPF is heavily optimised for capture recording, and provides functions to facilitate packet injection and kernel-level network monitoring [127].
- xPF (Extended Packet Filter) incorporates extensions for efficient statistics collection into the BPF model [38].
- FFPF (Fairly Fast Packet Filter) uses extensive buffering to reduce memory overhead, among other optimisations [12].
- Adaptive Pattern Matching optimises filter sets by attempting to find a near optimal permutation of filters. This is used to minimise redundancy in the filter CFG [69, 119].
- Zero-Copy BPF eliminates a redundant copy in BPF between kernel and user processes by creating a shared buffer between them [104].
- Swift packet filter uses a CISC (Complex Instruction Set Computing) based virtual machine to reduce instruction overhead and minimise filter update latency [133].

4.3 Routing Algorithms

Routing algorithms operate at a higher level of abstraction than filtering algorithms, processing specific field sets (usually the IP 5-tuple) rather than raw packet data [113]. Routing filters contain an ordered set of rules which specify the individual requirements for each field; a packet is said to match a filter if each of its targeted fields meet the requirements for their respective rules. This simplifies both the creation and evaluation of filter sets substantially by avoiding the complexity of handling arbitrary fields. This also allows routing algorithms to scale far more effectively than filtering algorithms to large filter sets.

Most routing algorithms return a single result, corresponding to the highest priority matching rule. There are however some parallel algorithms which support multi-match classification [40, 66], allowing a single packet to be matched by multiple rules simultaneously. This functionality is useful in circumstances where accuracy is a priority (such as in IDSs), where missed classifications can result in lost observations regarding network traffic. It is not as useful in contexts where classification is used to determine the single most appropriate action, as additional results slow performance and have limited practical use.

Routing algorithms are more varied in architecture than filtering algorithms, and have been implemented in software and on a wide range of sequential and parallel hardware. This section provides an overview of these approaches, and discusses five well understood sequential and parallel algorithms.

4.3.1 Approaches

Packet classification approaches may be loosely categorised into four overarching types: linear search, decision tree, parallel decomposition, and tuple space [113].

- Linear search is the most basic type, and operates by evaluating every rule until a match is found or the rule set is exhausted. While this method may be wasteful of processing resources in contrast to other algorithm types, it is simple to program and may be easily parallelised in both hardware and software. Linear searches are often used as a final component in more sophisticated and efficient algorithms, typically after the search space has been reduced to only a few possibilities [113].
- Decision tree algorithms are quite common due to their natural mapping to multi-variable classification [113]. While they are relatively diverse in design, all decision tree algorithms employ some form of branching control flow to sequentially eliminate irrelevant comparisons during execution. This is a highly efficient approach on sequential processors, but is relatively course grained and difficult to parallelise efficiently.
- Decomposition algorithms, which divide the classification process into several concurrent comparisons or evaluations, provide a fine-grained alternative which is better suited to parallel execution than decision trees [113]. Decomposition algorithms evaluate field values against filter rules in parallel,

Table 4.1: Example Filter Set, showing source and destination IP address prefixes for each filter. Adapted from [113].

Filter	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}
DA	0*	0*	0*	00*	00*	10*	*	0*	0*	0*	111*
SA	10*	01*	1*	1*	11*	1*	00*	10*	1*	10*	000*

and aggregate these partial results in a final step to match the packet to a filter. Decomposition algorithms are typically engineered to accelerate classification on massively parallel hardware, and have been implemented for both FPGAs [40, 41, 108] and GPUs [122, 136].

- Tuple space algorithms create tuples containing the number of significant bits in each evaluated field, and probe these (either sequentially or in parallel) to narrow the classification search space [110, 113].

The remainder of this section discusses a small selection of routing algorithms that use decision tree and decomposition methods.

4.3.2 Trie Algorithms

Trie algorithms are decision tree algorithms which employ tries to perform classification. A trie is essentially an associative array of string based keys, where each individual path through the trie combines to specify a unique match condition [10]. When a string is matched by a trie, each node tests a successive character index of the string, determining which successor node the data should be processed by. If no candidates are found, the string is not matched. Trie algorithms use bitwise tries, which operate over binary digits rather than characters. Bitwise trie structures help eliminate redundancy by combining common prefixes into a single string of nodes, and map well to both exact and longest prefix match classification [66, 113]. Examples of trie-based algorithms include Set Pruning Trees [21], Grid-of-Tries [111], and Extended Grid-of-Tries (EGT) [5]. Figure 4.3 provides an illustration of the Set Pruning Trees approach, using the filter set displayed in Table 4.1.

4.3.3 Cutting Algorithms

Cutting algorithms are a form of decision tree algorithm which view a filter with d fields geometrically, as a d dimensional object (or area) in d dimensional space

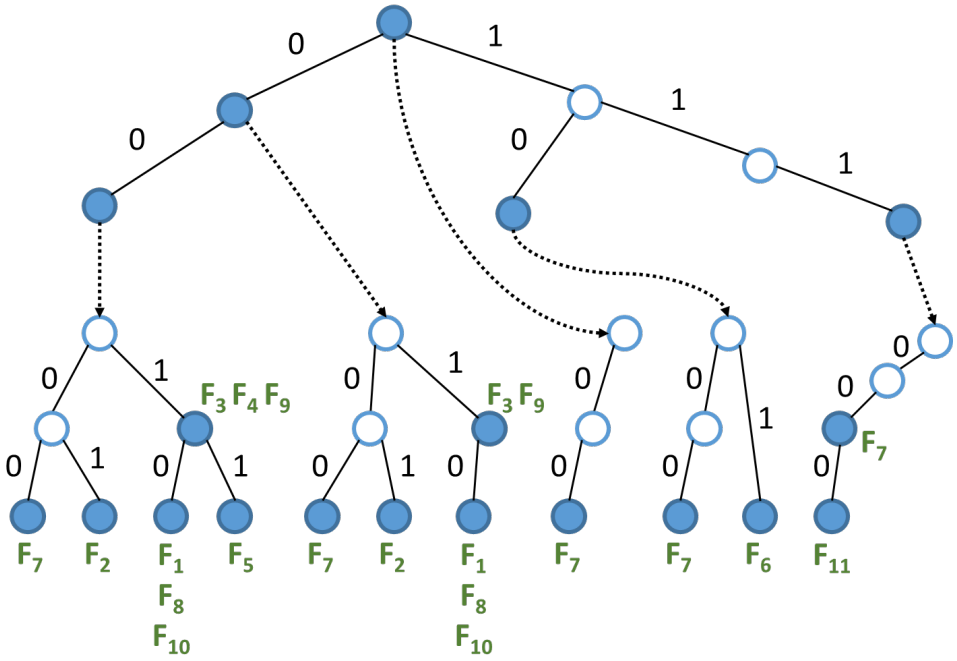


Figure 4.3: Set Pruning Tree created from the filter set shown in Table 4.1. Adapted from [113].

[30, 107]. The space occupied by a filter in a particular dimension is derived from the filter's range of values relating to that field (or dimension). Should a field value or range not be specified in a particular dimension, the filter simply fills the entire dimensional space.

Conceptually, cutting algorithms operate by cutting the d dimensional space into successively smaller partitions, until such time as the number of filters contained within a particular partition is below some specified threshold value [66, 113]. By treating each incoming packet as a point in this d dimensional space, the packet filtering problem can be expressed as selecting the partition within which the point falls [66, 113]. If the threshold value is larger than one, then the highest priority filter within the partition is accepted [30, 107]. Examples of cutting algorithms include Hierarchical Intelligent Cuttings (HiCuts) [30] and HyperCuts [107]. Figure 4.4 shows an illustration of HiCuts' two dimensional geometric representation of the example filter set provided in Table 4.2.

Table 4.2: Example filter set, containing 4-bit port and address values. Adapted from [113].

Filter	A	B	C	D	E	F	G	H	I	J	K
Port	2	5	8	6	0:15	9:15	0:4	0:3	0:15	7:15	11
Address	10	12	5	0:15	14:15	2:3	0:3	0:7	6	8:15	0:7

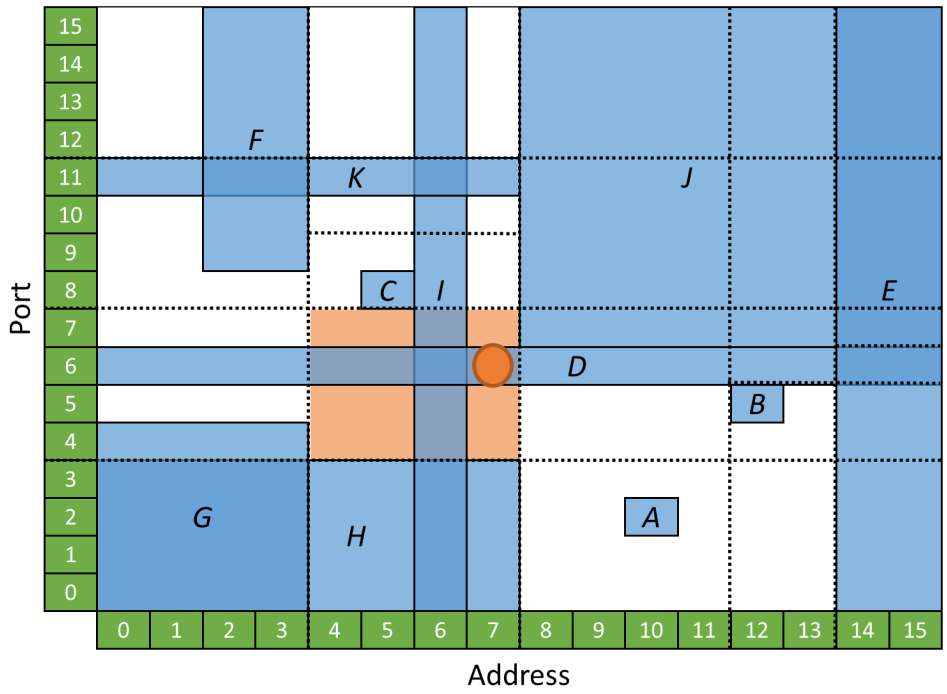


Figure 4.4: Geometric representation of a 2-dimensional filter over 4-bit address and port fields. Adapted from [113].

4.3.4 Bit-Vectors

Bit-vector algorithms are parallel decomposition algorithms which take a geometric view of packet classification, treating filters as d dimensional objects in d dimensional space, similar to Cutting algorithms [66, 113]. In each dimension d, a set of N filters is used to define a maximum of $2N + 1$ elementary intervals on each axis (or dimension), and thus up to $(2N + 1)^d$ elementary d dimensional regions in the geometric filter space. Each elementary interval on each axis is associated with a binary bit-vector of length N. Each index in this N-bit vector represents a filter, sorted such that the highest order bit in the vector represents the highest priority filter [66, 113].

All bit vectors are initialized to arrays of zeros. Then, when a filter in a specific dimension d overlaps an elementary range on d's axis, the corresponding bit-vector index is set to 1. Thus an elementary interval's bit-vector represents a priority ordered array of filters, where the value at each index represents whether a particular filter is active in the corresponding interval in that dimension. Each dimension is processed in parallel, producing a set of d bit-vectors, which, when aggregated through bitwise conjunction, produce a priority ordered list indicating all matching filters. Examples of bit-vector algorithms include Parallel Bit Vector [48] and Aggregate Bit-Vector classification [6]. An illustration of Parallel Bit-Vector partitioning for two fields (port and address) is shown in Figure 4.5.

4.3.5 Crossproducting

The Crossproducting method [111] is motivated by the observation that the number of unique values for a given field in a filter set is significantly less than the number of filters in that set [113]. For each field to be compared, a set of unique values for that field appearing in the filter set is constructed. Thus, classifying against *f* fields results in *f* independent Field Sets, with each Field Set containing the unique values associated with a particular field. When given a value from an associated packet field, the Field Set returns the best matching value in that set.

When classifying d fields, this results in a d-tuple, created by concatenating the results from each Field Set. These initial field matches may be done in parallel. The d-tuple result is used as a hash key in a precomputed table of crossproducts, which contains entries providing the best matching filter for all combinations of

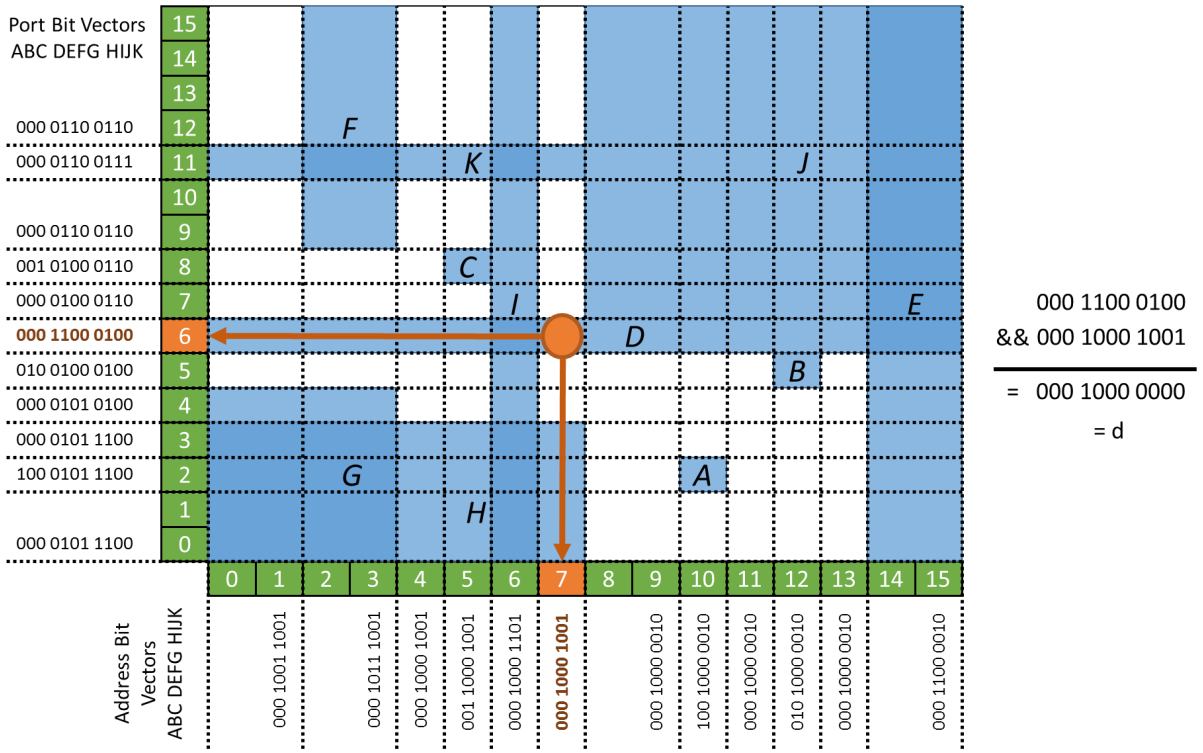


Figure 4.5: Example Parallel Bit-Vector classification structure over the filters depicted in Figure 4.4. Adapted from [113].

results [66, 113]. This method reduces an n-tuple comparison to n independent field searches, processed in parallel, followed by a single hash look-up. This comes at the expense of exponential memory requirements for the table of crossproducts [111]. Specifically, the table requires $\prod_{i=1}^{n} (\mid d_i \mid)$ unique entries, where $\mid d_i \mid$ is the cardinality of the set of unique entries for the i^{th} Field Set, and n is the number of fields being matched. An example of the Crossproducting algorithm, using three fields, is depicted in Figure 4.6.

4.3.6 Parallel Packet Classification (P^2C)

Parallel Packet Classification (P^2C) [121] is a relatively complex, multi-stage decomposition technique. For each field to be evaluated in the filter set, the field axis (for instance port number) is divided geometrically into its constituent elementary intervals, as in the bit-vector techniques. Above this axis, n layers are defined, where each layer contains a non-overlapping subset of filters, such that each filter is positioned over the range of elementary intervals corresponding to acceptable values for that field. The filters contained in each layer are selected such that

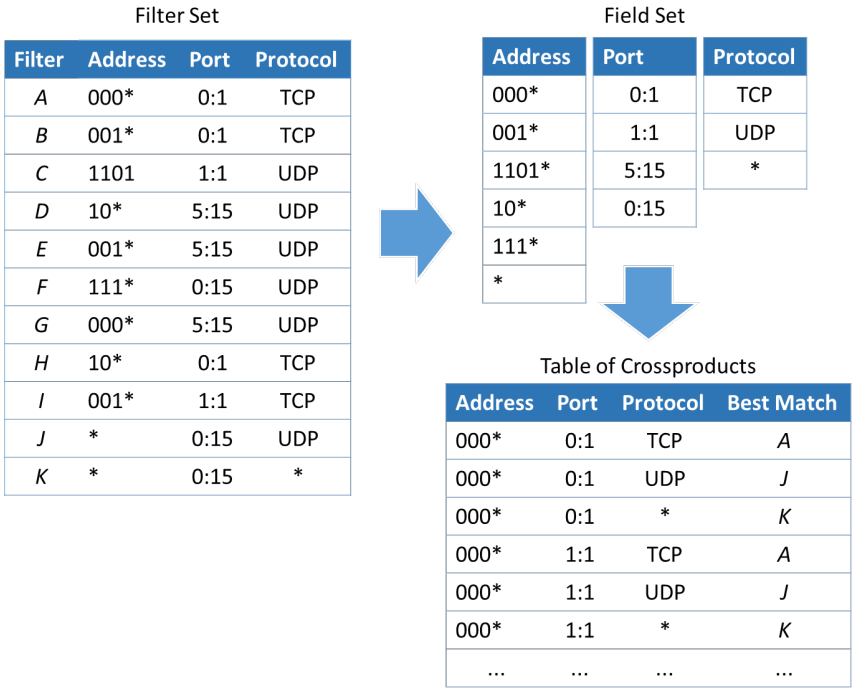
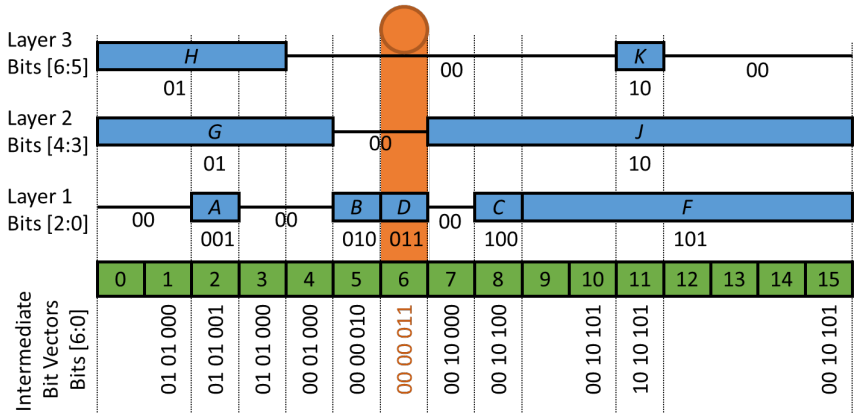


Figure 4.6: Example Crossproducting algorithm. Adapted from [113].

the number of layers necessary for non-overlapping subsets is minimised [66, 113]. The P^2C algorithm is illustrated in Figure 4.7, using the filter set provided in Table 4.2.

In each layer, the algorithm associates a locally unique binary value (using a minimum number of bits) to each filter contained in that layer, while empty regions are given a binary value of zero. An intermediate bit-vector is then created for each interval by concatenating the binary values of each layer in that interval. These intermediate bit-vectors are used to derive a ternary string (called the Ternary Match Condition) for each filter, which matches all intermediate bit-vectors associated with that filter. These ternary filter strings are then stored in a priority ordered list for classification. When a packet arrives, each field is processed in parallel, and the resulting bit vectors from each field are concatenated together to form a single binary string. This string is then matched against the precomputed filter strings in priority order to find the correct matching filter.



Ternary Match Condition	Filter
01 01 001	*A*
00 00 010	*B*
00 10 100	*C*
00 00 011	*D*
** ** ***	*E*
*0 10 101	*F*
0* 01 00*	*G*
01 01 00*	*H*
** ** ***	*I*
*0 10 *0*	*J*
10 10 101	*K*

Figure 4.7: Example P^2C range encoding, matching port field values (y-axis) of the filters depicted in Figure 4.4. Adapted from [113].

4.4 GPF

This section summarises the GPF algorithm [66, 67, 68, 70, 71] which provides the foundation for the research presented in this thesis. This section also provides an overview of the approach and architecture of this prototype, with emphasis on the processing abstraction used. GPF was implemented to assess the viability of GPU co-processors as accelerators for generalised packet classification and capture mining [66], specifically targeting compute capability of 1.3 Nvidia GPUs. The GPF algorithm was developed as an experimental prototype, and had a number of limitations which restricted its usefulness to a research context. Despite these limitations, GPF demonstrated promising performance, classifying at rates between 40 and 100 million packets per second for most filter sets when using an Nvidia GTX 480 [66]. This section concludes with a discussion of the primary limitations of the GPF algorithm.

4.4.1 Approach

The GPF algorithm is a non-divergent, massively-parallel filter-predicate processor created using C++ and CUDA; it is supported by a filtering DSL that was created using C# and ANTLR[1] (ANother Tool for Language Recognition) [94]. The architecture of GPF was influenced by decomposition based routing algorithms, which

[1] http://www.antlr.org/

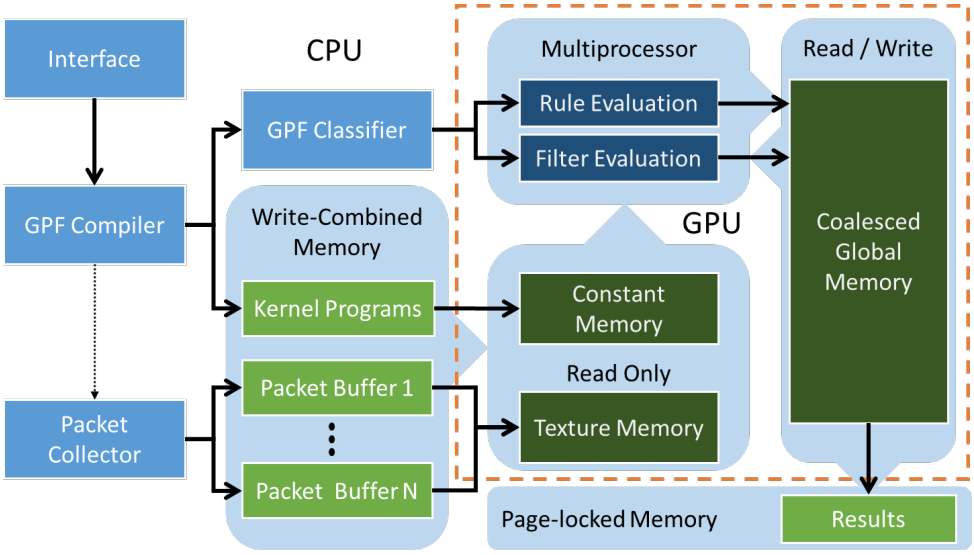


Figure 4.8: Overview of GPF Architecture

employ a parallel set of comparisons for each field that are subsequently aggregated in a final step [66]. Figure 4.8 shows the high-level architecture of GPF and its supporting host-side components. These components cooperate to evaluate multiple filter predicates concurrently without divergence, providing fast but feature limited multi-match filtering for arbitrary network protocols.

GPF uses a coarse grained SIMD approach that allocates each packet to an independent thread. Each thread then evaluates its prescribed packet against a filter program stored in constant memory. The algorithm treats packet records as an array of bits, and identifies header fields within these bits using a two-tuple containing the fields bit-offset and bit-length. Packet processing is divided into two distinct phases: a filter rule evaluation phase, where packet data comparisons are performed, and a filter predicate evaluation phase, where the results of rule evaluation are used to evaluate predicates. No divergence is necessary, as all threads process identical instructions, and all packet data is evaluated prior to the computation of any predicates. In addition, field comparisons may be grouped and ordered such that all operations can be performed in a single pass of the packet data, which eliminates all redundant global memory reads.

4.4.2 Filter Grammar

GPF employs an assortment of CPU-based components to provide functions for handling input and output data and compiling filter instructions. GPF uses a

Listing 8 Example GPF high-level filter code for identifying various protocols in the IP suite.

```
1  sub ipv4 { 96:16 == 2048 && 112:4 == 4 }
2  sub ipv6 { 96:16 == 34525 }
3
4  filter ip { ipv4 || ipv6 }
5  filter arp { 96:16 == 0x800 }
6
7  filter tcp { ipv4 && 184:8 == 6 || ipv6 && 160:8 == 6 }
8  filter udp { ipv4 && 184:8 == 17 || ipv6 && 160:8 == 17 }
9
10 filter icmp { ipv4 && 184:8 == 1 || ipv6 && 160:8 == 58 }
11
12 filter dns_query {
13     udpv4 && (272:16 == 53 && 288:16 > 1023)
14 }
```

simple DSL to specify filters, which is processed via an ANTLR grammar to produce optimised classification programs. A simple example of a filter set that targets numerous protocols in the TCP/IP protocol suite is provided in Listing 8.

Filters may be specified as either a `filter` type, or a `sub` (for subfilter) type. The only difference between these two types is that filter results are returned to the host, while sub results are discarded at completion. Both `filter` and `sub` predicates may be referenced by other `filter` and `sub` predicates to compose varied and complex filter conditions. While the high-level filter language supports parenthesis in filter expressions, the GPU filtering kernel does not, and so sub-equations in parenthesis are automatically converted into unnamed subfilters to be processed first.

The DSL filter specification is compiled into two to three distinct programs encoded as separate integer arrays. The first program comprises all the rule comparisons performed by the filter, grouped by field and ordered sequentially by field offset. The remaining programs encode the filter predicates and subfilter predicates (if they are used). Compilation also determines what regions of packet data are relevant to the filter set, which is used during pre-processing to crop unused bytes from the beginning and end of all packets. As a result, packet data records are stored as small uniform length byte arrays, and packet offsets can easily be determined by multiplying the thread's global index with the cropped record size. This also helps to reduce PCIE overhead when transferring packet data to the device for processing.

4.4.3 Classifying Packets

GPF uses two separate kernels to perform classification; the Rule kernel, which process the rule program, and the Filter kernel, which processes filter and subfilter programs. Each GPF kernel executes a set of integer encoded operations read from constant memory. As all threads access the same instructions, constant memory requests are never serialised and thus provide the best possible performance [86]. All threads perform identical operations at the same field offsets, but operate on different packet data. This provides two primary benefits:

1. It exposes the inherent redundancy of protocol field comparisons in filter sets, in that multiple distinct filters will share a significant proportion of field classifications between them [9, 113, 135]. By setting out to perform all possible classifications from the offset, this redundancy can be fully capitalised upon to ensure no repeated comparisons occur.

2. It allows for native and highly scalable multi-match classification, where a single iteration can return results for many filters concurrently. As a result of the high degree of redundancy in filter sets [113], the approach provides better per-filter throughput when larger numbers of filters are used.

The classifier processes packets concurrently, such that each thread reads from one or more regions of a specific packet record sequentially. As a result, threads read from sparsely distributed memory locations, which is not conducive to coalescing memory transactions, particularly on pre-Fermi GPUs which lacked global memory caches (see Section 2.5). As packet data is read but not written, it is bound to and read through a cached texture reference to improve performance [66, 86]. While a texture reference provides initial access to packet data, the same chunk of packet data may be used in multiple comparisons.

In order to avoid unnecessary global memory transactions, each thread maintains an 8 byte packet cache, which is loaded in 4 or 8 byte chunks. Once a chunk of packet data is cached, each unique field contained in that chunk is extracted and compared against all its relevant rules, until the chunk is exhausted and the next relevant chunk can be loaded. This process is guided by the rule evaluation program, which indicates when to load new cache chunks, which fields to extract, which comparisons to perform, and where to store results. Figure 4.9 shows an

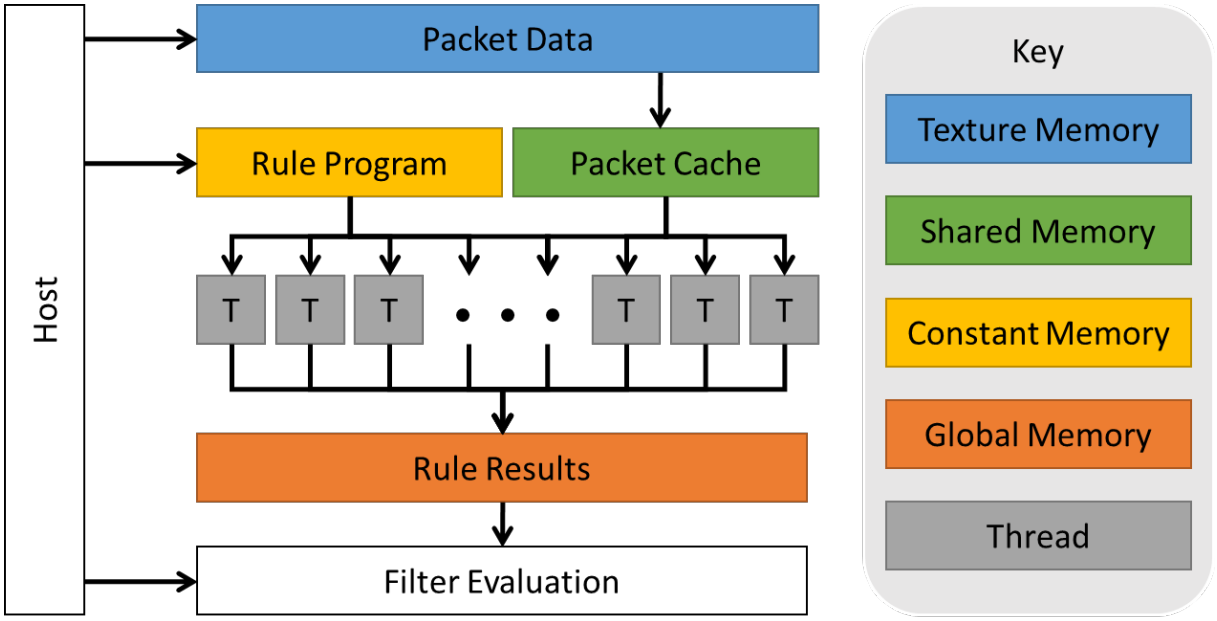


Figure 4.9: Memory architecture used by threads in evaluating filter conditions.

abstract overview of the memory architecture of the rule evaluation kernel, illustrating the flow of data between memory regions.

Once the rule evaluation kernel has completed, the predicate evaluation kernels are launched. The optional subfilter evaluation kernel is processed first, but may be omitted if no filters rely on subfilters. The filter evaluation kernel is invoked after the subfilter evaluation kernel has completed, using subfilter results as if they were produced by the rule kernel. These kernels differ only in where they store results; subfilter results are stored along-side the field comparison results generated in the rule evaluation kernel, while the filter results are stored in a separate region and transferred to the host at kernel completion.

Both kernels evaluate predicates in a nested loop; the outer loop performs logical disjunction on the results of the inner loop, which performs logical conjunction and negation on boolean values loaded from rule memory. An example of the process of encoding a single filter is provided in Figure 4.10. The integer values leading the OR and AND lines indicate the number of operands in the operation, and thus the number of loop iterations necessary to process each operand. The values of a, b, c and d refer to result arrays in coalesced global memory, while the 0 or 1 preceding them indicates whether or not to invert the value. Once all comparisons are complete, the results for each filter are transferred to the host and written to disk, while the next batch of packets is loaded and processed.

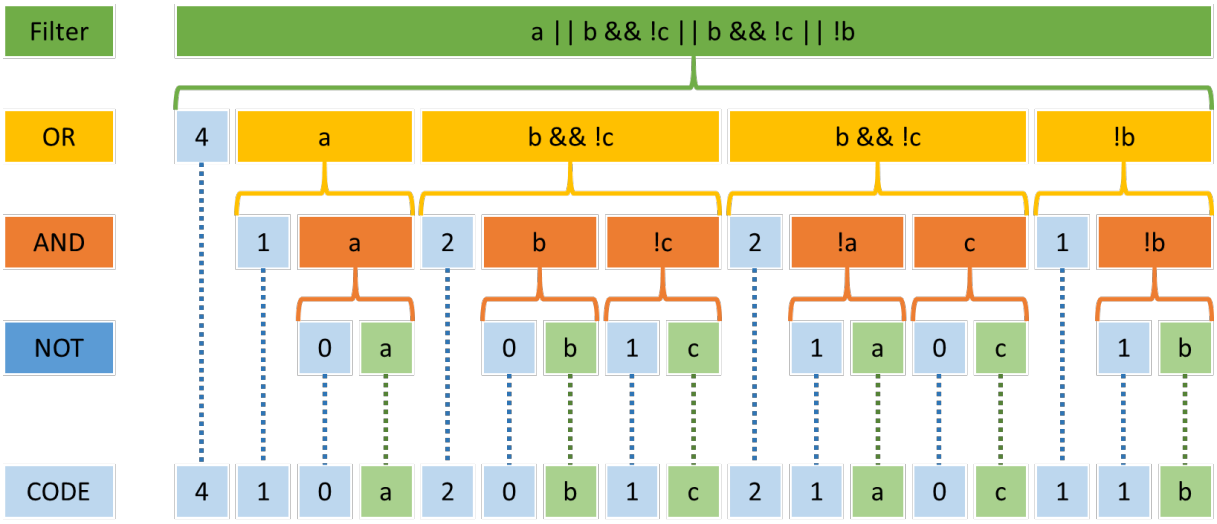


Figure 4.10: Encoding a single filter predicate for execution on the GPU. The number of filters to process is stored in constant memory.

4.4.4 Architecture Limitations

Due to its experimental nature, the GPF classifier has a number of limitations which restrict its usefulness in real world scenarios. This section details the primary limitations of the approach used above, in order to provide context for the design of the GPF+ algorithm discussed in Chapter 6.

- GPF does not properly handle packets with variable length protocol headers. As GPF does not diverge and executes only pre-compiled instructions, individual threads are incapable of adjusting processing based on the contents of packet header fields, and thus cannot handle variability in protocol offsets. Mitigating this limitation requires support for loading and retaining field data, performing transformations on this field data, and adjusting program flow dynamically on a per-thread basis, without introducing significant divergence overhead.

- GPF cannot prune redundant comparisons or filters at runtime, which causes performance to scale linearly with compiled filter set size [66]. While avoiding divergence altogether prevents serialisation within thread warps, it simultaneously prevents warps from avoiding computation that is irrelevant to all executing threads. This approach handles commonly occurring protocols and frequently used fields well, but becomes unnecessarily expensive when probing large sets of fields from rare or infrequently used protocols.

Listing 9 Example GPF filter code which replicates TCP port filters to handle both IPv4 and IPv6 packets.

```
1 sub tcpv4 { 96:16 == 2048 && 184:8 == 6 }
2 sub tcpv6 { 96:16 == 34525 && 160:8 == 6 }
3
4 filter http {
5     tcpv4 && (272:16 == 80 || 288:16 == 80) || tcpv6 && (432:16 == 80 ||
          448:16 == 80)
6 }
```

- GPF is limited in its capacity to access, analyse and collate packet field data from a raw packet byte stream. For instance, while any TCP or UDP port combination can be actively searched for, there is no mechanism to report the distribution of port values. Additionally, threads cannot return related field contents of flagged packets to the host, a function that would be useful in supporting both capture indexing and protocol analysis (see Sections 3.3.5 and 3.6).
- The inflexibility and complexity of programming filter sets using the prototype DSL limits the classification engine's usefulness outside of an experimental context. This stems from having to specify the bit offset from the start of the packet and the size of each relevant field statically by hand, which is somewhat clumsy and requires a measure of replication to handle different protocol compositions. For instance, TCP or UDP require separate definitions for IPv4 and IPv6 to account for the differing lengths of these protocols, as illustrated in Listing 9.

4.5 Related GPGPU Research

GPU accelerated classification is a relatively new research domain, with few related works currently available in the literature. The domain has seen a marked increase in related publications over the last two to three years however, and this trend is likely to continue as the domain matures. This section highlights a selection of related works which apply GPUs to accelerate routing algorithms and intrusion detection systems.

- **2008** – Vasiliadis *et al* [123] introduced Gnort, a GPU accelerated implementation of the Snort NIDS which offloads payload string-matching to a CUDA

co-processor to improve throughput. Pcap was used to supply and pre-classify packets into port groups [123] on the host; these groups were then processed separately against group-specific Snort rules using a GPGPU implementation of the Aho-Corasick string matching algorithm [2]. Vasiliadis *et al* later incorporated this functionality into the GASPP framework discussed below [124].

- **2014** – Zhou *et al* [136] described a GPU accelerated decomposition algorithm in 2014, based on binary range trees and bit-vectors, intended for fast 5-tuple based classification of large filter rule sets. The performance of this algorithm was subsequently compared to a CPU implementation, showing an overall improvement in packet throughput by a factor of two, at the expense of much higher average latency per packet processed [136].

- **2014** – Vasiliadis *et al* [124] described the GASPP (GPU-Accelerated Stateful Packet Processing) Framework for deep packet inspection and stateful analysis on live network traffic. The framework provides support for a number of GPU accelerated traffic processing functions including network flow tracking, TCP stream reassembly, packet reordering, string matching, cipher operations and packet manipulation [124]. GASPP executes a sequence (or chain) of modules, each performing a particular function, for a particular protocol, from within a discreet kernel. GASPP currently provides modules for Ethernet, IP, UDP and TCP, although the solution can support additional protocols through custom modules.

- **2014** – Varvello *et al* [122] investigated techniques to accelerate tuple-based classification against large rule sets for use in virtual switching. One of the primary focuses of this work is to accelerate classification beyond the standard 5-tuple in order to support a broader range of applications, and was evaluated processing both 5-tuple rules and more complex 12-tuple OpenFlow virtual switching rule sets [122]. The research investigated linear search and tuple search algorithms, and introduced an optimised form of tuple search, called Bloom search, which uses hash-based data structures called Bloom filters to improve encoding efficiency [122].

4.6 Summary

This chapter provided an overview of packet classification, focussing specifically on filtering and routing algorithms, as well as prior research.

Section 4.1 began the chapter by briefly summarising the domain. Discussion was limited to packet header classifiers, which operate by comparing field values contained in protocol headers to rules contained in filters. While both filtering algorithms and routing algorithms process protocol header fields, they have different and somewhat conflicting goals; filtering algorithms aim for high flexibility and protocol generality, while routing algorithms are optimised for high throughput and low latency execution.

Section 4.2 described several filtering algorithms designed for execution on CPUs, including the BPF algorithm used by Libpcap (see Section 3.5). Filtering algorithms utilise branching tree-based abstractions to sequentially parse packet fields and compare them to specific rules. Filtering algorithms are highly flexible but do not scale well to large filter sets. They are also difficult to adapt to GPU architecture due to their reliance on decision based branching, which results in serialisation on GPUs. Filtering algorithms execute relatively low level program instructions which locate, extract, compare and transform field data without requiring a priori knowledge of the protocol being processed.

Section 4.3 complemented the previous section by discussing routing algorithms, which lack flexibility but are well suited to parallelism. Routing algorithms process field sets rather than raw packet data, typically the IP 5-tuple of addresses, ports and protocol. This section began by briefly describing the four overarching approaches to this form of classification: linear search, decision tree, decomposition and tuple space. Of these approaches, decision trees and decomposition algorithms were considered in the most detail, through an examination of a selection of algorithms.

Section 4.4 discussed the architecture and primary limitations of the GPF algorithm. GPF was developed as a prototype while investigating methods to accelerate general packet classification using GPUs. The algorithm decomposed filtering into two distinct steps: building rule data and evaluating filters. Unlike filtering algorithms which employ branching control flow to guide processing, GPF filters are condensed into a single rule set, which executes every required evaluation for every

filter on every packet. This approach exploits the redundancy in filter sets by combining different rules relating to similar fields into a single operation. While the GPF algorithm produced promising results, it lacked important functions (including optional field and variable length protocol handling) that limited its usefulness outside of a research context.

Section 4.5 briefly summarised research related to classification on GPUs. The research described in this section is tangential to this project, and relates primarily to routing algorithms, deep packet inspectors and intrusion detection systems.

In the following part of the document, the design and implementation of GPF+, its supporting pipeline and example applications are discussed in detail.

Part III

Implementation

5

Architecture

PART III of this document discusses the design and implementation of a packet classification system specifically intended to accelerate analysis of bulk network traffic. This chapter aims to break down and compartmentalise the various functions of the classification system into a set of components. The system relies on a variety of different processes that span multiple domains, and thus discussing them as a unified whole is somewhat complex. Dividing the system into components that can be discussed independently provides a means of managing this complexity.

This chapter provides a general overview of the classification system, summarising the high-level architecture and introducing its various discreet components. It also describes the two simplest components – the capture buffer and pre-processor – which load and prepare packet data for classification. The classifier itself is described in greater detail in Chapters 6, while the DSL used program is discussed in Chapter 7. Example post-processors which use the system outputs are addressed last in Chapter 8. This chapter is structured as follows:

- Section 5.1 introduces the classification system, and addresses some high-

level aspects of design, including how programs are encoded, how classification is abstracted, and how raw capture data is handled.

- Section 5.2 explains the high-level architecture of the processes which comprise the classification process, and discusses how individual components communicate across thread and execution context boundaries.
- Section 5.3 discusses each of the system components in turn, contextualising their specific roles within the classification process.
- Section 5.4 discusses the capture buffer component, which is a simple but performance critical function responsible for reading in packet data for use by the system.
- Section 5.5 discusses the pre-processor, which parses packets from raw capture data and re-packages them for the GPU classifier. The pre-processor also handles the creation of capture index support files; these files are not currently used during classification, but are important to post-processors in facilitating capture navigation and the generation of capture-wide metrics.
- Section 5.6 provides a summary of the chapter.

5.1 Introduction

Processing large packet traces at high speed is a conceptually simple but resource intensive process, constrained at the system hardware level by I/O (Input/Output) performance, system memory and processing capacity. The implemented architecture takes advantage of the capabilities of modern CUDA GPUs to construct a fast, flexible and programmable packet classification pipeline. This pipeline is intended to accelerate the processing of arbitrarily large packet captures, and simplify the means by which they can be explored and analysed.

5.1.1 Implementation Overview

The implementation discussed in this part of the document is composed of a collection of components that cooperate to efficiently process large capture files. Figure 5.1 shows a simplified outline of the system architecture

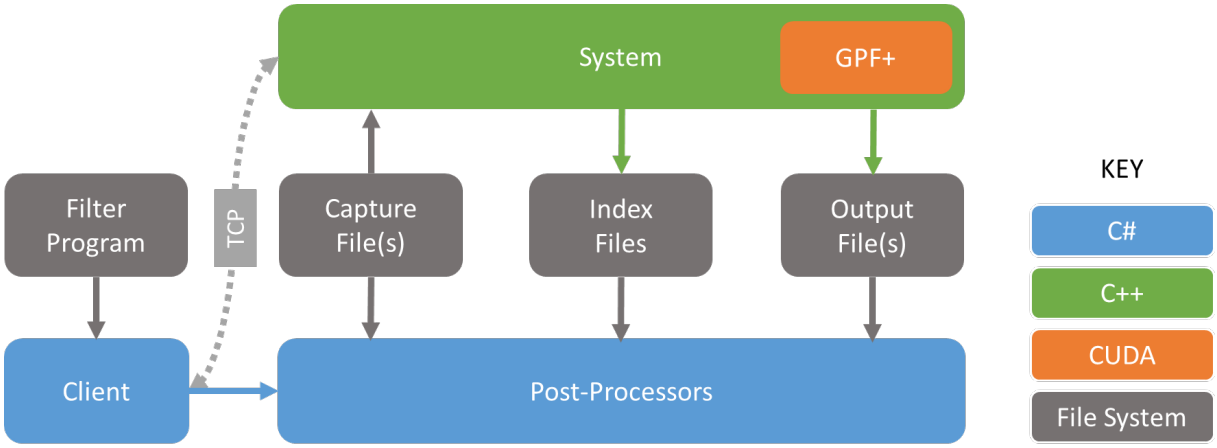


Figure 5.1: High-level implementation overview.

The primary component in this system is a CUDA-based general packet classifier. The classification process, referred to as GPF+ to distinguish it from prior work, is implemented as a device-side object that is instantiated within an executing kernel. This object facilitates the extraction of both bit-based filter results and integer encoded field values from raw packet data. GPF+ uses a stack-based abstraction that facilitates protocol pruning, reduces filter replication, handles variable length protocol headers, and simplifies filter creation. GPF+ is discussed in detail in Chapter 6.

The GPF+ classifier is supported by a multi-threaded system of host-side components executing in a C++ process that load, pre-process and deliver packet data to the GPU based classifier, while collecting and storing all classification outputs and index data. This system is specifically intended to accelerate the processing of large capture files (see Section 3.3), and is constructed using a pipeline of asynchronous threads connected via message queues. Its primary components are discussed later in this chapter.

GPF+ evaluates filter programs produced by a DSL component, implemented in C# and housed in a separate process (Client in the figure). The high-level grammar syntax defines protocols as object-like structures, and chains these together to create a protocol library. Filters then reference protocols and fields in this library to construct filter sets. This separates the description of protocol structure from filter definitions, allowing protocol structures to be reused in multiple filter sets without redefinition. The DSL and its grammar are described in Chapter 7.

The system provides three proof-of-concept post-processing functions which ap-

ply the outputs of the classification system to accelerate aspects of trace analysis. These post-processors include (in order of decreasing complexity) a component to visualise captures and filters, a function to generate (or distil) pre-filtered sub-captures from results, and a simple CPU process that maps and displays the composition of specific field results. The components are discussed in Chapter 8.

The architecture of the classification kernel and its supporting pipeline were derived through experimentation and extensive performance testing. This provided an experimentally informed indication of the relative performance of approaches to sub-problems. Experimental evaluation and synthesis of components was most heavily utilised in the development of the GPGPU processes, where even minor differences between implementations can result in differing occupancy, bandwidth efficiency, processing latency and/or serialization. In general, approaches were investigated to ascertain relative throughput, resource utilisation, occupancy and other metrics, and developed along the most promising lines.

5.1.2 Program Encoding

Program encoding has a significant influence on the design of the classification kernel and its supporting infrastructure, as it dictates to a large extent how filtering is performed. This section briefly introduces the basic premise of the GPF+ program encoding, and how it relates to both GPF and more traditional assembler style syntaxes.

In order to play to the strengths of the prevailing CUDA architecture, the original GPF kernel interpreted a rigidly structured stream of commands in fixed sequence, which ensured that all threads executed identical instructions. This approach avoided thread divergence overhead in exchange for additional redundancy during classification (see Section 4.4.2). The GPF classification approach could interpret pre-compiled commands efficiently, but it could not adjust processing based on packet or protocol contents, limiting its ability to handle complex packet records.

The feasibility of utilising a more traditional assembler style syntax to provide this flexibility was explored initially as an alternative to the approach established in GPF, but investigations into a correlating CUDA kernel architecture failed to approach competitive throughputs. GPF+ instead uses an abstraction derived from the process used in GPF that incorporates local state variables and restricted

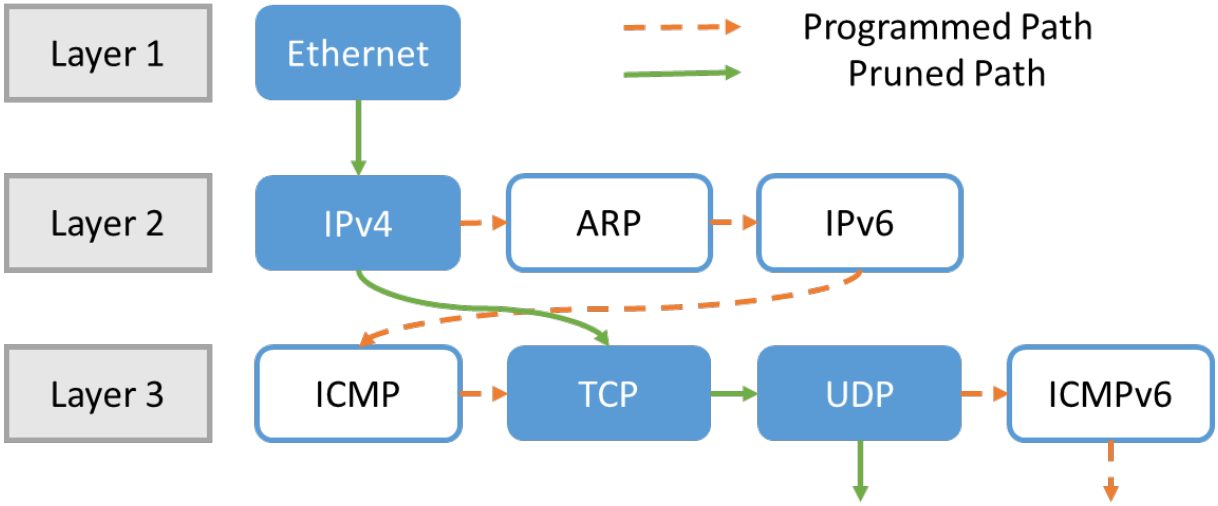


Figure 5.2: Layers and protocol pruning for a warp containing only TCP and UDP packets, encapsulated in IPv4 frames.

branching to improve efficiency and flexibility, while providing new functionality to complement filtering.

5.1.3 Processing Abstraction

The architecture of the GPF+ classifier extends the basic approach used in earlier work. It achieves this by incorporating a more refined processing abstraction that maintains and utilises local state variables to dynamically adjust processing to warp requirements at runtime. This section provides a brief overview of the processing abstraction used in this revised approach, and explains how it differs from that of the GPF prototype.

In the GPF prototype, packet data was treated as a byte stream containing fields of known size at predetermined bit offsets at compile time. This was by design, following the example of filtering algorithms, which treat all packets as byte arrays to ensure protocol independence and provide maximum generality (see Section 4.2). The GPF+ process uses a more developed abstraction, which organises classification into hierarchical layers of concurrent protocols, analogous to the protocol stacks in the TCP/IP and the OSI models (see Section 3.2).

GPF+ conceptualises packet records as being composed of layers of fixed and variable length protocol headers, each containing fields at known offsets within the

header. Each protocol layer contains one or more disjoint protocols, each of which may connect to one or more child protocols in subsequent layers. This abstraction allows for greater flexibility and helps to reduce redundant processing, as both individual protocols and entire layers may be skipped or pruned by the classifier through warp voting (see Section 2.7). An illustration showing the layer organisation of an example filter set is shown in Figure 5.2. In the figure, the programmed path represents the filter sequence encoded in the GPF+ program. The pruned path is the resultant path in a warp matching only IPv4 TCP and UDP packets.

This functionality is achieved through a small internal set of state variables which store context information needed to guide and optimise the processing of each layer, tailored to the thread warp in question. The updated approach also adds support for extracting, operating on and storing field values, and can apply these values within numeric expressions. Numeric expressions are currently used to handle protocol length fields exclusively, but could be extended to support more general computation. The implementation of the classification kernel is expanded on in detail in Chapter 6.

5.1.4 Handling Capture Data

A notable concern for developing a scalable system to support large capture processing is the associated memory allocation and storage costs of unbounded quantities of packet data and results. As the system must support capture traces far exceeding the memory capacity of even high-end desktops, it is impractical to rely on raw packet data stored in host memory (see Section 3.6.2). Similarly, as the amount of data contained within the generated results scales with respect to packet count and thus capture size, results data cannot be stored entirely in host memory either, as this would ultimately place limits on the maximum capture size. The system produces and subsequently employs compact index, filter and field results to allow post processors to operate independently of raw packet data, alleviating the need to repeatedly re-parse the raw capture. The implementation is designed to minimise interaction with raw capture data as much as possible, and requires only a single pass over the capture data to generate all outputs. This approach ensures scalability to large captures, and limits I/O overhead from capture data by avoiding repeated loads.

5.1.5 System Interface

The classification system is encapsulated within a C++ application that executes either as a stand-alone application or as a server for a client process. The system defaults to a server configuration, where it awaits a connection from a C# client application via a TCP channel. Once a connection has been made, the server receives execution arguments and compiled program code (emitted from the DSL) from the C# client, and initiates the main classification loop. Once the process completes, the server signals the client, indicating that the outputs have been generated and may be used.

The stand-alone configuration is more straight-forward, taking its inputs from the command-line to simplify automated or batch processing. This interface was included primarily to aide in testing, and currently lacks access to a native DSL compiler; at present, it relies on externally compiled programs created by the C#-based DSL in the current implementation.

5.2 Process Overview

This section describes the high-level components which comprise the critical functions within the classification pipeline. It should be noted for clarity that the high-level components discussed in this section are logically grouped into discreet units primarily to facilitate discussion, and may refer to multiple sets of processes which share some similarity of purpose or implementation, but do not necessarily interact directly.

5.2.1 Processing Captures

This section describes the processing of a packet capture within the framework chronologically, in order to introduce and contextualise the primary components within the system. Figure 5.3 shows the primary abstract system components, and how they are connected to facilitate capture classification.

The classification system relies on two separate processes; a C++ server application, which is responsible for performing classification, and a C# client application,

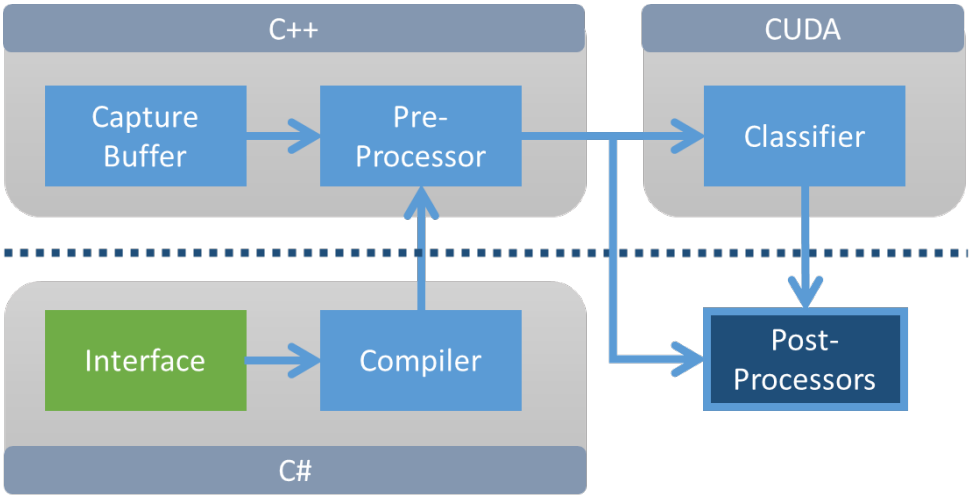


Figure 5.3: Abstract overview of components and connections of the implemented classification system.

which is responsible for program compilation, initiating classification, and presenting results. These processes communicate with each other via TCP, and can potentially execute on different hosts. Processing is initiated by the client process via a GUI (Graphical User Interface) input form, taking as input a packet capture file, a filter program, and process configuration. The program specification is passed to the DSL compiler, which processes it to produce the GPF+ program set for the classifier, as well as a collection of derived operational parameters and runtime constants. The client passes this configuration to the server to initiate the classification process.

On receipt of program instructions, the server initiates the main classification loop. This loop iteratively reads, pre-processes and classifies capture data, and writes the computed results to long term storage for future use. Raw packet data is buffered into memory by processes within the capture buffer component, and is passed asynchronously to the pre-processor in a different thread. The pre-processor parses packet records from the incoming raw capture data, copying packet segments into statically sized CUDA buffers; these are dispatched asynchronously to the classifier when filled. The pre-processor may additionally record indexing information, which is required by both the capture visualisation and capture distillation post-processing components (see Chapter 8).

The CUDA classification kernel is managed by a dedicated host thread within the server process. The host thread uses multiple streams to copy incoming packet data to the device and initiate classification asynchronously. The GPF+ kernel

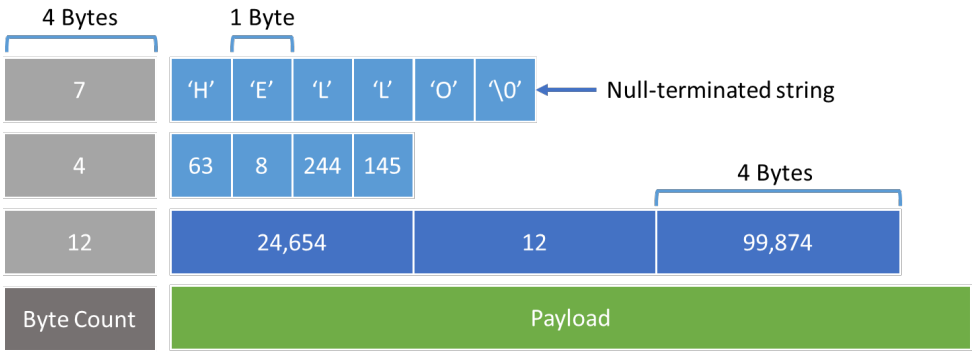


Figure 5.4: Example 0MQ messages.

evaluates the device-side data against the GPF+ program specification, producing results which are in turn copied asynchronously back to the host, and ultimately written to file. Once all packets have been processed, the client is signalled and the process ends.

5.2.2 Connecting Threads and Processes

Threads within the performance-critical C++ binary are connected together using 0MQ[1] (Zero Message Queue) sockets [33]. 0MQ is a high-performance, low-level C messaging API designed to facilitate communication between asynchronous distributed or concurrent processes executing on heterogeneous platforms, with minimal overhead and maximum portability. To this end 0MQ treats all messages as length specified byte arrays, leaving the programmer responsible for the conversion of variables, objects and data structures to and from the byte array message [33]. Examples of three 0MQ messages are provided in Figure 5.4.

Sockets typically communicate using the inproc (in-process) transport layer, which creates a flexible, direct and exclusive message pipeline between precisely two participating threads. An exception to this exists between the sockets which communicate between the C# client application and the C++ classification server. As components execute as separate binaries within both managed (C#) and unmanaged (C++) environments, they cannot communicate through inproc [33], and instead connect using a simple request-reply pattern over TCP.

For the most part, 0MQ sockets act as asynchronous and direct buffers between concurrently executing paired threads in a particular process, and pass large chunks

[1] http://zeromq.org/

of data via pointers to avoid unnecessary and expensive copies. As pointers cannot be passed between execution contexts, server threads communicating in this manner are tightly bound, and cannot be easily fragmented into distributed subprocesses [33].

5.2.3 Buffer Architecture

The host process is in essence a collection of independent but cooperative threads, coordinated through asynchronous inter-thread message passing. Components are connected to form a pipeline that carries large volumes of packet data from the packet reader, via the pre-processor, to the indexing and classification components. At the same time, output produced by the indexer and classifier is dispatched to a file writing component where it is ultimately written to long-term storage.

Due to the volume of data that this process needs to pass between components in the application's critical path, it is important to minimise memory copy overhead between participating threads. This is achieved primarily by utilising pairs of 0MQ sockets to pass pointers to data asynchronously over an inproc communication channel. A benefit of using 0MQ for this purpose is that it acts as a high-performance buffer between components. Each 0MQ communication channel is in effect a queue, storing messages in order of arrival until they are retrieved by the destination thread. If a thread attempts to retrieve a message when none are immediately available, 0MQ seamlessly ensures that the request blocks until a message becomes available. This greatly simplifies passing data between threads, as it removes the need for mutual exclusion or critical sections, and provides a measure of asynchronicity between producer and consumer threads [33]. Buffers are passed as 8-byte pointers to avoid expensive and unnecessary copy overhead between threads in the process. A simplified illustration of the flow of buffers between the primary system components is shown in Figure 5.5.

All buffers are statically allocated during the initialisation process, and are then reused repeatedly until processing completes. The implementation avoids dynamic allocation wherever possible, as it can negatively impact performance. Dynamic allocation refers to producer threads which allocate memory on demand during the course of processing, and expects the consumer thread to deallocate the memory once it has finished with it. This has the benefit of allowing the producer to execute with greater asynchronicity, without reliance on an external source to provide it

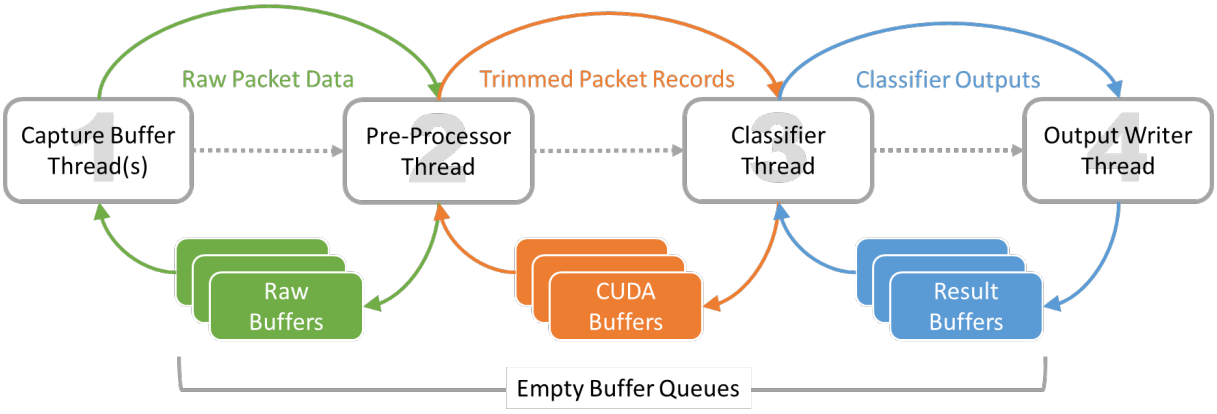


Figure 5.5: Buffers and system data flow.

with empty buffers. Dynamically allocating and deallocating memory is expensive however, and can increases pressure on host memory resources when allocating buffers for high data throughputs.

Static allocation, in contrast, does not have these drawbacks. Using static allocation, a small set of buffers are allocated once and passed back and forth between the producer and the consumer in a loop; they are only deallocated once all other processing has completed. As allocation only occurs during the construction phase of the object, allocation is a once-off cost that does not scale with capture size. In addition, the limited number of buffers in circulation prevent run-away memory utilisation, as an over-productive producer thread will eventually have to wait for an empty buffer to become available.

5.3 Components

This section summarises the components of the architecture and provides a more concrete outline of their roles within the classification process. This serves to contextualise discussion in the remainder of Part III.

5.3.1 Capture Buffer

The capture buffer is the primary input process and acts as the interface between long-term storage devices and the other components of the classification loop. The

capture buffer reads in raw packet data from capture files and dispatches that data to the pre-processor. Due to the relatively slow speed of long-term storage devices, reading capture data can present a potentially critical bottleneck when read from a slow medium. To overcome this to some degree, the input components include a simple hardware-independent mirrored file reader that can concurrently read from multiple copies of a capture on different storage devices to increase its throughput. This process is discussed in Section 5.4, after the system components have been described.

5.3.2 Pre-Processor

The pre-processor is responsible for preparing packet data for GPU processing, copying relevant sections of raw capture data into CUDA buffers for use by the classifier. The pre-processor is also responsible for index generation and storage, which it performs concurrently with packet record copying. The pre-processor and its functions are discussed in Section 5.5.

5.3.3 Classifier

The GPF+ classifier encapsulates the components responsible for processing packet data to produce the results requested in a filter program, and is capable of producing both filter and field results. Filter results contain the Boolean results of a particular user-specified filter predicate, encoded as a compact stream of bits, while field results store the numeric values contained within a particular header field as an array of 32-bit integers. Multiple field and filter results can be produced simultaneously in a single pass, even if they target entirely different protocols. These results are extremely useful when paired with the index files generated by the pre-processor, as together they provide an indexed database for the results of any performed classification or requested field value. Chapter 6 details how this is achieved.

5.3.4 Compiler

The compiler processes filter programs in the form of a high-level domain specific language constructed using the ANTLR API (using C# bindings) to produce the in-

structions which guide the main classification process. The compiler is implemented in C# primarily for the purposes of rapid development. Programs targeting the DSL are divided into a protocol library and a kernel (or main) method. The protocol library is comprised of multiple class-like structures which map out the structure of, and relationships between, required protocols. The kernel function leverages the structures defined in the library in order to specify the filter and field values to be collected during classification. The compiler outputs a compiled set of program instructions that can be interpreted by the GPF+ classifier, as well as a collection of runtime constants and pre-processor directives used in various parts of the end-to-end system process. Discussion of the compiler is deferred to Chapter 7, after the underlying machine has been described.

5.3.5 Post-Processors

Post-processing comprises a loose grouping of components which are not directly involved with the classification process, but apply the results of classification once the process is complete to provide useful end-user functionality. The post-processor components support three separate functions that serve as example applications. These include:

1. Visualisation of captures and filter results using OpenGL. Visualisation is facilitated through the use of the generated index files and classification results, and comprises components for high-level capture exploration in real-time through a basic graph-based interface.

2. Distillation of pre-filtered sub-captures using index files and filter results. Distillation uses the same inputs as visualisation, but uses these inputs to prune and filter the raw capture they relate to, producing smaller pre-filtered and/or temporally cropped captures. These smaller captures are easier to process and analyse in specialised applications such as Wireshark.

3. Generating simple field distributions from extracted field values. This function represents a simple program which derives the distribution of values contained by a specific protocol field within the capture, using extracted field data stored in an output field file as input.

Post-processing components are discussed in Chapter 8 as working examples of how the results generated by the classifier can be used to accelerate capture analysis.

5.4 Reading Packet Data

Buffering packet data into host memory is one of the most performance-critical procedures in the current classification system for two key reasons. Firstly, it is a prerequisite to all functionality, as all components are designed to interact with either packet data or the processed results of packet data. Secondly, it depends on long term storage, which is severely bandwidth limited, particularly when reading from HDD drives (see Section 3.3.4). As packet data is prerequisite to classification, the speed at which these files can be buffered into memory limits the speed at which the classifier can operate.

While RAID arrays provide a means of improving read performance [19, 95], they are independent of the classifier and are managed either in hardware or by the operating system (see Section 3.3.4). RAID arrays are not always convenient, as they rely on properly formatted data at a partition level, and require multiple comparable dedicated drives to operate efficiently [19, 95]. This requires investment of both time and resources on the part of the user. To provide an alternative for once-off processing that produces similar speed up but does not require multiple dedicated pre-formatted drive partitions, the system implements a simple optional mirrored read method that can interleave capture data collected from multiple file copies simultaneously.

In mirrored reading, file access is split across an arbitrary number of drives, each concurrently contributing a portion of the requested data. The approach does not perform any formatting or data-striping, and instead reads different segments (or stripes) from multiple mirrors of the file on different drives. This simplistic approach has the benefit of being able to support arbitrary numbers of dissimilar drives on demand with no significant setup, data re-formatting or pre-requisite partitioning.

The following subsections describe the standard capture buffering process and the mirrored reader in more detail.

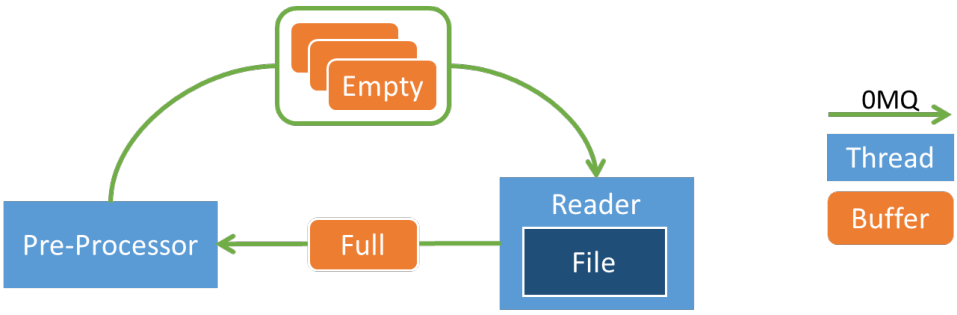


Figure 5.6: Reading from a single file source.

5.4.1 Reading from a Single Source

The simplest means of reading packet capture data is from a single source file, as it is an entirely serial process with no internal parallelism or concurrency to be managed. While reading from a single source is certainly slower than reading from multiple sources at once, it is more than sufficient performance-wise for most moderately sized captures. A high-level overview of the single file reader is provided in Figure 5.6.

The essential function of the reader is to get raw packet data into host memory as fast as possible, where it may be processed to produce filters and index files. This process is performed from within a dedicated thread which dispatches and reacquires buffers through separate 0MQ sockets. Before beginning the read process, the thread first determines the size of the capture and divides it conceptually into a sequence of 8 MB blocks. These are read sequentially into a revolving set of statically allocated 8 MB buffers. Once a buffer has been filled, it is dispatched asynchronously to the host process via 0MQ and replaced with an empty buffer from the empty queue. Dispatched buffers are processed sequentially by the host process, which returns each emptied buffer through a socket pair to the reader thread before acquiring another full buffer.

5.4.2 Reading from Multiple Sources

Reading captures from multiple sources is a slightly more complex process than its single file counterpart, due to the need to efficiently coordinate file reads, and interleave the results for consumption, by the pre-processor thread. The mirrored reader uses $n + 2$ threads to process data from n separate sources. Packet capture

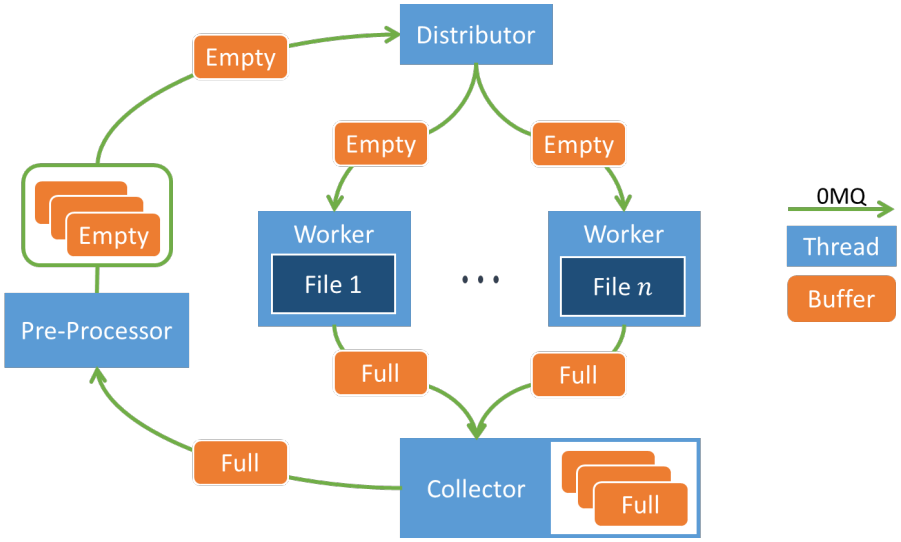


Figure 5.7: Reading from multiple file mirrors on distinct local drives.

data is read using n worker threads, each of which handles a separate input file. Worker threads are coordinated by a distributor thread, which supplies read offsets and empty buffers on request. Workers send filled buffers to a collector thread, which interleaves and re-transmits buffers to the pre-processor thread for indexing and trimming. The pre-processor returns the buffer to the distributor once it has been processed, where it joins the empty buffer queue to be reused. Figure 5.7 shows an overview of the mirrored file-reading process's primary components and connections.

The distributor thread is responsible for launching both the worker and collector threads, and provides workers with jobs to perform and buffers to write to. When reading from n files, the distributor uses an array of n pair sockets to manage connections to worker threads, which are polled in a loop for requests. The distribution of work is done on a first-come first-serve basis, providing a simple means for drives of different speeds to cooperate effectively. Workers which perform well complete their assigned tasks faster, and thus request work more frequently than those that perform badly.

Like the single source reader, packet data is read into 8 MB buffers which are dispatched to the pre-processor in sequence order. Buffers are organised into batches of eight contiguous buffers, such that each batch covers a unique 64 MB segment of the capture. Batches of contiguous buffers are issued to workers in order to reduce the number of seek operations needed during the reading process, without having to adjust buffer granularity. When a worker thread makes a request for work, the

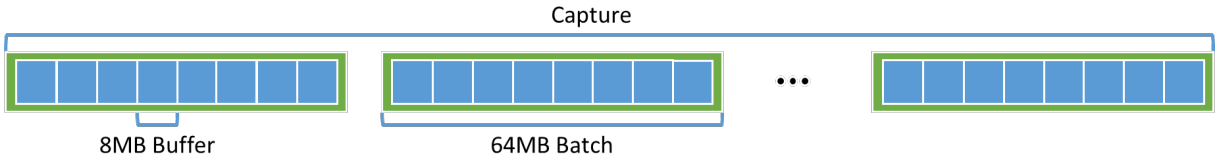


Figure 5.8: Dividing captures into batches of buffers.

distributor supplies it with a batch to process and passes the necessary buffers as additional messages that can be read on demand by the worker. This ensures that if the distributor cannot immediately supply enough buffers at the time of the request, the worker can still begin filling those that are available. Each dispatched buffer in the batch has an associated incremental identifier which identifies its relative position in the overall buffer stream – buffer 0 points to the first 8MB of data, buffer 1 points to the second 8MB of data, and so on (see Figure 5.8).

A job comes to a worker in the form of a start offset and a series of buffers to facilitate the operation. Workers send filled buffers, labelled with their identifiers, to paired sockets in the collector thread, which receives these messages in a polling loop. The collector thread is responsible for redistributing the buffers it receives in the correct order, and uses the unordered map data structure[2] from the C++ standard library to temporarily store buffers that arrive ahead of schedule. When the collector receives a message, it first checks the job's ID to see if it is the message it is waiting for. If the ID of the job matches the next expected ID, the job's buffer is sent to the pre-processor and the collector begins looking for the next job in the chain. If the ID does not match, the message is stored temporarily in an unordered map, indexed by job ID, and the collector moves to process the next received message. Once the expected message is received and resent, the collector retrieves and dispatches any buffers that directly follow the sent buffer in sequence that arrived early.

5.5 Pre-processing Packet Data

This section describes the process by which packet data passed from the capture reader is indexed and prepared for filtering. This process is quite simple in comparison to other aspects of the system, and not particularly time-consuming on its

[2] `http://www.cplusplus.com/reference/unordered_map/unordered_map/`

own.

5.5.1 Parsing Packet Records

The primary function of the pre-processor is to parse packet records from raw input buffers, so that they may be processed in the classification kernel. The process iterates through each packet, collecting header information and copying the required portion of the data into write-combined, page-locked buffers that can be transferred quickly over PCIe to the classifier. This copy is expensive, but also significantly reduces the amount of data needed to be transferred to the GPU. This is particularly true for packets with large payloads (such as those produced by streaming multimedia or file transfers for instance) where the relevant portion of the header may account for only a few percent of the packet's total size. A downside of this approach is that it has the reverse effect when packet sizes are particularly small. If a packet is smaller than the specified filter range (or window size), it is padded with null bytes to bring it to the correct size; this means that if the GPF compiler derives a window size for a filter that is larger than the average packet size, it would net-increase the amount of PCIe traffic and device storage required to process it, if only by a relatively small amount. While not ideal, the significant potential for transfer and storage reduction in larger packets outweigh the minor performance losses possible in smaller packets at this time, particularly since the former is far more likely in most scenarios.

Once the index information has been captured and the relevant data in the packet has been copied, the process moves to the next packet, iterating through the buffer until it is exhausted. The buffer is then returned, and a new buffer is acquired. As packets are not aligned to the beginning of each buffer when read, there is a high probability that a packet will overlap two buffers simultaneously – being partially contained at the end of the first and the beginning of the second respectively. These edge cases are handled with a 64KB overflow buffer that copies and stores both fragments for extraction. As the pre-processor iterates through each packet record, the index information and classifier data buffers are filled and dispatched to the index writer and classifier threads respectively.

Table 5.1: Index Files

File	Index Contents	Record Size
Packet Index	Byte offset for every packet in the capture.	64-bit
Time Index	Index of the first packet to arrive after each second.	64-bit

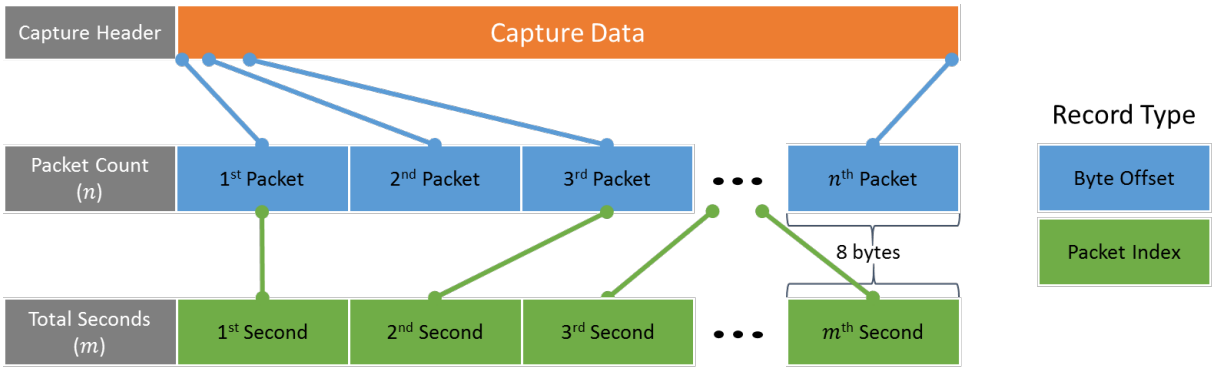


Figure 5.9: Overview of packet and time index files.

5.5.2 Index Files

The indexing component produces a packet index file and a time index file for the capture being processed, as listed in Table 5.1. The packet index file begins with a 64-bit packet count n, followed by n 64-bit bytes offsets. The use of 64-bit values is necessary, as 32-bit byte offsets would limit the maximum capture file size to only 4 gigabytes. The time index file begins with a 64-bit integer t indicating the total time span of the capture in seconds, followed by t 64-bit packet indexes (one for each elapsed second in the capture), which refer back to the packet index file. Specifically, each time index record contains the packet index of the first packet recorded t seconds after the capture start time. If no packets are recorded as arriving during a period spanning m time index records (t_n to t_{n+m-1}), all these records point to the next packet to arrive (t_{n+m}). An overview of the relationship between index files and the capture is provided in Figure 5.9.

5.5.3 Writing Index Files

Index data produced by the pre-processor is passed via 0MQ to an asynchronous listening thread, which writes the data to file. Packet and time index data are collected in separate buffers that are dispatched, once filled, to the listening index writer thread. The index writer determines the type of buffer data received from

the message header, and appends the chunk to the corresponding packet or time index file. Once the capture has been fully parsed, the index writer updates the record count in each index header to reflect the number stored, closing the files once this has completed.

5.6 Summary

This chapter served as an introduction to the classification approach and its requisite components, providing a context for component specific discussions contained in the remainder of this part.

Section 5.1 provides a concise overview of the implementation discussed in this part of the document. This section explained some high-level attributes of design relating to the classification kernel and its supporting host process. The section additionally described the basis for the program encoding used and its importance in informing kernel architecture, and introduced the GPF+ processing abstraction using GPF as a point of comparison.

Section 5.2 provided an overview of the high-level system architecture and its primary components. This section introduced the compiler, capture buffer, pre-processor, classifier and post-processor components, and elaborated on the basic process by which classification results are produced. The section also introduced the 0MQ framework, which connects components executing across multiple threads and execution contexts.

Section 5.3 provided a brief summary of the components introduced in the previous section, indicating where they are discussed in the document. This section also provides more specific context for each of the components, and details their primary roles.

Section 5.4 discussed the process of reading capture data using the capture buffer, either from a single source file, or by using multiple mirrored copies of a file to achieve higher throughputs.

Section 5.5 described the pre-processor thread, which receives buffers from the capture buffer to parse out packet records. The pre-processor packages packet data and passes it to the classifier, while optionally generating indexes of packet byte

offsets and arrival times. These index files were additionally shown to be useful in deriving traffic metrics.

6

Classifying Packets

PACKET classification is performed on the GPU through the execution of a CUDA kernel. This is in turn dispatched and maintained by a dedicated thread on the host system. The classification kernel executes a program compiled from a high-level DSL (see Chapter 7), evaluating filters and extracting field values from packet data delivered from the host (see Chapter 5). This chapter focuses specifically on the CUDA classification kernel's design and implementation. The chapter is broken down as follows:

- Section 6.1 introduces the classification process and kernel architecture described in the remainder of the chapter. This section describes the classification approach and addresses how processing threads avoid synchronisation, and are divided between multiple asynchronous streams.
- Section 6.2 describes the constant memory space, which houses a range of static variables and arrays used by the classification process.
- Section 6.3 details the system registers maintained in the classifier's runtime state memory. These variables are adjusted frequently during the course of execution, and are used to adapt processing to the packets being processed.

- Section 6.4 provides an overview of the classifier's global memory regions, which house raw packet records, working data and results.
- Section 6.5 discusses the packet cache, which acts as the interface to packet data stored in global memory.
- Section 6.6 provides an overview of the gather process, focussing on the layer processing function responsible for initiating cache loads, pruning redundant operations, and dispatching the field processing function.
- Section 6.7 describes the field processing function, which applies the packet data extracted from cache to produce temporary comparison results, extract field data, and update state memory.
- Section 6.8 details the filter process, which applies the temporary comparison results generated by the field processing function in the gather process to evaluate filter predicates.
- Section 6.9 concludes with a summary of the chapter.

6.1 Classification Process

This section expands upon the basic underpinnings of the abstract GPU classification process presented in Section 5.2, describing the process in more detail as a prelude to a discussion of its implementation. The following subsections provide a more thorough breakdown of how classification and filtering is performed, and conclude with an overview of the features provided by the approach. The remainder of the chapter focuses in greater detail on individual aspects of this process.

6.1.1 Process Overview

The classification process is implemented as a device-side object that executes within a CUDA kernel. This object evaluates packet records against a byte-based filter program, with the aid of constant and register memory, recording results as ordered arrays in device memory. Results include bit-based filter results computed from filter predicates, and field values extracted directly from protocol headers. Listing 10 shows how the GPF+ classification object is initialised and executed

Listing 10 Declaring and using the GPF+ classifier object.

```
1 __global__ void GpfProcessor(int stream_no)
2 {
3     GpfVm vm(stream_no);
4     vm.Gather();
5     vm.Filter();
6 }
```

from within a CUDA kernel. The object constructor is passed a single argument by the executing kernel, identifying the execution stream it is associated with (see Section 6.1.5). All other configuration variables and memory pointers are stored in constant memory to support fast global access from all classifier functions (see Section 6.2).

The classification object provides a constructor and two methods, dividing classification into three distinct phases: initialisation, gathering, and filtering. Initialisation is the simplest and shortest phase, during which the constructor populates the classifier's state and sanitises memory in preparation for use. The constructor takes a single argument, `stream_no`, which corresponds to the kernel's execution stream (see Section 2.8.2).

The gathering phase is responsible for extracting field values and performing field comparisons from packet data in device memory, directed by a program stored in GPU constant memory (see Section 2.6.2). The gather process is the only function which directly interacts with packet data, which it accesses through a register-based packet cache in order to reduce interaction with device memory, and thereby conserve bandwidth (see Section 2.5). Gathering additionally relies heavily on register-based state memory to track protocol information, prune redundant operations, and locate field offsets in packet data. The gather process iterates thirty-two times, in order to evaluate 32 consecutive packets in each thread. This is done to provide sufficient input data for the filtering phase.

The filtering phase is performed once all iterations of the gather process have completed, and uses the produced comparison results to evaluate filter predicates and store filter results. This process is both encoded and performed similarly to the predicate evaluation kernel used in GPF (see Section 4.4.3), but uses bit-based encoding in place of boolean values to encode results. Once filtering concludes, filter and field results can be copied from device memory back to the host.

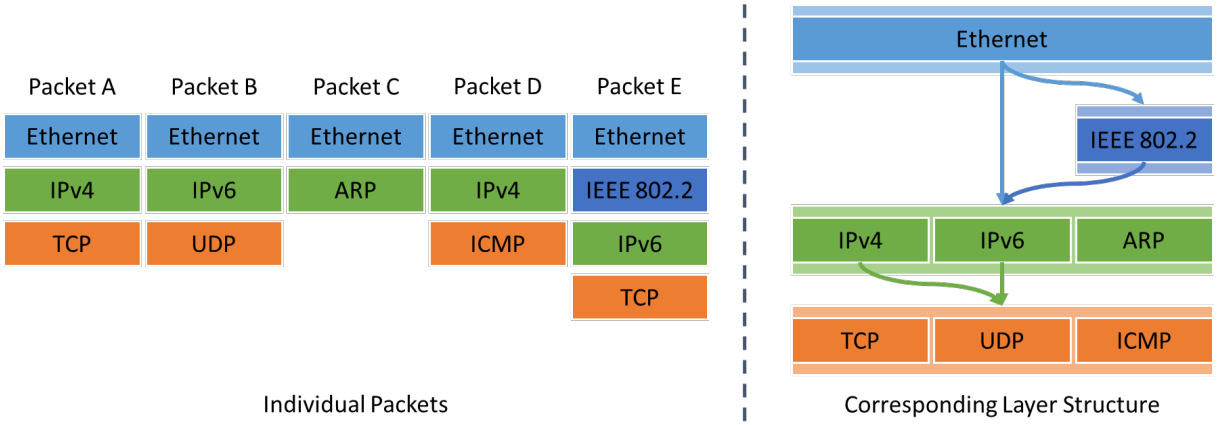


Figure 6.1: Example layer structure corresponding to protocol structure of the illustrated packets.

6.1.2 Layering Protocols

The gather process conceptualises packet records as an ordered stack of protocol headers, where each header in a packet is associated with a separate stack layer. Layers contains a set of one or more mutually exclusive protocols, such that a particular packet can match no more than one protocol in a layer at a time. The protocol used by a packet in a particular layer is determined in the previous layer. Layers may be skipped by the thread only if no protocols in the layer match the protocol specified by the previous layer. The process concludes when either all layers have been processed, or the executing warp fails to identify a valid child protocol matching a subsequent layer in any of its threads. A high-level illustration of a hypothetical layer structure that matches the structure of five example packets is shown in Figure 6.1. The second layer in this illustration, corresponding to the IEEE 802.2 protocol, is ignored in all but one packet[1].

As thread warps are essentially SIMD processing units, individual threads terminating execution or skipping certain header layers does not actually prune processing or significantly improve efficiency on its own; rather, threads are conceptually switched from an active state to an inactive state, cooperating with the remaining active threads in the warp to facilitate cache loads and storage operations. As long as a single thread in the warp depends on a particular protocol or layer, that protocol/layer will be evaluated (if not entirely processed) by all 32 threads, regardless of their internal state. Pruning is possible, however, if all threads in a

[1]The IEEE 802.2 protocol is associated with the data-link layer of the OSI model, and is not part of the TCP/IP suite.

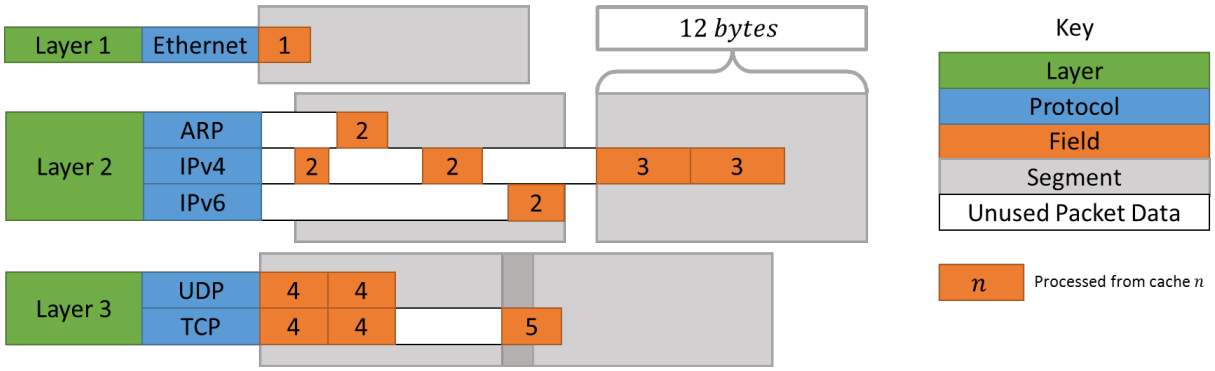


Figure 6.2: Illustration of Layer Processing Function

warp reach an inactive state, or if a protocol/layer is irrelevant to an entire warp. Through a unanimous warp vote, threads can abort processing, or skip an irrelevant protocol or layer, to speed up processing.

Layers are subdivided into one or more 12-byte cache segments. Segments are loaded in to cache only once to extract all relevant fields defined for each protocol in turn, helping to reduce redundant data loads by allowing a single cache load to be used to process multiple protocols. Fields are located by adding a bit offset (specified in the program) to the start offset of the cache segment, derived from the current protocol offset held in state memory. Cache loads and field extractions are handled by the packet cache, which is discussed in Section 6.5.

An illustration of the layer processing function applied to a hypothetical three-layer filter program is provided in Figure 6.2. The first layer contains only the root protocol (with leading bytes cropped) and a single relevant field, and thus requires one single cache load. Layers two and three contain multiple potential protocols and need more than one cache load to cover all included fields. The cache chunks in a layer may overlap if a prior chunk fails to fully encapsulate a field, as in the third layer of the graphic.

6.1.3 Warps and Synchronisation

In order to fully benefit from runtime layer and protocol pruning, the gather process and other processes comprising the classification virtual machine avoid block level synchronisation to ensure complete warp independence. Block level synchronisation constrains the processing speeds of all warps in a block to that of the slowest executing warp. With respect to the approach discussed, faster warps would

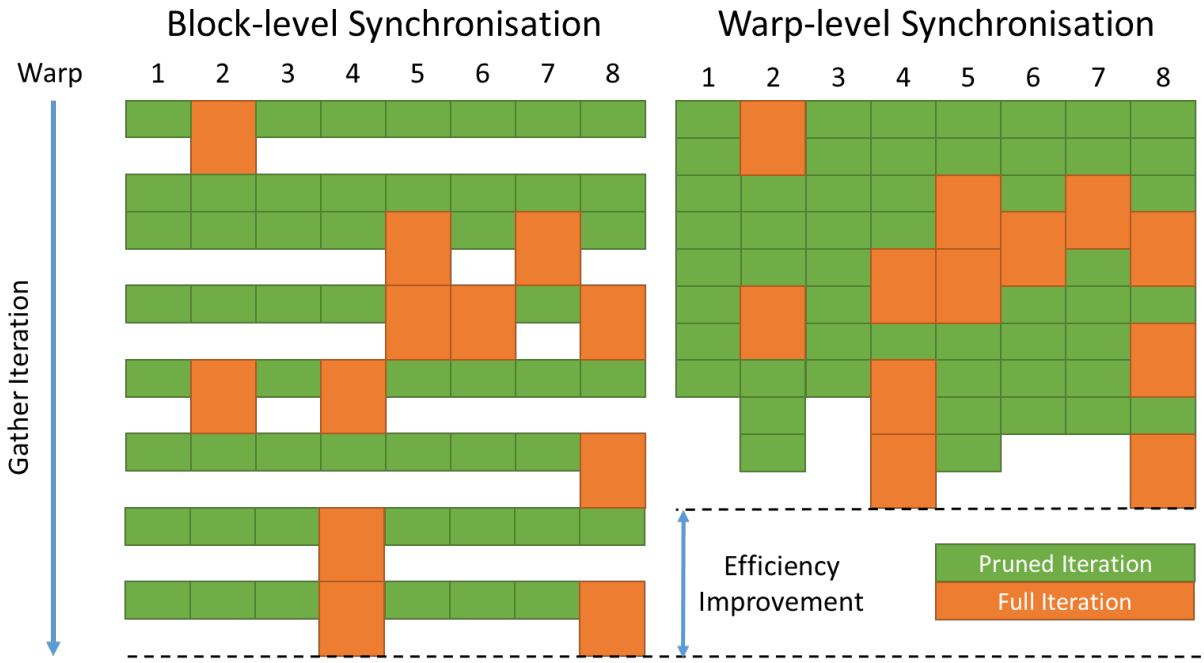


Figure 6.3: Simplified illustration of the effects of synchronisation.

necessarily block at synchronisation points within the gather process, wasting any potential time gained by pruning. In the worst case, fast warps would not be able to skip a layer, cut an iteration short, or avoid a protocol unless all other warps could as well. Where thread cooperation is necessary, data is passed through warp vote and shuffle functions, as these provide implicit and future-proof warp-level synchronisation [81] (see Section 2.7). A rough illustration of block- and warp-level synchronisation, and the effects they can have on execution efficiency, is provided in Figure 6.3.

The figure depicts the comparative efficiency of two blocks, each containing eight warps (256 threads), executing eight gather iterations using block and warp level synchronisation respectively. Warp-level synchronisation allows warps to execute with greater independence and fluidly within a block, only synchronising (implicitly) on block termination. This improves the utilisation of multi-processor resources, and provides finer thread granularity than explicit synchronisation at a block level. This allows individual warps to benefit more frequently from pruning operations, which is particularly powerful when requested filters or fields depend, in part, on infrequently occurring application-specific protocols.

Pruning reduces the set of layers and protocols that are processed by a warp to only those which are relevant to the 32 threads executing within it, thereby removing

all redundant divergence operations from the warp's execution path. It does not, however, impact upon the necessary divergence which occurs when more than a single protocol remains in a particular layer. As warps are SIMD units, this divergence cannot be avoided through pruning and necessarily results in some level of inefficiency. This divergence is managed in the architecture by merging similar abstract operations where possible (such as the caching packet data) and minimising divergent operations within potentially serialisable segments of code.

6.1.4 Assigning Packets to Threads

To avoid requiring block level synchronisation between the gather and filter processes, both processes must operate on the same packet records in each warp. That is, the comparison data used by the filter process to evaluate predicates must be produced in the gather process by threads in the same warp; this ensures that the filter process is not dependent on computation performed in other warp threads. If this is not enforced, synchronisation would be required to prevent faster warps from attempting to apply incomplete comparison results. This would in turn constrain all warps in a block to the performance of the slowest performing warp, which is wasteful of resources. With this in mind, execution is organised such that the i^{th} iteration the gather process produces and stores the integer result that the i^{th} thread of the warp will operate on in the filter process. Since there are 32 threads in each warp and 32 bits in each integer processed by the filter process, the gather process iterates 32 times.

Figure 6.4 provides a high level illustration of how packets are divided between warps in a block of 128 threads, within both the gather and filter processes. In the figure, each thread block processes 4,096 packets between four warps. Each warp evaluates 32 packets per iteration, ultimately processing 1,024 contiguous packets over 32 iterations. Each iteration produces a single 32-bit integer per warp (1 bit per thread), and thus 32 iterations produces 32 integers worth of filter data. These integers are processed in a single pass of the filter process, with the results of the i^{th} gather iteration providing the input for the i^{th} warp thread in the filter process. This avoids the need for block-level synchronisation, allowing warps to execute independently of one another.

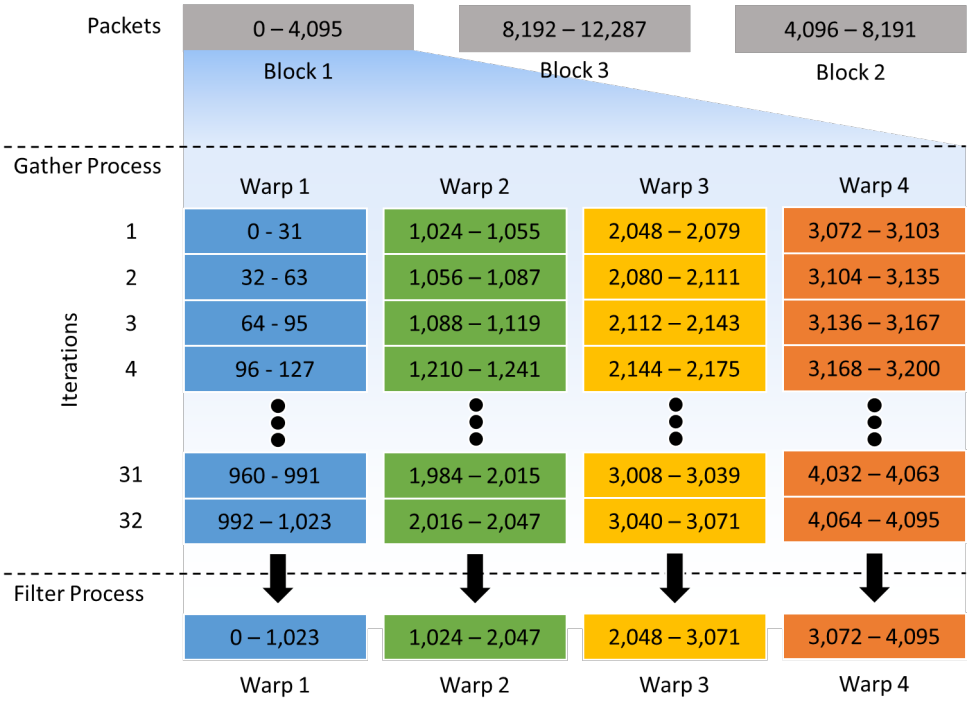


Figure 6.4: Breakdown of packet processing order for a 512 thread block by warp and process.

6.1.5 Execution Streams

The classification host thread executes the classification kernel using asynchronous streams (see Section 2.8.2). The current implementation uses four streams by default (as this provided the best balance of performance and memory utilisation), but this is configurable at runtime. Each time the classifier is executed, it operates on a new buffer of packet data within a dedicated stream. This allows data transfer operations occurring in one stream to overlap with computation performed in another, better utilising device resources.

Each stream is assigned its own sub-regions within global memory to read data from and write data to. The execution stream number is passed as an argument to the kernel so as to read and write data within the correct regions of global memory (see Section 6.1.1). The execution stream is the only argument passed to the kernel; all other configuration data is made accessible to the kernel through the constant memory space (see Section 6.2). No other aspect of execution is affected by the execution stream.

6.2 Constant Memory

Constant memory is used to provide fast access to runtime constants derived during the compilation process (see Section 7.3.2) and host-side initialisation.

The constant memory space stores the GPF+ program instructions, an integer lookup table, global memory pointers, and several constant integers. Constant memory is typically only written to twice during capture classification; it is initially populated prior to the first kernel invocation, and is slightly modified prior to the final kernel invocation, adjusting the packet count to reflect a partially full final buffer.

6.2.1 Program Memory

The GPF+ classifier maintains two program regions for the gather and filter processes respectively, which remain unaltered for the entirety of capture classification. The gather program is currently allocated 14 KB of constant memory storage, while the filter program is allocated 2 KB. The gather program is provided more storage space, as the filter program is usually comparatively small. The program memory capacity provided is sufficient to encode hundreds of filters simultaneously.

Programs are accessed as a linear array of bytes, indexed by a register-based program counter which changes dynamically during the course of execution. This encoding as bytes differs from the encoding used in the GPF prototype (see Section 4.4.3), which used integers as basic elements to allow for integral values (required for filter comparisons) to be encoded directly into the program instruction stream. This approach was quite wasteful of the limited constant memory resources, as the instruction codes which make up the majority of the program require far less than a byte to successfully encode. Encoding instructions as bytes instead of integers thus quadruples the instruction capacity per KB of program memory. As a consequence, however, program memory can no longer properly contain the integral values needed for runtime comparison operations. These values are now contained within a separate lookup table of 32-bit integers, indexed by 8-bit values encoded in the program byte stream.

6.2.2 Value Lookup Table

The value lookup table is a 1 KB constant value lookup array with a limited capacity of up to 256 32-bit integral records. The limited capacity of the value table is a consequence of using bytes for indexing, but could be worked around if this proved necessary. For instance, the lookup table could be divided into n banks of 256 integers stored in an $n \times 256$ element integer array, and indexed using a pair of bytes. Alternatively, the command width of program memory could be widened by using 2-byte short values instead of 1-byte char values, at the cost of bloating the program memory footprint by a factor of two.

This appears unnecessary in the general case, due to the Match Condition Redundancy and Matching Set Confinement properties of filter sets [113]. The former states that there is a high level of redundancy within filter comparison values, while the latter states that the number of unique match conditions for any m-bit field is significantly less than the 2^m values it can represent. As the value lookup table only contains unique records, its limited capacity is sufficient for potentially hundreds of filters, although this is dependant on the level of redundancy within a particular filter set.

6.2.3 Memory Pointers

Device memory pointers for packet data, results memory and working memory are stored in constant memory, and are globally accessible to all functions within the kernel. All pointers use an integer type to provide optimum read bandwidth, and are accessed simultaneously by all participating threads to avoid read serialisation (see Section 6.4).

6.2.4 Runtime Constants

Runtime constants are stored as integers in GPU constant memory in order to minimise latency. Constants are read simultaneously by all threads, and thus avoid additional latency due to access serialization (see Section 2.6.2). These constant values cannot be altered between individual streams when employing asynchronous execution, but may be adjusted between kernel invocations on the host. This is

Table 6.1: Table of Kernel Runtime Constants

Group	Constants
Packet	Packet Count
	Layer Count
	Packet Size
Working	Working per Stream
	Working per Block
	Working per Warp
	Values per Packet
Result	Filter Memory per Stream
	Field Memory per Stream
	Filter Results per Packet
	Field Results per Packet
	Filter Array Size
	Field Array Size

only done prior to the final invocation of the classifier in order to change the packet count to reflect the number of packets in the final buffer. Runtime constants are primarily used as control variables in loops, and in combination with state memory to read from and write to the correct offsets in global memory.

The classification kernel relies on thirteen specific constants, which are listed in Table 6.1 and described below. While some of these constants could be derived at runtime, it is more efficient to pre-calculate frequently used transformations and avoid unnecessary processing overhead.

Packet Count The number of packets processed in each stream.

Layer Count The number of distinct protocol layers included in the program.

Packet Size The cropped size (in bytes) of each packet record in device memory.

Working per Stream The allocated working memory region size (see Section 6.4.2) per execution stream, in bytes.

Working per Block/Warp The amount of working memory used per thread block/warp, in bytes.

Values per Packet The number of distinct working values stored per packet.

Filter/Field Memory per Stream The allocated filter/field memory region sizes per execution stream, in bytes.

Filter/Field Results per Packet The number of distinct filter/field results stored per packet.

Filter/Field Array Size The total memory required to store a single result for all packets in a stream.

6.3 State Memory

State memory is composed of globally accessible internal registers that store dynamic, contextual information that guides the execution process. State memory is comprised of fifteen distinct values, which are tightly packed into five registers. State memory variables are maintained within vector types, which allow multiple smaller values (such as `short` and `char` types) to be stored in a single 32-bit register. Without vectorised state variables, each 8-bit and 16-bit state variable would consume a separate 32-bit register, both wasting memory in the register file and increasing register pressure. The increased register pressure alone would prevent the kernel from ever achieving full occupancy (see Section 2.6.1), and would therefore significantly impact performance by not being able to utilise a device's full processing capacity [81].

While the use of vector types reduces register requirements of state memory by roughly two thirds, it does have a minor associated cost; because multiple variables share a single register, the likelihood of register read-after-write conflicts is increased (see Section 2.6.1). This can be designed against to an extent by spacing out multiple consecutive operations to components of a single vector, providing time for the register to be written before attempting to access it again. To maintain minimal register utilisation, signed and unsigned vector types are also used to store loop control variables and other sufficiently small runtime variables. These are maintained within a particular scope, rather than state memory, so that the registers they utilise may be reallocated for other operations on scope closure.

An exhaustive list of the defined global state memory variables, and how they are stored across vectors, is provided in Table 6.2. A short summary of each of these variables follows:

Packet Index The index of the current packet being processed by the thread, relative to the beginning of the kernel's data stream. Each thread is allocated

State Variable	Data Type	Sub-Variables	Applicable To
Packet Index	`int`	–	Gather
Offsets	`short2`	Program Offset	Gather & Filter
		Protocol Offset	Gather
Protocol State	`uchar4`	Active	Gather
		Current	Gather
		Next	Gather
		Length	Gather
Thread State	`uchar4`	Warp	Gather
		Stream	Gather & Filter
		Iteration	Gather
		Alignment	Cache
Cache State	`uchar4`	Reciprocal A	Cache
		Reciprocal B	Cache
		Reciprocal C	Cache
		Cache Lane	Cache

Table 6.2: Summary of State Variables

32 non-contiguous packets to process (one packet per iteration of the gather process, with a 32 packet stride between iterations) and thus this value increments a total of 32 times in the thread's lifetime.

Program Offset The current command offset from the start of either the gather of filter operation's program memory, depending on the current execution context. The program offset is incremented frequently (although not after every operation), but cannot currently be manually adjusted.

Data Offset The number of bytes from the start of the packet record to the start of the current protocol layer. The data offset is incremented by the length of the current protocol just before switching to the next protocol at the beginning of each new layer. It is reset to zero at the beginning of each new iteration of the gather process.

Active Protocol The unique numerical ID of the protocol that the thread warp is currently processing from program memory. This value is set before processing fields for a particular protocol, and thus changes regularly as each layer tests each of its applicable protocols.

Current Protocol The unique numerical ID of the protocol associated with the current layer of the packet being processed. Each packet can only be associated with a single protocol ID per layer, which is used to determine if an

operation applies to the current thread or not by comparing it to the ID of the current active protocol. The current protocol uses the reserved ID of `0xFF` to indicate that a thread has reached the end of processing, which prevents the thread from matching any subsequent active protocols.

Next Protocol The unique numerical ID of the next expected protocol after the current protocol. This value is initialised to `0xFF` at the beginning of each iteration of the gather process, but may be set by a successful switch statement during the course of evaluating the layer. This effectively terminates a thread if no switches relevant to the current protocol succeed before the end of the layer is reached.

Protocol Length The length of the current protocol in bytes, which is initialised to the appropriate protocol's default value at the onset of a layer. This state variable is unique, in that it is the only state variable that can be written to directly or via an integral expression from high-level filter code during the course of evaluating a layer. The protocol length is used to update the data offset on conclusion of a layer, which allows layers to support protocols of different and variable lengths.

Warp The index of the warp that contains the current thread within the executing thread block. This value is rarely used for tasks other than reading and writing working memory, but is used frequently enough over the course of a threads life-time to benefit from reserved storage.

Stream The stream index of the kernel invocation that contains the thread. The stream index is the only variable not passed through or derived from constant memory (as all streams share the same constant memory space), and is instead passed as the only argument of the kernel function. The stream index is used by both the gather and filter functions of the classifier to determine which areas of memory to read and write to, as each stream is provided its own working space and results storage.

Gather Iteration The current gather iteration, where each iteration corresponds to a separate and distinct packet processed by the thread. This value is incremented on completion of each iteration of the gather process, until it reaches a value of 32. At this point, the gather process terminates and the filtering process begins.

Alignment The byte alignment of the start offset of the currently loaded cache data, as a 2-bit (0-3) offset from a 4-byte alignment boundary. This value is used to handle the disparity between the 1-byte alignment of headers in packet data, and the 4-byte alignment requirement imposed when reading data as integral values from global memory. Its use is discussed in Section 6.5.3.

Cache Reciprocal A/B/C The reciprocal thread indices for each of the three register shuffle operations used by the packet cache. These operations are performed after a 16-byte coalesced read to get the correct packet data to the correct thread. Due to their compact size, these values are pre-calculated and stored once in the threads constructor, to avoid repeating all three calculations hundreds of times during the course of 32 iterations of the gather process.

Cache Lane The thread's prescribed lane in the cache unshuffling process. It is required to both retrieve the right packet data from global memory, and to initially calculate the three cache transforms discussed above. As with the cache reciprocals, the cache lane state variable could be derived at runtime, but was included to avoid repeated re-calculation.

6.4 Global Memory

Global memory is housed in device DRAM and contains packet data, results memory, and working memory. All global memory regions are allocated once during system initialisation, and subsequently reused across multiple kernel invocations. Global memory regions are therefore fully accessible to all streams, and accessing the correct offsets in device memory spaces therefore requires support from constant and state memory. Global memory requirements are determined during system initialisation from setup parameters and compiled program attributes, and are not affected by the composition of individual captures.

6.4.1 Packet Data

Packet data is passed to the classification host thread from the pre-processor in page-locked, write-combined buffers (see Section 2.8.1), which are copied asyn-

chronously to the device in an assigned stream. The kernel is launched asynchronously in that stream immediately thereafter, and begins executing on the device as soon as the buffer copy is completed. Packet data is contained as an array of integers in device memory, and is accessed through the packet cache. The packet cache handles both the loading of raw data from global memory, and the extraction of fields from that data for the classification process. The packet cache is discussed in detail in Section 6.5.

6.4.2 Working Memory

Working memory is a region in global memory that temporarily stores bit-based comparison results generated during the gather process. Like other regions in global memory, working memory is partitioned evenly between participating streams. Each stream partition is itself sub-divided into a set of integer based result arrays, where each array stores the results of a particular comparison or predicate for every packet contained within the stream. These results are not returned to the host, and are instead used as elements in the filter process.

Working memory is the only region of global memory that is accessed from both the gather process (which operates on a single packet per thread) and the filter process (which operates on 32 packets per thread). The gather process issues the majority of working memory write transactions, but does not read data from it. Prior to writing to working memory in the gather process, all comparison results are collected as a 32-bit integer in the first thread of the warp; this thread alone writes the comparison results to working memory. The filter process reads this comparison data, using the contained records to produce filter results and additional working results, the latter of which may then be used by a subsequent filter.

Working memory elements are assigned numeric identifiers at compile time, which correlate to their array's section of working memory. Working results are organised in device memory by warp to avoid block level synchronisation between the gather and filter processes (see Section 6.1.3). Figure 6.5 illustrates this layout when n working variables are defined.

The correct read and write offsets for the gather and filter process are derived using values in constant and state memory, as shown in Listing 11. The offset of the current warp is determined first, and is subsequently used to calculate both

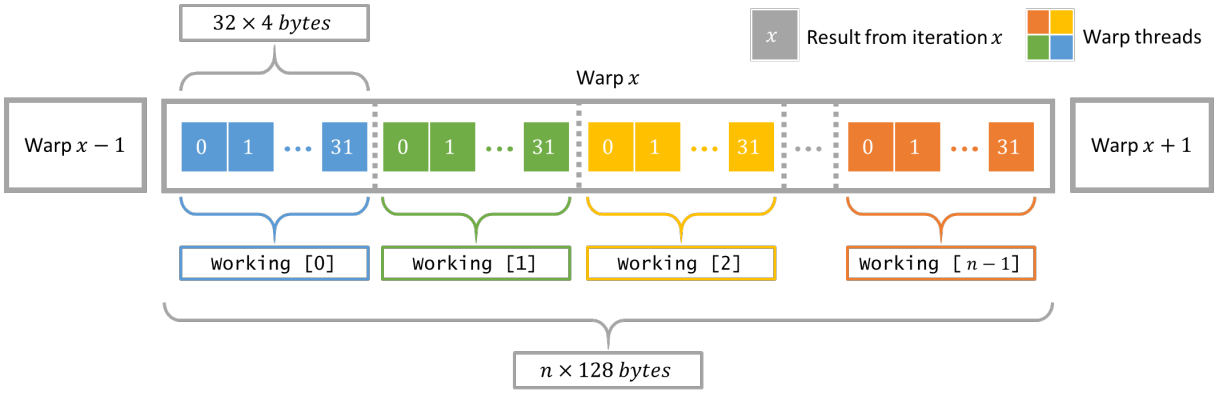


Figure 6.5: Layout of working memory by warp for n working variables.

Listing 11 Deriving the gather and filter offsets for a warp in working memory.

```
1 int warpOffset = Constant.WorkingPerStream * State.Stream + Constant.
     WorkingPerBlock * blockIdx.x + Constant.WorkingPerWarp * State.Warp;
2
3 int gatherOffset = warpOffset + workingIndex * 32 + State.Iteration;
4 int filterOffset = warpOffset + workingIndex * 32 + threadIdx.x & 31;
```

gather and filter offsets. The gather process uses this warp offset to transform the working index values (supplied by the GPF+ program) into an offset in working memory. This write is issued from the first thread in the warp, and occurs once per warp per iteration.

The filter process reads the integer elements stored by the gather process, with each thread accessing an element produced in a different gather iteration. This pattern coalesces perfectly within the warp. The filter process additionally uses this offset to write new elements, which are created to temporarily house sub-predicates (such as those needed to handle parenthesis) needed by subsequent filters. Further details on how working memory is applied within the gather and filter processes are provided in Section 6.7 and 6.8 respectively.

6.4.3 Results Memory

The results of operations which are intended to be transferred back to the host process are stored in a global memory region referred to as results memory. Results memory currently has two separate and distinct banks that store field values and filter results. Each bank is sized appropriately at compile time, and shared equally between all executing streams. Field values are stored as 32-bit integers, while

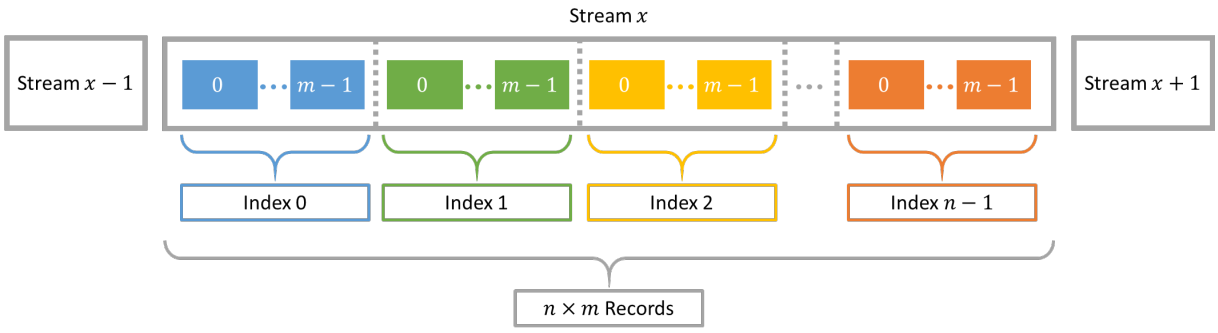


Figure 6.6: Memory layout for n filter or field results in a stream containing m packets.

Listing 12 Deriving the gather and filter offsets for a warp in results memory.

```
1 streamOffset = Constant.ResultMemoryPerStream * State.Stream;
2 arrayOffset = Constant.ResultArraySize * resultIndex;
3
4 int filterOffset = streamOffset + arrayOffset + blockDim.x * blockIdx.x +
      threadIdx.x;
5 int fieldOffset = streamOffset + arrayOffset + State.PacketIndex;
```

filter results are stored as 1-bit boolean results encoded within integers. Field values are currently only writeable by the gather process, and can only be read on the host.

Outputs are stored in their respective banks within equally sized contiguous arrays (one for each filter or field extraction operation), organised in packet index order. For instance, if a program defines n separate filters and each stream processes m packets at a time, then the portion of the filter memory bank accessible to a given stream is equally divided into n adjacent arrays of m filter results. The layout of field and filter records is illustrated in Figure 6.6. The psuedocode provided in Listing 12 shows the offset calculations necessary to determine the stream offset in device memory, and the result array offset within the stream. The `Result` prefix used in psuedocode is a stand-in for either `Field` or `Filter` prefixes, depending on the context. These offsets are used by the gather process to locate field results, and by the filter process to locate filter results.

This pattern coalesces fully on writes from both the gather and filter processes, which helps to reduce bandwidth utilisation and latency. Once a particular stream has completed, the relevant portion of both the field and filter banks are copied back to a buffer on the host to be stored. The buffer is passed to an output writer thread using a 0MQ channel, which appends each result array contained in the

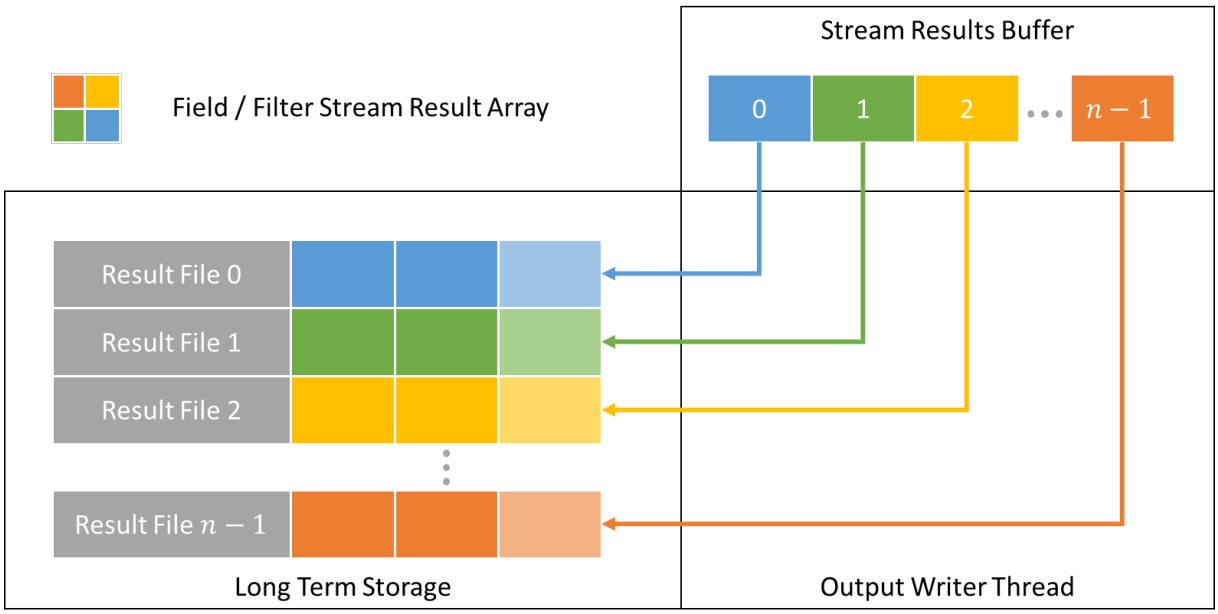


Figure 6.7: Writing four filter / field results from a single buffer to multiple result files.

buffer to a dedicated file. Figure 6.7 illustrates this for four separate filters or extracted fields.

6.5 Packet Cache

High performance classification requires efficient access to packet data as a necessary prerequisite, as all useful classification results and processing outputs generated by the classifier depend on reading and evaluating at least one fragment of packet data. As packet records are relatively large, and are distributed in memory such that they interfere with coalescing [66], accessing packet data directly without optimisation introduces a bandwidth bottleneck that could severely compromises kernel performance. The classifier is supported by a packet cache to mitigate this bottleneck, reducing redundant device memory accesses, and improving bandwidth efficiency on those loads that are required.

The packet cache is implemented as a set of state-based utility functions within the classification object executed by the kernel. The packet cache uses a 16-byte integer array in register memory to temporarily store chunks of header data, thereby helping to reduce dependency on global memory reads. The packet cache functions

include a cache load function which fills the cache array, and a field extraction function which extracts field values from the cache array. These two functions are supported by several state memory variables, and are guided by a single compiler-generated instruction in program memory.

6.5.1 Approach

The cache load function, by allowing threads to coalesce reads more often, reduces the performance impact of reading chunks of packet data from GPU device memory. The implementation of this function uses groups of four cooperating threads to load 16 byte cache chunks from four consecutive packets, one at a time. On completion, the first thread's cache contains the first integer of each packet chunk, the second thread's cache contains the second integer, and so on. As threads read from adjacent indexes, these reads coalesce, improving bandwidth efficiency [18, 22]. These integers are then redistributed (or transposed) between group threads using warp shuffle operations [22], such that the first thread cache contains all 16 bytes of the first packet, the second thread contains all 16 bytes of the second packet, etc. Larger caches would promote higher coalescing at the expense of additional on-chip register storage and warp shuffle overhead to store and organise results.

An illustration of the derived cache load process is provided in Figure 6.8, which charts the four global memory copies and three shuffle operations performed to read and reshuffle packet records. Figure 6.9 shows the performance of this approach on a GTX Titan when applied to fully caching 128 byte records, in comparison to caching records using a direct and fully uncoalesced approach. Performance was measured using Nvidia Nsight 4.2 for Visual Studio 2012, with each configuration being timed ten times and averaged. All tests utilised a 16 byte cache in register memory, sufficient to hold four integers at a time.

These results show significant improvement in both L2 and RO cache performance, but lower overall performance when using texture objects or CUDA arrays. The implementation of the load process uses the Read Only cache to load packet data, as it performed particularly well in initial tests, achieving a maximum throughput over 30% higher than any other direct or shuffled read configuration.

The remainder of this section discusses the caching process in more detail.

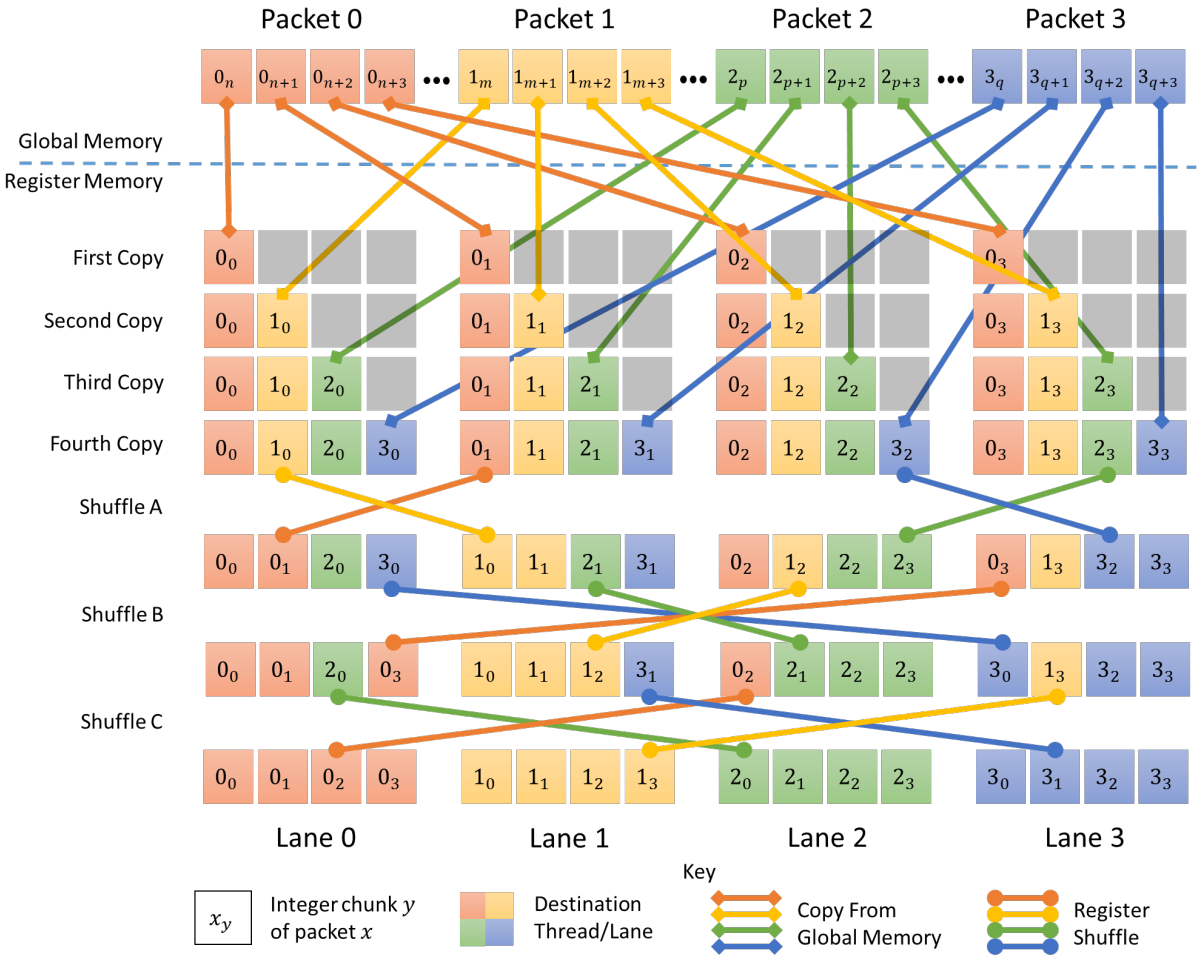


Figure 6.8: Illustration of the cache load process for a four-thread shuffle group.

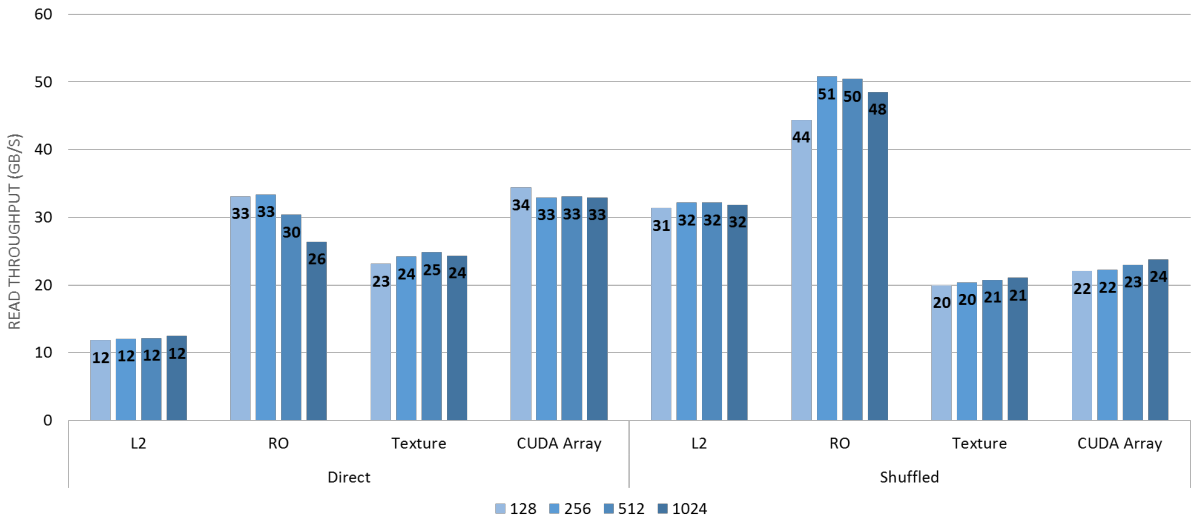


Figure 6.9: Average read throughput achieved using direct and shuffled access.

Listing 13 Initialising runtime-constant cache state variables.

```
1 State.CacheLane = threadIdx.x & 0x3;
2
3 State.ReciprocalA = (5 - State.CacheLane) & 0x3;
4 State.ReciprocalB = 3 - State.CacheLane;
5 State.ReciprocalC = (2 + State.CacheLane) & 0x3;
```

Table 6.3: Possible Reciprocal values

Cache Lane	0	1	2	3
Reciprocal A	1	0	3	2
Reciprocal B	3	2	1	0
Reciprocal C	2	3	0	1

6.5.2 Cache State Variables

The packet cache has 5 dedicated state memory variables that it uses to read and extract data, of which four remain constant with respect to a particular thread (see Table 6.2). These state variables reside in the same vectorised register, and are set in the constructor as shown in Listing 13. The fifth variable, `Alignment`, changes on each cache load; this state variable is more complex, and is discussed separately in Section 6.5.4.

The `CacheLane` state variable can only take on one of four values, as there are only four possible values that each reciprocal value can hold. These are summarised in Table 6.3. For each operation in turn, the reciprocal values indicate to the thread which cache array index it should exchange, as well as the cache lane it should exchange with. For example, let $C_x(y)$ be the integer at index y in cache lane x. `ReciprocalA` is used during the first shuffle operation to swap the value in $C_0(1)$ with $C_1(0)$, and $C_2(3)$ with $C_3(2)$, corresponding to Shuffle A in Figure 6.8. `ReciprocalB` and `ReciprocalC` follow the same pattern, together redistributing all integers to the correct location within their respective thread's cache.

6.5.3 Value Alignment

Despite loading and storing 16 bytes of data during each load, only 12 bytes of the retrieved data is actually usable. The remaining 4 bytes provide necessary slack space to handle cache loads that do not start on a 4-byte boundary. The need for this slack space arises due to protocol headers being located at unknown

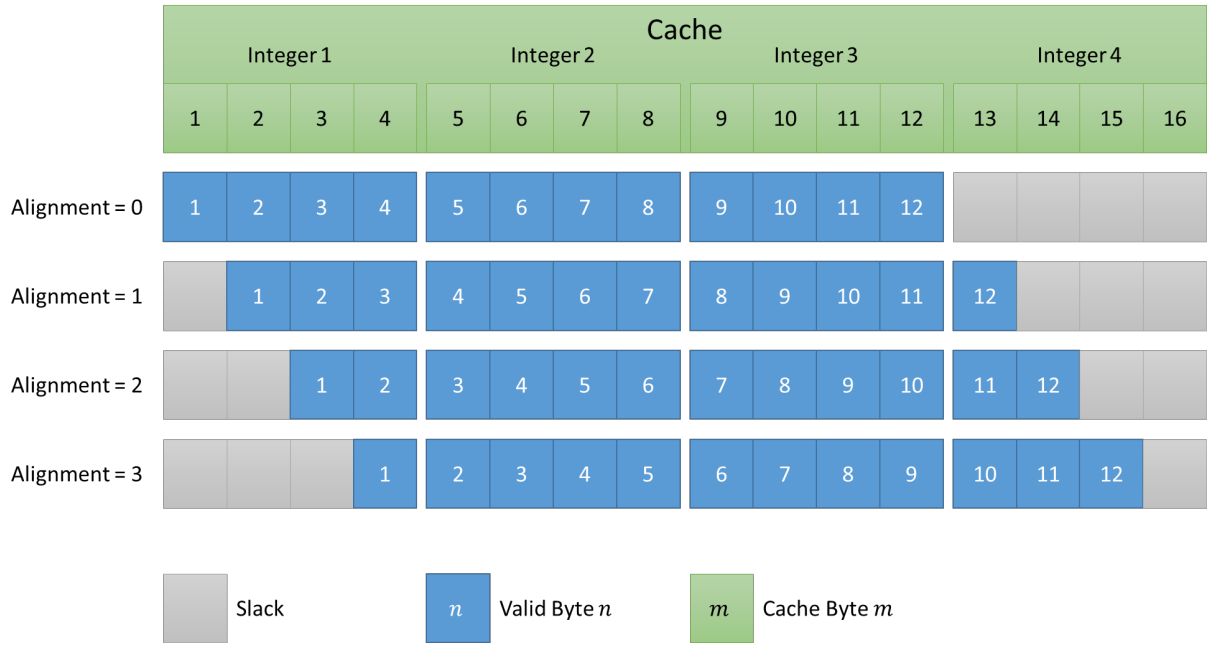


Figure 6.10: Effect of chunk alignment on byte positioning within cache registers.

offsets and aligned on byte rather than integer (4-byte) boundaries. As a result, protocol header offsets are not guaranteed to coincide with integer boundaries, an may potentially start on any byte offset. This complicates matters, as integer loads will not work if they do not lie on a 4-byte boundaries [87].

As a result of this difference in alignment, when a cache chunk is loaded there is a distinct possibility of acquiring and storing up to three leading bytes, which wastes up to 75 % of a cache integer. Figure 6.10 shows the effect of all possible alignments on the storage of a 12-byte cache chunk within cache registers, illustrating why four integers cannot store 16 bytes of packet data when alignment might vary between threads.

Alignment is maintained as an attribute of the Thread State vector in state memory (see Section 6.3) and is used to correctly locate each field's byte offset in cache. Alignment is calculated at the start of the cache load through a bitwise modulus (see Section 6.5.2), but is only used during field extraction (see Section 6.5.5).

6.5.4 Filling Cache

The cache load function is called at the start of each new layer, populating the thread's cache with a 16-byte segment of packet data. Psuedocode for the function

Listing 14 Cache load function psuedocode.

```
int ChunkOffset = GatherProgram[ProgramOffset++];
int Reciprocals[] = { State.ReciprocalA, State.ReciprocalB, State.
    ReciprocalC };

//find byte offset and alignment
int byteOffset = State.ProtocolOffset + ChunkOffset;
int intOffset = byteOffset >> 2;
State.Alignment = byteOffset & 3;

//get pointer to cache chunk
int startPacket = State.Stream * State.PacketCount + State.PacketIndex & 0
    xFFFFFFFC;
int* startOffset = Constant.PacketPtr + startPacket * Constant.PacketSize;

//load chunk cooperatively
for (int k = 0; k < 4; k++) {
    int currOffset = __shfl(intOffset, k, 4) + State.CacheLane;
    cache[k] = __ldg(startOffset + Constant.PacketSize * k + currOffset);
}
//shuffle caches and reverse byte order
for (int k = 0; k < 3; k++) {
    int index = Reciprocals[k];
    int working = __byte_perm(cache[index], 0, 0x0123);
    cache[index] = __shfl(working, index, 4);
}
//reverse byte order on stationary indexes
cache[State.CacheLane] = __byte_perm(cache[State.CacheLane], 0, 0x0123);
```

is provided in Listing 14, which encodes the process illustrated in Figure 6.8.

The first step of the process involves deriving both the integer offset of the thread's associated cache chunk, and an integer pointer to the start of the first cache chunk in the shuffle group. To find the integer offset of the thread's associated cache chunk, the byte offset of the current protocol header (read from state memory) is added to the protocol-local byte-offset of the cache chunk (read from program memory).

While the chunk offset is constant for all packets, the byte offset of the header may vary between cooperating threads based on the results of prior layers. Note that `intOffset` points to the start of the integer in which the first byte of the cache chunk is contained, and not the byte itself. `Alignment` stores the byte index of the first byte of the cache in the integer at `intOffset`, and is a necessary component of the field extraction process (see Section 6.5.3 and Section 6.5.5). The thread-specific value `intOffset` is used to locate the offset of each cache chunk in global memory. These offsets are represented in Figure 6.8 by the variables n, m, p and q.

At the beginning of iteration $k \in [0, 4)$ of the loading loop, the offset derived in the k^{th} thread is broadcast to all participating thread lanes (via a register shuffle) so that each thread knows the exact offset of the k^{th} packet's cache chunk. Each thread then adds its own lane offset to the base offset received, as the k^{th} thread lane is responsible for reading the k^{th} integer of all chunks. The load operation itself uses the `__ldg` load function to take advantage of the performance benefits of Read-Only cache (see Section 2.5.4).

Once the loading process completes, each packet cache chunk is evenly distributed across all four participating threads. These values are redistributed in the final loop using the register shuffle intrinsic function and the reciprocal indexes stored in state memory (see Section 6.5.2). Packet data is stored in big-endian network order (most significant byte first), while Windows PCs and x86-based processors use little-endian encoding (least significant byte first) [58]. Due to this, an additional step is needed to swap the byte-order of integers in the cache on Windows PCs so that they correspond to the correct values. This is facilitated by the hardware accelerated `__byte_perm` intrinsic function. This step could be omitted if compiled for big-endian machines.

Once this process has completed, the stored cache data remains accessible until the next time the load function is called. During this period, multiple field values may be extracted from the cache by the classifier for any protocol in the layer.

6.5.5 Field Extraction

The purpose of field extraction is to locate specific n-bit binary string from the 12 bytes of valid cache data obtained during the loading process (where $n \leq 32$) and return it as a 32-bit integral value. While fields larger than 32 bits (such as a 128-bit IPv6 address) cannot be extracted in a single operation, it is possible to handle them by subdividing them into multiple shorter bit strings and processing those components individually. Each n-bit string will therefore be contained over either one or two integers, depending on whether the field crosses a 32-bit boundary in cache memory or not. Fields which fall partially outside of cache must be handled within a different cache load. This is illustrated in Figure 6.11.

The extraction process takes two inputs – `BitOffset`, and `BitLength` – which describe the field's position within the 12 byte cache. The `Alignment` state variable is used to adjust the bit offset to the correct position in the first register of

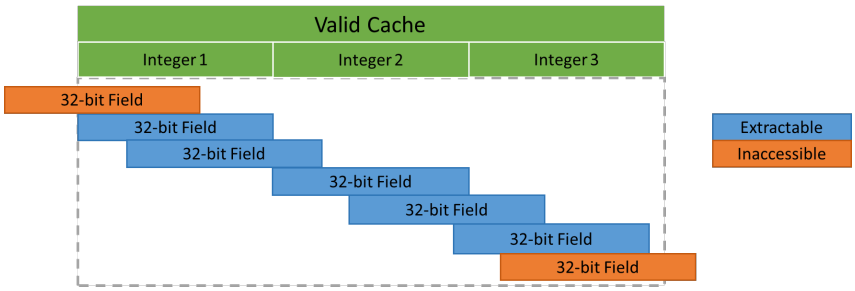


Figure 6.11: Example valid and invalid reads from cache, assuming 32-bit fields.

Listing 15 Psuedocode for the field extraction process.

```
1  int BitOffset = GatherProgram[ProgramOffset++] + State.Alignment * 8;
2  int BitLength = GatherProgram[ProgramOffset++];
3  int IntOffset = BitOffset >> 5;
4
5  // mask leading bits
6  uint value = cache[IntOffset] & (0xFFFFFFFF >> (BitOffset & 31));
7
8  //check containment
9  if ((BitOffset & 31) + BitLength < 33) {
10     value = value >> (32 - (BitOffset & 31) - BitLength);
11 }
12 // merge with trimmed trailing integer
13 else {
14     int remaining = (BitOffset + BitLength) & 31;
15     value = (value << remaining) + (cache[IntOffset + 1] >> (32 - remaining)
           );
16 }
17 return value;
```

cache memory. These values are then used to extract the field value into a register, `value`, which is ultimately returned. This is done through bit shifts and bit-wise conjunction operators rather than division and modulo operators to improve instruction throughput (see Section 2.8.4). Basic C psuedocode for the function is shown in Listing 15.

The operation begins by masking the leading bits (if any) from the first relevant integer, storing the result in `value`. The process then diverges depending on whether the contained field spans one or multiple integers. If the offset of the last bit of the field is also contained in the first integer, the trailing bits are shifted off of `value` and it is returned to the calling process. If the field overlaps two integers, however, the bits from the second integer need to be concatenated to `value`. The bits in `value` are shifted to provide room for the remaining bits and, after shifting off the trailing bits from the second integer, merged with `value` through simple addition.

At this point the field value is properly extracted, and can be returned to the calling process.

6.6 Gather Process

This section describes the gather process, so named because it is responsible for navigating each protocol header layer of each raw packet to extract and store all relevant information in a more efficient format. All subsequent host and device-side work operates on fully coalescing bit strings and pre-filtered integral arrays generated by the gather process.

The gather process executes as a hierarchy of nested loops, controlled by nested instructions contained in program memory, and supported by the packet cache and state memory. The gather process is composed of two dedicated nested functions, referred to as the layer and field processing functions respectively. These functions are outlined in Figure 6.12. The outer layer processing function (described this section) manages the switching of stack layers, prepares the thread cache and state memory for data gathering, and prunes redundant computation from individual warps. The inner field processing function (described in Section 6.7) is responsible for interacting with packet data, and stores filter comparisons, field data, and the results of expression evaluations in working and results memory (see Section 6.4).

The remainder of this section focusses on the layer processing function. Field processing is discussed separately in the following section.

6.6.1 Layer Processing Function

The layer processing function is called once for each iteration of the gather process, and manages all filtering and data gathering operations. It is responsible for navigating raw packet records, managing layer switching and runtime pruning, pre-loading the packet cache with raw packet data, and dispatching the field processing function. This is achieved through a hierarchy of three nested loops which divide processing into simple groupings.

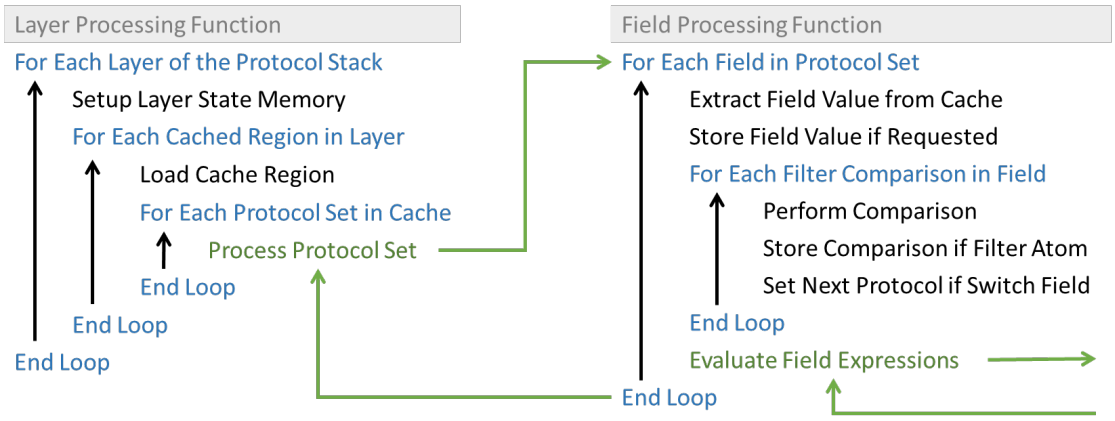


Figure 6.12: Abstract representation / psuedocode of the layer processing function.

1. The outer-most loop manages the transition between different protocols in the packet header. This loop may skip iterations if the warp contains no applicable threads, and may terminate if all threads in a warp reach a finalised state.

2. The central loop fills the packet cache in preparation for filtering. This process was discussed previously in Section 6.5.

3. The inner-most loop dispatches processing of warp-applicable, protocol-specific sets of fields that overlap with the currently loaded cache. The loop may skip protocol sets if a warp contains no applicable threads.

Listing 16 describes the structure of the relevant portion of the gather program in Extended Bachus-Naur Form (EBNF) [117]; fields are discussed separately in Section 6.7. A complete listing for the full gather process can be found in Appendix A.2. The remainder of this subsection will discuss these loops from outer-most to inner-most, detailing important functionality and tying it to commands embedded in the EBNF program.

6.6.2 Layer Switching

The outer loop's primary function is to manage the transition between protocols and to facilitate layer level pruning. At the start of each new layer, the gather process evaluates the layer header to determine the relevancy of the layer. The

Listing 16 Partial EBNF describing the encoding of the gather program.

```
gather program = { stack layer } ;

(* Layer *)
stack layer = layer header , cache chunk count , { cache chunk } ;
layer header = protocol count , { layer protocol } , skip offset ;
layer protocol = protocol id , protocol length ;

(* Cache *)
cache chunk = local offset , protocol count , { protocol set } ;

(* Protocol Set *)
protocol set = protocol id , skip offset , field count , { field set };

field set = ... ;

protocol count = number ;
protocol id = number ;
protocol length = number ;
skip offset  = number ;
cache set count = number ;
local offset = number ;
protocol count = number ;
field count  = number ;

number = digit , { digit } ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

layer header contains a simple list of protocol definitions comprising the protocol's unique numeric identifier and its default length. Each thread iterates over these definitions, comparing the protocol identifier to the `NextProtocol` register in state memory to determine if that protocol is relevant to the threads associated packet. The `NextProtocol` register is initialised to the root protocol's identifier at the start of each iteration of the gather process (or alternatively, each packet), and is updated each time the next protocol in the packet header is identified.

If a match is found, the `CurrentProtocol` register is set to the value in the `NextProtocol` register, and the `DataOffset` register is incremented by the value stored in the `ProtocolLength` register. This associates the thread with both the layer and the protocol, and updates the thread's local `DataOffset` to point to the beginning of the current header. The `ProtocolLength` register is then re-initialised to the default length specified in the definition, and the `NextProtocol` register is reset to a null value.

Once all protocol definitions have been reviewed, threads perform a warp vote to determine if the layer can be skipped. If all threads fail to find a matching protocol definition, the warp vote succeeds and the layer is skipped. Specifically, the `ProgramOffset` is incremented past the current layer using the skip offset provided in the gather program, and the next layer iteration is immediately started. If the vote fails, at least one thread in the warp is active, and the process begins iterating over each of the cache chunks contained within the layer.

The outer loop performs one additional warp vote at the end of each iteration, which passes if all threads have a null value stored in the `NextProtocol` register. This only occurs if all threads in the warp fail to identify a subsequent protocol during the course of layer evaluation, and thus a passing warp vote indicates the warp is finalised (no threads in the executing warp have relevant operations left to perform). If the warp is finalised, the outer loop exits and completes the process early, effectively pruning any remaining layers.

6.6.3 Caching and Protocol Set Dispatch

The central loop refills the packet cache, which acts as the data source for all protocol set processing functions. The cache takes a single argument, specifying the byte offset of the first integer in the cache load, from the start of the packet record.

This byte offset is found by summing the `DataOffset` register (the byte offset to the start of the layer) to the local offset specified in program memory. The derived byte index indicates the thread local start position of the cache chunk in the thread's current packet record, which is passed as an argument to the cache load function (see Section 6.5).

Once the cache has been populated with the data specified in the current cache chunk, the inner-most loop begins iterating through each protocol set associated with the cache chunk, determining which sets must be evaluated and which sets can be ignored. A protocol set is effectively the set of all operations targeting a specific protocol within a specific cache chunk. This grouping allows the process to skip all operations targeting a specific protocol through a single warp vote. Without this grouping, pruning protocol redundancies would require a separate warp vote in each individual field evaluation, which would become expensive in larger filter programs.

The warp vote to skip a protocol set is performed at the start of each iteration of the inner-most loop. The vote identifies redundant protocol sets by comparing the protocol identifier associated with the set to the `CurrentProtocol` register in state memory. A protocol set is considered redundant and is skipped if the warp vote shows that no threads in the warp are associated with the same protocol as the protocol set. If the protocol set is relevant however, the `ActiveProtocol` state register is set to the value of the protocol's unique identifier, and the protocol's fields are processed by the inner field processing function. This is discussed in the next section.

6.7 Field Processing

The field processing function executes within the layer processing function of the gather process. This function extracts and derives field and filter results for a particular protocol within a specific cache chunk, updating the `ProtocolLength` or `NextProtocol` registers in state memory when necessary. Figure 6.12 provides an overview of the process. It is composed of an outer field processing loop with two consecutive and optional inner loops that handle filter comparisons and expression evaluations respectively. Listing 17 shows the EBNF for the relevant portion of the gather program. This section explores the outer field processing loop, inner filter

Listing 17 Partial EBNF describing the encoding of the gather program.

```
field set = field offset , field length , store index , filter count, {
    filter comparison } , expressions ;

filter comparison = comparison operator , lookup index , switch id ,
    working index ;

field offset = number ;
field length = number ;
store index = number ;
filter count = number ;
expressions = ... ;

comparison operator = "0" | "1" | "2" | "3" | "4" | "5" ;
lookup index = number ;
switch id = number ;
working index = number ;

number = digit , { digit } ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

comparison loop, and expression handling. Expressions are discussed in Section 6.7.3.

6.7.1 Extracting Fields

The first step performed during the processing of each field involves extracting the value of the field from the cache and storing it in a temporary local register (referred to as the `value` register for simplicity). Field extraction is performed by the cache read function, as described in Section 6.5.5. This function takes the field offset and field length supplied by the gather program as inputs. Once the field has been extracted, it may be stored in result memory (see Section 6.4.3) at the index specified by the `store index` field in program memory. This occurs if the index is not equal to the reserved value of `0xFF`. If a valid index is specified, threads write their extracted field values to consecutive (and thus fully coalescing) indexes of the field's result array.

6.7.2 Field Comparisons

The next phase of the function is optional: it iterates through and performs every filter comparison targeting the field, storing the results as a bit string. The filter

Table 6.4: Mapping of integer identifiers to comparison operators.

Comparison Operator	==	!=	<	>	<=	>=
Integer Identifier	0	1	2	3	4	5

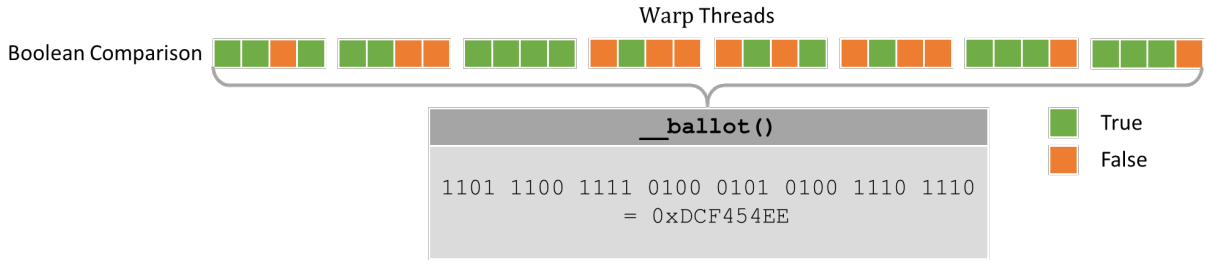


Figure 6.13: Converting a warp-wide comparison result into a single 32-bit integer.

comparison involves comparing the extracted field value to a 32-bit value stored in the lookup table (see Section 6.2.2), and using one of the six supported comparison operators. These operators are represented by the integer identifiers in program code (see Table 6.4). Each filter comparison is performed in conjunction with a protocol identifier test, which filters out results from threads that are not associated with the active protocol. The produced boolean variable may be stored as in working memory (see Section 6.4.2) for the filter process if filter comparisons are defined, or used to set the `NextProtocol` register in state memory if the field is the target of a switch statement.

Comparisons are stored if the working index supplied in the gather program is not equal to `0xFF`. If a comparison is to be stored, the boolean result of the comparison is immediately compressed from a collection of 32 bytes into a single 32-bit integer value using a warp ballot (see Section 2.7.1). The ballot produces an identical integer in all 32 threads, where each bit of the integer corresponds to the comparison result of the corresponding thread in the warp. This format is eight times more compact and bandwidth efficient than a boolean array used in GPF, and through bitwise operations can evaluate 32 times more results per operation in the filter process. Once the ballot function has completed, the first thread in the warp writes the resultant integer to working memory (see Section 6.4.2). If a comparison is used in a switch statement and results in a true value, the associate protocol identifier is written to the thread's `NextProtocol` register to facilitate layer switching (see Section 6.6.2).

The remainder of the function is dedicated to evaluating integral expressions, which are used to calculate protocol lengths from header field data.

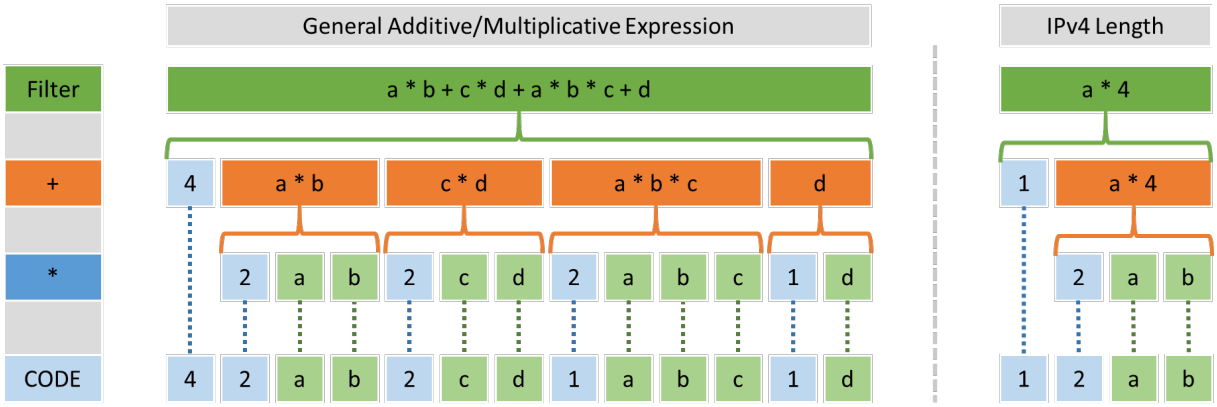


Figure 6.14: Example of expression encoding.

6.7.3 Protocol Length Processing

The final phase of the field function processes integer based expressions, which are currently exclusively used to transform and store the protocol length fields contained in header data; this information is required by the outer layer function when switching from a variable length protocol to a child protocol.

Protocol length fields are typically specified as the number of bytes or 4-byte words in the header and thus only require a multiplication process to evaluate. This could be achieved quite simply: multiply the field by a program encoded multiplier greater than or equal to zero, where zero indicates no expression, for instance. This would be achieved at the expense of a generic approach; the function would not be generalisable to arbitrary expressions comprising addition, subtraction, multiplication and parenthesis operators. With these operators it would be possible to perform numerical calculations of greater complexity during the course of execution, which could be used to transform field values, local variables and state memory registers programmatically. This is not within the scope of this implementation but is a possibility for future work.

As a compromise, the implementation provides the architectural foundation for processing varied numeric expressions, but explicitly supports only addition, multiplication and parenthesis operators. The majority of this section discusses this limited implementation; the following section discusses possible approaches to incorporating subtraction and division and extending expression evaluation functionality. EBNF showing the program syntax for expression evaluation is provided in Figure 18, and may also be found in Appendix A.3. If a field defines no expressions, expression evaluation is skipped by setting the field's expression count to zero.

Listing 18 Partial EBNF describing the encoding of expressions.

```
expression = expression count , { sum } ;
sum = product count , { product } , value ;
product = value count , { value } ;
value = type id , index ;

expression count = number ;
product count = number ;
store index = number ;
value count = number ;
read type = number ;
read index = number ;
store type = "0" | "1" ;

number = digit , { digit } ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Expressions are evaluated within a three-tier nested loop; the outer loop iterates through each individual expression, while the inner loops process additive and multiplicative processes, respectively, within each expression. For simplicity these loops will be referred to, from outer-most to inner-most, as the expression, sum and product loops. These loops process a series of embedded values, which are encoded as two tuples comprising a source identifier and a read index. The source identifier indicates the region of memory where the data is stored, and currently has two defined values that point to the lookup value table in constant memory, and the `ProtocolLength` register in state memory.

The inner-most product loop processes a section of program memory composed of a value count, followed by a series of one or more value tuples. Starting with a local register m containing the multiplicative identity ($m = 1$), the product loop uses the value count to iterate over each value pair, loading an integer from the indicated location, multiplying the retrieved integer with the value contained in m, and finally storing the result back in m. Once the loop completes, m contains the product of all referenced values in the product section. If an expression does not employ multiplication, any contained product sections will reference exactly one value tuple.

The sum loop iterates through a sequence of one or more product sets, accumulating the results of each set in a local register s initialised to the additive identity ($s = 0$). By nesting multiplication within the addition loop, operator precedence is maintained; multiplication between values will always be performed before addition and thus $a + b * c$ will be correctly evaluated as $a + (b * c)$ and not $(a + b) * c$. The

result of the expression is contained in s after all iterations of the sum loop have completed, and may then be stored in either system memory or temporary memory (lookup memory is constant and not writeable at runtime). The storage location is specified using the same syntax as value tuples, but exposing only writeable locations. System memory thus only exposes the `ProtocolLength` register, as the field value register is read-only.

6.8 Filter Process

The filter process generates bit-based filter results for each packet in the processed capture. This function combines comparison results (stored in working memory) using bitwise logical operators. In comparison to the gather process, the filter process is relatively simple, and quite similar to the expression evaluation process described in Section 6.7.3. Both these processes are based on the filter process used in prior work [66], which evaluated predicates using stored boolean values and boolean logic (see Section 4.4). The use of bitwise operators over boolean operators is extremely important to improving the performance of the filter process, as it allows each thread to classify thirty two packets in a single bitwise operation, while greatly improving storage and bandwidth efficiency. The predicate syntax remains identical to the syntax shown in Figure 4.10.

The EBNF for the filter program evaluated by the process is provided in Listing 19, while the psuedocode for the process is shown in Listing 20. Note that the `WORKING` and `RESULT` macros correlate to the transformations discussed in Sections 6.4.2 and 6.4.3 respectively.

The filter process is constructed using a three tier nested loop, and uses the same approach as expression evaluation to maintain precedence rules. The outer layer iterates through each filter, storing results in either results memory or in working memory. Results memory stores the final filter bit-strings that are returned to the host, while working memory stores the results of parenthesised sub-predicates in filter definitions. As working memory is also used by the gather process, later iterations of the filter process treat gather results and prior filter results in working memory as being identical.

Predicates combine bit strings using bitwise negation (`~`), conjunction (`&`) and disjunction (`|`) operators, and are handled by the central and inner-most loops. Dis-

Listing 19 EBNF for filter program encoding.

```
program = filter count , { filter } ;
filter = or count , { group } , store location;
group = and count , { element } ;
element = invert , read index ;
store location = store type, write index;

filter count = number ;
or count = number ;
and count = number ;
invert  = "0" | "1" ;
store type = "0" | "1" ;
read index = number ;
write index = number ;

number = digit , { digit } ;
digit = "0" | "1" | "2" | "3" | "4" |
      "5" | "6" | "7" | "8" | "9" ;
```

Listing 20 Psuedocode for the filter process.

```
foreach filter in program
{
    int ans = 0;
    foreach group in filter
    {
        int sub = 0xFFFFFFFF;
        foreach element in group
        {
            if (invert)
                sub &= ~WORKING(element.index);
            else
                sub &= WORKING(element.index);
        }
        ans |= sub;
    }
    if (store type == 0) WORKING(filter.index) = ans;
    else RESULT(filter.index) = ans;
}
```

junction has the lowest precedence of the three operators and is performed by the central loop, which combines the partial results returned by the inner-most loop. The inner most loop in turn performs bitwise conjunction, while negation is performed as the bit string is loaded into the filter process. Negation is only performed if the invert flag preceding the specified index is set to one. After the predicate has been processed by the two inner loops, the outer loop writes the result to either working memory or result memory using a coalescing pattern, at a rate of 1024 results per storage operation per warp.

6.9 Summary

This chapter described the construction of the GPF+ packet classification kernel. This discussion divided the classifier into two sub-processes. The first is the gather process, which applies a DSL compiled gather program to the data supplied by the packet cache in order to navigate protocols and compare and extract fields. The filter process follows, and uses the working comparison results produced by the gather process to compute and store arbitrarily complex predicates. These processes are unified in a device side C++ object, which is initialised and executed from within an encapsulating CUDA kernel.

The classification process was introduced in Section 6.1. This section first provided an overview of the classification approach, introduced the component processes discussed above, and described the protocol layering abstraction used. This was followed by an overview of how packets are divided among executing threads to avoid block-level synchronisation. The section concluded by discussing how asynchronous streams are used to improve device utilisation by overlapping kernel execution with asynchronous memory copies.

Section 6.2 described the constant memory region, which contains globally accessible attributes, memory pointers, and program data that remain consistent both throughout the process, and also between streams. This section also demonstrated that all threads in a warp access constant variables simultaneously, avoiding serialisation.

Section 6.3 summarised the dynamic variables contained in register-based state memory. It was also shown that these variables are tightly packed into vector types to minimise register utilisation.

Section 6.4 discussed the various regions of global memory which hold system inputs and outputs, including packet records, working memory and results memory. Attention was given to the different memory mappings used by working and results memory, which facilitate warp level synchronisation and fully coalesced access to generated outputs.

Section 6.5 focussed on the packet cache, which locally caches segments of packet data, and extracts fields from these segments on behalf of the gather process. It was shown that the packet cache redistributes reads to better promote coalescing, using warp shuffle operations to reorganise data after the read operations complete. The section further described how alignment was managed using state memory, and how this effectively reduced the capacity of the cache from 16 bytes to 12 bytes, due to the need for slack storage. The final part of this section discussed the field extraction process, which extracts field values from cache registers for use by the gather process.

Section 6.6 described the main loop of the gather process, responsible for pruning operations and cache filling. The section began with an overview of the layer processing function's implementation, and detailed the structure of the program which guides it. The remainder of this section broke the process down into layer switching, caching, and protocol set dispatch, discussing each in turn.

Section 6.7 detailed the field processing function, which extracts and processes every field from a specific, relevant protocol, contained in a particular cache segment. This section described how field values are extracted, how field comparisons are evaluated and converted into compact bit strings, and how protocol lengths are calculated using encoded field expressions.

The chapter concluded with an overview of the filter process in Section 6.8, describing the temporary bit-based comparison results generated by the gather process in order to evaluate complex predicates encoded in the filter program. The filter process uses bitwise operators to simultaneously evaluate 32 packets per thread per operation, significantly improving throughput while dramatically reducing memory overhead.

The following chapter discusses the GPF+ DSL and compiler, which are used to generate program and configuration inputs for the classifier from a high-level program specification.

7

Generating Programs

THE DSL discussed in this chapter provides a human-readable programming interface that reduces the complexity of defining programs for the GPF+ classifier. This is necessary, as the classification kernel discussed in the previous chapter relies on multiple program inputs that are too complicated to reasonably create by hand. This chapter introduces the DSL, its grammar syntax, and the compilation procedure applied to convert high-level programs into GPF+ classifier inputs. The chapter is broken down as follows:

- Section 7.1 provides an overview of the grammar's construction, and introduces its two major sections: the protocol library and the kernel function.
- Section 7.2 explains the grammar syntax used, with examples for clarification.
- Section 7.3 details the compilation process, which converts high-level GPF+ programs into classification directives.
- Section 7.4 concludes the chapter with a summary.

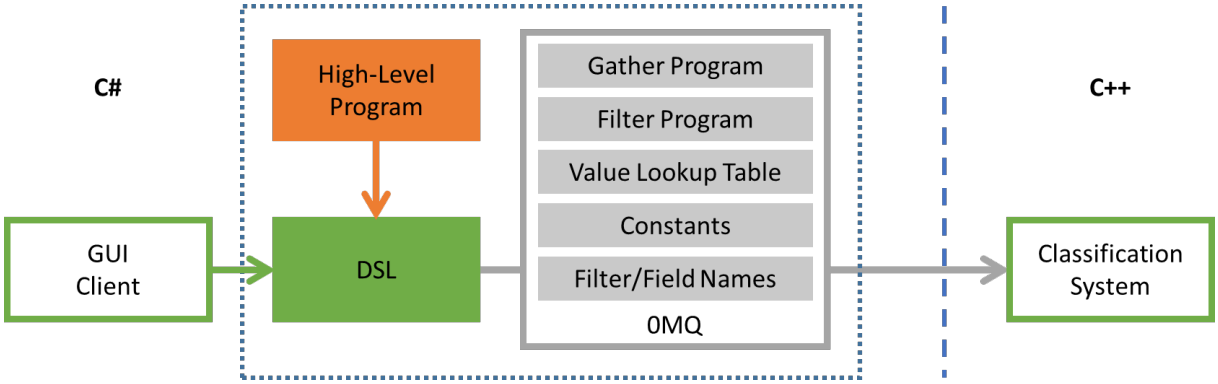


Figure 7.1: Overview of compilation and delivery of program inputs to the GPF+ classifier.

7.1 Grammar Overview

This chapter describes the design and implementation of the Domain Specific Language (DSL) used to create programs for the GPF+ classifier, previously described in Chapter 6. This DSL was developed in ANTLR 4 and C# 4.5, and is responsible for translating a high-level filter program into various low-level directives. These are used to both configure the process, and guide processing from the constant memory space (see Section 6.2). An illustration of the DSL's relation to other components, and a summary of its primary outputs, is shown in Figure 7.2.

The grammar design is loosely based on the core DSL used in GPF (see Section 4.4.2), reworked to increase flexibility, efficiency, reusabilty and readability. ANTLR handles lexical analysis (which converts high-level code into streams of tokens) and parsing (which translates token streams into data structures) [94]. The DSL then walks these data structures to prune redundancies and emit compiled programs.

The grammar specification is divided into the protocol library and kernel function, which loosely correlate to the gather and filter processes in the GPF+ kernel (see Section 6.1) . The protocol library maps out the general structure of each protocol, and their connections to child protocols, while the kernel function specifies the predicates and field extractions to be performed by a specific program. When a protocol, field or comparison in the library is referenced in a kernel function, the DSL is able to infer all necessary pre-requisite operations needed to reach that reference by following the connections defined in the library. Only the parts of the library relevant to the kernel function are used, allowing for large and detailed

Listing 21 Example GPF+ high-level program targeting the IP protocol and TCP/UDP service ports.

```
1  //protocol library
2  ...
3  protocol IP {
4      field Length [4:4] { $length = $value; }
5      field Protocol [72:8] {
6          .TCP == 6;
7          .UDP == 17;
8      }
9      switch (Protocol){
10         case UDP: goto Ports;
11         case TCP: goto Ports;
12     }
13 }
14 protocol Ports {
15     field Src [0:16];
16     field Dst [16:16] { .DNS == 53 }
17 }
18 //kernel function
19 main() {
20     filter tcp_http = IP.Protocol.TCP && (Ports.Src == 80 || Ports.Dst ==
           80);
21     filter dns_dest = Ports.Dst.DNS;
22     field ip_proto = IP.Protocol;
23 }
```

libraries to be constructed without impacting on compiled program size.

The high-level syntax structure is primarily inspired by C++ style languages, with protocol structure specified in class-like objects, and kernel processes encapsulated in an entry point method. An example GPF+ high-level program showing both the protocol library and and kernel function is provided in Listing 21. The protocol library contains two connected protocol definitions, the second of which generalises TCP and UDP protocols to access the port of either protocol in a single operation. TCP and UDP may also be declared and handled separately for greater control.

Similarly to the object model it is based on, the protocol library and its definitions are intended to be easily extended and reused across multiple GPU kernels, in order to avoid the need for replication across multiple programs. This was originally planned to operate similar to header files in C++, allowing generic reusable libraries to be included rather than repeatedly specified. Due to scope pressure however, this functionality has been left for future work; the current implementation uses a single file containing both the kernel program and its complete supporting protocol library as input.

The DSL processes input files to produce the gather and filter programs used by the classifier, as well as the majority of configuration details stored in constant memory (see Section 6.2). This information may then be passed via 0MQ to the classifier via the server interface to be used immediately, or stored for use by the command line interface at a later time (see Section 5.1.5).

7.2 Grammar Syntax

This section outlines the grammar syntax and the means by which it is parsed into memory. High-level GPF+ programs are composed of a collection of protocol definitions, each containing their own field information and comparisons. These protocols are connected together using a simple syntax, derived from switch statements, into a conceptual protocol tree. This tree may then be flattened into the layer structure used in the gather program (see Section 7.3). The kernel function uses the structure of and connections between protocols, supporting the translation of simple predicates (such as `TCP.SourcePort == 443`, or `TCP || UDP`) into an optimised program set that contains all the necessary preliminary steps (for instance, whether the packet is an IPv4 or IPv6 datagram) while ignoring irrelevant processing (such as if the packet is an ARP packet). This process is illustrated in Figure 7.2 using a simple filter for incoming SSH (Secure Shell) traffic.

In the figure, the filter is used to prune all irrelevant protocols, fields and comparisons from the protocol tree, leaving only those required to perform the filter and field extractions defined in the kernel function. This reduces the more general protocol library to only the elements required to process the supplied filters. As a result, large sets of protocol and field definitions have no impact on program size or execution duration, as all known redundant information is automatically pruned. All remaining protocol redundancies (i.e. included protocols that do not occur in the packet set) are removed through runtime pruning (see Section 6.6).

In addition to eliminating unnecessary computation, this approach helps facilitate better protocol reuse within programs, as it allows protocols to be referenced in multiple paths (and eventually programs) without replicating the protocol definition. With respect to the current implementation, this allows protocols to support multiple parent protocols or paths without replicating the protocol definition in each path (see TCP in Figure7.2). The original GPF grammar required separate

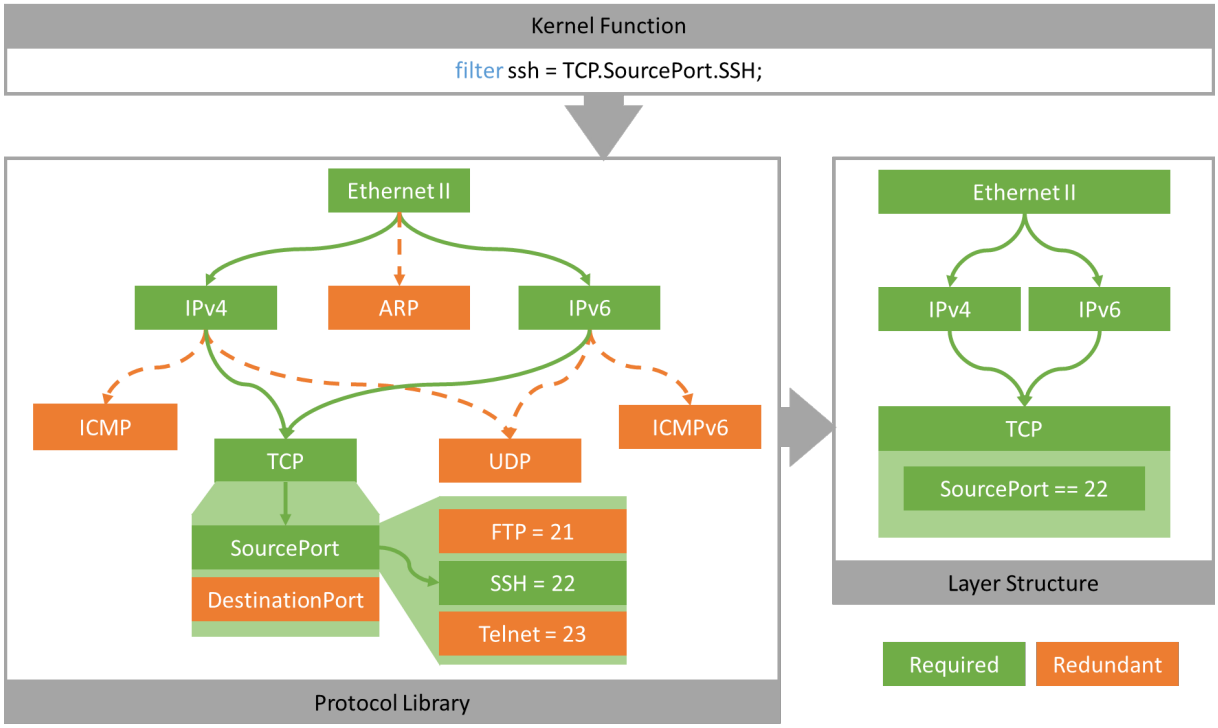


Figure 7.2: Pruning the protocol library to produce a GPF+ layer structure.

definitions for protocols such as TCP and UDP to account for IPv4 and IPv6 parent protocols in both the DSL and emitted GPF code; the GPF+ DSL and byte code, in contrast, only require one.

The remainder of this section describes the grammar syntax in more depth, with the first three subsections describing the protocol library. Section 7.2.1 introduces the syntax for specifying protocols, fields and comparisons. Section 7.2.2 explains connecting protocols using switch statements. Section 7.2.3 discusses handling variable length protocols through length field processing. The final subsection focusses on the kernel function syntax, showing how filters and field extractions are defined. The complete EBNF for the grammar is too long to be included here, but can be found in Appendix A.

7.2.1 Protocols and Fields

Protocols are the basic elements in the protocol library, containing all the fields, filters and connections that apply to them. Specifically, protocol definitions encapsulate a set of field specifications which in turn may contain zero or more named

comparisons. Protocols and fields are declared using the `protocol` and `field` keywords respectively, while named comparisons within fields are prefixed with a period character. The position and length of fields within a protocol are specified with square brackets of the form $[x : y]$ where x is the zero-based bit offset of the field within the protocol and y is the bit length of the field. Field sizes are currently limited to 32 bits, but larger fields can be accommodated by subdividing them into multiple 32-bit fields. This could be performed automatically by the compiler in a future build.

Field comparisons are contained within the field body following the field declaration; the declaration is terminated with a semi-colon if no comparisons are defined. Field comparisons support the standard operators (==, !=, <, >, <=, >=) which are used to compare the field value with a numeric constant, specified as either a decimal value, hexadecimal value, or IPv4 address. An example program defining a subset of the Ethernet II protocol is shown in Figure 22.

Listing 22 Example specification for the Ethernet II protocol

```
1  protocol EthernetII
2  {
3      field MacSource[0:48];
4      field MacDestination[48:48];
5      field Protocol[96:16]
6      {
7          .IP == 0x800;
8          .IPv6 == 0x86DD;
9          .ARP == 0x806;
10         .IEEE802 <= 1500;
11     }
12 }
```

7.2.2 Protocol Switching

Switching to new protocols is achieved by borrowing the syntax of the `switch` statement from C style languages virtually unaltered. The switch statement takes an existing field identifier as the condition and uses named filter comparisons as case values. Switch cases currently only support the `goto` operator, which takes another protocol as an argument, essentially adding that protocol as a child node of the current protocol if the switch case succeeds. It is worth noting that circular and self-references are illegal by design, following the practices of other filters like

Listing 23 Protocol switching example.

```
protocol EthernetII
{
    ...
    field Protocol[96:16]
    {
        .IP == 0x800;
        .IPv6 == 0x86DD;
        .ARP == 0x806;
        .IEEE <= 1500;
    }
    switch(Protocol)
    {
        case IP: goto InternetProtocol;
        case ARP: goto ARP;
        case IEEE: goto IEEE802_3;
    }
}
protocol IEEE802_3 { ... }
protocol InternetProtocol { ... }
protocol ARP { ... }
```

DPF and BPF+ [9, 23]. Specifically, no protocol can be a direct child or a descendent of itself. Currently each protocol may only contain a single switch statement, and may not employ `goto` statements outside of this context. This may change in future work to improve flexibility by allowing multiple switch statements within a protocol, supporting more general computation within switch statements, and allowing switch cases to span multiple commands. An example of protocol switching is provided in Listing 23.

It is worth noting that protocol switching may additionally be used as a means of facilitating optional fields such as the 802.1Q VLAN (Virtual Local Area Network) tag in the Ethernet header (see Appendix B.1), or other optional fields. This is achieved by switching to a separate virtual protocol that includes the optional field and the remainder of the protocol, as shown in Listing 24. Thus, while optional fields are not explicitly supported, they can be processed partially through creative use of the switch statement.

Handling optional fields in this manner is not particularly efficient, as it fragments protocols into multiple layers, each requiring an expensive cache fill. Dedicated support for optional fields was originally included in the design of the classifier and DSL, but was scoped out of the final implementation as optional fields are typically rare, and can generally be emulated via protocol switching, as shown

Listing 24 Supporting optional fields through switching.

```
protocol EthernetII {
    field Protocol[96:16] {
        .IP == 0x800;
        .VLAN == 0x8100; ...
    }
    switch(Protocol) {
        case IP: goto InternetProtocol;
        case VLAN: goto IEEE802_1Q; ...
    }
}
protocol IEEE802_1Q {
    field PriorityCodePoint [0:3];
    ...
    field Protocol [16:16] {
        .IP == 0x800; ...
    }
    switch (Protocol) {
        case IP: goto InternetProtocol; ...
    }
}
protocol InternetProtocol { ... }
```

above. Improved support for optional fields is intended for future work (see Section 12.4).

7.2.3 Protocol Length

Protocol length is typically derived at compile time using the bit offset and length of the last protocol field, if it is included. If the final protocol field has been omitted, the default protocol length can be defined by appending a default length in square braces to the protocol definition. Specifying a default length is optional, and is useful in variable length protocols when the compiler cannot otherwise infer protocol length from field definitions. When protocol length is not fixed and may vary between packets, the length of the protocol may be specified explicitly or calculated from the value of a header field, using the `$length` and `$value` registers exposed within the field body.

The `$length` register corresponds to the `ProtocolLength` register in classifier state memory (see Section 6.3), which is used to determine the byte offset of the next protocol in the stack after each layer completes. The length of a variably sized protocol is typically specified as an integer within a field header (generally

Table 7.1: Referencing the Library

Reference	Type	Example
Protocol	`filter`	`TCP`
Comparison	`filter`	`TCP.SourcePort >= 1000`
Filter	`filter`	`TCP.SourcePort.DNS`
Field	`int`	`TCP.SourcePort`

the number of bytes or n byte words in the header), which is exposed within that field by the read-only `$value` register. Figure 25 shows how the length of an IPv4 packet can be set at runtime using a simple multiplication expression.

Listing 25 Example IP protocol length.

```
1 protocol IPv4 [160] { //default length
2    field IHL[4:4] { //number of 4-byte words
3       $length = $value * 4; //convert to bytes
4    } ...
5 }
```

Division, subtraction and other operators are not supported, as the GPF+ classifier currently only supports addition and multiplication operators when calculating length (see Section 6.7.3). Other operators may be included in future work, if expression evaluation is expanded beyond simple length processing to include user defined variables and output arrays. Alternatively, length calculation could be replaced with an optimised scaling function, such as in DPF [23] (see Section 4.2.4), allowing for expression evaluation to be performed in a separate processing phase better suited to the task.

7.2.4 The Kernel Program

The kernel program is an encapsulating or entry method which specifies the functions to be performed by the filter program, utilising the protocol structure defined in the library. The kernel program can produce output data containing either filter results or field values, using `filter` and `int` data types respectively.

Results arrays are declared like single variables, and conceptually contain the results for each packet processed. The `int` data type declares an array of 32-bit integers used for field extraction, while the `filter` data type allocates a bit array where each bit stores the result of the associated filter for a single packet (see

Section 6.4.3). The filter library is invoked from within the kernel program by referencing a protocol, field, or comparison defined within it. The protocol library can be invoked in four distinct ways, summarised in Table 7.1. These library reference types are restricted by context; functions which return bitwise `filter` values may not be invoked from within integral field values stored in `int` kernel declarations, and vice versa. Library references of a filter type may be combined through standard boolean operators to form arbitrarily complex predicates, which are eventually computed during the classifier's filter process (see Section 6.8). In contrast, `field` declarations can currently only store references to a single raw field value.

Listings 26 shows a complete example protocol library and kernel function, which specify four filter operations and two field extractions from protocols in the IP suite.

7.3 Program Generation

This section describes the process by which a filter specification is processed to emit a compiled GPF+ program set. Program compilation is initiated after the high-level program has been converted to a token stream by the ANTLR-produced lexer object [94], and is conceptually divided into three phases:

1. The high-level program is parsed into data structures.
2. The protocol tree is pruned of redundancies.
3. The reduced tree is processed to emit a complete program.
   (a) The reduced tree is converted into an intermediate layer-based data structure.
   (b) The intermediate data structure is processed to emit a low-level program.

During the parsing phase, the high-level program is parsed into objects in memory, organised into a tree of protocols and a collection of kernel function definitions. Once parsed, the kernel function definitions are used to prune irrelevant structures from the protocol tree, leaving only the records required to perform the requested processes. In the emission phase, the reduced protocol tree and the kernel

Listing 26 Complete example GPF+ protocol library.

```
1  //protocol library
2  protocol EthernetII {
3     field Protocol[96:16] {
4        .IP == 0x800;
5        .IPv6 == 0x86DD;
6     }
7     switch(Protocol) {
8        case IP: goto IPv4;
9        case IPv6: goto IPv6;
10    }
11 }
12 protocol IPv4 {
13    field IHL[4:4] { $length = $value * 4; }
14    field Protocol [72:8] {
15       .TCP == 6;
16       .UDP == 17;
17    }
18    field SourceAddress [96:32];
19    field DestinationAddress[128:32];
20    switch(Protocol) {
21       case TCP: goto TCP;
22       case UDP: goto UDP;
23    }
24 }
25 protocol IPv6 {
26    field PayloadLength [32:16] { $length = $value; }
27    field NextHeader [48:8] {
28       .TCP == 6;
29       .UDP == 17;
30    }
31    switch(NextHeader) {
32       case TCP: goto TCP;
33       case UDP: goto UDP;
34    }
35 }
36 protocol TCP {
37    field SourcePort[0:16];
38    field DestinationPort[16:16];
39 }
40 protocol UDP {
41    field SourcePort[0:16] { .SomeApp == 1234; }
42    field DestinationPort[16:16];
43 }
44 main() //kernel function
45 {
46    filter ipv6_tcp_udp = IPv6 && (TCP || UDP);
47    filter srcport443 = TCP.SourcePort == 443 || UDP.SourcePort == 443;
48    filter SomeAppv6 = !IPv4 && UDP.SourcePort.SomeApp;
49
50    filter ClassBSrcAddr = IPv4.SourceAddress > 128.0.0.0 && IPv4.
         SourceAddress < 192.0.0.0;
51
52    int tcp_destport = TCP.DestinationPort;
53    int destaddress = IPv4.DestinationAddress;
54 }
```

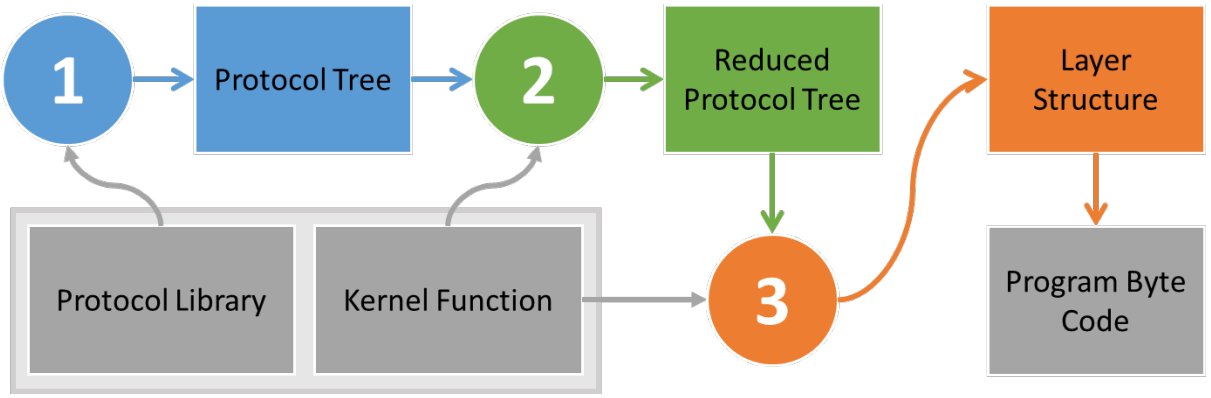


Figure 7.3: Overview of the compilation process.

program are used to generate new data structures aligned more closely with the virtual machine architecture. These new structures are then used to generate byte code and execution parameters for the GPF+ classifier. These stages are shown in Figure 7.3, and fit closely with the high-level illustration shown previously in Figure 7.2. The following subsections discuss these stages in order.

7.3.1 Parsing

The protocol library is parsed into memory as a tree structure (see Figure 7.4), similar to those used in decision tree filters such as BPF and BPF+ [9, 54]. Each protocol is a node in the tree and may have any number of child protocols branching from it. At the root of the tree is the physical layer protocol, which is determined by the network interface and is specified in the capture's global header; all other protocols are determined at runtime by inspecting the contents of packets. Due to scope restrictions, the root protocol must be specified at the top of the program file, and thus a program can only target one interface. This presents an inconvenience rather than a severe limitation, circumvented by copying and slightly adjusting the specification to use a different root protocol.

Protocols are parsed individually into a collection during the parsing process, and are subsequently connected through a simple recursive descent, initiated by the root protocol after the library has been parsed. The process creates and fills two separate hash set collections for each protocol it reaches, containing the set of all ancestor protocols (passed down from parent to child) and descendent protocols (passed up from child to parent) respectively. To prevent circular dependencies, the parsing process results in an error if a protocol is added to both ancestor and

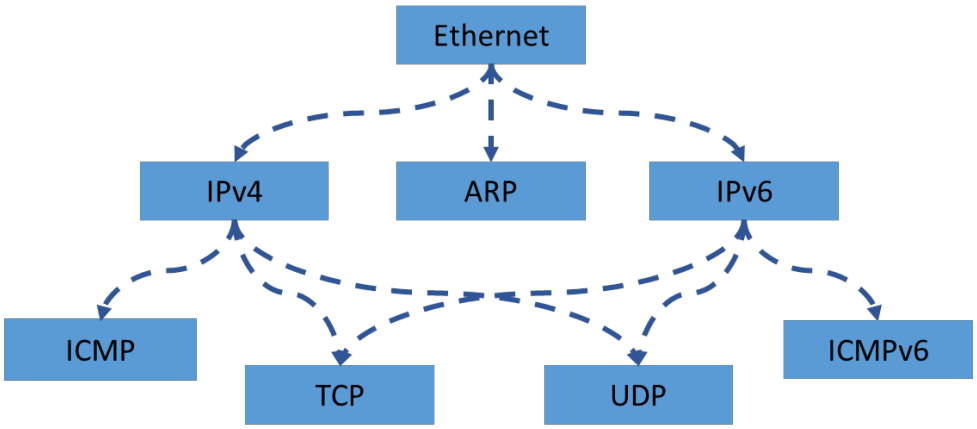


Figure 7.4: Example protocol tree.

descendent sets of another protocol, or if a protocol adds itself to one of its own sets. Aside from these sets, protocols are rather simply composed. Each contains a linked list of field definitions, and a switch definition if one is specified. In turn, each field contains a linked list of filter comparisons, and a linked list of expressions containing any calculations that are performed within that field.

The kernel function is parsed next, and used to populate three hash sets of protocol, field and filter object references. These hash sets contain all required protocols, fields and and filters specified in the kernel function.

- Fields and filters are simply handled. They are assigned a storage index in results and working memory respectively, and are added otherwise unaltered to the field hash set.
- Comparisons are automatically converted to filters by providing them with a unique, system generated identifier. For instance, the comparison `TCP.SourcePort == 53` is handled by adding `.XYZ == 53` as a filter to the `TCP.SourcePort` field in the tree (where `XYZ` represents a unique system-generated, internal and transparent identifier). The filter `TCP.SourcePort.XYZ` is then added to both the protocol tree and the filter hash set, assigning storage in working memory.
- Protocols are identified within the switch statements of parent protocols, and thus do not need to be included within a dedicated layer. The protocol hash set contains these references, rather than the protocols themselves. To populate the protocol hash set, each referenced protocol is used to generate a set of

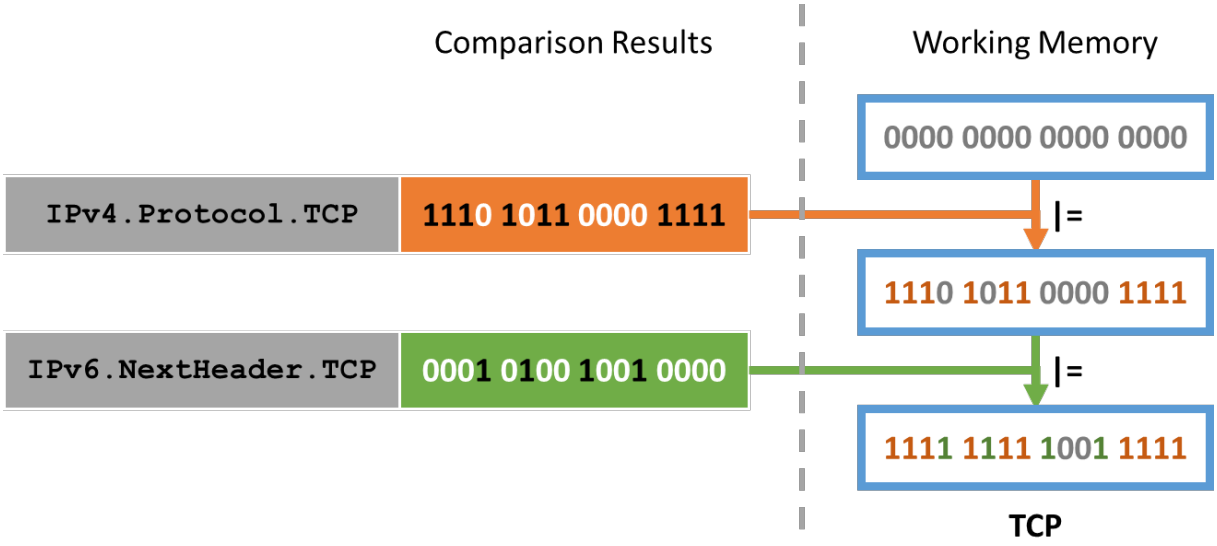


Figure 7.5: Matching TCP protocol from multiple parent protocol comparisons.

parent protocols, each containing a switch statement that points directly to the referenced protocol. All relevant switch statements in the parent set are assigned the same working memory index, before being added to the protocol hash set. For example, a protocol reference of `UDP` would add the equivalent of `IPv4.Protocol.UDP` and `IPv6.NextHeader.UDP` to the protocol hash set, both pointing to the same working memory array.

Assigning the same working memory index to filters in disjoint protocols allows a child protocol to be matched by multiple parents without additionally requiring a separate working array for each parent. This is possible because no more than one parent protocol can connect to any one child in a single packet. To account for different paths taken by different packets, the classifier uses bitwise OR operations to update existing stored values, rather than fully overwriting previously collected results (see Section 6.7.2). An illustration of this process when matching the TCP protocol from IPv4 and IPv6 headers is shown in Figure 7.5.

7.3.2 Pruning Redundant Entries

The filter, field and protocol hash sets, once constructed, are used to trim redundant elements from the protocol library (see Figure 7.6). The first step in this process involves producing target sets for both protocols and fields.

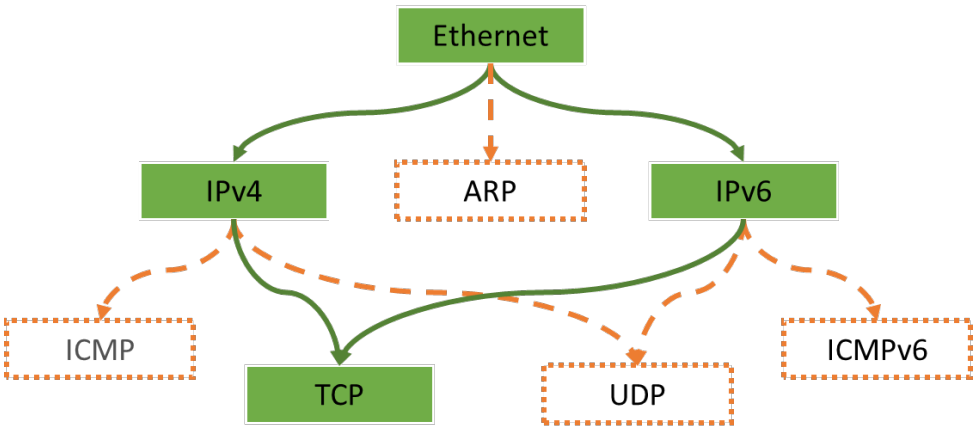


Figure 7.6: Pruning the example protocol tree.

- The protocol target set contains all required protocols in the library, and is created by finding the union of all encapsulating protocols in all hash sets.
- The field target set similarly contains all required fields. It is created by finding the union of all fields in the field hash set, all encapsulating fields in the filter hash set, and all switch fields in the protocol hash set.

The protocol target set is applied first to recursively test every protocol in the library for relevance, starting from the root protocol. Protocols are retained in the tree if they meet one of the following criteria:

1. They are the root protocol.
2. They are contained in the target protocol set, and thus contain a required field or filter.
3. They are both a descendant of the root protocol and an ancestor of a target protocol, and are therefore required to reach a targeted field or filter.

The next pruning step removes all unnecessary fields from the remaining protocols. Fields are retained in the library if they meet one of three criteria:

1. They are contained in the field target set, and were therefore directly referenced.
2. They contain the length calculation for a protocol that has a target protocol as a descendant.

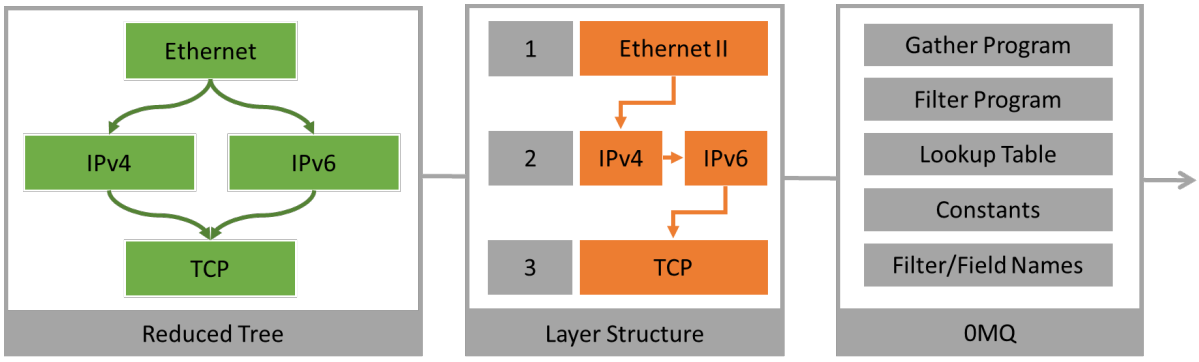


Figure 7.7: Generating DSL outputs.

3. They are used by a switch operation that connects to a protocol contained in the target set.

Note that length and switch fields are omitted if their containing protocol does not connect to a child that is a valid target protocol.

Once all unnecessary fields have been removed, the remaining fields are inspected to remove unnecessary filters. This process leaves only those filters which are required to evaluate specified comparisons (and thus have an assigned working memory index), or are otherwise needed to switch to target protocols. At this point the protocol tree has been stripped of all unnecessary definitions, and is ready to be converted to a layer structure, which closely resembles the structure of the target language. This is the topic of the next section.

7.3.3 Emitting Programs

Program emission uses the pruned protocol library, in conjunction with the kernel function definitions, to produce a set of inter-connected protocol sets, organised into a layer-based data structure similar to that of the targeted classifier. The emission process then flattens this data structure into program byte code and configuration constants, which may then be dispatched to the classifier via 0MQ, or stored as compiled GPF+ programs on disk. This transformation is illustrated in Figure 7.7.

To begin, the library is first converted into a temporary data structure composed of an ordered set of layers, architecturally similar to the layer abstraction used in the classifier's gather process. Each layer contains one or more protocols which

in turn contain their associated fields, filters, expressions and switch functions. Once the layer structure has been constructed, the contained protocols are divided into the minimum number of cache chunks required to contain each layer. This information is used during code emission to specify cache load offsets and field offsets (see Sections 6.5 and 6.7.1) .

Once complete, the kernel function definitions (stored alongside the kernel hash sets) are in turn converted to collection of objects, tied to the correct storage index for each referenced library element. At this point, the sets of protocol layers and kernel function objects contain all the necessary information to generate the program byte code and meta data; this information is used during memory allocation to correctly size GPU global memory and host buffers, and to populate the classifier's constant memory space (see Section 6.2). The emission process builds programs by walking the optimised layer and kernel data structures, using linked lists to accumulate the generated byte streams. These linked lists are converted to fixed arrays on completion. Once the programs and other outputs have been generated, they are either sent to the classifier process via TCP, or stored on disk for command-line use (see Section 5.1.5).

The DSL outputs are encapsulated in a C# object, which provides functions to handle transmission and serialisation. The variables and arrays generated by the DSL and stored in this class are listed in Table 7.2, along with their size and data type. These program components are either transmitted over TCP (using 0MQ) to the classification server, or serialised to file for reuse. A brief summary of each of the variables contained in the output object is included below:

Data Start The byte offset from the start of each packet record of the first cache load function. Used by the pre-processor to trim leading bytes from raw packets (see Section 5.5)

Data Length The number of bytes to include from each packet. This value is currently inflated by the classifier by 20 bytes (to provide additional slack in case of optional fields) and rounded to the nearest integer boundary.

Working/Filter/Field Count The number of unique working/filter/field values extracted by the program. These values are used to properly size GPU results and working memory (see Section 6.4), and to derive runtime constants (see Section 6.2.4). These values are also used, in combination with `DataLength`, to determine how many packet records are stored in each buffer.

Group	Name	Length	Type
Constants	Data Start	8×4 bytes	Integer
	Data Length		
	Working Count		
	Filter Count		
	Field Count		
	Lookup Count		
	Layer Count		
Program	Gather Program	Varies	Byte []
	Filter Program		
	Lookup Table		Integer []
Names	Filter Names	Varies	String []
	Field Names		String []

Table 7.2: Summary of DSL outputs.

Lookup Count The number of integer values stored in the value lookup table (see Section 6.2.2).

Layer Count The number of layers in the gather program, used from constant memory to control the number of layer iterations in the gather process (see Section 6.6).

Gather/Filter Program The byte code for each program, prefixed by an integer indicating their byte length (see Section 6.2.1).

Lookup Table The integer contents of the lookup table.

Filter/Field Names The string identifiers of each defined filter and field extraction operation in the kernel function. Each string is prefixed by an integer length, and each set of strings is pre-fixed by a total count. These strings are used by the classifier to name filter and field files (see Section 6.4.3).

7.4 Summary

This chapter described the high-level domain specific language used to specify filter programs, and the compilation process used to convert specifications into classifier byte code. The chapter began with a brief overview of the grammar and the APIs used to support it in Section 7.1. The APIs used include ANTLR, which facilitates lexing and parsing; .NET (via C#), which optimises, restructures and emits parsed

data structures as byte code; and 0MQ, which relays compiled programs to the classification server process. This section also introduced the high-level structure of the grammar, which is divided between the protocol library and kernel function. The protocol library contains reusable protocol definitions connected together to form a protocol tree, which is referenced by the kernel function to evaluate filters and extract field data.

Section 7.2 detailed the grammar syntax used for creating filter programs. Protocols, fields and filters are defined within class-like structures, connected through switch statements, with support for adjusting protocol length using dedicated registers. These structures are referenced within the kernel function to abstract away the complexity of filter creation. Section 7.3 concludes the chapter by stepping through the parsing, optimisation and code emission phases of the compilation process, which together convert the high-level specification into GPF+ program byte code. This byte coded is encapsulated within an object which handles transmission to the classifier and serialisation operations.

Having examined the host process, classification kernel and DSL, which together constitute the packet classification process, the following chapter briefly explores three interconnected example applications which apply generated results to accelerate aspects of packet analysis.

8

Post-processing Functions

THIS chapter discusses three related example applications that utilise the outputs of the classification process (discussed in Chapters 5, 6, and 7) to accelerate aspects of long-term capture analysis. The purpose of discussing these applications is to provide working examples that demonstrate the the classification system's use, and to evaluate its outputs and performance. As these functions fall outside the main classification process (they merely apply the results of the process within a specific context) and exist primarily as proof of concept implementations, the discussion is less technically involved than in prior chapters.

- Section 8.1 describes a simple function that calculates the distribution of values within a field file and displays them as a sorted bar graph.
- Section 8.2 describes an example application which utilises filter and index data to rapidly visualise large long-term captures within an OpenGL-based Windows Forms control. This control plots average traffic volume, packet arrival rate and packet composition over time.

- Section 8.3 discusses an example capture distillation function, which applies filter, index and packet data to generate pre-filtered capture files. These are easier and less resource-intensive to process in existing protocol analysis software.
- Section 8.4 concludes the chapter with a summary.

8.1 Simple Field Distribution

The simple field distribution is a function which operates on a single field result file (see Section 6.7) to produce a break down of the distribution of its stored values. Determining this distribution is easily achieved with the use of a .NET dictionary object, using unique field values as the dictionary keys, and the associated counts as dictionary values. In brief, the function iterates through each field stored in the field file, either adding a new dictionary key (with an associated dictionary value of one) if it is the first occurrence of the field value, or incrementing the associated count of a value if it already exists in the dictionary. The resultant field value / count pairs can then be sorted and displayed with relative ease. Sample output showing the top field values for Ethernet's ethertype field, as well as TCP and UDP source ports, is shown in Figure 8.1. This figure is composed of three cropped screen captures showing the most significant values, annotated with field specific headers, and the names of protocols or applications most commonly associated with each port, for legibility.

The produced distributions can be used to quickly ascertain the most significant active protocols or field value ranges within a capture, which provides a means to investigate a captures contents. While processing extracted fields on the host is significantly faster and easier than processing raw capture files (see Section 11.3.3), it is important to note that accelerating this function using CUDA is not particularly difficult, and would likely be hundreds of times faster [8, 34].

The CUDA Thrust API[1] [34], for instance, provides the `reduce_by_key` function which reduces key/value pairs using GPU acceleration, outputting results equivalent to the CPU implementation discussed above [34]. Specifically, the function reduces a set of non-unique key / value pairs to a set of unique keys, each paired with

[1] `https://developer.nvidia.com/Thrust`

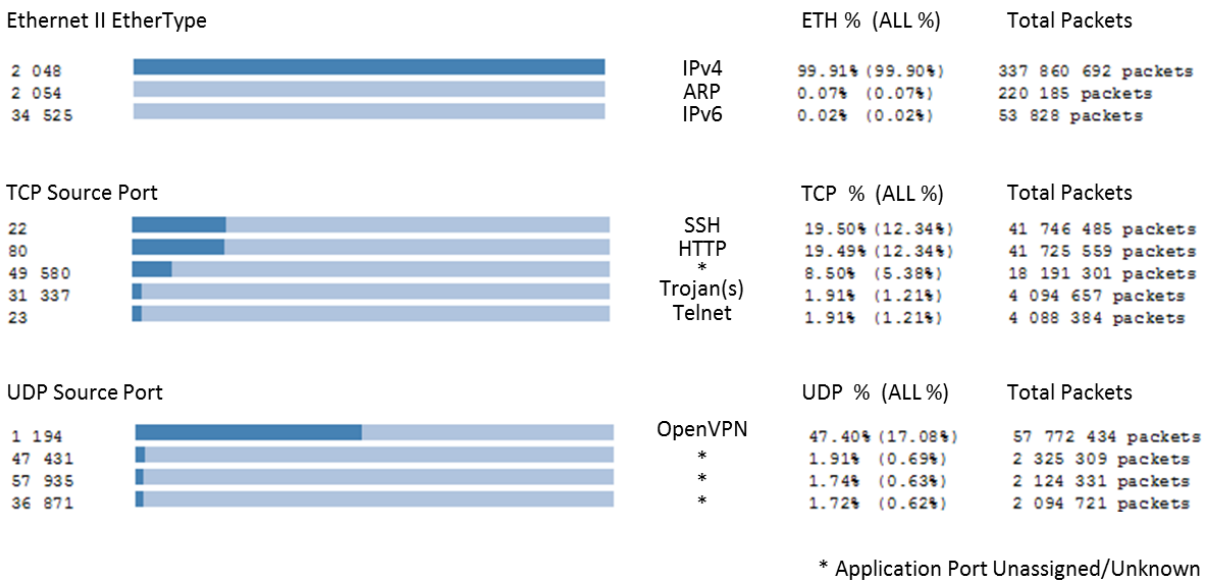


Figure 8.1: Sample distributions showing top field values for 338 million packets.

the sum of all values associated with that key. Thrust provides a simple template-based wrapper that makes it relatively easy to employ, and helps to reduce CPU overhead while significantly accelerating computation [34].

While it significantly outpaces the simple CPU implementation discussed above, the Thrust API does not provide performance gains equivalent to a well constructed CUDA kernel. Segmented reduction [8], for instance, can perform a similar reduce-by-key operation at a rate of over 10 billion fields per second on a GTX Titan, outperforming Thrust by a factor of three to four in included benchmarks [8]. Either of these approaches could be incorporated with little effort; they have not, however, been implemented in this study, as their performance is already established.

8.2 Visualising Network Traffic over Time

The second application is a user-interface which provides functionality to visualise and explore the dynamics of capture traffic over extended periods of time. The application's implementation is a relatively minimal proof of concept that relies on OpenGL for rendering performance; it was used in place of existing libraries to mitigate the possibility of external code introducing unexpected overhead at runtime,

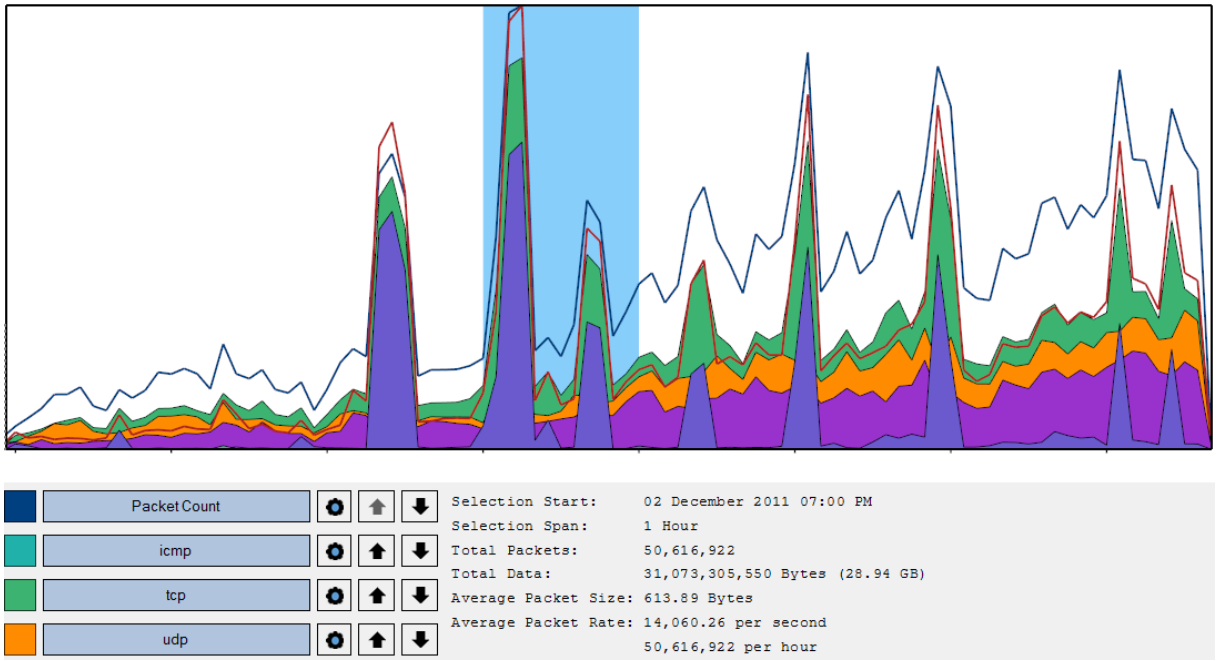


Figure 8.2: Screen capture of a visualised capture, showing statistics for the 1-hour period highlighted.

which could transparently skew performance results. As the visualisation component is not novel, and is only intended as an application for testing purposes, a detailed consideration of its components falls outside of the scope of this document. This section instead limits discussion to a high-level architectural overview of the control that displays the traffic graphs, describing the processes which convert classification outputs to graph vertex data.

Visualisation depends heavily on index files (see Section 5.5.2) and filter results (see Section 6.4.3) generated during packet processing, in order to allow for real time exploration of large captures without exhausting host memory. The interface is implemented in C# to facilitate rapid prototyping and experimentation, with visualisations constructed in OpenGL[2] (Open Graphics Library) [106] through the OpenTK[3] (Open ToolKit) C# wrapper [92].

The graph control displays an overview of the capture by using the per-packet and time-stamp index files to calculate the number of packets and amount of data over various time intervals (see Section 8.2.3). This may then be expanded and contracted in near-real-time, with rendering facilitated by GLSL (OpenGL Shading

[2] https://www.opengl.org/
[3] http://www.opentk.com/

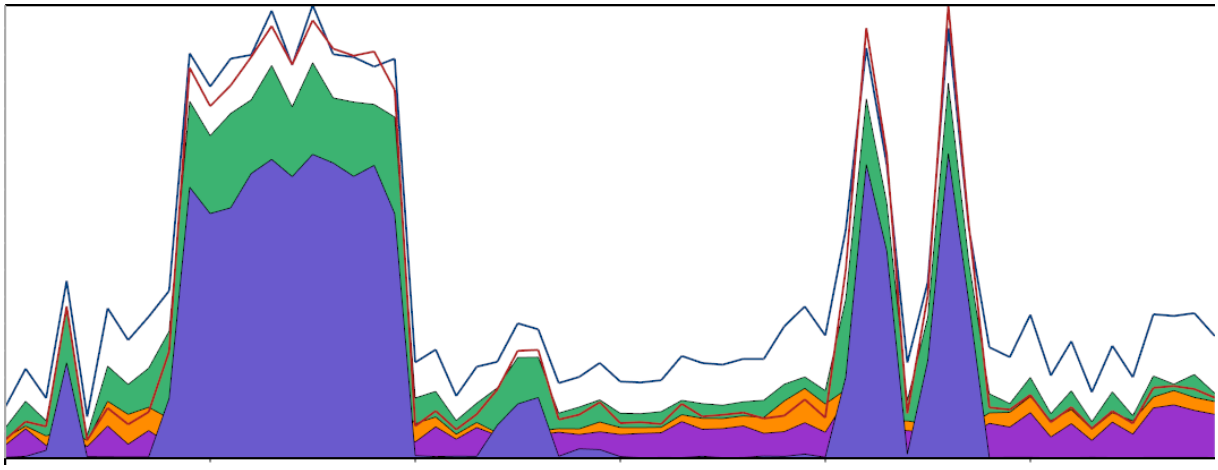


Figure 8.3: Expanded view of highlighted section in Figure 8.2.

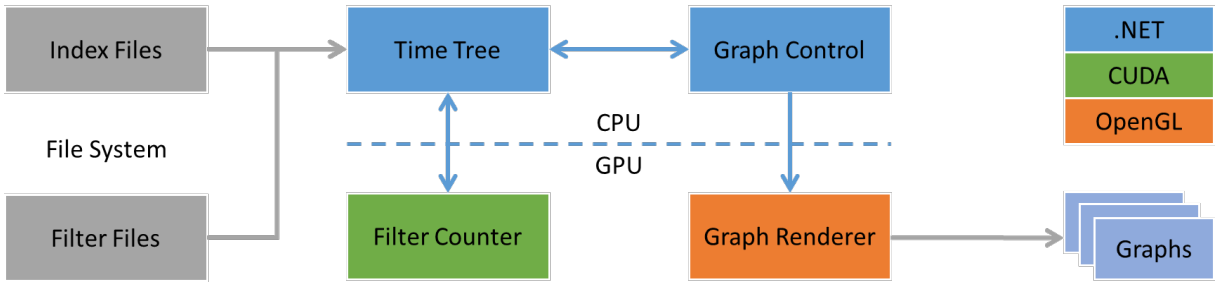


Figure 8.4: Simplified overview of visualiser process.

Language) [43] programs which execute on the GPU. This section provides two images generated by the visualiser that illustrate this. Figure 8.2 shows the default rendering level which spans the capture, with an hour long segment highlighted. Figure 8.3 shows an expanded and more detailed view of the highlighted segment, displayed by double clicking it in the UI (User Interface). In the graph images, the large dark blue line indicates the number of packets arriving at that time, while the red line shows the volume of packet data arriving at that time. These line graphs are scaled such that their respective maximum values meet the top of the view port, highlighting relative rather than absolute performance. All coloured areas represent the packets matching each filter, scaled proportional to the maximum packet count.

The rendered graph collection provides a means of viewing traffic dynamics over time, simplifying detection of hostile or anomalous activity, and helping to identify trends in traffic. A simplified overview of the graph creation process is provided in Figure 8.4. The following subsections explain and discuss this process in greater detail.

8.2.1 Managing Memory Utilisation

Avoiding over-utilisation of finite host memory resources is a primary concern for an application specifically aimed at visualising arbitrarily large packet traces, as inputs are expected to regularly exceed host memory capacity. As the interface is intended to visualise packet sets that could potentially span terabytes, it is extremely impractical to depend on raw packet captures as a primary resource for visualisation. While iterative buffering could limit the host memory overhead of processing the capture directly, it would still require parsing the packet capture at least once to initialise visualisation data, which would bottleneck performance at the throughout limitations of file I/O (see Section 5.4). The visualiser instead depends on index files and filter results generated during classification, as these require significantly less storage than raw capture files, and can be loaded faster, stored more easily and accessed more efficiently than raw packet records. Index and filter files are used to derive metrics (see Section 8.2.3) that are in turn appended to a tree data structure; the packet index and filter files themselves are not retained in memory, and do not need to be fully read to be used. This prevents memory utilisation from scaling linearly with capture size, and helps to secure high throughput by limiting the impact of slow file access. In contrast, time index files are stored in entirely in memory as they scale with capture duration rather than packet count, at a rate of roughly 240 MB per year. As the duration of even long-term captures rarely exceed a year or two, storing time indexes in memory helps to reduces disk contention during processing without a significant risk of unbounded memory utilisation.

8.2.2 Structuring Visualisation Data

The graph control depends on efficient access to packet index data to quickly adjust to different temporal scales (from years to minutes for instance). While packet index data can require significant storage for long term captures, most of the offsets encoded in the index data are not required directly by the process (see Section 5.5.2). As a result, relevant information can be loaded in piecemeal on demand, and retained in a tree structure that summarises data over increasingly finer intervals of time.

Each tree node covers a particular unit of time at a specific level of scale – either minute, hour, day, month or year – and contains detailed information on that spe-

cific segment of the capture, including but not limited to total packet count, total data volume and matching counts for each filter for the time period. Larger units of time act as parent nodes to a collection of smaller units. For instance, a node covering a particular day will typically contain 24 child nodes that subdivide the hours of that day. Child tree nodes are only added to a parent node if a portion of the capture being processed intersects with the child's specific subdivision, so nodes on the edge of captures may not contain a node for every subdivision unit. Figure 8.5 provides a simplified high-level illustration of the organisation of the tree structure as it applies to a theoretical capture.

This hierarchical structure allows the index data to be represented and retained at variable resolutions, scaling from per second metrics to yearly averages by traversing between parent and child nodes. As previously discussed however, packet index data scales linearly with packet count and thus has the potential to grow too large to be contained in system memory. To avoid over-utilising memory, not all levels of the hierarchy are initially populated; only nodes on levels from the root down to and including the captures default render level are added on initialisation, with remaining elements loaded sparingly on demand. The captures render level is an adjustable property that indicates which layer of the tree to use when generating vertex data; it is initialised to a level appropriate to the timespan of the visualised capture, using a simple set of heuristics. Figure 8.6 provides an example of this, showing the default tree constructed for an arbitrary long term capture spanning several months, with the render level initialised to one day. Each individual day node is populated with metrics (see Section 8.2.3), which are passed up through the tree to calculate and store parent node metrics.

When the render level is lowered from its initial setting to a smaller unit of time, additional nodes are created, filled and appended to the tree. These nodes are created on demand for the specific subsection of the capture being rendered to avoid loading unnecessary data and conserve host memory. For instance, if a particular hour in a day-long capture is expanded and requires minute resolution nodes to be appended to the tree as a result, only minutes falling within the expanded hour are parsed into nodes and appended to the tree. This means that the system only needs to retain high resolution nodes for a few short and relevant segments of the capture, reducing read overhead and conserving host memory, whilst dramatically reducing the work load required to visualise the capture.

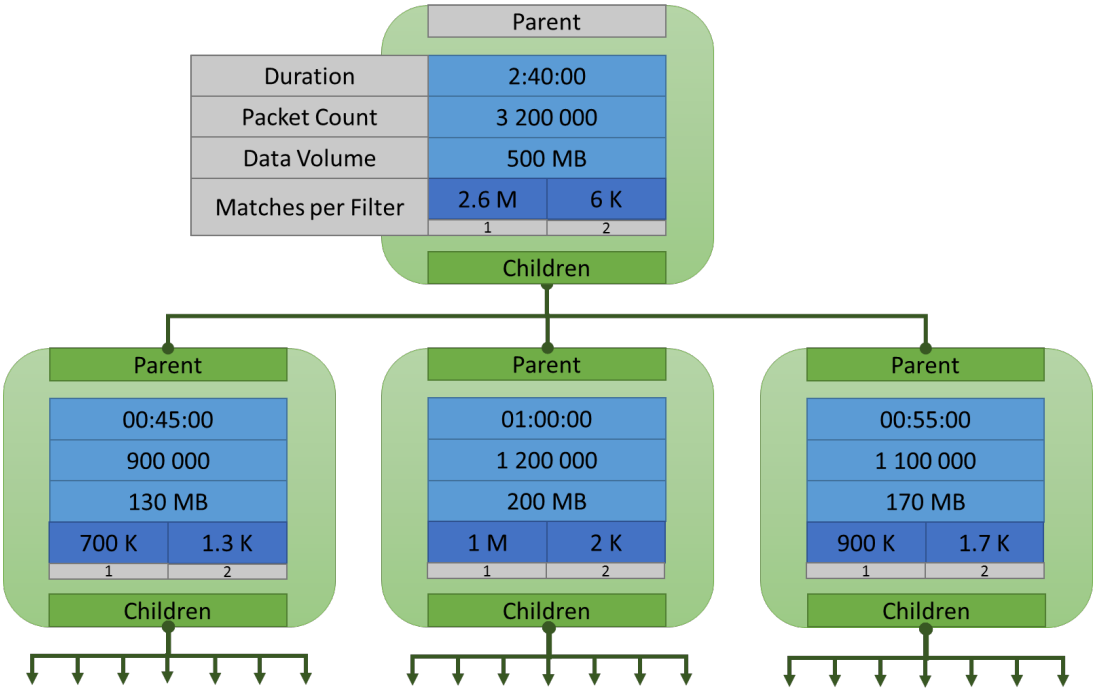


Figure 8.5: Simplified illustration of the uppermost nodes for a hypothetical capture, with two defined filters.

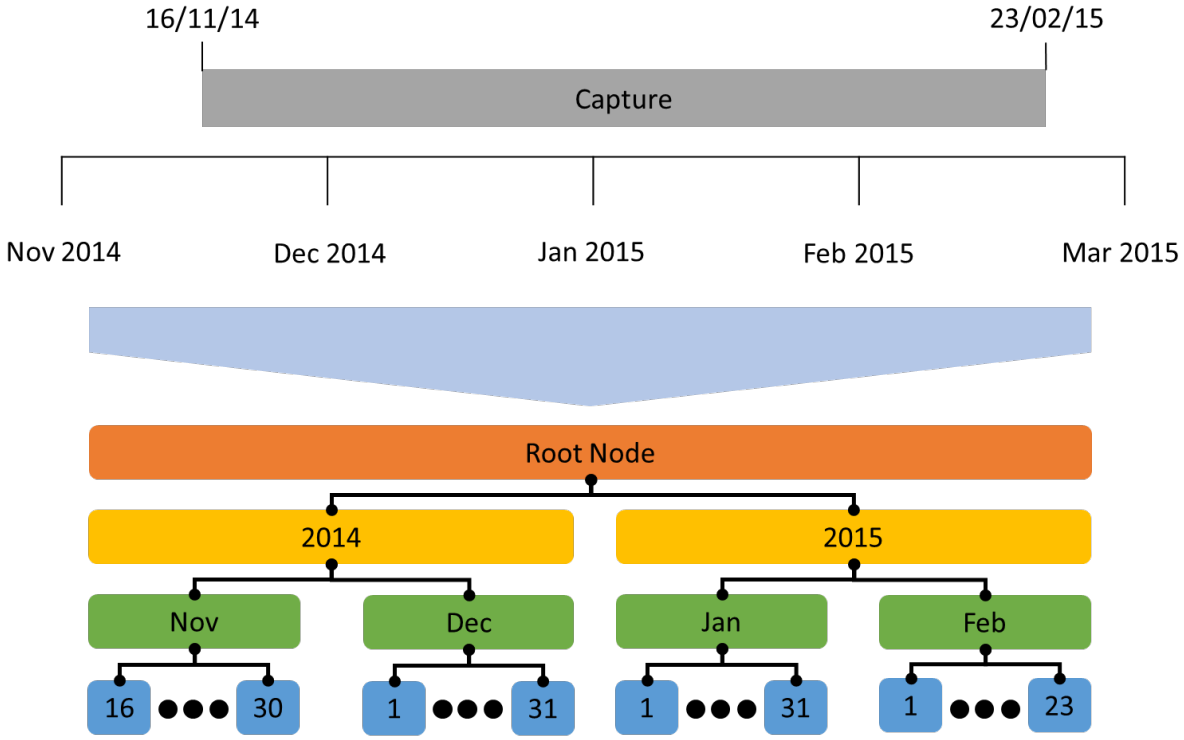


Figure 8.6: Default tree construction for an arbitrary capture spanning just over three months.

8.2.3 Visualised Metrics

Nodes in the time tree store inclusive packet count, data volume and filter match metrics for the time span that they encapsulate. These metrics are established through interaction with the capture's packet index and time index files, as well as with all filter results files related to the capture. This section describes how each of these metrics is derived from the classifier's outputs.

8.2.3.1 Packet Count

The packet count reflects the number of packets that arrived during the time interval encapsulated by the tree node. It is the simplest metric to derive and relies only on the capture's time index file for input. Let the process of accessing records in the time index file be represented by the function f, such that $f(a) = b$ (where a is a specific second and b is the index of the first packet to arrive a seconds after the start of the capture). Let s and t be arbitrary temporal indices which satisfy $0 \leq s < t$, and let p and q be packet indexes such that $p = f(s)$ and $q = f(t)$. Then the packet count C between s and t is given by $C = q - p = f(t) - f(s)$. This allows the packets between any two arbitrary seconds within a capture to be counted with two 8-byte reads, regardless of relative proximity, providing an appropriate mechanism to determine the number of packets contained in a tree node's time span.

8.2.3.2 Data Volume

The traffic volume of a node indicates the total number of bytes spanned by a node, and is derived using the packet index file in conjunction with the packet count C and the packet indices p and q derived previously. Let the function g represent the process of accessing the packet index file such that $g(b) = c$, where b remains the index of a packet and c is the byte offset of that packet in the capture file. Let x and y be byte offsets such that $x = g(p) = g(f(s))$ and $y = g(q) = g(f(t))$. Taking into account that each packet record in the capture contains a 16 byte packet header, the total volume V of packet data between p and q is given by $V = y - x - 16C = g(q) - g(p) - 16(q - p)$.

It is worth noting that the index files may be used to derive other metrics, such as average packet arrival rate and average packet size [74]. These metrics indicate the average number of packets received per second and the average size of individual packets, respectively. Average arrival rate is given by $\frac{C}{t-s} = \frac{q-p}{t-s}$, while average size is given by $\frac{V}{C} = \frac{y-x}{q-p} - 16$. These metrics are not visualised in the current implementation, but are displayed when a particular tree node (or time segment) is highlighted in the interface (see Figure 8.2).

8.2.3.3 Filter Match Count

The filter match count is a measure of how many packets in a given interval match a particular filter. Each node stores one count for each filter, which is derived by summing the number of 1 bits in segments of bitwise predicate results (i.e. finding the Hamming weight of the bitstream). Unlike the metrics generated from index files, which only require accessing the first and last elements for a given node, generating the filter match count requires processing all recorded bits for the period between s and t in all filter files. The visualiser employs a high-speed CUDA-based filter counting kernel to facilitate this.

The counting kernel is a post-processing function provided by the classification server, invoked from the C# client through a 0MQ socket connection over TCP. The client reads and packages filter results data as byte arrays, cropped to the results relevant to the node and masking edges where necessary. Each array segment sent from the client is received by the C++ server in its own separate GPU execution stream and returns a single integer result to the host on completion.

The counting kernel is shown in Listing 27. The kernel begins with each thread performing a coalesced 32-bit integer read from the filter segment into register memory. The integer is then immediately processed using the hardware accelerated `__popc` (or population count) intrinsic function, which counts the number of 1 bits set in a register at a throughput of 32 operations per warp (or alternatively one instruction per thread) per clock cycle [87]. This allows the warp to convert the filter results of 1,024 packets into 32 partial sums in a single clock cycle.

Once complete, each thread in the warp contains an initial count covering a specific set of 32 packets. These are summed through a warp shuffle based reduction, which provides fast and implicitly synchronous register copies between threads in

Listing 27 Counting kernel implementation.

```
1  __global__ void CountingKernel(int* filterSegment, int segmentSize,
       unsigned long long int* resultPtr)
2  {
3      int index = blockDim.x * blockIdx.x + threadIdx.x;
4      int count = index < segmentSize ? __popc(filterSegment[index]) : 0;
5
6      count += __shfl_xor(count, 16);
7      count += __shfl_xor(count, 8);
8      count += __shfl_xor(count, 4);
9      count += __shfl_xor(count, 2);
10     count += __shfl_xor(count, 1);
11
12     if ((threadIdx.x & 31) == 0) atomicAdd(resultPtr, count);
13 }
```

a warp. To sum the counts stored across all registers in a warp, threads invoke the `__shfl_xor` function, which is specifically intended to facilitate a butterfly reduction pattern (see Section 2.7.2). The reduction requires $log_2(32) = 5$ warp shuffle operations, after which point the combined total matches for the warp is replicated in every warp thread. Finally, the first thread in the warp uses an atomic function to add the warp's total count to the stream specific global count, stored in device memory, which is retrieved by the host on completion of the execution stream, and sent via 0MQ to the client application. This provides high throughput, as 3.5 devices support high speed global atomics [81].

Once the filter counts for all nodes have been processed and stored, the tree is used to generate vertex data for the render process. This process is discussed in the following subsection.

8.2.4 Rendering Graphs

The capture visualisation control renders a set of stacked graphs (one for each filter/metric) over a given time interval and at a specific render level. The time frame defaults to the capture length, but may be adjusted by drilling down into subsections of the capture; the render level is initially determined heuristically, but may be increased or decreased through keyboard shortcuts.

The graphs displayed by the control are rendered using separate vertex arrays that are generated on demand (when the render level or time frame is changed), using

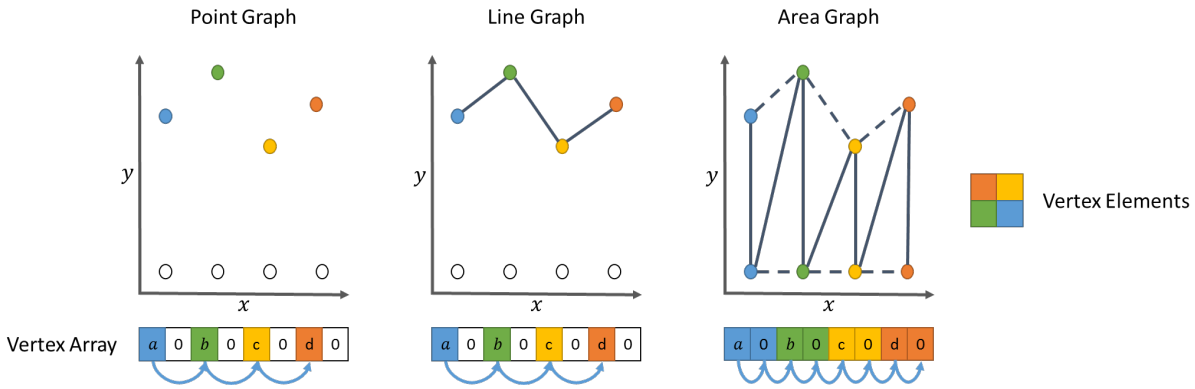


Figure 8.7: Rendering point, line and area graphs from an array of vertex data.

LINQ (Language-Integrated Query) expressions to process the metric values contained within the time tree [56]. Vertex data is only generated for visible portions of the graph, and thus as the render level becomes more fine grained, the size of the section of the capture that is processed is simultaneously reduced. This ensures the number of data requests and storage space required per render does not scale exponentially as the render level is decreased, similarly to node population as discussed in Section 8.2.2.

Each graph is encoded as an array of two dimensional Cartesian coordinates of the form (x, y), where x is the node's start time and y is an unscaled metric value contained by the node. Each vertex array encodes the y values for the graph, using the value's index offset in the array to infer an x value. These coordinates may be rendered as either a scatter plot, a line graph, or a solid graph through simple GLSL vertex and fragment shaders, by varying the draw function and array stride. These graph types are illustrated in Figure 8.7. The vertex shader applies a simple world transformation in order to position and scale the generated vertices for display, while the fragment shader simply applies a graph specific colour to each pixel. Graph render order, draw type and graph colour can be easily adjusted at this point, as these adjustments do not require vertex data to be reprocessed.

8.3 Capture Distillation

Capture distillation is a process that repackages subsets of packets from large capture files into much smaller pre-filtered captures. Distillation is performed by the main C++ server process to maximise read performance. It is invoked, however,

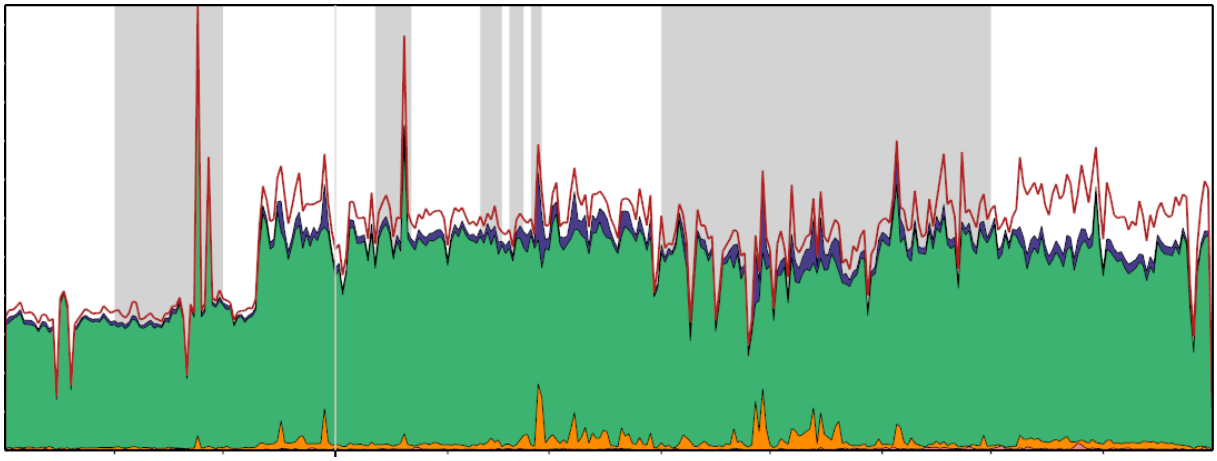


Figure 8.8: Multiple selected packet ranges within the graph control.

from the .NET visualisation control discussed in the previous section, which simplifies the selection of arbitrarily sized subsets of packets in large captures by allowing highlighted sections, at varying render levels, to be marked using a hot key (see Figure 8.8).

On invocation within the client process, each selected subset within the graph control is converted into one or more tasks, which are dispatched to the C++ distillation function through a 0MQ TCP socket. A task is composed of two offset-length pairs containing index and byte ranges. The index range specifies the range of packets contained in the task, while the byte range indicates the portion of the capture in which the packets are stored. These ranges are derived from the index files generated during classification. To simplify the server side process, tasks are sized by the client such that they do not exceed the capacity of the read process's input buffers. Received tasks are processed one after another in sequence, and may be broken down into three general processing steps: reading, filtering and writing. A simplified overview of the process and its component steps is provided in Figure 8.9.

The first step of the process is performed in a dedicated I/O thread, and involves buffering the relevant portion of the packet capture into memory. This is achieved using the byte offset and length pair contained in the task specification. Once loaded, the buffered data is passed to a subsequent thread where the remainder of processing takes place. Using the supplied index range in the task specification, in conjunction with the packet index file and filter results file, the process selectively copies packets which passed the filter from the buffer to a new capture file, while skipping over and ignoring those that failed. Once every packet index in the task

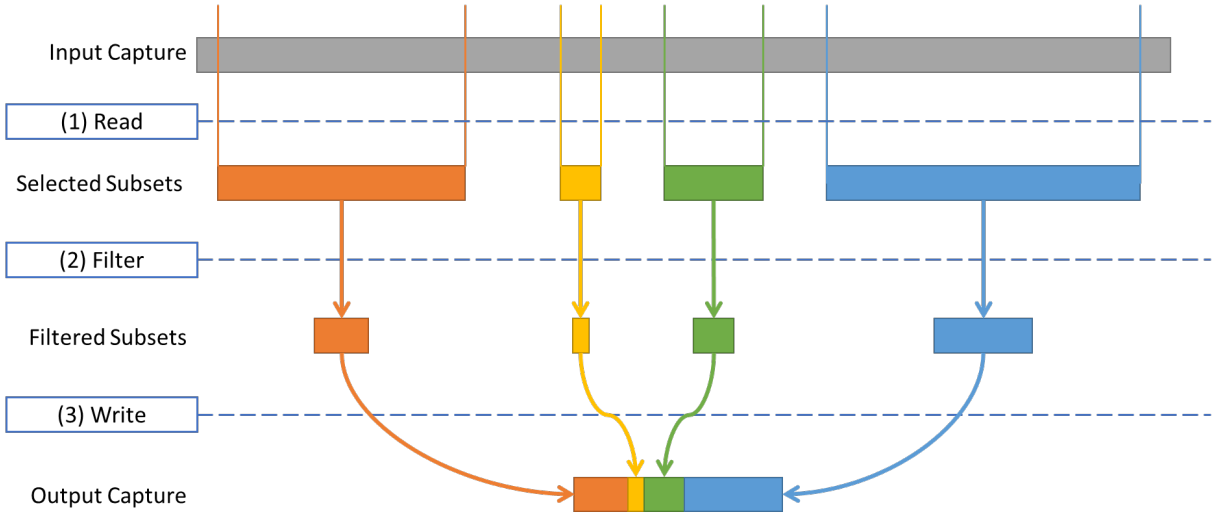


Figure 8.9: Distillation process overview.

has been processed, the function begins processing the next task, continuing until no further tasks remain. The server signals its completion to the client prior to terminating, after which the produced capture can be analysed in most protocol analysis suites. The client will optionally load the capture automatically in Wireshark on completion if requested in the GUI.

8.4 Summary

This chapter briefly investigated three example applications which apply the outputs of the classification system to facilitate simple but useful capture analysis functions.

The first example application, described in Section 8.1, uses extracted field data to create and display field value distributions, which can then be used to identify the proportion of packets in a capture that use a particular protocol or field value. The distribution is derived by an unsophisticated C# function, with the intention of illustrating performance gains achievable by ad hoc CPU applications that apply classification outputs.

The second example application, discussed in Section 8.2, uses index and filter results to render an adjustable high level visualisation of the capture file using OpenGL. The visualiser uses a tree structure, built using index and filter files, to

hold detailed metrics of large captures. This structure is used to render high level overviews of capture, supported by a CUDA kernel to quickly count filter results.

The final application discussed was the distiller function, which uses the visualiser, filter results and index files to generate cropped and pre-filtered captures. This function helps to facilitate rapid analysis of large captures in existing protocol analysis software by eliminating unnecessary packets, and trimming captures to a manageable size.

Having described the design and implementation of the components within the processing pipeline in some detail, the following part of the document evaluates and discusses the performance of these components both collectively and in isolation.

Part IV

Evaluation

9

Testing Methodology

THIS part of the document discusses the performance results for the system components discussed in Part III. This part is organised into three chapters, inclusive of the current chapter, in order to divide discussion into manageable, related segments with a logical progression:

- This chapter introduces the testing methodology, reports the system configuration used, and describes the input programs and captures employed during the testing process.
- Chapter 10 evaluates the performance of the CUDA-based classification kernel in isolation. The classification kernel is considered separately, as it is the most important component in the system and the main focus of this research.
- Chapter 11 investigates the performance of the end-to-end system, including both the classification process (inclusive of both GPU and CPU components) and the post processing functions which employ system outputs.

The current chapter begins by providing an overview of testing goals, system configuration and the verification methods used. This is followed by

an overview of the three packet captures used in testing and the nine filter programs applied to them. The chapter concludes with a short summary.

9.1 Performance Criteria

The GPF+ classifier and its supporting system were evaluated with respect to four general performance criteria (throughput, accuracy, efficiency and scalability), which relate directly to the research goals (see Section 1.3.3). Flexibility and usability are not included, as these were addressed through system design. The approaches taken to evaluate the system based on these criteria are briefly elaborated upon in the remainder of this section.

- Throughput was measured by timing the execution of various processes, and extrapolating performance metrics from these records using the known size and packet counts of the processed captures. Results were reported in milliseconds for the CUDA classification kernel, and in seconds for the classification system. These timings were used to determine the number of packets processed per second (packet rate) and the volume of data processed per second (data rate).

- The accuracy of the produced results were verified through the distillation of smaller filtered captures using produced results. These were checked for consistency using existing verified software (see Section 9.3).

- Efficiency was measured in terms of device resource utilisation and host memory overhead. Metrics relating to device performance were collected through Nvidia Nsight [85], while host memory utilisation was measured by examining the peak working set memory of tested host processes through Windows Task Manager [57].

- Scalability was assessed by testing nine filter programs of increasing complexity, over three captures of increasing size (see Sections 9.4 and 9.5). These results are subsequently compared to establish device size scalability with respect to filter program complexity, and host side scalability with respect to capture size.

Table 9.1: Test system configuration (host).

CPU	Make	Intel
	Model	Core i7-3930K
	Cores	6
	Base Frequency	3.2 GHz
RAM	Type	DDR3 1600
	Total Memory	32 GB
	System Memory	24 GB
	RAM Disk	8 GB
Operating System		Windows 7 x64
PCIe Interface		PCIe 2

Table 9.2: Technical comparison of test graphics card specifications.

	GTX 750	GTX Titan
Manufacturer	Gigabyte	MSI
Version	GV-N750OC-1GI	06G-P4-2790-KR
Micro-architecture	Maxwell	Kepler
Compute Capability	5.0	3.5
CUDA Cores	512	2688
Device Memory (MB)	1024	6144
Memory Type	GDDR5	GDDR5
Core Base Clock (MHz)	1020	837
Memory Bandwidth (GB/s)	80	288
Release Price	$119	$999
Release Date	February 18^{th}, 2014	February 21^{st}, 2013

9.2 System Configuration

This section describes the hardware and software configuration used during testing. Testing was performed using the system configuration shown in Table 9.1 [37] and two separate graphics cards: a low-end Maxwell-based GTX 750 [82]; and a high-end Kepler-based GTX Titan [80]. Both cards used unmodified 340.62 Geforce drivers, acquired through the NVIDIA Geforce website[1]. The technical specifications of these cards is given in Table 9.2.

Testing used both SATA II HDDs and SATA III SSDs as storage mediums for packet captures (see Table 9.3). Outputs were written to an 8 GB RAM disk in system memory to avoid both unnecessary competition with the capture reading

[1] \url{http://www.geforce.com/drivers}

Table 9.3: Drives used for capture storage and retrieval during testing.

	HDD	SSD [20]
Interface	SATA II	SATA III
Drive	Various	Crucial MX100
Model No.	Various	CT256MX100SSD1
Drive Count	4	2
Drive Capacity	1.5TB - 2TB	256 GB
Peak Read Speed	±120 MB/s	±500 MB/s

process and skewing of end results from poor write performance. The virtual RAM-based drive provides high read and write performance, at the expense of a quarter of available system memory. This is not typically necessary, as system outputs are far less expansive than inputs, and are written in intermittent fragments on demand to any specified drive, generating little contention. This configuration does, however, reduce the likelihood of output processes interfering with achieved performance.

9.3 System Verification

Verification testing was performed prior to other aspects of testing to ensure the process produced accurate results. Verification testing was achieved through internal and external mechanisms, including Wireshark, post-processing visualisation and distillation functions, and a simple bit-string visualiser for low level filter result checking (see Figure 9.1).

Initial verification was performed by comparing results generated by Wireshark to those stored in filter results files. This involved comparing the number of packets passed by the filters, as well as randomised low-level inspection of the permutation of accepted and rejected packets. The former ensured that the right number of packets were classified, while the latter ensured they were stored in the correct locations. This approach required testing against small files however, as larger captures could not be loaded successfully into Wireshark for comparison. In addition, while the approach was useful in spotting major programming bugs, its utility was diminished when attempting to verify hundreds of millions of records. As a result, this method was used for initial verification during development, while final verification was provided instead by the visualisation and distillation components.

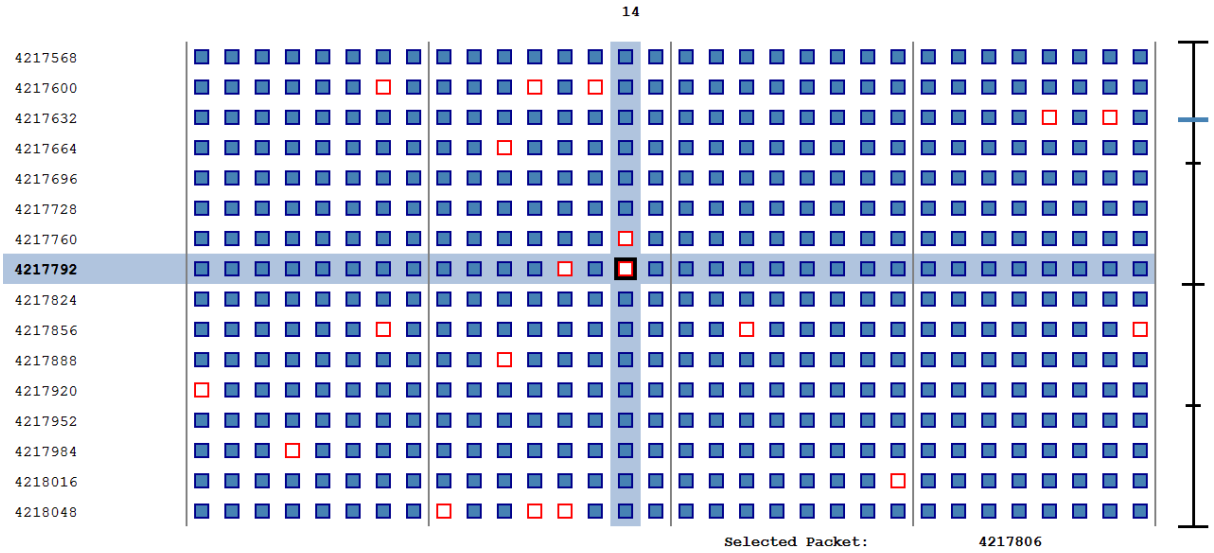


Figure 9.1: Bit-string visualiser showing per-packet results of a filter results file.

The visualisation and distillation functions provided an easy means to extract small (and electively pre-filtered) groups of packets from large captures using the filter outputs of the system. These captures were lightweight enough to load and filter in Wireshark without issue, providing a mechanism to inspect and compare the contents of unfiltered and filtered captures. A selection of non-uniform results were used to filter, extract and distil an assortment of sub-captures small enough to be processed within Wireshark from each packet capture used. These distilled captures were tested to ensure that (a) an equivalent Wireshark filter returned all packets in the capture, and (b) the negation of the filter in Wireshark returned no packets.

While this method is, ultimately, still based on sampling select portions of results, and thus does not prove the absence of error, it demonstrates that filter results are at least accurate over the tens of thousands of contiguous packets within each sample. As this would be highly unlikely if the process had any significant tendency toward misclassification or malfunction, it provides sufficient verification for the approach employed.

9.4 Capture Files

Testing was performed using three packet sets, referred to alphabetically for simplicity, the details of which are summarised in Table 9.4, as well as in Appendix C.4.

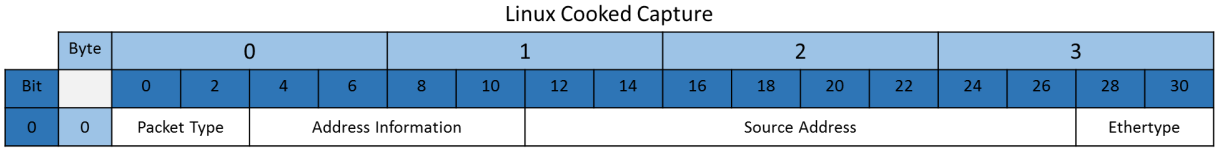


Figure 9.2: The Linux Cooked Capture pseudo-protocol header.

Table 9.4: Packet sets used in testing.

Packet Set	A	B	C
Total Packets	26,334,066	260,046,507	338,202,452
Average Size	70 bytes	155 bytes	480 bytes
Average Rate	0.9 /s	15 /s	12,150 /s
File Size	2.1 GB	41.4 GB	156 GB
Duration	11 months	6 months	8 hours

Packet sets A and B store packets containing their original Ethernet frame headers, while packets in set C employ Linux Cooked Capture pseudo-protocol headers instead (see Appendix D.1). The Linux Cooked Capture format [130] provides a standard format to record packets from multiple heterogeneous interfaces, and is also used in cases where the link-layer header is either inconsistent or absent. Cooked Capture headers contain the same two byte Ethertype field as Ethernet, but place the field at byte 14 rather than byte 12 in the header [130]. An overview of the Linux Cooked Capture header is shown in Figure 9.2.

9.4.1 Packet Set A

Packet set A contains packets collected via an Ethernet interface from a /24 network telescope between the 1^{st} of October 2009 and the 31^{st} of August 2010 [39]. The majority of packets in this capture are unrequested TCP SYN packets, with a small contingent of UDP packets and other packet types. As a result, the capture does not include much payload data, and thus packet sizes remain relatively small. An overview of the capture, generated by the visualisation component from system outputs, is shown in Figure 9.3. This is the only included capture that can be loaded into Wireshark.

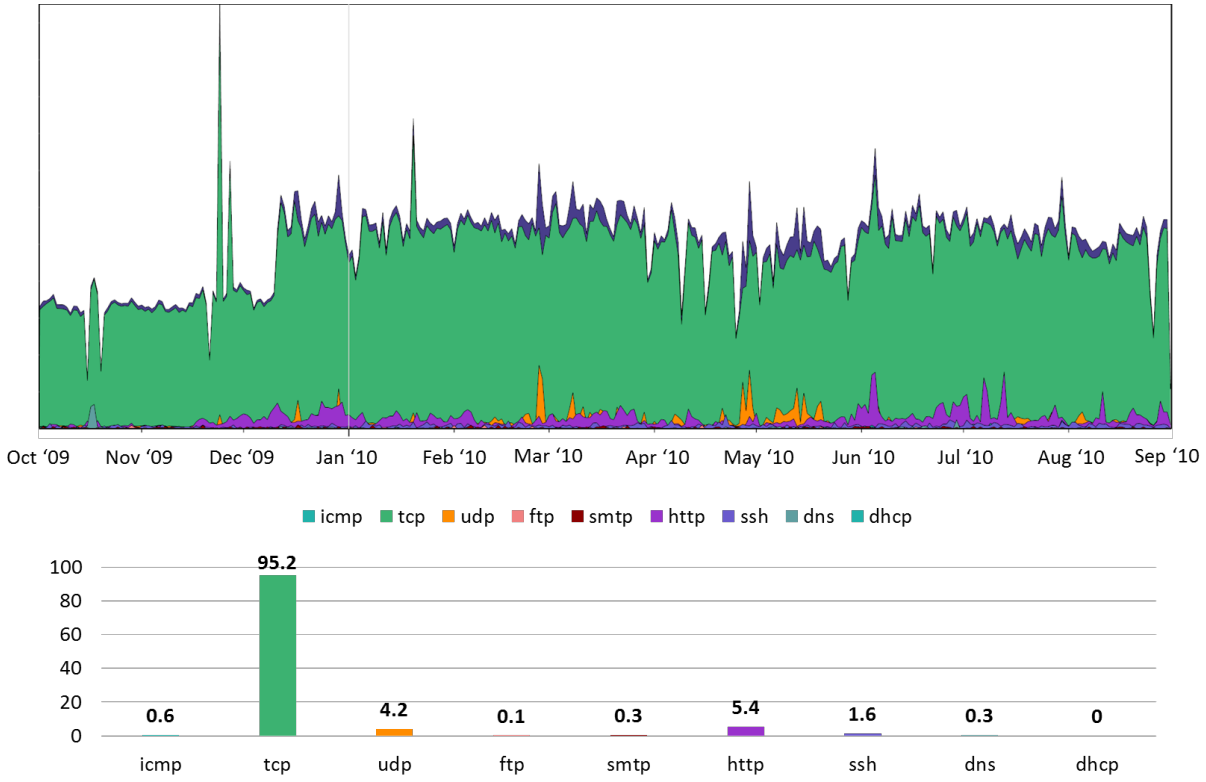


Figure 9.3: Overview of Packet Set A.

9.4.2 Packet Set B

Packet set B is a moderately sized long term capture, consisting almost exclusively of UDP and TCP DNS traffic, collected from an Ethernet interface between the 21^{st} of October 2010 and the 1^{st} of May 2011. Capture B is just under twenty times larger than packet set A (despite being collected over a shorter time period), due to a higher average packet rate ($\approx 16.6 \times A$) and larger average packet size ($\approx 2.2 \times A$). The capture is illustrated in Figure 9.4.

9.4.3 Packet Set C

Packet set C is a large and dense short term capture, collected between 3:56 PM and 11:55 PM on December 2^{nd} 2011, that contains a wide variety of protocols captured from a fast live network. The capture was originally recorded as 1,600 separate 100 MB capture files, in order to manage analysis with existing CPU tools; these were combined to form a single large capture in order to test the system's

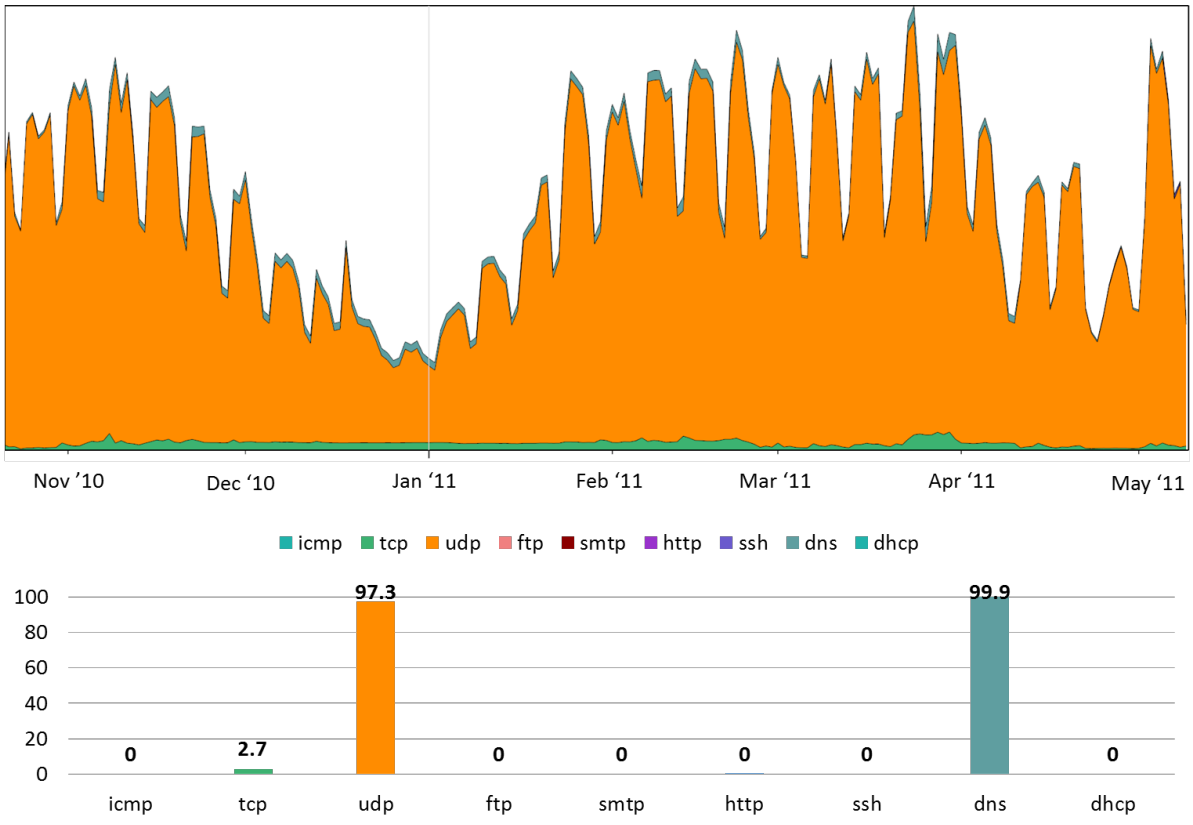


Figure 9.4: Overview of Packet Set B.

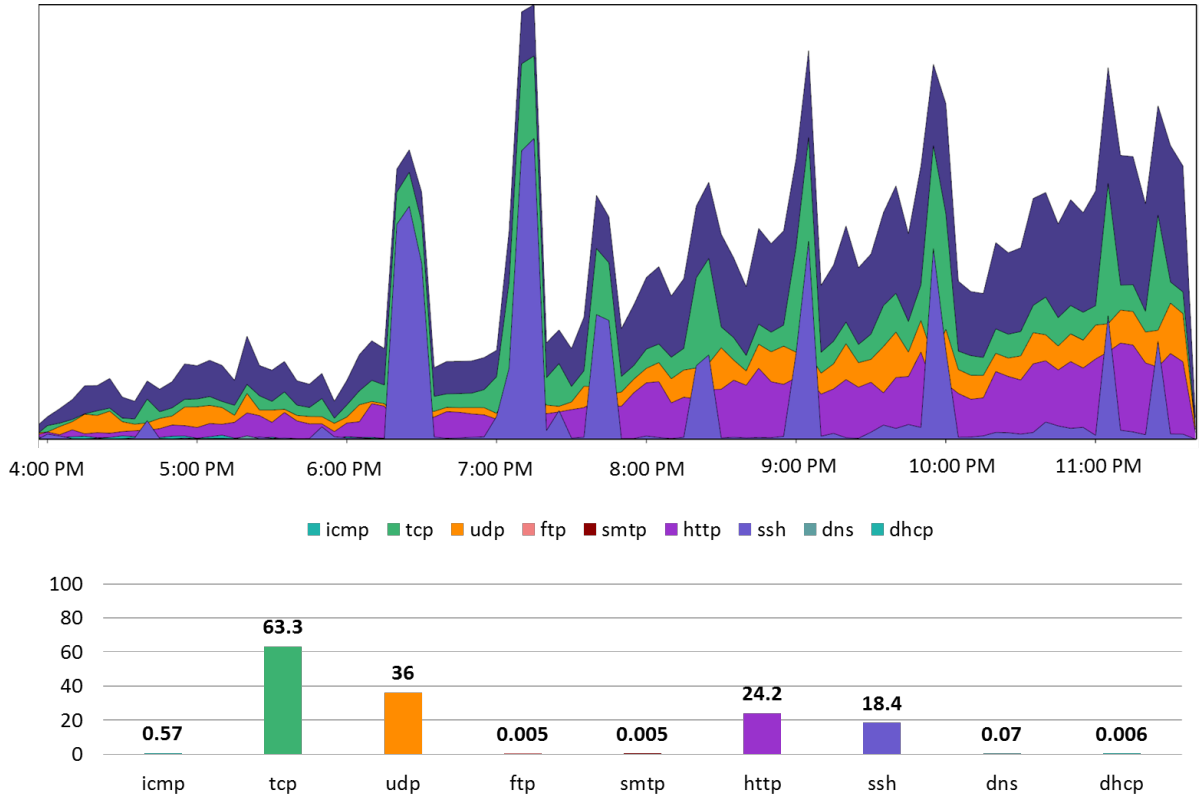


Figure 9.5: Overview of Packet Set C.

scalability to large capture files. An overview of the capture is provided in Figure 9.5.

While the capture contains only eight hours of recorded live traffic, it is more than four times larger than capture B – again due to a significantly higher average packet arrival rate ($\approx 810 \times B$) and larger packet size ($\approx 3.1 \times B$). These averages are skewed upward by short (± 10 minutes) intermittent bursts of SSH traffic, carrying significantly larger payloads; this is illustrated in Figure 9.6, which shows the SSH spikes in isolation from other protocols, with a red line showing the corresponding average packet size.

9.5 Testing Programs

Testing was performed using nine different filter programs of varying complexity and result cardinality. The listings for these programs are contained in Appendix D. Programs have been grouped into three categories, or sets – referred to as set

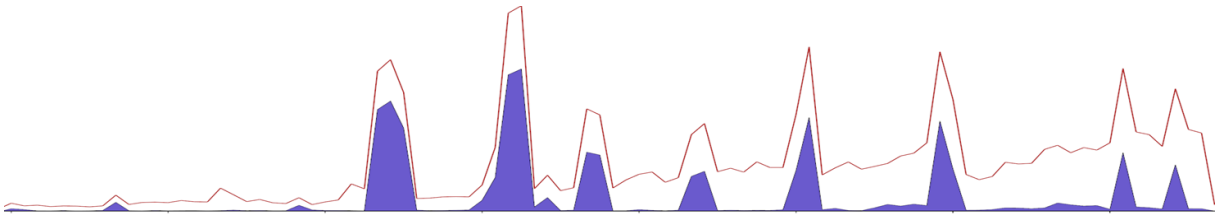


Figure 9.6: Average packet size (red line) superimposed over SSH packet count (purple area).

A, B and C – which differentiate filters based on the types of results returned: filter results, field values, or both. Programs within sets are numbered in order of increasing complexity.

- Set A contains programs that return only filter results.
- Set B contains programs that return only field values.
- Set C contains programs that return both filters and field values.

Filtering requires kernels to read similar volumes of data to field extraction, but consumes far less bandwidth when storing and returning results. By contrast, however, field extractions require fewer operations than filtering to evaluate, as they translate to a single coalesced memory write. Measuring the performance of each function in isolation and in unison provides a means of identifying bottlenecks in either function, and determining efficiency and performance scaling when executing both operation types simultaneously.

Testing was performed using two versions of each program. The program used while evaluating packet sets A and B uses Ethernet II as the link-layer protocol, while the program for packet set C uses the CookedCapture pseudo-protocol instead [130] (see Figure 9.2).

9.5.1 Program Set A

These three programs produce filtering results only, and are intended to test filtering performance in isolation from field extraction.

Listing 28 Program A1 kernel function.

```
1  main() {
2      filter ip = IP;
3  }
```

Listing 29 Program A2 kernel function.

```
1  main() {
2      filter tcp = TCP;
3      filter udp = UDP;
4      filter icmp = ICMP;
5      filter arp = ARP;
6  }
```

Listing 30 Program A3 kernel function.

```
1  main() {
2      filter icmp = ICMP;
3      filter tcp = TCP;
4      filter udp = UDP;
5      filter ftp_src = TCP.SourcePort == 21 || TCP.DestinationPort == 21;
6      filter smtp = TCP.SourcePort == 25 || TCP.DestinationPort == 25;
7      filter http = TCP.SourcePort == 80 || TCP.DestinationPort == 80;
8      filter ssh = TCP.SourcePort == 22 || TCP.DestinationPort == 22 ||
9              UDP.SourcePort == 22 || UDP.DestinationPort == 22;
10     filter dns = TCP.SourcePort == 53 || TCP.DestinationPort == 53 ||
11             UDP.SourcePort == 53 || UDP.DestinationPort == 53;
12     filter dhcp = UDP.SourcePort == 68 && UDP.DestinationPort == 67 ||
13             UDP.SourcePort == 67 && UDP.DestinationPort == 68;
14     filter dhcp6 = UDP.SourcePort == 546 && UDP.DestinationPort == 547 ||
15             UDP.SourcePort == 547 && UDP.DestinationPort == 546 ||
16             TCP.SourcePort == 546 && TCP.DestinationPort == 547 ||
17             TCP.SourcePort == 547 && TCP.DestinationPort == 546;
18 }
```

Listing 31 Program B1 kernel function.

```
1 main() {
2     int proto = IP.Protocol;
3 }
```

Listing 32 Program B2 kernel function.

```
1 main() {
2     int proto = IP.Protocol;
3     int srcaddr = IP.SourceAddress;
4     int destaddr = IP.DestinationAddress;
5     int srcport = Ports.SourcePort;
6     int destport = Ports.DestinationPort;
7 }
```

Program A1 This program contains a trivial filter set that classifies all packets against the IPv4 protocol. The generated program contains only a single protocol layer, and thus does not invoke protocol switching. A1's kernel function is shown in Listing 28. The complete program is provided in Appendix D.2.1.

Program A2 This program contains four filters that identify all TCP, UDP and ICMP(v6) packets (over either IPv4 or IPv6), as well as all ARP packets. The program produces two protocol layers, and requires a single protocol switch from Ethernet to either IPv4 or IPv6. A2's kernel function is shown in Listing 29. The complete program can be found in Appendix D.2.2.

Program A3 This program contains ten filters that primarily target protocols on the transport layer of the protocol stack. In addition to TCP, UDP and ICMP(v6) packets, it locates FTP, SMTP, HTTP, SSH, DNS, DHCP and DHCPv6 packets. The resultant program contains three separate protocol layers, and additional protocol switches between IPv4/IPv6 and their respective transport protocols. Listing 30 shows the kernel function for A3. The complete program source is available in Appendix D.2.3.

9.5.2 Program Set B

These three programs perform field extraction operations exclusively, and are intended to evaluate this function in isolation.

Program B1 This program is a simple field extraction program that records the

Listing 33 Program B3 kernel function.

```
1  main() {
2      int ethertype = Ethernet.EtherType;
3      int tcpsrcport = TCP.SourcePort;
4      int tcpdestport = TCP.DestinationPort;
5      int tcpseqno = TCP.SequenceNumber;
6      int tcpackno = TCP.AcknowledgmentNumber;
7      int udpsrcport = UDP.SourcePort;
8      int udpdestport = UDP.DestinationPort;
9      int udplen = UDP.Length;
10     int icmptype = ICMP.Type;
11     int icmpcode = ICMP.Code;
12 }
```

IPv4 protocol field from the Internet layer of processed packet data. The program requires two protocol layers, reading field data from within the IPv4 protocol layer. Similarly to A1, this program is primarily intended for comparison to other more complex field extraction programs. Listing 31 shows the kernel function used in B1. The full program source can be found in Appendix D.3.1.

Program B2 This program extracts each field of the IPv4 5-tuple (source address, destination address, protocol, source port, and destination port). This program uses fields from both the Internet and Transport layers of the protocol stack (see Section 3.2), and thus uses three separate protocol layers in the compiled program. B2 is the only program which uses 4 cache loads, using a second load in the internet layer to extract IP address values. B2's kernel function is shown in Listing 32. The complete program source may be found in Appendix D.3.2.

Program B3 This program is the most complex pure field extraction program, recording ten fields from the Ethernet, TCP, UDP and ICMP protocols. Specifically, the program extracts:

- Ethernet EtherType.
- TCP source port, destination port, sequence number and acknowledgement number.
- UDP source port, destination port and length.
- ICMP type and code.

Listing 34 Program C1 kernel function.

```
1  main() {
2      filter ipv4 = IPv4;
3      filter ipv6 = IPv6;
4      int proto = IPv4.Protocol;
5      int nextHeader = IPv6.NextHeader;
6  }
```

Listing 35 Program C2 kernel function.

```
1  main() {
2      filter tcp = IPv4.Protocol.TCP || IPv6.NextHeader.TCP;
3      filter udp = IPv4.Protocol.UDP || IPv6.NextHeader.UDP;
4      filter icmp = ICMP;
5      filter arp = ARP;
6      filter ssh = ServicePorts.SourcePort.SSH ||
7              ServicePorts.DestinationPort.SSH;
8      filter dns = ServicePorts.SourcePort.DNS ||
9              ServicePorts.DestinationPort.DNS;
10     int srcport = ServicePorts.SourcePort;
11     int dstport = ServicePorts.DestinationPort;
12     int icmptype = ICMP.Type;
13     int icmpcode = ICMP.Code;
14 }
```

Like B2, this program uses three protocol layers. The kernel function used in B3 is shown in Listing 33. The full program can be found in Appendix D.3.3.

9.5.3 Program Set C

The final three programs test programs which invoke both filtering and field extraction processes.

Program C1 This program contains two filters targeting IPv4 and IPv6, and two field extractions storing IPv4's Protocol and IPv6's Next Header fields. Like B1, this program requires two layers to evaluate. The complete source for the program can be found in Appendix D.4.1.

Program C2 This program is composed of six filters (four involving simple predicates) and four field extractions. The program filters for TCP and UDP packets in layer two, as well as ICMP, ARP, SSH and DNS packets in layer three. The program additionally extracts the source and destination ports of TCP / UDP

packets, as well as the type and code fields contained in ICMP packets. The filter program source is provided in Appendix D.4.2.

Program C3 This program is the largest included, simultaneously evaluating all ten filters performed in A3 and all ten field extractions performed in B3. This allows for comparisons between the performance of these three programs, and provides a means of demonstrating performance scalability to larger programs. The program executes over three protocol layers, similarly to both A3 and B3. C3's kernel function contains all definitions in Listings 30 and 33, and is too long to list here. The complete source for the program can, however, be found in Appendix D.4.3.

9.6 Summary

This chapter introduced and discussed the methodology, hardware configuration and inputs used throughout classification kernel and system testing in the following two chapters.

The chapter began by discussing the main performance criteria to be evaluated during the course of testing in Section 9.1. Section 9.2 outlined the system software and hardware configuration, including details regarding the GPU accelerators and storage devices used. Section 9.3 elaborated briefly on how results were verified using the distillation post-processor and Wireshark to verify the accuracy of generated results.

The final sections discussed the packet capture and classification program inputs used during testing. Section 9.4 briefly described the three capture files used, and Section 9.5 discussed the nine filter and field extraction programs against which the capture files were processed.

The following chapter considers the performance of the classification kernel in isolation from the rest of the system, while the subsequent chapter examines end-to-end system and post-processor performance.

10

Classification Performance

THIS chapter reports on the performance of the classification kernel described in Chapter 6, using the performance analysis functionality supplied by Nvidia Nsight 4.2 [85]. This discussion does not consider host side overhead, which is discussed separately in Chapter 11. The remainder of the chapter is structured as follows:

- Section 10.1 provides an overview of kernel testing, discussing the kernel test configuration and collected results in general.
- Section 10.2 reports on the performance of the individual filtering programs included in set A (see Section 9.4.1)
- Section 10.3 reports on the performance of the individual field extraction programs included in set B (see Section 9.4.2).
- Section 10.4 reports on the performance of the individual mixed programs included in set C (see Section 9.4.3).

Table 10.1: Kernel execution configuration.

	GTX 750	GTX Titan
Buffers Size	128 MB	256 MB
Stream Count	4	4

- Section 10.5 compares the results from program sets A, B and C collectively, in order to show and discuss observed scaling.
- Section 10.6 concludes the chapter with a summary.

10.1 Kernel Testing Overview

The classification kernel described in Chapter 6 was analysed on both GTX Titan and GTX 750 graphics cards, using the nine programs introduced in Section 9.5 and the four captures listed in Section 9.4.

Tests were executed using a block size of 128 threads, which allows 16 blocks to execute on each multi-processor simultaneously (the maximum supported on Kepler GPUs). Kernels were configured to use four asynchronous streams of execution (see Section 2.8.2) to improve occupancy and hide access latency, balanced to minimise buffer memory utilisation. Higher stream counts do not seem to significantly impact kernel performance, but do require additional host-side buffers to adequately maintain in the current implementation.

Due to differences in available device memory, different buffer sizes were used when executing on different devices (see Table C.2). The GTX Titan, which has over 6 GB of available device memory, was configured to use four 256 MB buffers, requiring 1 GB of device memory overall. The GTX 750, however, only has 1 GB of device memory, and so uses four 128 MB buffers to provide room to hold system outputs. A summary of this configuration is shown in Table 10.1.

10.1.1 Measurements

This section briefly discusses the validity of results. Each capture was processed once in each execution configuration, and timed using the high precision performance analysis functionality provided by Nvidia Nsight. This has negligible impact

on accuracy, however, due to the functionality of Nsight and the size of packet captures used. Specifically:

1. The timings provided by the Nvidia Nsight monitor [85] are highly stable and not prone to significant variance, as all kernels were replayed (or repeated) by Nsight ten times during the performance analysis process [85]. Each kernel execution time reported by Nsight is thus already an average of multiple iterations of the kernel.

2. Due to the size of the captures, in comparison to that of device memory, captures B and C required tens of iterations of the kernel to complete. Each of these iterations was measured independently by the Nsight Monitor as an internal repetition, which could then be averaged.

3. As timings are initiated only after files have been fully loaded, the measured performance is less susceptible to variability introduced by OS file caching (this is not the case for the encapsulating system however, where it can have a dramatic impact due to the storage bottleneck).

10.1.2 Results Presentation

The tests performed in this chapter evaluate the performance of the classifier while processing each of the nine filter programs defined in Section 9.5. Each test includes a high-level figure that shows the overall performance for each test, and a table conveying aggregated metrics of each kernel invocation. These are discussed below.

The figure shown for each test contains three separate graphs; a linear graph of total kernel processing time (in seconds), and two logarithmic graphs showing the achieved packet rate (in millions of packets per second) and data rate (in gigabytes per second). All data rate estimations are adjusted to account for capture-specific overhead that may otherwise inflate results; the data rate presented, therefore, reflects the amount of packet data (rather than capture data) processed, calculated by subtracting the combined global and record header lengths from the total capture size.

The table included with each test presents additional lower-level information relating to the execution of the individual kernels, rather than the process as a whole.

The values reported are averages taken from all kernel invocations needed to process a particular capture on a particular device. The results ignore metrics collected from the last kernel invocation for each capture, as these invocations process partial buffers that could skew the results attained from Capture A (which uses few repetitions). Each table includes seven different metrics, which bear some explanation:

Packets The number of packets (in millions) processed by each kernel in each stream.

Time (ms) The average number of milliseconds required for each kernel to complete.

Time (σ) The standard deviation, in milliseconds, between all recorded kernel timings. Kernels which processed fewer than three full buffers do not have a standard deviation, and are instead marked with an asterisk (*).

Achieved Occupancy The average occupancy achieved between executed kernels. This value is calculated by the Nsight analysis function as a percentage of device resources[85].

SM Activity The percentage of execution time in which a Streaming Multiprocessor (SM) was actively processing a warp, averaged across all SMs. This value is also calculated internally by Nsight[85].

GPU Serialization The percentage ratio of the number of instructions replayed compared to the number of instructions actually issued [55].

Executed IPC The average number of discreet instructions executed per clock cycle [85].

Some runtime aspects do not vary between programs and are thus not reported individually. In all tests, each thread uses a total of 27 registers and each multiprocessor processes 16 blocks at a time. These values ensure optimal performance, as each thread can support up to 32 registers per thread and still maintain full occupancy, and each Kepler multiprocessor can support up to a maximum of 16 blocks.

10.2 Filtering Programs

This section details the performance of the classification kernel while processing three pure filter programs from program set A. Tests in this set do not record any field information, and thus produce only bit-based filter results, which require very little device-side bandwidth to store.

10.2.1 Program A1

The A1 program is a very simple program that never moves beyond the link-layer protocol, and only produces a single result; it is used to benchmark best case performance of classification, primarily for the purpose of comparison to other filter programs. The full filter program used may be found in Appendix D.2.1. An overview of the performance results for program A1 are provided in Figure 10.1 and Table 10.2.

The results displayed in Figure 10.1 show a good approximation of best case performance, achieving comparable packet throughput across captures on each device. As the packet rate is relatively unaffected by capture composition, the observed data rate scales linearly relative to average packet size. Capture B contains twice as much payload data per packet compared to capture A, and thus achieves twice the throughput of capture A for processing the same number of packets. Similarly, capture C accumulates over three times the volume of payload data per packet compared to capture B, and thus achieves a data rate roughly three times higher than that of capture B. Together, this shows that processing time is primarily bound by the number of packets processed, and is largely unaffected by packet size. This is expected, as all packets are cropped to uniform lengths prior to processing.

The performance metrics presented in Table 10.2 show that kernel execution time demonstrates little variance between results, with a sub-millisecond standard deviation in all tests. In addition, both devices managed to maintain high overall occupancy and multiprocessor activity. The GTX Titan achieved roughly six times the packet throughput of the GTX 750, with slightly higher overall occupancy and a significantly higher number of instructions executed per clock cycle. The GTX 750, however, required negligible serialization and achieved slightly higher SM activity. These differences are primarily attributable to differences between microprocessor and memory architectures, and remain relatively consistent throughout all tests.

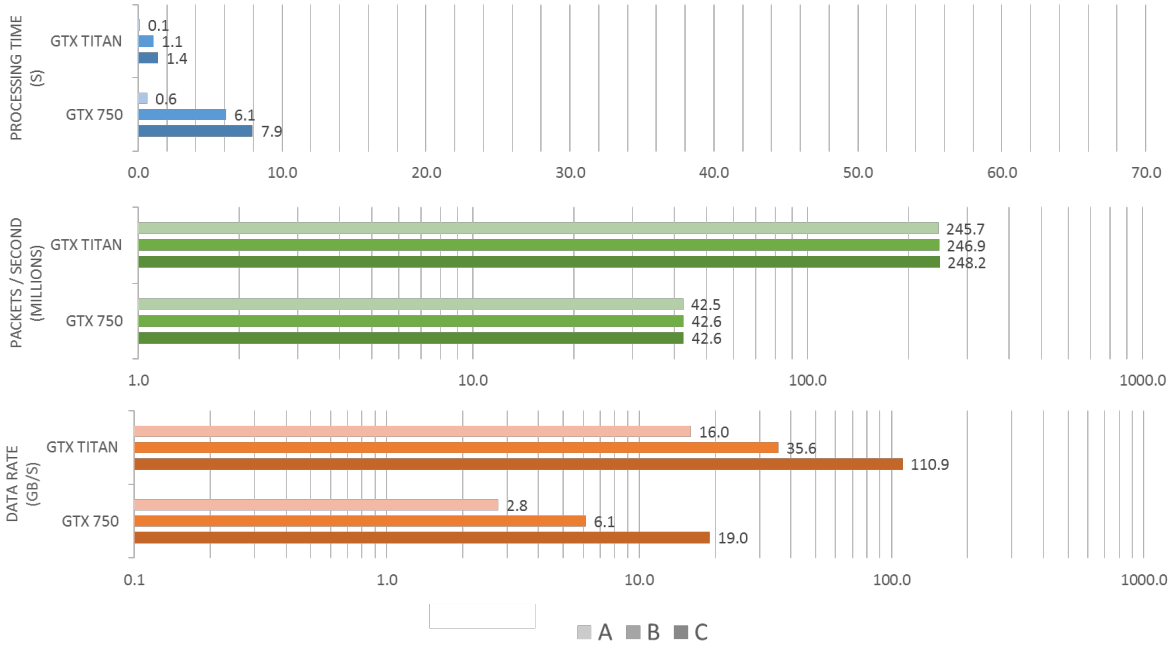


Figure 10.1: Program A1 Performance Results

Table 10.2: Program A1 Kernel Performance

		Overview			Performance Metrics			
		Packets (Millions)	Time		Achieved Occupancy	SM Activity	GPU Serialisation	Executed IPC
			ms	σ				
GTX 750	A	7.8	185	0.2	93.9%	99.7%	0%	1
	B	7.8	185	0.2	93.7%	99.8%	0%	1
	C	7.8	185	0.2	93.7%	99.8%	0%	1
GTX Titan	A	15.8	65	*	96.6%	97.2%	7.3%	2.5
	B	15.8	64	0.5	96.5%	98.4%	7.3%	2.5
	C	15.8	64	0.6	96.4%	98%	7.3%	2.5

* - Insufficient Iterations

Table 10.3: Program A2 Kernel Performance

		Overview			Performance Metrics			
		Packets (Millions)	Time		Achieved Occupancy	SM Activity	GPU Serialisation	Executed IPC
			ms	σ				
GTX 750	A	6.4	361	0.4	94.8%	99.7%	0%	1
	B	6.4	361	0.6	94.6%	99.7%	0%	1
	C	6.4	362	0.6	94.6%	99.7%	0%	1
GTX Titan	A	12.8	127	*	97.4%	99.1%	7.1%	2.4
	B	12.8	127	1.4	97.3%	99%	7.1%	2.4
	C	12.8	128	2.6	97.3%	99%	7.1%	2.4

* - Insufficient Iterations

10.2.2 Program A2

The A2 program classifies four separate filters that target both the Data-link and Internet layers of the protocol stack. While more complex than A1, the program does not actually interact with any Transport layer protocols, and thus does not require the evaluation of expressions to determine the offset of a later protocol (i.e. all protocols are treated as having a constant length). The full filter program used may be found in Appendix D.2.2. An overview of the performance results for program A2 are provided in Figure 10.2 and Table 10.3.

The results in Figure 10.2 show a similar pattern to that of program A1, with kernels achieving roughly equivalent packet throughput on each device regardless of actual packet capture composition, resulting in a data rate that correlates directly with average packet size. The only significant difference is in overall achieved performance, with the A1 program outperforming the A2 program by a factor of just under 2.5 times. This scaling is in line with expectations, as the program A1 traverses twice as many layers and produces four times as many filter results as program A2. The program displays high overall performance, producing 70 and 400 million distinct filter results per second on the GTX 750 and GTX Titan, respectively. In keeping with the observed performance in program A1, the GTX Titan outperforms the GTX 750 by close to a factor of six.

Table 10.3 indicates slightly higher variance between kernel execution times, which is partially attributable to runtime pruning through warp voting. Both devices achieved slightly higher occupancy, and the GTX Titan achieved higher SM activity and lower serialization overall, at the expense of evaluating slightly fewer instructions per clock cycle. The GTX 750, in contrast, shows greater overall stability, producing very similar metrics to those of program A1.

10.2.3 Program A3

Program A3 is the final and most complex filter specific program. It contains ten separate filters collectively utilising 29 independent comparisons between all evaluated predicates. The majority of defined comparisons operate on fields within the Transport (3rd) layer of the protocol stack (see Section 3.2). As a result, the length of the Internet layer protocol needs to be evaluated, adding extra work. The full program specification may be found in Appendix D.2.3. The performance results for program A3 are shown in Figure 10.3 and Table 10.4.

The performance results shown in Figure 10.3 are in line with expectations, with an achieved packet rate close to one third of that achieved in program A2, despite processing more filters, substantially more comparisons, and the necessary evaluation of protocol length fields. Unlike prior programs however, there is greater variance in achieved packet rate between captures. This may be attributable in part to runtime optimisations. For example, capture C is the only capture that includes IPv6 packets, which requires not only more protocol evaluations per layer in certain warps, but also doubles the number of protocol length calculations that need to be performed within these warps. Capture B, by contrast, contains predominantly UDP datagrams encapsulated in IPv4 headers, which allows many warps to skip processing both the IPv6 protocol and the TCP protocol through warp voting. As captures A and C both contain an abundance of TCP packets, and capture B contains very few, the kernel is able to optimise out TCP and IPv6 comparisons at runtime more easily when processing capture B.

In addition to this, the results shown in Table 10.4 indicate that capture C was configured to classify more packets per kernel iteration than in other captures. This is a product of two factors. Firstly, the A3 filter program requires more storage per packet than other filter specific programs, as it depends on headers in the transport layer. This reduces the number of packets that can be contained in each buffer, and thus the number that can be processed by each kernel. Secondly, and more importantly, Capture C employs a program that targets packets encapsulated within the Linux CookedCapture [130] pseudo-protocol (see Section 9.4 and Appendix D.1). While this program targets an EtherType field identical to standard Ethernet, it is positioned differently in packet data (at an offset of 112 bits from the start, rather than 96 bits). This slight difference in byte offset, in combination with the greater memory requirements per record in the A3 program, and runtime size adjustments

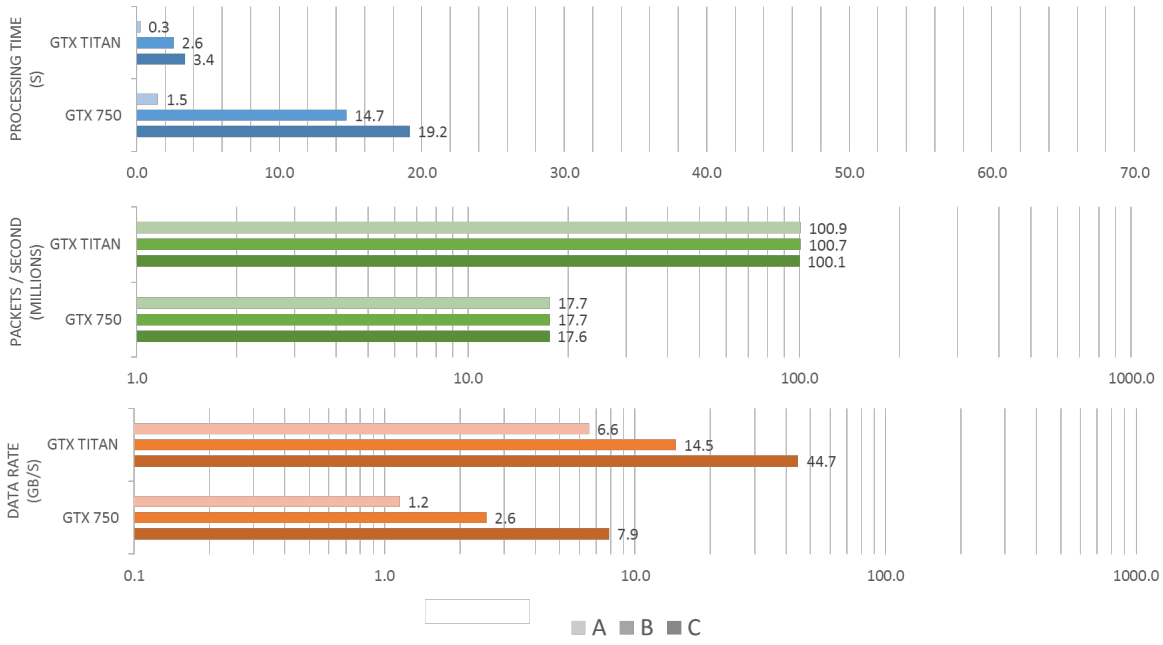


Figure 10.2: Program A2 Performance Results

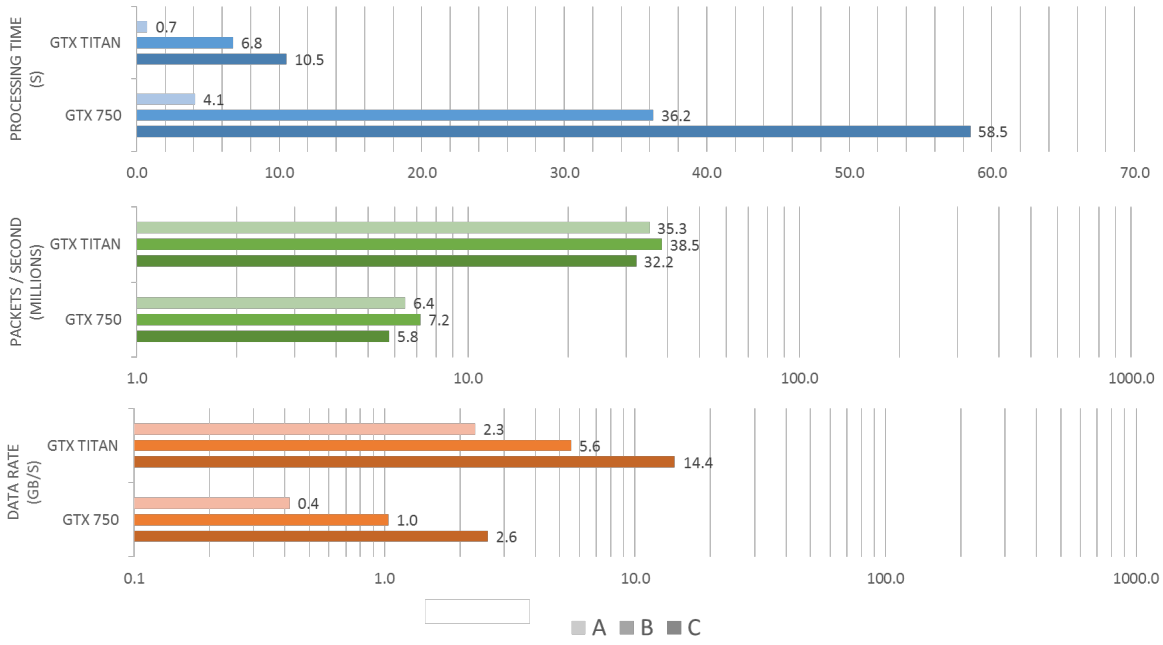


Figure 10.3: Program A3 Performance Results

to ensure proper alignment, allow the classification host process to use a slightly larger grid size during processing.

Aside from this difference, Table 10.4 shows several other slight differences when compared to previous tests. Firstly, the GTX 750 showed far higher variance between kernel calls than the GTX Titan, and slightly lower overall occupancy and SM activity. This appears to be due in part to starvation, as the supplied grid sizes for both tests was comparatively small (particularly with respect to the GTX 750). In addition, the slight difference in packet count for capture C shows a significant impact on SM activity, achieved occupancy, and the number of instructions executed per cycle. This is most notable on the GTX Titan.

In summary, these observations indicate that in some more complex and memory constrained cases, performance may be affected by both packet composition and grid size.

10.3 Field Extraction Programs

The next set of evaluation programs test the performance of field extraction in isolation. Field extraction is a simpler function than that of filtering, requiring neither function-specific inter-thread communication, or post extraction result aggregation. Storing field data is however significantly more bandwidth intensive than storing filter data, as a single field record uses 32 times as many bits of bandwidth per result. Similarly to the previous section, this section presents the results of three programs ranging in difficulty from a single field extraction to a moderately complex set of ten extractions.

10.3.1 Program B1

The B1 program includes a single operation that extracts the protocol field from the IPv4 protocol header residing in the Internet layer of the protocol stack. As a result, this program is more cache intensive than program A1 (which operates on the Data-link layer only) as it requires two cache loads rather than one to complete. The full filter program used may be found in Appendix D.3.1. An overview of the performance results for program B1 are provided in Figure 10.4 and Table 10.4.

Table 10.4: Program A3 Kernel Performance

		Overview			Performance Metrics			
		Packets (Millions)	Time		Achieved Occupancy	SM Activity	GPU Serialisation	Executed IPC
			ms	σ				
GTX 750	A	2.2	335	7.0	92%	99.2%	0%	1
	B	2.2	301	8.9	91.8%	99.3%	0%	1
	C	2.3	400	6.9	94.4%	99.4%	0%	1
GTX Titan	A	4.3	122	2.5	91.8%	96.9%	6.4%	2.4
	B	4.3	112	4.0	90.8%	96.3%	6.4%	2.3
	C	4.6	143	2.2	96.5%	93.0%	6.4%	2.6

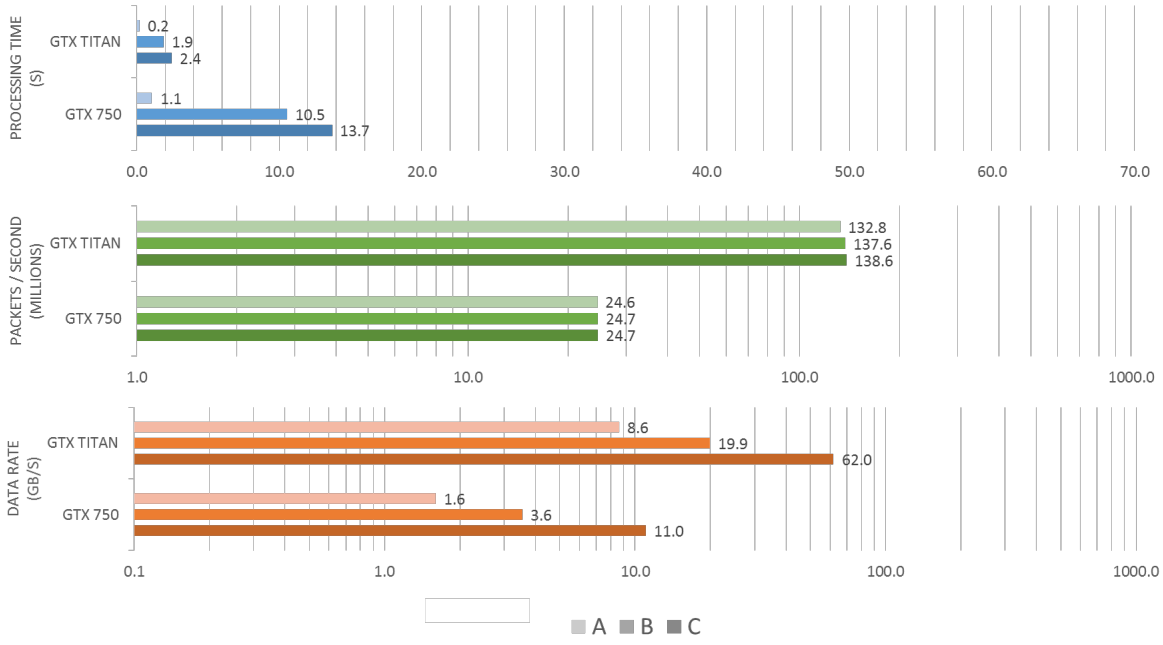


Figure 10.4: Program B1 Performance Results

Table 10.5: Program B1 Kernel Performance

		Overview			Performance Metrics			
		Packets (Millions)	Time		Achieved Occupancy	SM Activity	GPU Serialisation	Executed IPC
			ms	σ				
GTX 750	A	5.6	227	0.3	94.3%	99.5%	0%	1
	B	5.6	227	0.3	94.4%	99.6%	0%	1
	C	5.6	227	0.3	94.3%	99.7%	0%	1
GTX Titan	A	11.2	84	2.3	96.1%	96.1%	7.6%	2.4
	B	11.2	81	0.9	96.4%	96.5%	7.6%	2.4
	C	11.2	81	0.8	96.5%	97.0%	7.6%	2.5

Figure 10.4 shows a similar performance pattern to previous tests, with relatively consistent packet rates across captures, and thus a data rate scaled in proportion to average packet size. In terms of packet rate, program B1 falls between A1 (~245 million/s) and A2 (~100 million/s), although it is much closer in performance to A2 than A1. This is most likely a by-product of the additional cache load incurred by operating on two layers, which only program A1 avoids.

With respect to the kernel metrics shown in Table 10.5, program B1 performed similarly to program A1. While capture A shows a noticeably higher standard deviation (in combination with slightly lower overall performance in Figure 10.4), this seems to be a result of anomalous noise during execution; capture A is particularly small, requiring few iterations and a short execution cycle to completely process, which can inflate the impact of system noise when performance is extrapolated from results. With the exception of this performance anomaly, kernel performance metrics fall in line with established expectations.

10.3.2 Program B2

Program B2 collects five fields from the network and transport layers of the protocol stack, collectively referred to as the IP 5-tuple. These fields include the source and destination addresses, transport protocol from the Internet layer, and the source and destination ports in the transport layer where relevant. The full program specification may be found in Appendix D.3.2. The performance results for program B2 are shown in Figure 10.5 and Table 10.6.

Figure 10.5 shows somewhat better results than that of program A3, but fall far short of those achieved for A2. This is interesting, as B2 employs more cache loads

Table 10.6: Program B2 Kernel Performance

		Overview			Performance Metrics			
		Packets (Millions)	Time		Achieved Occupancy	SM Activity	GPU Serialisation	Executed IPC
			ms	σ				
GTX 750	A	2.2	225	0.8	94.3%	99.1%	0%	0.9
	B	2.2	224	0.9	94.2%	99.1%	0%	0.9
	C	2.4	242	0.8	94.4%	99.3%	0%	0.9
GTX Titan	A	4.5	83	0.5	96.3%	96.2%	7.1%	2.3
	B	4.5	83	2.1	96.2%	97.2%	7.1%	2.3
	C	4.8	95	4.6	95.3%	91.9%	7.1%	2.2

than any other program in the set, but process fewer protocols in the transport layer than other programs (as TCP and UDP ports are grouped as a single protocol). If performance was primarily determined by the number of cache loads, one would expect B2 to show lower overall performance than other programs which employ only three cache loads. This does, however, indicate that additional layers do not have as pronounced an impact on performance beyond three layers.

Table 10.6 shows similarities with program A3, in that the volume of packet data per record transferred to the device was large enough to produce different grid sizes for the two programs. This produced slightly better performance on the GTX 750, but seemed to have a negative impact on the performance of the GTX Titan in all areas but GPU serialization. The impact of this can be seen in the reported packet rate for capture C, which is slightly lower than would otherwise be expected in comparison to other results.

10.3.3 Program B3

Program B3 is the most complex field extraction specific program, collecting ten field values from the Data-link, Internet and Transport layers of each packet. Of these extracted fields, four are specific to the TCP protocol, three are specific to the UDP protocol, and two are specific to the ICMP protocol. The full program specification may be found in Appendix D.3.3. The performance results for program B3 are shown in Figure 10.6 and Table 10.6.

Figure 10.6 shows that program B3 supplies marginally lower throughput than program B2, despite extracting and storing twice as many field values from each packet. This may potentially be the result of B2 employing an additional cache

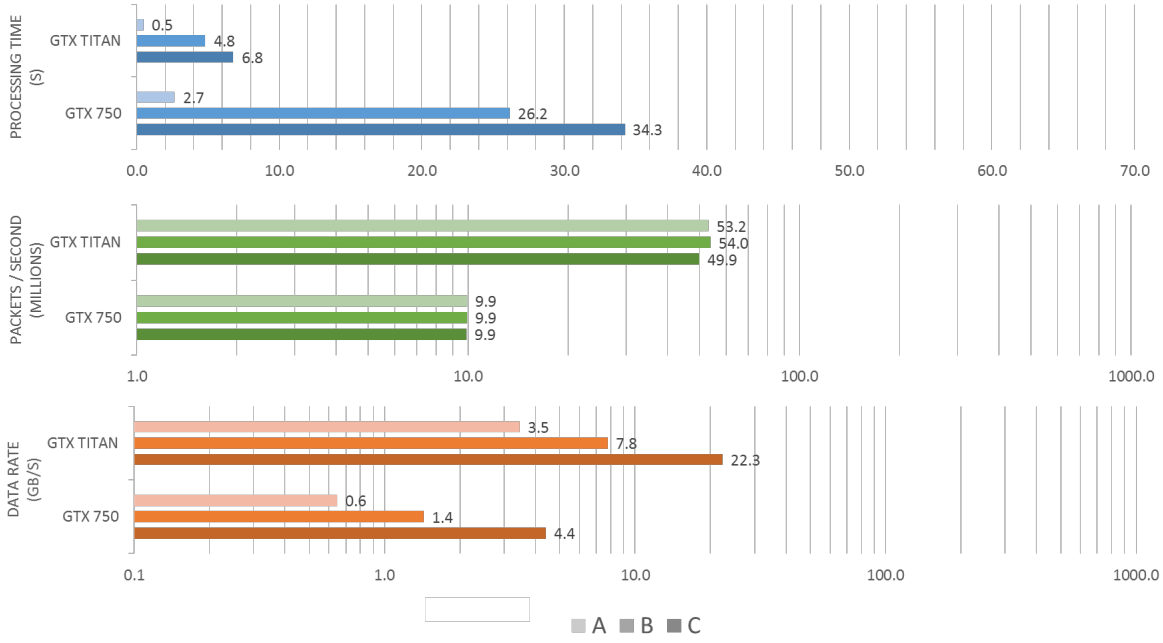


Figure 10.5: Program B2 Performance Results

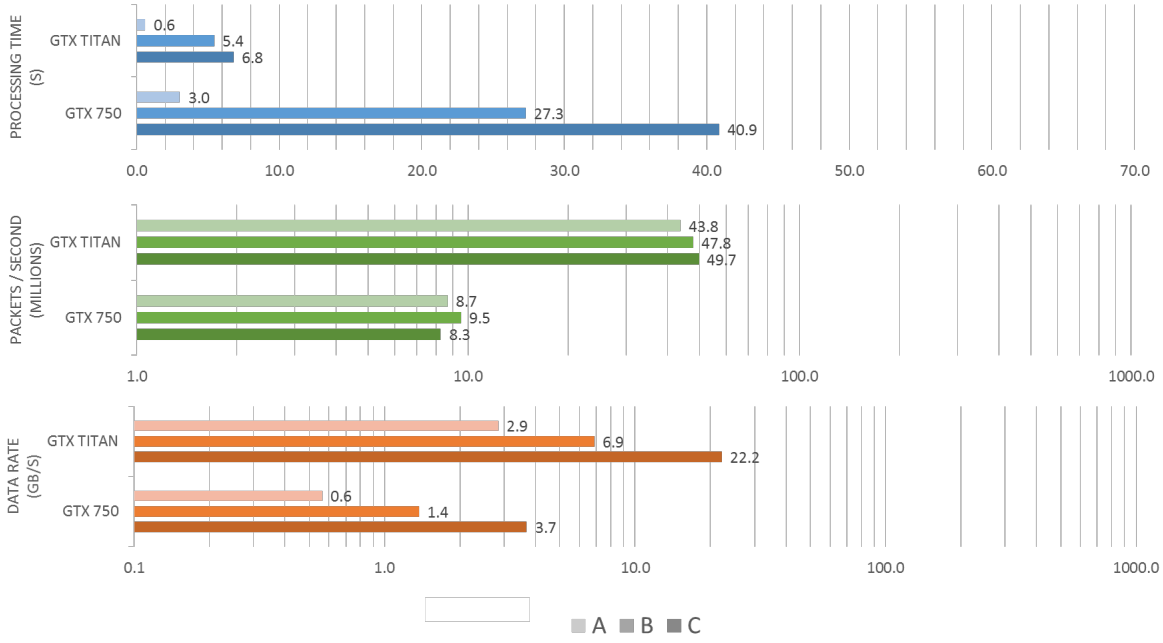


Figure 10.6: Program B3 Performance Results

load. The achieved packet rate shows similar variance to other programs operating within the Transport layer, with the exception of capture C's performance on the GTX Titan, which performs favourably in comparison to previous tests.

Table 10.7 provides a potential reason for this anomaly; namely, a significant difference in grid size between the kernel processing capture C and the kernels evaluating other captures which, in contrast to the previous set of results, seems to have improved overall performance on the GTX Titan. The GTX 750 results follow a pattern more closely resembling previous Transport layer programs, seemingly because the execution grid size remained constant for all captures on this device.

10.4 Mixed Programs

This section presents the results recorded for three programs that combine both filtering and field extraction. As with the previous set, programs range in difficulty from simple to moderately complex in order to show how performance is affected by program complexity.

10.4.1 Program C1

Program C1 is the simplest program in the set, containing two filters and two fields that fall within the IPv4 and IPv6 protocols. The program uses two layers to encode all protocols. The full filter program used may be found in Appendix D.4.1. An overview of the performance results for program C1 are provided in Figure 10.7 and Table 10.8.

Figure 10.7 shows that program C1 achieved a packet rate roughly mid-way between that of program B1 and program A2, falling within the same range as other programs that operate on two layers. This provides some indication that the process performs well when evaluating full programs, maintaining a packet rate in line with previous results. This observation is corroborated by the metrics presented in Table 10.8, which shows a similar performance pattern to simple programs in previous tests, with no significant deviations from established norms.

Table 10.7: Program B3 Kernel Performance

		Overview			Performance Metrics			
		Packets (Millions)	Time		Achieved Occupancy	SM Activity	GPU Serialisation	Executed IPC
			ms	σ				
GTX 750	A	1.9	213	2.1	93.5%	98.9%	0%	0.9
	B	1.9	194	3.5	93.4%	98.9%	0%	0.9
	C	1.9	224	3.1	94.1%	99%	0%	0.9
GTX Titan	A	3.7	84	1.5	94.7%	91.5%	6.5%	2.1
	B	3.7	78	2.1	93.9%	92.6%	6.6%	2.1
	C	4.8	96	1.6	95.6%	90.6%	7.1%	2.2

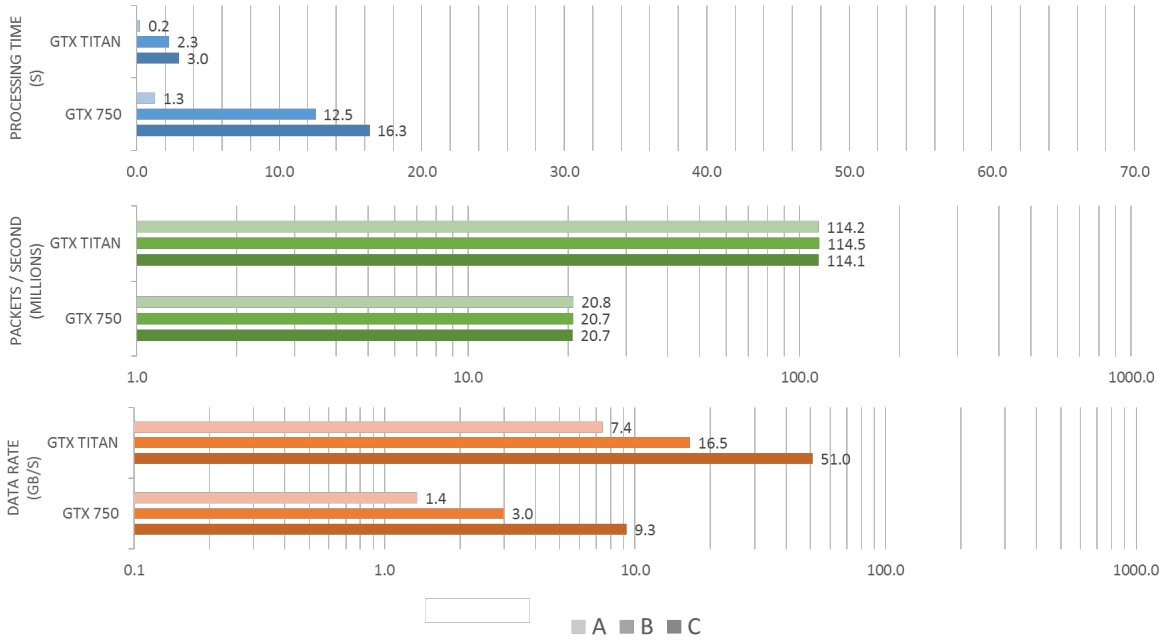


Figure 10.7: Program C1 Performance Results

Table 10.8: Program C1 Kernel Performance

		Overview			Performance Metrics			
		Packets (Millions)	Time		Achieved Occupancy	SM Activity	GPU Serialisation	Executed IPC
			ms	σ				
GTX 750	A	4.6	222	0.3	94.4%	99.6%	0%	1
	B	4.6	223	0.5	94.4%	99.6%	0%	1
	C	4.6	223	0.4	94.4%	99.6%	0%	1
GTX Titan	A	9.2	81	0.5	96.1%	95.7%	7.5%	2.4
	B	9.2	81	0.8	95.8%	97%	7.5%	2.4
	C	9.2	81	0.9	95.9%	97%	7.5%	2.4

Table 10.9: Program C2 Kernel Performance

		Overview			Performance Metrics			
		Packets (Millions)	Time		Achieved Occupancy	SM Activity	GPU Serialization	Executed IPC
			ms	σ				
GTX 750	A	3.3	362	2.0	94.3%	99.4%	0%	0.9
	B	3.3	356	0.9	94.6%	99.4%	0%	0.9
	C	3.3	361	4.9	94.4%	99.5%	0%	0.9
GTX Titan	A	6.5	140	1.1	96%	94.5%	6.7%	2.3
	B	6.5	136	1.1	96.2%	94.2%	6.7%	2.4
	C	6.5	134	2.7	95.9%	94.3%	6.7%	2.4

10.4.2 Program C2

Program C2 contains six filters (of which four are simple predicate strings) and four field extractions that target protocols on the Data-link, Internet and Transport layers of the protocol stack. The program makes no distinction between TCP and UDP packets, and processes their service ports collectively within a single pseudo-protocol definition, similarly to program B2. Classification performance results for program C2 are provided in Figure 10.8, while Table 10.9 tabulates the metrics relating to the execution of individual kernels.

The per packet classification performance of program C2 falls close to, and roughly between, that of program B2 and B3, and outperforms program A3 by an average margin of over 30%. While the program performs comparably overall to similar three-layer programs, fitting previous observations, the typical variance in performance between individual capture sets compared to other three layer programs is noticeably less pronounced. With respect to kernel metrics, program C2 behaved comparably to previous programs, with high overall occupancy and SM activity, and low standard deviation. The GTX Titan achieved a slightly higher occupancy, while the GTX 750 achieved higher overall multiprocessor activity.

10.4.3 Program C3

Program C3 is the most complex program overall, being composed of all operations from both the filter-intensive A3 program and the field-intensive B3 program. This makes the program particularly useful for evaluating how throughput scales with added program complexity, as the performance of each component program has

Table 10.10: Program C3 Kernel Performance

		Overview			Performance Metrics			
		Packets (Millions)	Time		Achieved Occupancy	SM Activity	GPU Serialization	Executed IPC
			ms	σ				
GTX 750	A	1.2	219	4.9	91.3%	98.4%	0%	1
	B	1.2	189	6.1	90.9%	98.4%	0%	0.9
	C	1.3	254	5.0	93.5%	98.7%	0%	1
GTX Titan	A	2.4	84	1.5	85.1%	93.7%	6.3%	2.4
	B	2.4	75	2.8	85.2%	94%	6.4%	2.3
	C	2.5	90	3.7	91.0%	95.6%	6.3%	2.6

been established in isolation. Performance results for the program are given in Figure 10.9, while kernel metrics are displayed in Table 10.10.

Despite the complexity of the program, C3 achieves a promising packet throughput lower than, but still comparable to, that of program A3. For instance, while the total time required to process capture C against both programs A3 and B3 on the GTX Titan was slightly over 17.3 seconds, when combined into a single program these same operations took only 12.1 seconds (only 1.6 seconds longer than A3 on its own). This supports the hypothesis that performance is more significantly influenced by the number of independent layers (and thus distinct cache loads) than by the number of filters evaluated or values extracted.

With the exception of lower than average occupancy, the presented kernel metrics are generally comparable to prior complex programs. The decreased occupancy may be partially attributable to the significantly increased number of memory transactions serviced per packet, due to the number of field and filter results evaluated. Decreased occupancy may also be related to the small grid size used; this could be improved on the GTX Titan (by using more execution streams of larger buffers), as it has surplus memory capacity far exceeding the GTX 750.

10.5 Performance Evaluation

Having concluded the presentation of results on a test by test basis, this section provides an evaluation of these results collectively, focussing predominantly on program scalability. To aid in this evaluation, Figure 10.10 plots the average packet throughput (in million of packets per second) and effective computed data rate (for

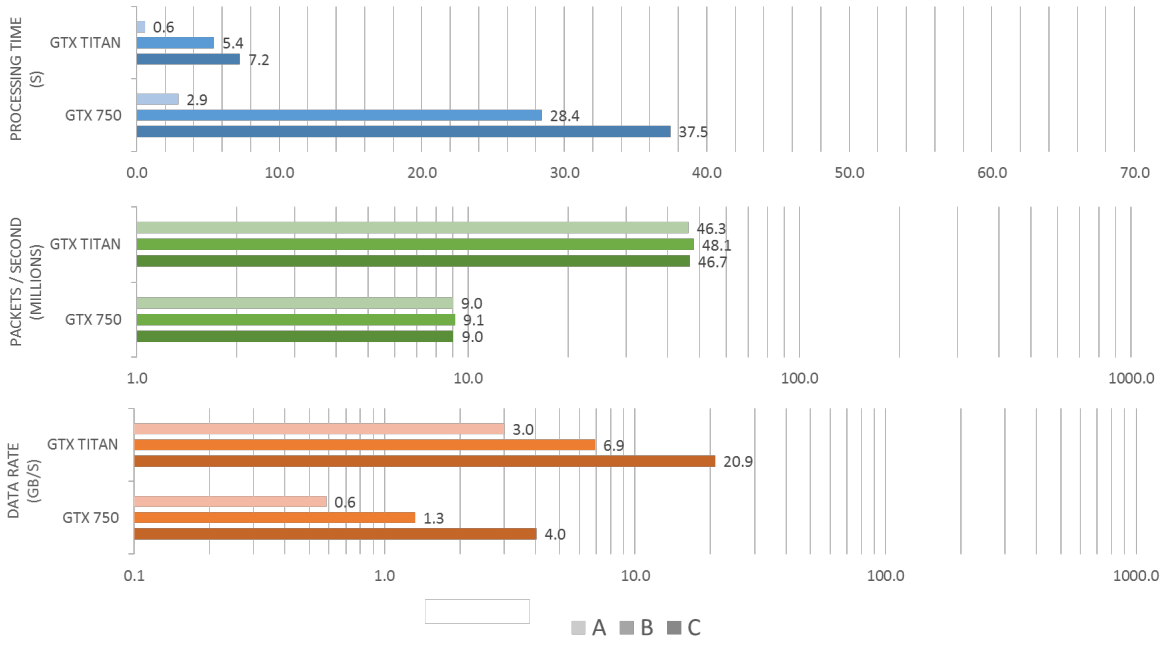


Figure 10.8: Program C2 Performance Results

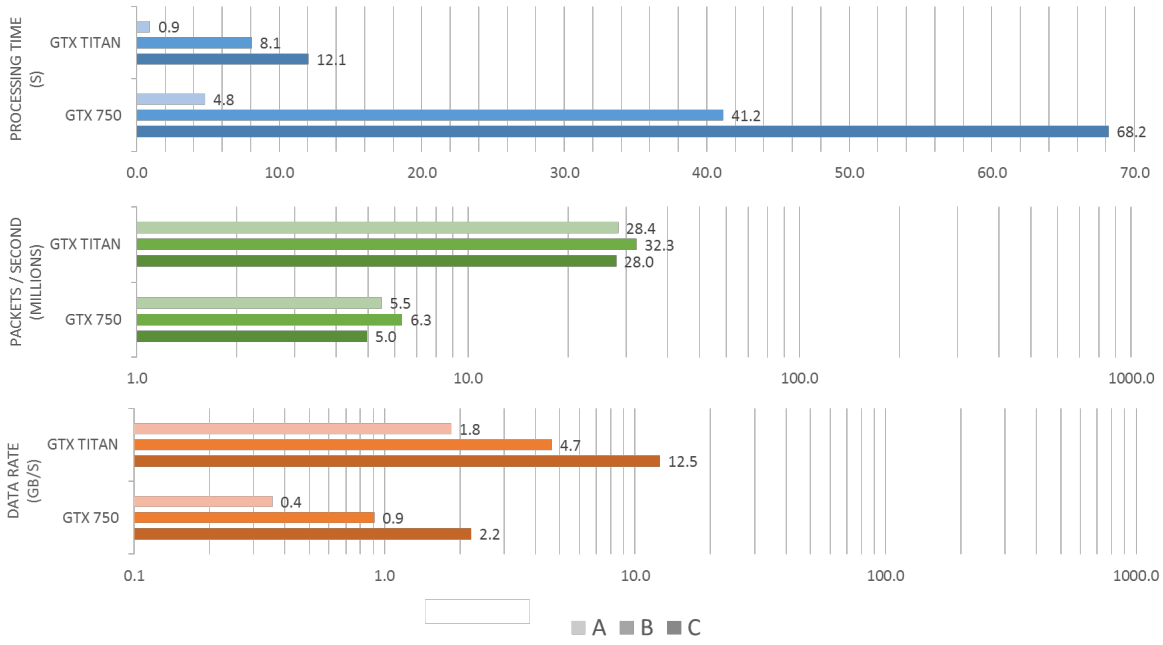


Figure 10.9: Program C3 Performance Results

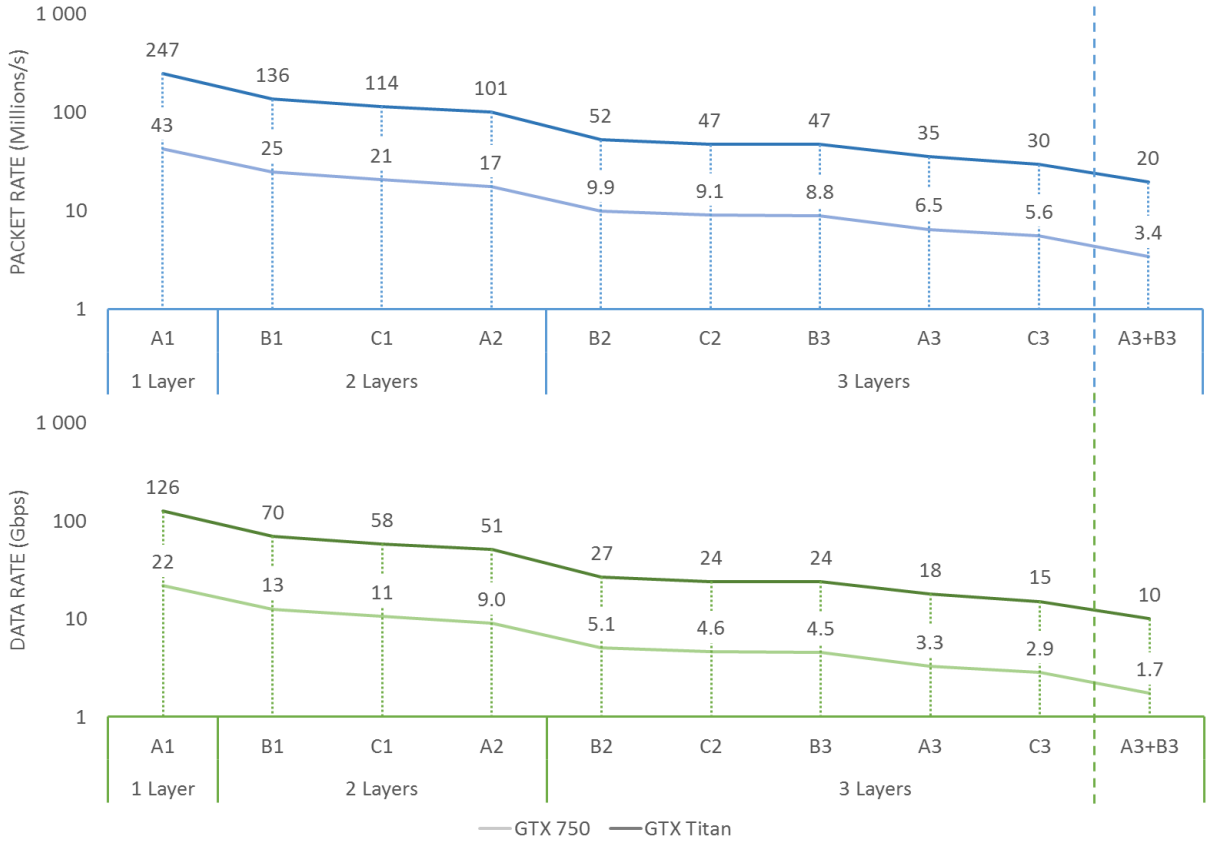


Figure 10.10: Average packet rate per test program, in decreasing order of performance.

64 byte packets, in Gbps) for each test and device, in descending order of performance, using a logarithmic y-axis.

The figure supports the hypothesis that throughput is primarily correlated with layer count, with performance declining significantly between programs with differing numbers of layers. Performance declined only marginally, however, with respect to different volumes of filter and field extraction operations, indicating improved scaling to larger and more diverse programs. Consider, for example, the packet rates of A1, B1 and C1: the throughput of A1 is almost twice that of B1, while B1 is only about 20% faster than C1. This is significant because A1 and B1 differ by one in terms of layers processed, but are otherwise comparable in terms of work performed: A1 produced 1 filter while B1 produced 1 field. In contrast, programs B1 and C1 share the same number of layers, but have very different operation counts (C1 processes 2 field and 2 filters, double that of A1 and B1 combined). The figure also shows that performing A3 and B3 at the same time is 50% more efficient than performing these operations separately. This provides an indic-

ation that in programs with similar layer counts, performance scales sub-linearly with respect to the number of protocols, filters and fields processed (i.e. it is more efficient to process multiple protocols in a single kernel than it is to process them separately).

The performance discrepancy between A1 and B1, which differ only in the number of cache loads performed, seems to be due to added global memory overhead resulting from the cache function. Most results seem to support this observation with the noted exception of B2, which is both the fastest performing three layer program, and the only program to employ a fourth cache load. While further testing is required to be certain, it seems from results that the higher ratio of computational work to memory transactions helps to hide memory transaction overhead as filter sets scale in size, thereby limiting the impact of memory contention and access latency overhead in more complex programs.

Long complex predicates also appear to have an appreciable impact on performance, as evidenced by program A3. This is likely tied to how predicates are evaluated, as each predicate atom must be stored in the gather process and subsequently loaded from coalesced (but still comparatively slow) global memory. The process used to store filter results during the gather process is particularly bandwidth inefficient, as only the first thread in each warp actually writes data to device memory (see Section 6.7.2). All other threads are left idle, wasting bandwidth. This was done for both simplicity and to avoid over-utilising register resources, but could be improved by utilising either spare registers or shared memory as a temporary store. This would allow multiple iterations to be written in a partially coalescing pattern to device memory at a later point in execution.

A final aspect of execution, notable in Figure 10.10, is the comparable performance of the GTX 750 and GTX Titan. In all instances, the GTX Titan was just over half an order of magnitude (5x to 6x) faster than the GTX 750. As the GTX Titan contains 5.25x as many cores as the GTX 750, this performance ratio follows the comparative computational resources of the devices quite closely.

10.6 Summary

This chapter reported on results collected from the CUDA based classifier in isolation from other system components using Nvidia Nsight 4.2. Performance analysis

was performed using nine separate programs, subdivided into three distinct categories: filtering programs, field extraction programs and mixed programs.

Section 10.1 provided an overview of the testing performed, relating how measurements were taken and describing how results were presented. Performance results and execution metrics were subsequently provided for each program in each set and briefly discussed in Sections 10.2, 10.3 and 10.4 respectively. Section 10.5 concluded the chapter by comparing all collected results, indicating that the classification process scales to process additional filters and fields relatively well, but does not scale well with respect to the number of layers used. This was primarily attributed to the expected poor performance of global memory, although the correlation became less clear as filter programs grew in complexity.

Device performance results were generally promising, showing high device occupancy and activity, limited serialisation and relatively consistent performance (with standard deviations often below a millisecond). Performance across devices was also shown to scale relative to the number CUDA cores available, although differences in multiprocessor architecture had some additional impact on achieved performance. Specifically, Maxwell architecture achieved higher overall occupancy and utilisation in most tests, and did not suffer from instruction serialisation.

The following chapter investigates the performance of the end-to-end classification process, including CUDA classification, capture buffering and indexing operations, as well as post-processor functions.

11

System Performance

THIS chapter investigates the performance of the complete capture processing system, inclusive of capture buffering, indexing and classification functions. System testing is divided into two subsections that consider processing throughput and host resource utilisation respectively. This chapter also investigates the performance of the post-processor components, which apply the outputs generated by the system within the contexts of specific applications. The chapter contains the following sections:

- Section 11.1 evaluates and discusses the throughput of the end-to-end classification system using a variety of input and processing configurations.
- Section 11.2 evaluates the memory utilisation of the classification system, and the storage requirements of output files.
- Section 11.3 examines the performance of each of the post-processing functions, and briefly discusses the implications of results.
- Section 11.4 concludes the chapter with a summary.

11.1 Processing Throughput

This section reports on the capture processing rate achieved using the system (inclusive of initialisation, memory allocation, capture reading, indexing, classification, and output generation). Inputs for this evaluation include the three captures previously described in Section 9.4. These captures vary significantly in size and thus processing time; for the purposes of comparison, collected timings are converted using the known capture size into an achieved packet and data rate, in packets/s and MB/s respectively.

The system was tested on both the Nvidia GTX 750 and the Nvidia GTX Titan (see Appendix C) using five different capture buffering configurations in three different execution modes. The purpose of these tests is to show how GPU device power, execution modes and storage mediums affect system performance. In addition, the tests are intended to evaluate how performance scales when using multiple files sources, and how this compares to a comparable RAID 0 (Striped) medium.

11.1.1 Testing Configurations

Performance testing was relatively broad, covering 90 different unique configurations timed over a combined total of 900 iterations, processing over 17 terabytes of capture data collectively. Each unique combination of GPU device, packet capture, storage medium and function was executed and timed ten times and subsequently averaged to derive the reported result. The target capture was cycled between every test to prevent disk caching from artificially inflating processing speed; as the combined size of these captures is just under 200 GB, captures are purged from host cache by the time they are reprocessed. All outputs are written to an 8 GB RAM disk for consistency and in order to minimise interference from the output file writing process, which could otherwise artificially inflate processing time.

An overview of the system configuration can be found in Section 9.2 or Appendix C. The capture buffer configurations used are summarised in Table 11.1 and described below:

HDD x 1 Captures are read from a single HDD source.

Table 11.1: Capture buffering configurations.

	Drives	Interface	Striped	Mirrored	Identical	Unused
HDD x 1	1	SATA II	No	No	–	No
HDD x 4	4	SATA II	No	Yes	No	No
SSD x 1	1	SATA III	No	No	–	Yes
SSD x 2	2	SATA III	No	Yes	Yes	Yes
SSD RAID 0	2	SATA III	Yes	No	Yes	Yes

Table 11.2: Execution configurations for end-to-end system testing.

Execution Configuration	Classify on GPU	Index on Host	GPF+ Program
Filter	Yes	No	C2
Index	No	Yes	None
Both	Yes	Yes	C2

HDD x 4 Captures are read from four unstriped HDD file mirrors simultaneously. Drives vary by make, model, capacity and level of prior utilisation.

SSD x 1 Captures are read from a single SATA III SSD source.

SSD x 2 Captures are read from two unstriped SATA III SSD file mirrors. Drives share identical make, model and capacity, and were both otherwise unused.

SSD RAID0 Captures are read from software managed RAID 0 array of two striped SATA III SSD drives, which were otherwise unused.

Testing was performed in three distinct execution configurations that vary the work done by the process. These configurations are summarised in Table 11.2. The use of program C2 in this testing (rather than the more complex C3 program) was motivated by the ratio of output volume (particularly with respect to capture C) to available output space in RAM disk. Storage utilisation is discussed in more detail in Section 11.2.

The results of system performance testing are presented in Figures 11.1 and 11.2, for the GTX 750 and GTX Titan respectively. Results have been converted to MB/s for the purpose of normalisation. In addition to these graphs, Figures 11.3 and 11.4 shows the total processing time (in seconds) and the standard deviation (in milliseconds) between all iterations of a particular test. The following sections discuss various aspects of measured performance, using these graphs for reference.

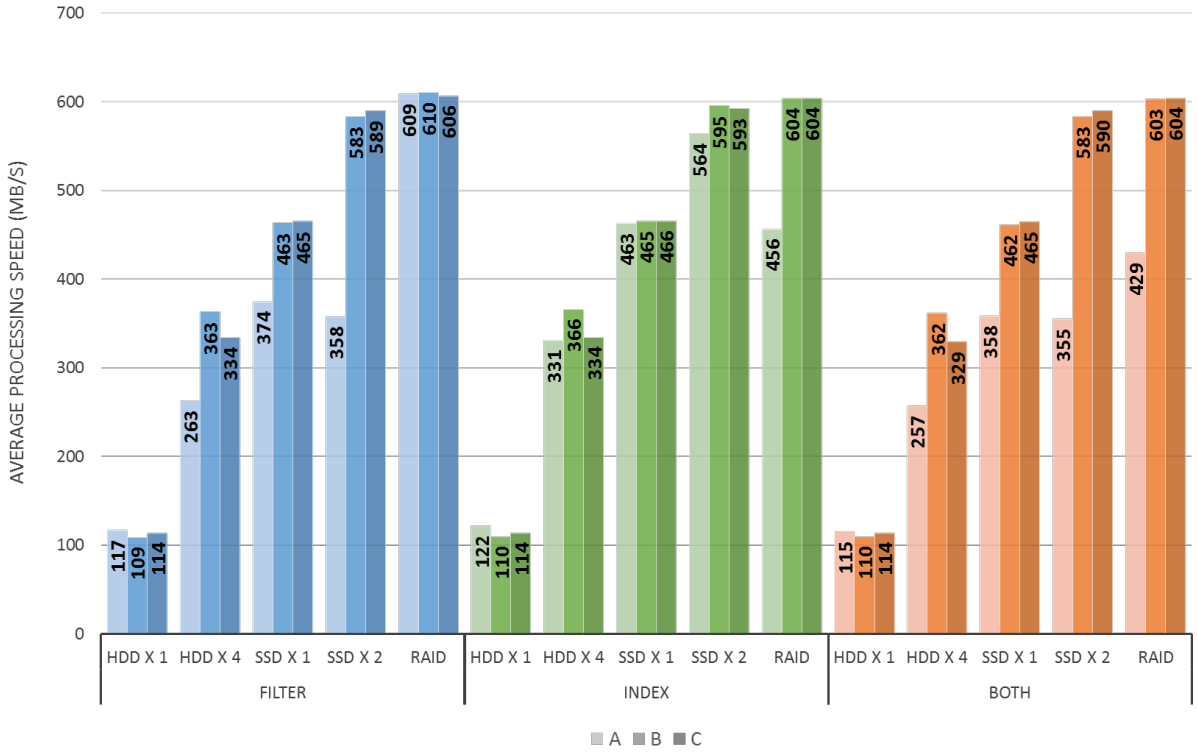


Figure 11.1: Average capture processing speed using GTX 750

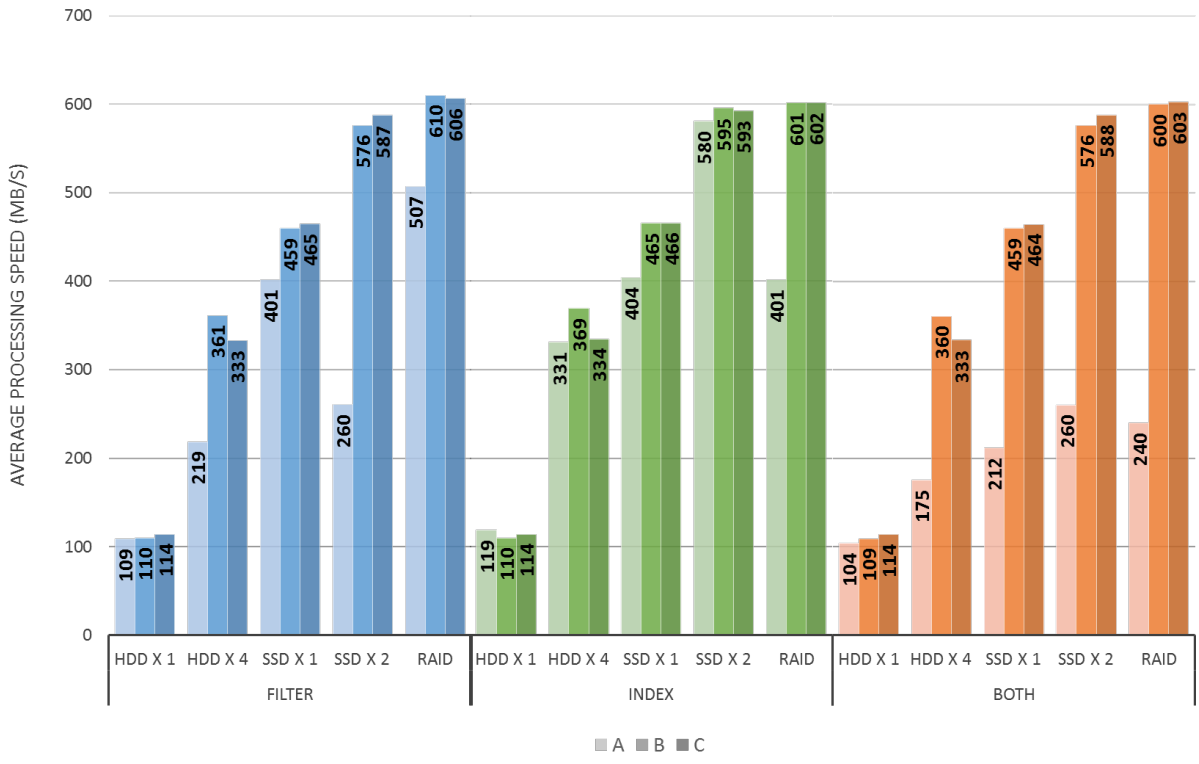


Figure 11.2: Average capture processing speed using GTX Titan

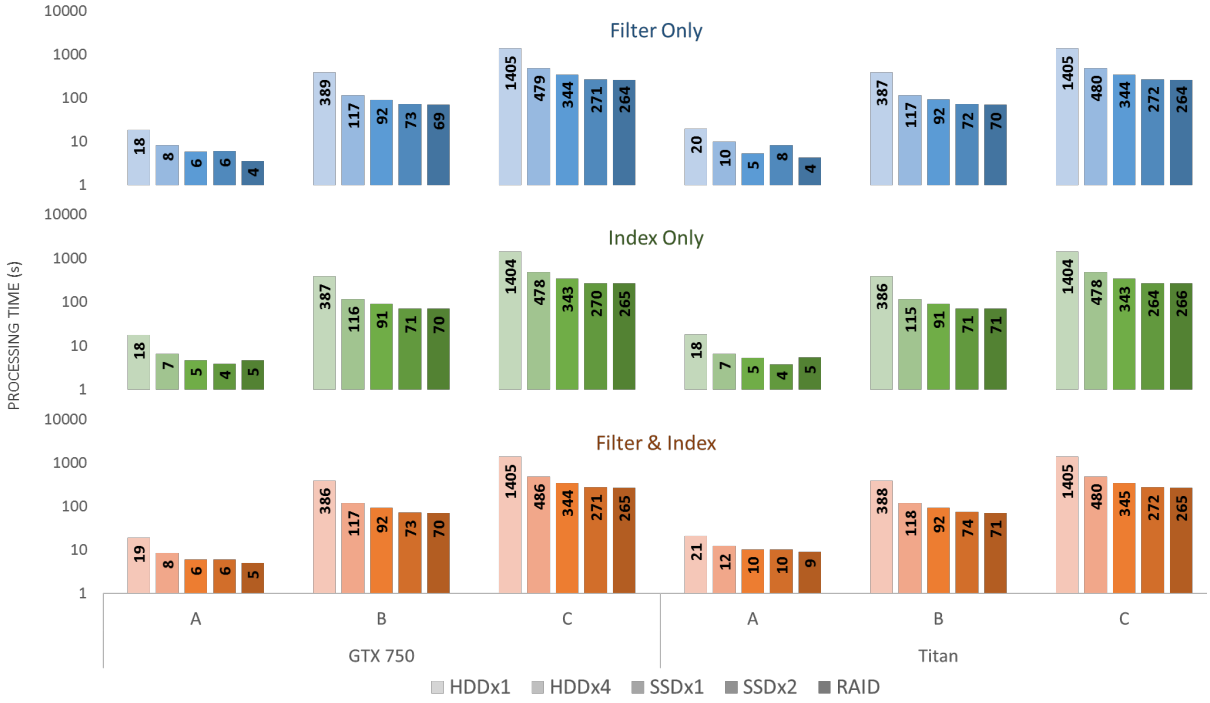


Figure 11.3: Total capture processing time, in seconds.

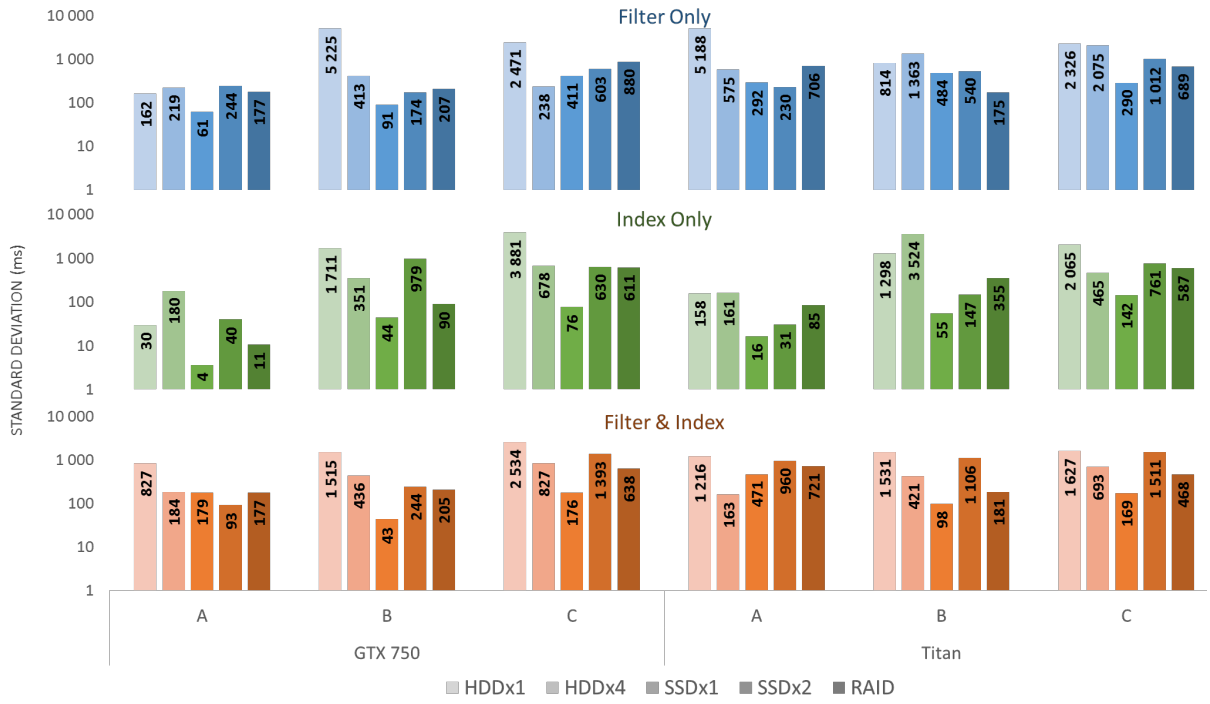


Figure 11.4: Standard deviation of capture processing speed, in milliseconds.

11.1.2 Execution configuration

Figures 11.1 and 11.2 show that system performance scales roughly in line with the performance of the input source, with only minor variations between capture inputs (discussed in Section 11.1.3). The process is relatively unaffected by other aspects of the configuration, such as the GPU used.

There was no significant difference between configurations using the GTX 750 and those using the GTX Titan, despite prior measurements showing that the latter outperforms the former by close to a factor of six (see Section 10.5). Furthermore, these results are comparable to index-only results which do not utilise the GPU at all. This indicates that GPU performance was not a bottleneck during execution, with even the comparably slow GTX 750 being capable of outperforming the fastest storage medium used.

Similarly, the combination of functions performed during processing (filtering, indexing or both) showed negligible difference in terms of achievable performance, with all variations showing speeds consistent with the storage medium used. Thus, like the GPU device, the combination of functions employed does not present a bottleneck to system performance on any of the storage mediums used.

11.1.3 Capture size

This section briefly evaluates how capture size affects performance. Results indicate that while captures B and C (which span 41 GB and 156 GB, respectively) perform consistently well in each test with only minor variations between them, capture A (which spans 2.1 GB) generally performs far worse. This is most noticeable in results which employ filtering, and seems to be exacerbated when using the GTX Titan. This is partially a product of setup time being a more significant component of processing time for small captures, and partially a result of the ratio of buffer allocations to buffer usage. Allocation of large memory segments is a somewhat expensive but once-off process, and is most noticeable when processing small captures with large buffers. As the GTX 750 uses smaller buffers than the GTX Titan, less memory is allocated during setup, which allows the GTX 750 to perform more efficiently in these edge cases. It is least noticeable during index processing, as indexing uses substantially smaller buffers that are less expensive

to allocate. This performance trade-off is in-line with expectations, as the design goals prioritised efficient processing of large captures over small captures.

11.1.4 File source

The recorded average performance correlates most significantly with the bandwidth of the file source, and indicates that achievable throughput is bottlenecked by file buffering in all tested configurations. The file mirroring approach performs surprisingly well overall, facilitating close to (and sometimes exceeding) RAID 0 performance when utilising identical SSDs, but without the need for similar drives or striped drive formatting. Captures are also natively mirrored and thus far more fault tolerant than a RAID 0 array, as the data is not lost if one drive fails. It also easily incorporates additional ad hoc mirrors to files from any source medium, which is not possible with a low-level striping approach. This is possible because capture files are always processed sequentially in their entirety; if access were random, RAID 0 would significantly outperform mirroring as mirroring would be unable to efficiently break down and interleave each random read operation.

While the mirroring approach is only an efficient option for simple file streaming, it is seemingly sufficient within this context, easier to setup and adjust than a true RAID 0 array, and substantially more flexible with respect to hardware. This comes at the expense of increased storage requirements, which scale by the number drives used (similar to a RAID 1 array). While this provides redundancy in case of drive failure, the associated cost of this redundancy when using many similar and expensive drives may compare less favourably to a true RAID 0, which can better utilise the available space.

11.1.5 Results stability

The standard deviation between iterations for each test shown in Figure 11.4 is relatively small, ranging between the extremes of tens of milliseconds to several seconds processing time, in contrast, ranged from 4 to 1,400 seconds (23 mins), depending on capture size (see Figure 11.3).

Standard deviation was typically highest when using a single HDD drive for filtering, and generally lowest when performing indexing from a single SSD. Standard

deviation does not correlate significantly with capture size or processing time, but rather seems to be primarily the product of operating system overhead and random drops in disk performance not directly related to the application. Instances where the standard deviation result is high typically correlate with result sets containing only one or two outliers. As a case in point, while the standard deviation for filtering capture A from a single HDD on the GTX Titan was over 5 seconds, eliminating a single outlier from the calculation – which reported a time twice as long as all other iterations – results in a standard deviation of only 97 milliseconds. Tests with higher standard deviations are therefore not indicative of high variability between all ten executions, but rather result from a small subset of outliers that, either due to buffer performance or resource availability, failed to perform at the same rate as other iterations.

Provided drive performance remains consistent, these results indicate that system throughput is quite stable and predictable.

11.2 Resource Utilisation

This section reports on system resource utilisation during capture processing. The classification process did not interfere with or slow other processes, and maintained a low, relatively consistent CPU load of roughly 2% ~ 3% across all tests. Memory utilisation and file size varied more widely, and are discussed in greater detail in the remainder of this section.

11.2.1 Memory Requirements

Figure 11.5 shows the achieved steady-state peak working memory utilisation for the system process (as measured by Windows Task Manager) for each of the nine filter programs listed in Section 9.5, as well as a control test where only indexing is performed. Measurements were taken when using 128 MB buffers and 256 MB CUDA packet buffers, correlating to the buffer sizes used in other tests by default for the GTX 750 and GTX Titan respectively. All tests used packet capture C, due to its comparatively large size, and were performed using four streams on the GTX Titan. This does not limit the value of results, as host side memory allocations are

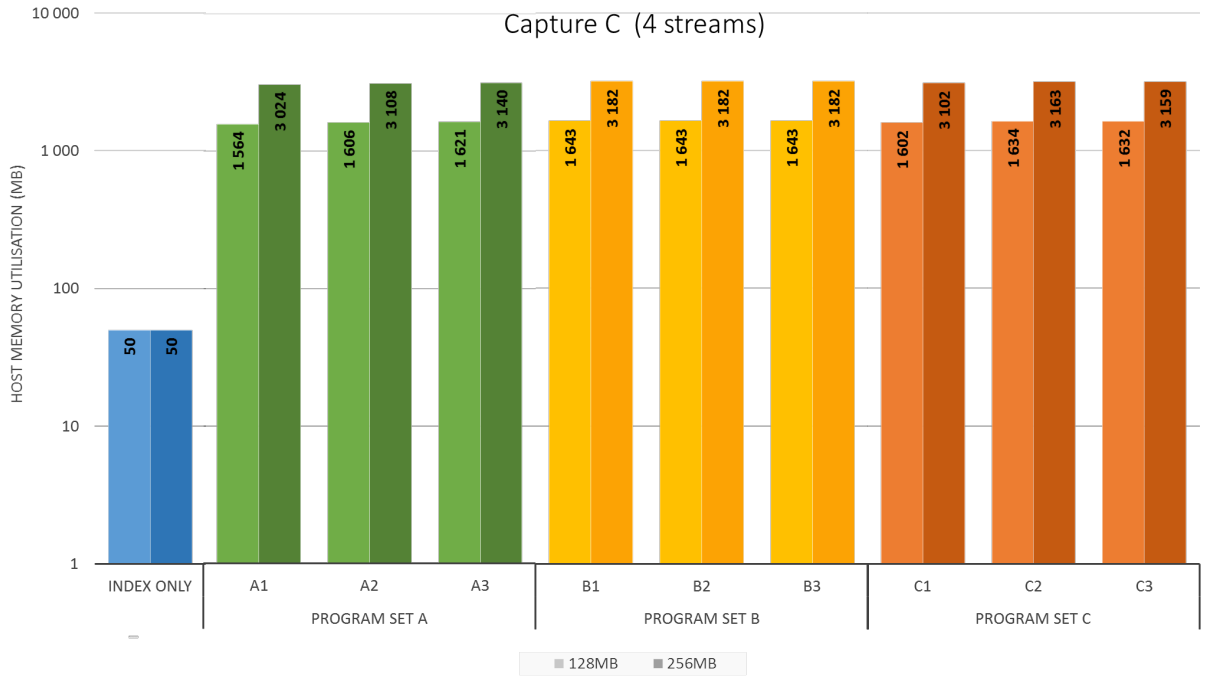


Figure 11.5: Peak working memory against buffer size for different configurations.

not determined by inspection of either capture size or target device, but by function, filter program, and combined CUDA buffer size (i.e. stream count $\times$ buffer size).

The results in Figure 11.5 confirm that memory utilisation is primarily determined by the specified buffer size, with very little variation evident between the simplest and most complex filter programs. This lack of significant variance is expected, as the system process dynamically scales the number of packets evaluated per stream to fit within the specified buffer size, using information in filter program to determine per-packet memory requirements. As the process uses four streams, with each stream triple buffered on the host, the process allocates memory slightly more than 12 times the specified buffer size (1536 MB for 128 MB buffers / 3072 MB for 256 MB buffers). It is thus possible to significantly reduce memory requirements by reducing either the buffer size, the number of streams, or the number of buffers per stream. The remainder of allocated space is comparatively minor, and is attributable to system and indexing overhead, as well as output buffers.

11.2.2 Storage Requirements

The system produces four output file types during the course of processing, which are summarised in Table 11.3. Figure 11.6 shows the uncompressed size of each of

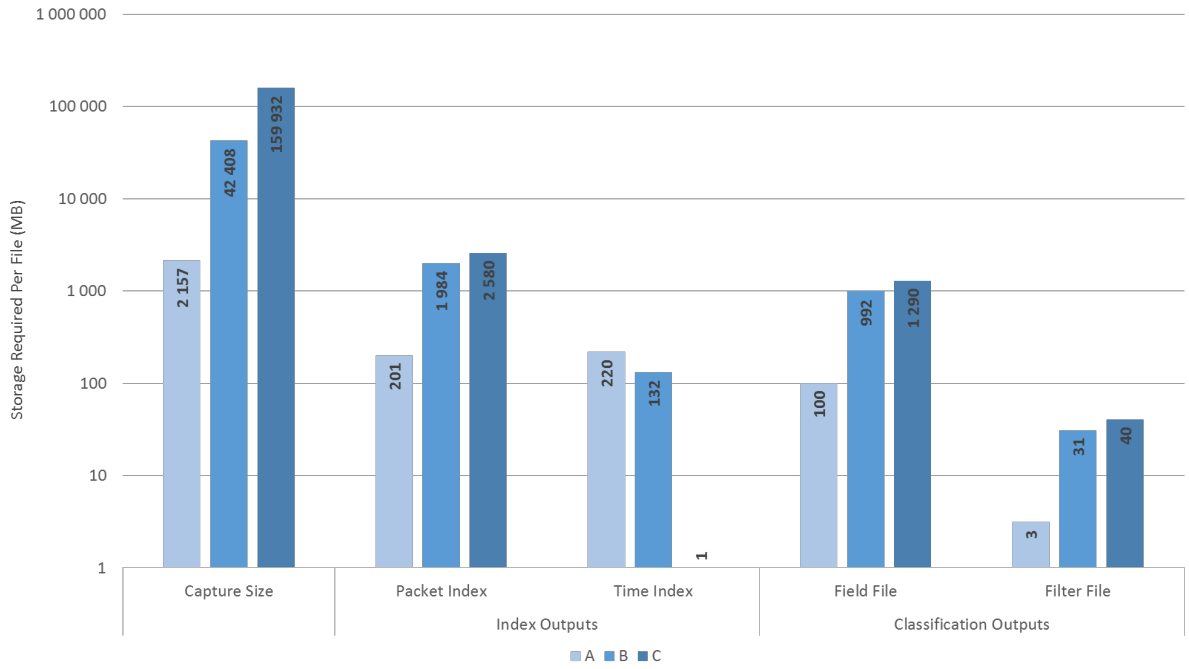


Figure 11.6: Output file size by source capture and type.

Output Files	Produced By	Scales With	Number Created
Packet Index	Indexer	Packet Count	1 per Capture
Time Index		Capture Duration	1 per Capture
Filter Results	Classifier	Packet Count	1 per Filter
Field Values		Packet Count	1 per Field

Table 11.3: Overview of output files.

these file types in comparison to the size of the capture that produced them. With the exception of the time index file, all output files scale linearly relative to packet count; the time index file instead scales linearly with capture duration, as detailed below. Indexing produces a single file of each type, while filtering creates a unique file per individual result.

The following list discusses each of the output file types in turn:

- The packet index file, consisting of 64-bit records, stores the offsets of 2^{17} (131,072) packets per MB. This is the most expensive single file in terms of storage space for the majority of captures, scaling inversely in comparative storage efficiency with respect to average packet size. In comparison to raw capture size, the packet index typically consumes one to two orders of magnitude less space.

- The time index file stores a single 64-bit index file offset per recorded second for a given capture file, and thus requires just over 600 KB of storage per day, or 240 MB per year. Capture A's extremely low packet arrival rate and long duration thus requires a significantly larger time index file (220 MB) than that of capture C, which spans a few hours despite its size, and thus requires only 200 KB of storage.
- Field files are comprised of 32-bit records, and thus achieve twice the packet density per MB compared to the packet index file. As a separate field file is required for each extracted field however, overall disk space utilisation can become dominated by field files if large numbers of fields are returned by a program. Indeed, due to the size of these files, program C3's outputs could not be contained within the limited capacity of the RAM disk used in system testing; the field files generated by this program when processing capture C required nearly 13 GB of storage and were responsible for over 80% of the total space required for all files generated by the program. In contrast, program C2 produced 7,982 MB of output data when processing capture C, which just fit within the maximum capacity supported by the RAM disk (8 GB).
- Filter files consist of single bit records, requiring 32 times less storage space than field files generated from the same capture – 2^{23} (8,388,608) packets per MB specifically. Filter files are highly efficient with respect to storage space in comparison to other file types; even in the worst case, filter files were well over three orders of magnitude smaller than the original capture file.

11.2.3 Performance Comparison

This section provides a simple comparison between the system and Wireshark in order to afford some context for collected performance results.

The comparisons are made on the basis of metrics relating to the processing time per capture, achieved data rate, and peak memory utilisation. The measurements taken relating to Wireshark reflect the time taken and memory utilised while opening each capture. These are compared to the measurements collected for the system process while processing program C2.

Captures B and C are far too large to be contained within host memory, and could only be partially processed within the Wireshark GUI. As a result, metrics for

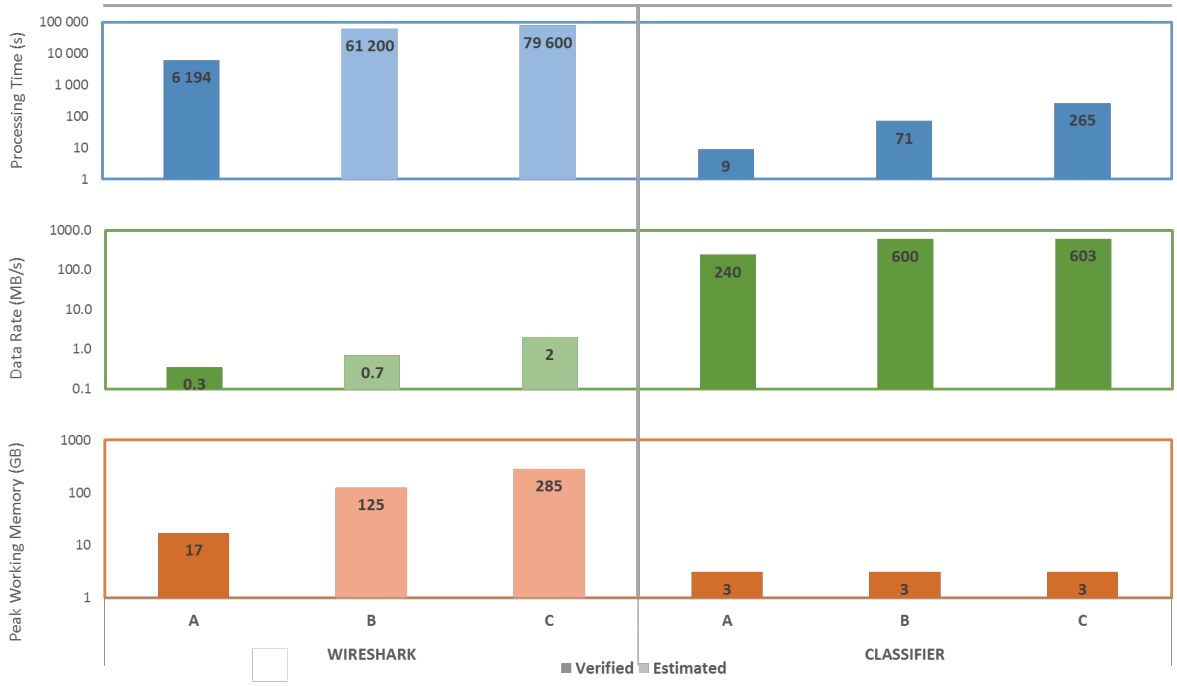


Figure 11.7: Comparison between Wireshark and system performance, showing processing time, data rate and peak working memory utilisation by capture file.

the Wireshark GUI relating to these captures were extrapolated from partial results, and should therefore be treated as rough estimations only. The comparative processing times, data rate, and peak working memory utilisation metrics when operating on captures stored on a striped SATA III SSD array are shown in Figure 11.7.

The results show that the classification system performed between 300 (capture C) and 900 (capture B) times faster than Wireshark estimates, and just under 700 times faster than the only verified time (capture A). In addition, memory utilisation was substantially lower than Wireshark in all tests, and did not scale with respect to capture size.

11.3 Post-processor Performance

This section investigates the performance of the post-processor functions described in Chapter 8. These are ancillary functions that apply the outputs generated by the classification process, and have been included primarily as working proof-of-

concept applications of classifier results. These results are included to show how generated system outputs perform when used as inputs to external ad hoc systems.

11.3.1 Filter Result Counting

The filter result counter is a simple GPU accelerated function provided by the C++ server process, which sums the number of '1' bits in a filter results string in order to determine how many packets passed a particular filter in a given results segment. The function is used to generate metrics and to visualise filter results (see Section 8.2.3.3). This section shows the performance of the CUDA function in isolation, as timed by the performance analysis functionality in Nvidia Nsight 4.2.

The performance of the function was measured while counting synthetic arrays of random binary data. The amount of data processed was incremented from 2 KB (16,384 packets) to 512 MB (2^{32} packets) by powers of two; each data volume configuration was subsequently evaluated on both the GTX Titan and GTX 750 graphics cards listed in 9.2. Results were verified on the host to ensure accuracy. Figure 11.8 shows the time taken in milliseconds, and the resultant count rates achieved in billions of filter results per second, for each configuration; the figure uses a logarithmic axis for both graphs.

The counting mechanism demonstrates an extremely high throughput, ranging from around five billion filter results per second for 16,384 packets (2 KB) to 700 billion and over for sets equal to or larger than 33.5 million results (4 MB) on the GTX Titan. The GTX 750 outperforms the GTX Titan for smaller sets equal to or less than 32 KB (262,166 records), but gradually scales down to slightly under one third of the GTX Titan's performance for segments of 4 MB or larger. Given that the GTX Titan has over five times as many CUDA cores as the GTX 750 however, this indicates significantly better overall efficiency and performance on Maxwell architecture. This observation is supported by captured performance metrics, which show significantly lower serialisation and higher average instructions per warp when using the GTX 750.

The counting throughput achieved is substantial, taking just 5.5 ms and 18.5 ms to count 512 MB of results on the GTX Titan and GTX 750, respectively. For context, a complete set of filter results from capture C only spans around 40 MB, and takes around 500 μs and 2.3 ms to process on these respective cards. Put differently,

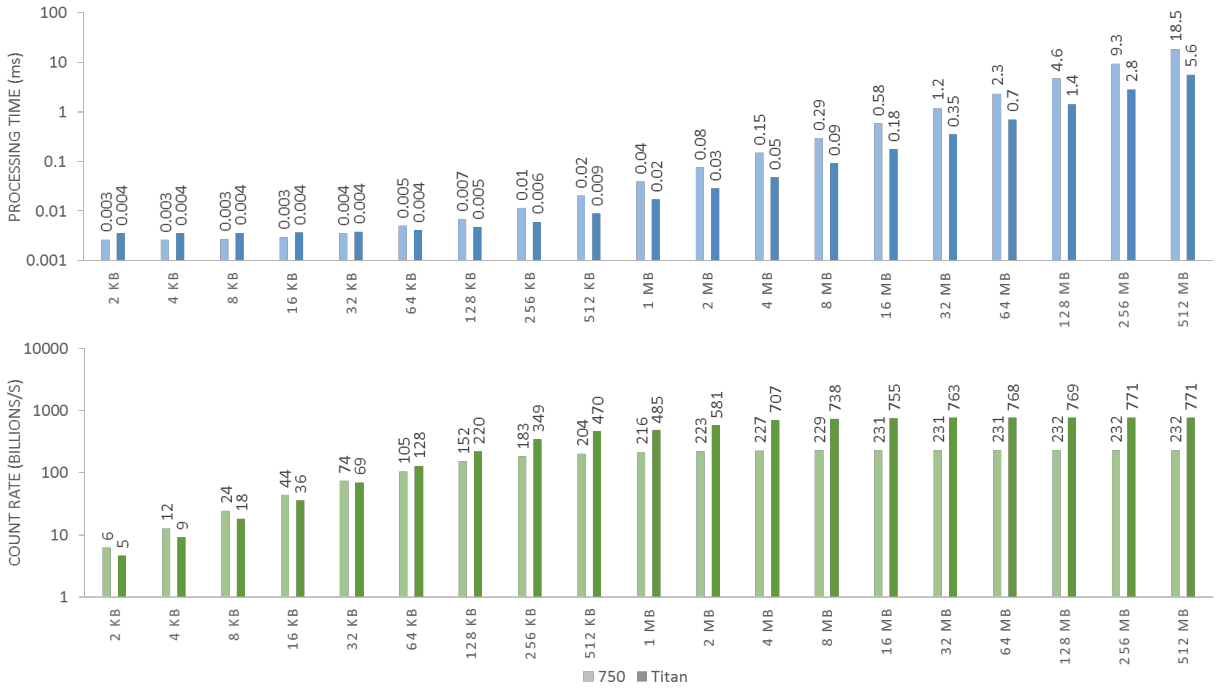


Figure 11.8: Achieved packet count rate in billions of packets per second by filter data processed.

if a capture of similar density per MB to capture C produced 512 MB worth of results for a single filter, the capture would approximate to 2 TB of packet data, and relative throughput of the counting mechanism with respect to the capture would exceed 360 TB/s.

The high efficiency of this function, in conjunction with the small size of filter files, renders filter counting virtually instantaneous from a user perspective for all captures containable on contemporary commodity storage mediums. This is useful for capture analysis by accelerating the process of calculating arbitrary capture metrics, based on the results of filtering. This function is used as key component in capture and filter visualisation discussed in the next section.

11.3.2 Capture Graph Construction

This section reports the recorded performance of the C# based capture visualiser (see Section 8.2), focussing specifically on the time required to construct and populate the internal tree data structure, which generates and organises the vertex data of each included graph using the files generated by the classification system. Once

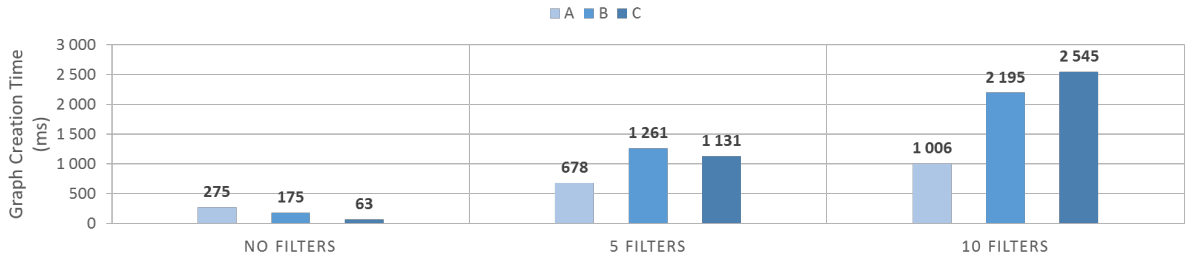


Figure 11.9: Time in milliseconds to construct and populate the capture graphs internal data structure, by filter count and capture.

constructed, graphs are easily and rapidly rendered directly from system memory using OpenGL shader programs. This function includes all necessary memory allocation, index and filter file access, and GPU accelerated filter counting performed within the visualiser.

Figures 11.9 and 11.10 show, respectively, graphs of the time taken in milliseconds to construct the internal data structure, and the total cumulative system memory used by the C# Visualiser and the external C++ Server (which provides counting functionality through a TCP socket connection). Results are shown for all three captures listed in Section 9.4, using three filter configurations: no filters, five filters (program A2) and ten filters (program A3).

As shown in Figure 11.9, graph construction time varied between tens of milliseconds to roughly two and a half seconds, depending on the size of the time index file and the number of filters processed. When no filters were present, processing time and memory utilisation was loosely proportional to the capture time span; capture A, which has the longest duration but least number of records, requires significantly more processing time and memory than capture C, which spans a far shorter interval but contains nearly 13 times as many records. This is a result of holding the time index file permanently in system memory for fast access; as the size of the time index scales with duration at a rate of around 240 MB per year (see Section 11.2.2), there is little concern of over-utilisation of system memory, even for decade long captures. As the filter counts are increased however, the process is forced to parse and evaluate more filter data, gradually slowing the processing of captures relative to filter file size.

While visualiser memory utilisation follows a similar trend, it does not expand proportionally to processing time, as processed filter counts require only a single integer of storage per render unit, and this unit varies depending on capture length.

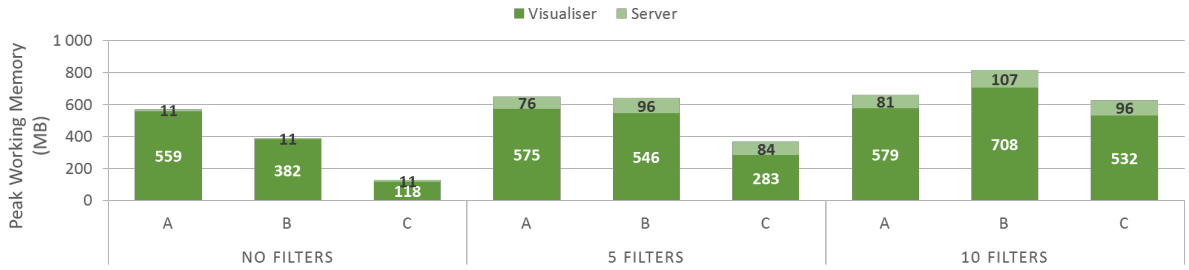


Figure 11.10: Peak working memory utilisation for the visualisation process and server process during graph construction, by filter count and capture.

As filter files are quite compact, they are read and temporarily held within system memory during the construction process to accelerate filter counting. This increases the peak memory utilisation of the visualiser, in relative proportion, to both the number of filter files processed and individual file size. Once the construction process completes, memory utilisation relaxes to levels comparable to unfiltered results. The server used significantly less memory than the visualiser client, consuming around 11 MB when left unused, and reaching a peak of 107 MB when processing 10 filters from capture B. The memory utilisation of this process is primarily governed by the buffer size used, which is determined at runtime by analysing the time index file.

While the visualiser does not perform at rates comparable to the GPU accelerated counting kernel it utilises, its processing time and memory utilisation are negligible in comparison to the resources required by Wireshark over the same capture data, and are significantly lower than those required by the classification system. Thus, once the processing of a capture has produced the required index and filter files, the generated solution can be reloaded in a matter of seconds, regardless of its original size.

This post-processing need not be restricted to counting alone; relatively simple GPGPU programs can easily accelerate other operations on filter bit-strings, such as conjunctions, disjunctions and negations. This functionality is already contained within the classification loop (see Section 6.8), and a stand-alone version would be simple to extrapolate. This would allow for fast post-classification mechanisms to combine filter outputs and rapidly create increasingly complex predicates from exiting filter results, without needing to re-parse the capture file.

Figure 11.11: Field distribution processing metrics

11.3.3 Field Distributions

The generation of field distributions is the simplest post-processing function implemented. It simply iterates through each stored field value in a specified field file, using the field values to build a dictionary object comprised of key-value pairs; each unique field value is used as a key, with the associated value indicating the number of times that it occurred in the file. Once the function has completed and the dictionary is built, results are displayed in a bar graph drawn using the standard Microsoft .Net draw functionality. The performance of this process is shown for field files collected from each of the three test captures in Figure 11.11. The Figure shows four metrics:

Processing time The number of seconds taken to read and process a single field file from a particular capture.

Field processing rate The average number of fields, in millions, processed each second.

Field file read performance The average read throughput achieved for each file.

Capture relative performance The effective throughput achieved relative to original raw capture size.

The calculation of a particular field distribution is relatively slow in comparison with either previous GPU accelerated functions or the classification process itself, managing to process field files at a rate of roughly 60 MB/s. Processing time compares favourably, however, to handling raw captures directly, with the achieved relative throughput scaling linearly with the average packet size of a given capture. Capture B, which has an average packet size close to double that of capture

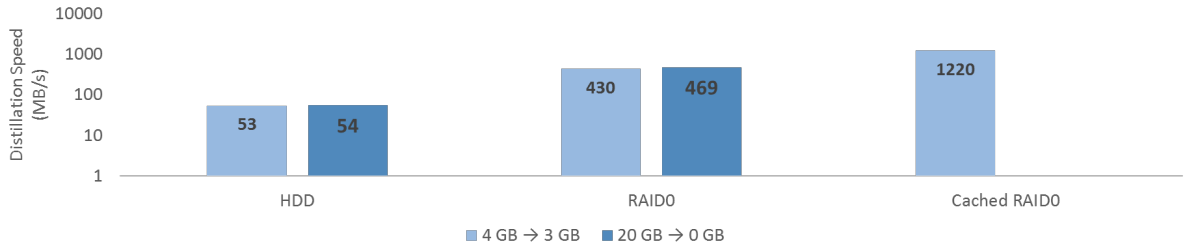


Figure 11.12: Distiller performance for two different filtering operations.

A, achieves a relative throughput roughly twice that achieved while processing capture A. Similarly, capture C, containing packets averaging three times the size of capture B, achieves a throughput roughly three times greater than observed while processing capture B.

The performance achieved relative to capture size is promising, given the simplicity of the implementation and the limited computational resources utilised. Specifically, it shows that the outputs of the classification process can be used to dramatically accelerate analytical operations in generic and sequential user applications.

11.3.4 Distillation

The final function evaluated was the distillation process, which extracts and repackages packets from a specific time interval of a large capture into a smaller capture file, optionally filtering packets against a supplied set of filter results.

The function buffers all packet data between the supplied time indices from a single source, performs filtering operations in a separate thread once data is buffered in host memory, and typically only writes a subset of parsed input to a file on a separate drive. The use of a different output drive ensures that the read and write processes do not compete for disk resources, providing better asynchronous performance.

Figure 11.12 shows the observed average performance for the distillation procedure when processing two separate hour-long segments of capture C. The first segment contained 4 GB of traffic, and was filtered for TCP traffic to produce an output capture spanning just under 3 GB. The second segment contained just under 20 GB of data, and was filtered against a null filter which effectively drops all packets. Both segments were processed three times from HDD and SSD RAID0 storage

using a cold cache, while the 4 GB segment was additionally processed three times using a warm cache (the second segment was too large to be cached).

While the distiller is not quite as efficient as the classifier (due primarily to its implementation simplicity), it still achieves promising results. Performance scales relative to the throughput of the input medium, although with some loss in efficiency. This is most pronounced when reading from the HDD source (around 50% efficiency), but improves appreciably when using the RAID 0 array (70% ~ 80% efficiency). When relying on a warm cache, the process scales to over 1.2 GB/s, showing that the performance achieved when using RAID 0 is not a hard limit, but rather a result of inefficiencies in the architecture. Filtering seems to provide a slight benefit to performance, likely because it reduces the amount of data that has to be written to disk. It is worth noting that the distiller's performance when using the RAID 0 array ($\pm$450 MB/s) is around 100 MB/s faster than pcap's capture reading process, which achieves a throughput of between 300 MB/s and 350 MB/s from the same file source (see Section 3.5).

Despite its simple implementation, the distillation process is both fast and powerful, providing access to arbitrary pre-filtered segments of large captures in any protocol analysis suite, often within a few seconds. As previously discussed in Section 9.3, verification of distilled captures was performed (distilled capture size permitting) with Wireshark, which confirmed that both the classification and distillation processes produced accurate results.

11.4 Summary

This chapter reported the recorded and derived performance results for the complete classification system, dividing results into three main sections.

Section 11.1 reported on system performance – inclusive of capture buffering, indexing operations, GPU classification and output generation – and focused primarily on observed processing time, and the associated achieved packet and data rates. Tests were performed using both the GTX 750 and GTX Titan graphics cards using a variety of input, filtering and processing configurations. Processing time was shown to display only minor variance between iterations (which was seemingly attributable to operating system overhead) and scaled to roughly match the read

speed of the input source, with negligible impact evident from other configuration attributes or capture file composition.

Section 11.2 investigated resource utilisation, focussing on host memory and disk storage specifically. Results showed that host memory utilisation achieved a stable constant value rapidly, and that this value scaled closely with the initial capture buffer configuration rather than capture size. This provides evidence that the system will scale to extremely large captures, far exceeding the capacities tested, without depleting system resources. The section also discussed the storage requirements of output files, showing that they consume a fraction of the capacity of a raw capture.

Section 11.3 concluded the chapter by discussing the three post-processing functions developed to investigate the usefulness of the system in real-world environments. These functions performed well given their implementation simplicity. The visualiser was able to render 10 separate filter graphs, covering over 150 GB of capture data, in only a few seconds, with minimal memory requirements. This was made possible by the filter counting kernel, which was shown to scale to hundreds of billions of packets per second, provided sufficient input data. The simple field distribution function performed similarly well, summarising the field of a large capture in a fraction of the time required when parsing directly from raw packet data. Finally, the distillation function quickly extracted and filtered arbitrary capture segments, producing small targeted captures at high speeds.

The following part discusses the conclusions of this research, and addresses potential avenues for extension.

Part V

Conclusion

12

Conclusion

THIS thesis has reported on the results of research outlined in Chapter 1, with the aim of improving and extending the speed and versatility of general packet classification through the use of GPU devices. The approach is based on previously conducted exploratory research into the domain [66], which produced a limited prototype for general classification. The implementation of GPF+ incorporates dynamic state registers, a more sophisticated cache, and an extended and refined processing abstraction, which together facilitate greater flexibility during classification, improved scalability to larger filter sets, increased processing efficiency, and enhanced usability through a refined abstraction. The revised and more extensive approach was employed within a multi-threaded pipeline of specialised components in order to greatly accelerate the analysis of large captures, supported by a small collection of post-processors which render the results produced by the process both usable and useful.

This chapter provides a conclusion to the research presented, containing a summary of the preceding chapters and and overview of outcomes achieved. The chapter is divided into the following sections:

- Section 12.1 provides a summary of the topics discussed in previous chapters of this document.
- Section 12.2 summarises the features and characteristics of the components produced as outcomes of this research.
- Section 12.3 discusses the research outcomes in relation to research goals, and considers some of their potential applications.
- Section 12.4 concludes the chapter with the discussion of avenues for further research beyond the scope of the present work.

12.1 Research Summary

This research was undertaken to construct a flexible, scalable and efficient means of general packet classification optimised for GPU hardware. This approach was applied specifically to the problem of analysing large packet traces, achieved through the construction of supporting host-side capture processing infrastructure and post-processing functions. The research was introduced in Part I, which summarised the research goals, methodology and outputs.

Part II established the foundations for the research conducted, providing necessary context to the reader for the design and implementation discussed in Part III. Chapter 2 served as an introduction to the GPU platform, with a particular focus on the CUDA programming model and API. This chapter familiarised the reader with the GPGPU domain and CUDA API, providing the necessary background on GPU accelerated processing. Similarly, Chapter 3 provides context for packet capture files and packet records. In particular, this chapter familiarises the reader with packets, protocol models, capture files, and capture file processing. Finally, Chapter 4 discussed packet classification in detail, describing filtering and routing approaches to packet header classification, as well as the GPF prototype [66], which incorporated filtering and routing elements to perform basic general classification on GPU hardware.

Part III described the abstract architecture and implementation of the various system components used to facilitate classification and apply results. Chapter 5 first introduced the classification system, and elaborated on the context and functions

of its constituent abstract components. This chapter also described the system's main host side architecture, including buffering and pre-processing components. The design and implementation of the GPF+ classification object was considered in Chapter 6. The chapter discussed the various memory regions (constant, state and global) and functions (caching, gathering and filtering) which cooperate to efficiently classify and extract field values from raw packet data. This process is guided by programs and configuration compiled from a high-level DSL, which was described separately in Chapter 7. This chapter introduced the two components of DSL programs (the protocol library and kernel function) and their syntax, and explained how they are transformed into optimised low-level programs. Having discussed the primary components of the classification system, Chapter 8 concluded the part by discussing three different post-processing applications, including: a simple function to compute field distributions; a capture visualiser that quickly renders interactive, high-level overviews of capture files; and a capture distiller, which generates pre-filtered sub-captures using index and filter data.

Part IV evaluated the GPF+ classifier and its supporting system, discussed in the previous part, through broad performance testing. Chapter 9 provided an overview of evaluation, describing the testing methodology, configuration, and verification method, and additionally summarised the various capture and filter programs used. Chapter 10 discussed performance results of the classification kernel in isolation from host-side processes. The classifier was tested against a range of different programs to assess the performance of the device side classifier. Chapter 11 concluded evaluation by considering overall system performance from the perspective of the user. This chapter investigated the throughput and resource utilisation of the main classification loop (comprising the capture buffer, pre-processor and classifier) and post-processing functions, demonstrating the efficiency, scalability and throughput of the system within a real-world context.

12.2 Research Outcomes

This section discusses the primary outputs of this research. The GPF+ classifier is discussed first, followed by the capture processing system and post-processing components.

The GPF+ classifier heavily refined the initially proposed architecture of GPF [66, 71, 72, 73], and adapted it to modern GPU architecture. GPF+ introduced a

more sophisticated caching mechanism, completely replaced the interpretive read process with a more flexible and scalable gather process, and optimised the filter process to improve filtering efficiency by a factor of 32. GPF+ achieved this by incorporating a compact state machine into each executing thread, which tracks protocol offsets, efficiently prunes redundant protocols, and evaluates simple length expressions to improve classification flexibility.

Performance evaluation of a variety of filter sets showed the approach executed efficiently on both Kepler and Maxwell devices, achieving high occupancy and multiprocessor utilisation with only limited serialisation. Throughput was shown to scale sub-linearly with respect to large filter and field sets with comparable protocol header counts, and near-linearly with respect to the number of cores available on the GPU device. Throughputs on the GTX Titan ranged from just under 30 million packets per second in the worst case to nearly 250 million packets/s in the best case. Filter programs using two layers achieved throughputs slightly over 100 - 130 million packets per second, while filter programs with three layers achieved 30 - 50 million packets per second (see Section 10.5).

The following list provides an overview of some of the more significant features of the classification approach developed during the course of this research:

- The layering abstraction and protocol encapsulation proved an effective replacement for decision trees on GPU hardware, supporting the flexibility to prune unnecessary protocols at runtime whilst avoiding heavy divergence in executing warps (see Section 6.1.2). Runtime pruning ensures filters are only ever processed in warps where they are relevant, providing greater execution efficiency and improved scalability when processing large and diverse filter sets (see Section 6.6). Additionally, layering allowed valid protocols in a warp to cluster reads to reduce memory overhead (see Section 6.5.4), and eliminated the need for protocol duplication during processing when header offsets differed. This was supported by performance results, which showed only minor drops in performance due to filter and field complexity.
- The classifier relies on complex and highly contextually sensitive low-level programs that are too complex to be reliably coded by hand. Filter creation is simplified greatly by the incorporated high-level DSL, which facilitates reuse of protocol definitions by separating protocol structure from filter operations (see Section 7.2). Protocol structures are described once, and subsequently

referenced to define any number of filters and field extractions in the kernel. Unreferenced fields and protocols (as well as redundant comparisons and switches) are automatically removed at compile time (see Section 7.3). This greatly benefits the usability of the classifier, allowing optimised and highly parallel programs to be constructed from simple high-level descriptions.

- The classifier is implemented as a relatively self-contained device-side object that can be declared and used within any arbitrary kernel, provided the classifier's constant memory space is populated appropriately before use. While the object uses up to 27 registers while executing (see Section 10.1.1), it consumes only eight persistent registers to store state and cache memory when idle, with the rest existing only within the scope of specific methods. The implementation does not employ any explicit synchronisation (see Section 6.1.3), which allows for complete execution independence between warps in an executing block. As a result, the object could be declared and used within a kernel that also processes its outputs – either directly, or through dynamic parallelism (see Section 2.8.5).

The remainder of the research focussed on applying this algorithm to the acceleration of capture analysis. The classifier was used in a multi-threaded pipeline of components to buffer, index and classify packet data (see Section 5.2). A collection of post-processors, which applied outputs of this pipeline to analyse large traces, was also included. This system of components significantly accelerated large capture analysis through filtering, visualisation and distillation, demonstrating the usefulness of the classifier and its outputs in this domain. The following list discusses the research outcomes associated with this secondary objective.

- The capture processing pipeline was highly successful, achieving a stable throughput of roughly 600 MB/s from a software RAID of SATA III SSDs, even under high load (see Section 11.1). Throughput was ultimately limited by the storage medium, and may continue to scale if faster mediums are used. Notably, the pipeline achieved throughputs equivalent to indexing operations when classifying on both the GTX 750 and GTX Titan (see Section 11.1.4), despite classifier testing showing the GTX Titan outperformed the GTX 750 by close to a factor of 6 (see Section 10.5). This indicates that GPU performance was not a bottleneck during execution, with even entry level cards being capable of outperforming two SSDs in RAID 0.

- Despite its simplicity, the mirrored file reading mechanism was as effective as a RAID at accelerating capture buffering when using the same drive pair (see Section 11.1). This makes it useful mechanism when a RAID array is unavailable. Resource utilisation remained constant for all captures tested (see Section 11.2.1), indicating that the process can scale to any capture size.

- Outputs are stored as arrays of integers or as bit-strings, which are easy to process in external contexts and adapt to arbitrary functions. These outputs were applied successfully to field distribution calculations (see Section 8.1), capture visualisation (see Section 8.2), and capture distillation (see Section 8.3). As index, field and filter information is well ordered and condensed (see Sections 5.1, 6.4.3 and 11.2.2), output files can be read into memory and processed far more quickly than a raw capture file (see Section 11.3). Outputs are currently uncompressed, however, and would benefit from space efficient encoding, such as those explored in capture indexing approaches (see Section 3.3.5).

- Post-processors were used to quickly analyse protocol composition and display the traffic dynamics for hundreds of millions of packets in seconds (see Section 11.3.1). In addition, the distillation post-processor extracted pre-filtered sub-captures from traces otherwise too large to process (see Section 11.3.4). This makes large packet captures more accessible to analysis, as they can be fragmented and pre-filtered into chunks small enough to easily process in applications such as Wireshark.

The GPF+ algorithm significantly extends and improves the comparatively simple architecture of GPF, providing greater flexibility, efficiency, scalability and usability than its prototype counterpart. Accessing packet data appears to be the most expensive aspect of the process, although this should improve as Nvidia micro-architectures evolve. The classifier and its supporting components have been shown to greatly accelerate the analysis of large capture files, producing non-volatile results that are easy to process in external contexts, without heavy memory utilisation or processor load. This provides a solid foundation on which to expand and optimise the approach in the future (see Section 12.4).

12.3 Conclusions

The overarching goal of this research was to provide a solid framework for general protocol analysis through experimental implementation. The research aimed to improve upon and extend prior research done by the author into GPU accelerated general classification [66, 71, 73, 74], and apply it to the problem of capture analysis for the purposes of practical evaluation. This proved highly successful, with the packet classifier, capture processing system and results post-processors meeting or exceeding initial research goals (see Section 1.3.3).

This section discusses the various components and potential applications of this research, within the context of the research goals originally set.

12.3.1 Extending GPF functionality

The GPF+ classifier was designed to extend protocol independent filter queries and field extraction to a GPU context, maintaining the generality of the initial approach to support a wide array of potential applications, while providing added flexibility, scalability, efficiency and usability. The GPF+ classifier heavily extends and re-works the algorithm developed in prior work [66], taking advantage of the increased flexibility and device-side resources of modern GPUs.

- The classification approach developed provides sufficient flexibility to support variable length fields and handle optional fields, greatly improve the coverage of the process. The approach provides support for a wider variety of protocol types, improves classification accuracy, and provides better handling for multiple classification paths. In addition, the state memory (see Section 6.3) and warp voting (see Section 6.5.4, 6.6.2 and 6.6.3) mechanisms provide an architectural foundation on which to extend functionality, through finer grained branching (to support optional fields), and greater access to state registers (to manipulate data offsets programmatically). The classifier also includes limited support for expression evaluation (see Section 6.7.3), which could be extended with relative ease to allow arbitrary computation (to generate metrics, etc.) during classification.
- The GPF+ classifier is implemented as a fully device-side object (see Section 6.1), rather than as a collection of independent but cooperative kernels, as was

the case with the earlier GPF prototype [66]. The classifier object occupies 8 registers (for state memory and the packet cache), uses no shared memory, and achieves close to full occupancy during execution (see Chapter 10). This makes it potentially possible to employ the classifier within an executing kernel, rather than as a separate pre-processor. For instance, a simple NIDS function could use the classifier to produce a set of field values and filter bitstrings in device memory, which it could then immediately process within the same kernel, or pass on to a subsequent kernel. This also applies to the generation of index files for capture indexing approaches (see Section 3.3.5); all outputs in device memory could be processed immediately, rather than relying on sequential host-side processes to pre-extract relevant fields [27]. This provides added flexibility in how the classifier can be applied, and benefits scalability, efficiency and usability as well.

- The classifier performs all aspects of general header classification, including protocol parsing, field comparisons and predicate evaluation, in parallel on the GPU. This allows the process to scale more easily to available GPU resources, as no host side interactions are necessary during processing. As long as the constant and global memory regions are populated before invocation (see Sections 6.2 and 6.4), the classifier could potentially be allocated, initialised and used within dynamic kernels on demand (see Section 2.8.5). This could allow a dynamic process to scale classifier resources to adjust to changing load, for example, by increasing or decreasing the number of separate executing instances.

- The high-level DSL significantly improves the usability of the programming interface and the scalability of filter programs. While the current implementation of the DSL is constrained to a very limited scope, it incorporates the basic foundations necessary to flesh out a more refined and feature-rich syntax. In particular, it generalises to arbitrary protocols, limits protocol replication, and manages protocol relationships transparently (see Section 7.2.2), greatly simplifying filter program creation. Programs are easy to optimise (see Section 7.3.2), which additionally helps to improve execution efficiency.

The GPF+ provides greater coverage than the original GPF, and improves on the flexibility, scalability, usability and efficiency of the approach in a number of different ways. As a result of its improved generality, it has many potential applications,

particularly within the domains of network security and protocol analysis, where routing algorithms are not always sufficient.

12.3.2 Accelerating Capture Processing

The second goal of this research was to accelerate capture classification through the development of a multi-threaded pipeline of high-performance components. The pipeline was designed with similar priorities to the classifier, but with stronger emphasis on scalability and efficiency in order to support arbitrarily large capture files.

- The multi-threaded reader allows captures to be read from multiple interfaces simultaneously (see Section 5.4.2), providing performance comparable to a RAID 0 array with ad hoc drives (see Section 11.1.4). This feature can significantly reduce capture processing time without requiring specially formatted, identical disks. This process could be extended to incorporate multi-GPU support, by dividing workers between GPUs. It could also be employed, in combination with the pre-processor (see Section 5.5), to interleave packets from multiple capture sources into a stream of packets ordered by time stamps. This could be used to quickly combine captures or process multiple captures simultaneously, which could be useful for generating metrics from multiple trace files.

- All host side components are connected via 0MQ (see Section 5.2.2), which facilitates asynchronous interaction. The use of 0MQ to provide the backbone of capture processing makes it easier to span these components out across multiple processes or remote hosts at a later point [33], with little change necessary to the architecture. This pipeline could be extended into a fully modular approach, by providing a standard interface to connect component's 0MQ sockets. For instance, an API front-end could be connected to the classifier to facilitate pcap-like functionality, allowing processes to pass buffers directly to the classifier and retrieve results, without relying on other host-side components. Alternatively, a post-processing component could replace the output writer, applying results immediately to a specialised task, rather than storing them for later use.

- The classification pipeline has relatively fixed memory requirements (see Section 11.2.1), and unlike Wireshark, does not scale memory utilisation with capture size (see Section 3.6.2). This allows extremely large captures to be processed without exhausting host memory resources, even on 32-bit machines. The process does not consume significant CPU resources either (see Section 11.1), as the majority of computation is performed on typically underutilised GPU devices. The process is thus resource efficient, and can feasibly scale to any capture size without exhausting system memory or slowing system performance. As the process is lightweight, it could conceivably be employed as a service which categorises inbound traffic for live traffic monitoring or logging purposes.
- The system produces persistent outputs that are simple to navigate and utilise, and can be reused to avoid repeated processing. Despite being uncompressed, these results are much smaller than the raw captures they refer to (see Section 11.2.2), and thus can be read from disk far more quickly than if accessed from raw packet data. Additional compression could compact them further for the purposes of either archiving or index based acceleration [26, 46]. As results are persistent, they may be shared for the purposes of research without exposing the contents of packet data.

The classification pipeline greatly accelerates capture filtering, saturating all storage interfaces tested while performing multiple classifications and field extractions in parallel. The process is resource efficient, scalable to captures of any size on even memory limited machines, and produces efficient, reusable outputs.

12.3.3 Supporting Protocol Analysis

The final goal of this research was to apply the classifier and capture processing system outputs to solve problems associated with protocol analysis, and to demonstrate the utility of generated results in a real-world context.

- The capture visualiser (see Section 8.2), while limited to a single graph type, rendered simple interactive overviews of hundreds of millions of packets (and billions of arbitrary filter results) in seconds (see Sections 11.3.1 and 11.3.2), for any arbitrary set of filters. This functionality could easily be extended

to provide a much wider variety of general and specialised graphs, potentially supported by a specific classifier program that extracts context-specific information. Alternatively, visualisation could be performed through third-party Business Interfaces (BIs) or dashboards, which provide a wide variety of visualisation and graphing options for incoming data. This could be applied within intrusion detection systems, network administration and monitoring applications, and network related research to quickly chart arbitrary protocol attributes within large packet sets.

- The ability to distil small, pre-filtered captures rapidly from the visualiser (see Section 8.3) greatly improves the accessibility of large packet traces. In addition, it directly applies system outputs (and multiple post-processors) to dramatically improve the usefulness of existing protocol analysis software when analysing large captures. The function allows the system to be used as a powerful pre-processor that crops and pre-filters captures to manageable sizes faster than they can be read by pcap (see Section 11.3.4), allowing the full functionality of Wireshark or other analysers to perform detailed analysis on only the remaining packets.

- The visualiser and field distribution post-processors work independently of packet data, and can be used without access to the original capture. This allows system outputs to be used to perform high-level analysis of large packet sets remotely, without requiring the complete raw data set. By carefully selecting the fields and filters that are included, captures can be mapped with fine grained control over what information is shared. This can be used to address the security and privacy concerns of sharing traces collected on live networks, which may contain significant volumes of private, protected and personally identifiable information [16]. Sharing captures in this way requires less storage space and processing time, and provides much greater control over what information in the capture is accessible to those processing it.

The implemented post-processors apply classification results to provide powerful functionality that both simplifies and accelerates protocol analysis, particularly with respect to large packet captures. More importantly, they demonstrate the versatility of results and the performance benefits they provide over repeatedly re-parsing captures.

12.4 Future Work

A primary goal of future work is to extend functionality of the GPU classifier in order to provide finer execution control and flexibility, as well as improved usability. The current implementation incorporates many new features, and lays the foundation for many more, but still lacks efficient handling for several important or otherwise potentially useful functions. Another area worth investigating and extending is the syntax and functionality of the DSL, which is currently feature-limited due to scope constraints. Many other areas are also of interest, as both the classifier and its supporting components present many opportunities for optimisation and extension.

A selection of desirable extensions that could be used to improve flexibility, usability and performance are summarised in the following list:

- User defined variables – Support for user defined variables could be used to supply temporary and reusable local storage. This could be used to hold accumulated metrics and packet data, share field values between protocols, and provide a means to extend field extraction in order to facilitate more complex and generalised computation. On current hardware, these variables could be housed in either global memory or shared memory (currently unused). Global memory would provide far greater storage capacity, while shared memory would provide higher throughputs. Shared memory is a limited resource, however, and avoiding its use allows it to be applied elsewhere. Global memory throughput is expected to improve dramatically in Pascal micro-architecture (see Section 2.2.5), making the performance trade-offs between these mediums unclear. Future research into this functionality will thus largely depend on the impact of future memory architecture on global memory performance.

- Driver encapsulation and live traffic monitoring– The current implementation relies on external software (specifically the WinPcap driver [103]) to manage the collection and storage of packet data from a specific network interface, which writes this data as a capture on a local hard drive. This is inefficient for live monitoring, as it relies on the unnecessary storage and retrieval of packet data on bandwidth-limited local drives. While the WinPcap driver could be used for managing a live interface, it is not optimised for the

intended GPU platform, and performs tasks that are not necessary (such as capturing timestamp information for every packet). To address this, the general pcap driver could be replaced with a specialised driver for GPF+ that captures and prunes packet records on arrival, and copies relevant portions directly into a write-combined page-locked memory buffer for dispatch to the GPU. In addition to filling processing buffers, the driver could additionally handle both packet and time indexing operations, recording the eventual offset positions of each arriving packet, and the time delta in which each packet falls. As there is no recorded capture to apply classification results or indexing information to, this information can instead be used for monitoring purposes.

- Decisions and looping – The system currently uses decisions (in the form of warp voting) to facilitate runtime pruning and protocol branching (see Section 6.6). These mechanisms could be extended to facilitate decisions and loops through `if` and `while` statements, useful in directly evaluating optional and variable length fields, respectively. Additionally, switch statements (currently dedicated and restricted to switching between protocols) could be expanded to support more general computation beyond simple `goto` statements (see Section 7.2.2). This functionality may prove useful when attempting to process complex protocols or payload data.

- System register read/write access – GPF+ provides direct read/write access (and expression evaluation support) to the length state variable, in order to handle variable length protocols (see Section 6.7.3). This could be extended to other state variables, such as protocol state registers and read offset registers (see Section 6.3). Combined with more general decision operations, access to system registers would afford much finer control of program execution, and could be used to better support optional fields.

- Large field support – The current implementation only directly supports fields up to four bytes in size (see Section 6.5.5), and requires larger fields to be split manually into multiple sub-fields, that are subsequently written to independent arrays in results memory (see Section 6.4.3). This could be improved by splitting large fields into sub-fields transparently within the DSL and providing additional functions to write multiple sub-fields to a single output array, in the correct order, on the device. This would provide proper support for large field values, such as 128-bit IPv6 addresses.

- Index-guided processing – The capture processing pipeline process captures

sequentially, as this is necessary to determine packet offsets in the capture file. As a result, buffers produced by mirrored file readers have to be interleaved into a sequential queue and processed in order (see Section 5.4.2). This sequential component is not actually necessary, provided index files have already been generated for the capture. Index files could be used to divide captures into several large segments, each read by a dedicated worker thread that writes to a specific execution stream (or GPU). Index files may additionally be used to navigate untrimmed packet records on the device. With the support of NVlink and unified memory (see Section 2.2.5) [25], packet data could be directly accessed from host memory through NVlink, avoiding the buffer copy to device memory. As with other potential functions, this depends heavily on the effectiveness and applicability of Pascal's improved memory architecture.

- Stateful inspection through multi-pass analysis – The current implementation has been constructed to evaluate all packet header layers in a single GPU kernel invocation. This process is suited to classification of independent packets, but lacks capabilities for higher level operations such as stream-based processing and stateful analysis. To expand the functionality of the platform to facilitate analytical operations over collections of packets, the existing header classification kernel can be used to generate initial results within a multi-pass pipeline of kernels. Results collected from this first pass (such as protocol composition, port numbers, IP addresses, fragmentation information, etc.) can be used in subsequent passes to compute metrics relating to specific streams, or to select appropriate packets for deep packet inspection or payload processing operations.

Additional research avenues include but are not limited to the following:

- A wrapper API to simplify the use of the pipeline and its components within external applications.
- Full support for multiple GPUs to allow the system to further accelerate processing, in order to scale to high-speed live interfaces.
- Expanded expression evaluation to allow for more diverse calculations within executing kernels, for use in computing arbitrary metrics and runtime statistics.

- Results compression to reduce the storage footprint of generated index and result files.
- Grammar extensions in both the DSL and GPF+ kernel to simplify and accelerate common filter related tasks, such as IP address masking.

12.5 Other Applications

The previous sections of this chapter have focussed exclusively on revisions and extensions to better support packet filtering. The classification methodology detailed in this work has potential applicability in other domains, however, and this section aims to highlight some of these potential applications.

- Network security applications – The GPF+ classifier may be applied to support/supplement network security applications, including firewalls and IDSs. The developed approach is better suited to batch processing tasks that do not require low latency processing, as GPU-based classification relies on the simultaneous processing of large numbers of packet records. This is partially mitigated by utilising multiple separate execution streams to subdivide traffic, but cannot be avoided entirely. While this dilutes its usefulness as a real-time classifier due to inflated latency on incoming packets, it may still be useful as a secondary processor for more rigorous evaluation of traffic.
- Packet processing for Big Data / Cloud environments – This research has focussed on desktop systems as a deployment environment for GPF+, and has not considered its use in large-scale distributed computation within a Big Data or cloud environment. GPF+ is well suited to this task, due to its ability to scale across multiple GPU-based processors such as Tesla servers with ease. As many cloud and Big Data platforms support GPU virtualisation, GPF+ could potentially be used to accelerate the transformation of packet data from raw, unstructured binary arrays to structured and semi-structured data points within a Big Data distribution (such as Hadoop [4]), or perform monitoring across virtual machines in the cloud using Nvidia GRID [88].
- Grid-based sensor network – GPF+ may be employed as a processor on a distributed grid platform such as BOINC (Berkeley Open Infrastructure for

Network Computing) [120], which could be useful as a tool similar to a network telescope for researching and analysing large scale networks or Internet traffic. For example, using a specialised GPF+ program, each node on the grid could compute individual metrics for the local host and/or nearby neighbours as a background process, and pass this reduced information periodically to a database or Big Data solution for detailed analysis. The generated metrics could be used to analyse long term traffic dynamics for large scale network traffic, collected from either local or geographically distributed hosts. Alternatively, it could be used to monitor the health of an internal network, or detect and trace malicious activity that occurs behind the boundary firewall.

- Generating inputs for Machine Learning – GPF+ outputs may be used as training data for supervised machine learning applications, providing a foundation from which an artificial intelligence may use generated meta-data, in combination with raw packet transmissions, to learn to identify packets and protocols, build a baseline for the behaviour of a particular network over time, or rapidly identify malicious transmissions. Depending on the input programs used, supervised learning may use results for broad-based categorisation, or for specialised domain specific tasks. Machine learning may be integrated into a Big Data platform or grid, providing very large and detailed data sets to train against.

- Packet processing for simulated and virtual networks – Virtual networks have applications in network and security related research, and use large collections of interconnected virtual machines to simulate or study network traffic and malware propagation. Through the virtualisation technologies included in Nvidia GRID 2.0, each virtual machine may use virtualised GPU resources to perform detailed categorisation, metric extraction, and traffic analysis on its virtual interface(s). As the GPF+ high level language provides the flexibility to process arbitrary protocols, the metrics captured can be specialised to particular tasks, protocols, or types of traffic.

- Log file and database processing – The GPF+ processor is specifically designed for packet analysis, but the techniques used could be generalised to other data sources that follow a structured or semi-structured schema. The process could, for instance, be adapted to process log files or relational and non-relational tables. The current implementation is only suitable for fixed length fields, or variable length fields with specified lengths; processing fields

which terminate on the occurrence of specific characters would require additional work.

12.6 Concluding Remarks

The components described in this research have proven highly successful in improving general packet classification on GPU hardware, accelerating large capture processing, and applying capture results to simplify and accelerate protocol analysis. In addition, these components form a solid foundation for future expansion and refinement, and provide ample opportunities for optimisation. It is expected that the performance of the approach will continue to improve as GPU micro-architectures mature, and current bottlenecks (such device memory access and transfer overhead) diminish in importance.

References

[1] **Advanced Micro Devices, Inc.** *AMD Accelerated Parallel Processing OpenCL Programming Guide rev 2.7*. Online, November 2013. Last Accessed: 29/06/2015.
URL `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf`

[2] **Aho, A. V. and Corasick, M. J.** *Efficient String Matching: An Aid to Bibliographic Search*. *Communications of the ACM*, 18(6):333–340, June 1975. ISSN 0001-0782. doi:10.1145/360825.360855.

[3] **Alcock, S., Lorier, P., and Nelson, R.** *Libtrace: A Packet Capture and Analysis Library*. *ACM SIGCOMM Computer Communication Review*, 42(2):42–48, March 2012. ISSN 0146-4833. doi:10.1145/2185376.2185382.

[4] **Apache Software Foundation**. *Hadoop*. Online, January 2016. Last accessed: 04/02/2016.
URL `http://hadoop.apache.org/`

[5] **Baboescu, F., Singh, S., and Varghese, G.** *Packet classification for core routers: is there an alternative to CAMs?* In *Annual Joint Conference of the IEEE Computer and Communications.*, volume 1 of *INFOCOM '03*, pages 53–63 vol.1. March 2003. ISSN 0743-166X. doi:10.1109/INFCOM.2003.1208658.

[6] **Baboescu, F. and Varghese, G.** *Scalable Packet Classification*. *ACM SIGCOMM Computer Communication Review*, 31(4):199–210, August 2001. ISSN 0146-4833. doi:10.1145/964723.383075.

[7] **Bailey, M., Gopal, B., Peterson, L. L., and Sarkar, P.** *Pathfinder: A pattern-based packet classifier*. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI '94, pages 115–123. 1994.
URL http://www.researchgate.net/publication/220851768_PathFinder_A_Pattern-Based_Packet_Classifier

[8] **Baxter, S.** *Segmented Reduction*. Online, 2013. Last Accessed: 23/06/2015.
URL https://nvlabs.github.io/moderngpu/segreduce.html

[9] **Begel, A., McCanne, S., and Graham, S. L.** *BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture*. *ACM SIGCOMM Computer Communication Review*, 29(4):123–134, August 1999. ISSN 0146-4833. doi:10.1145/316194.316214.

[10] **Black, P. E.** *trie*. Online, February 2011. Last accessed: 23/06/2015.
URL http://xlinux.nist.gov/dads/HTML/trie.html

[11] **Borgo, R. and Brodlie, K.** *State of The Art Report on GPU Visualisation*. Online, February 2009. Last accessed: 23/06/2015.
URL http://www.comp.leeds.ac.uk/viznet/reports/GPU_report/GPUSTARReport_html.html

[12] **Bos, H., de Bruijn, W., Cristea, M., Nguyen, T., and Portokalidis, G.** *FFPF: Fairly Fast Packet Filters*. In *Proceedings of the Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI '04, pages 24–24. USENIX Association, Berkeley, CA, USA, 2004.

[13] **Braden, R.** *Requirements for Internet Hosts – Communication Layers*. RFC 1122, Internet Engineering Task Force, October 1989.
URL https://www.ietf.org/rfc/rfc1122.txt

[14] **Buck, I.** *Brook Spec v0.2*. Online, October 2003. Last Accessed: 23/06/2015.
URL http://graphics.stanford.edu/papers/brookspec-v0.2/brookspec-v0.2.pdf

[15] **Buck, I.** *NVIDIA's Next-Gen Pascal GPU Architecture to Provide 10X Speedup for Deep Learning Apps*. Online, March 2015. Last Accessed: 26/05/2015.
URL http://blogs.nvidia.com/blog/2015/03/17/pascal/

[16] **CAIDA**. *Promotion of Data Sharing*. Online, May 2011. Last Accessed: 21/06/2015.
URL `http://www.caida.org/data/sharing/`

[17] **CAIDA**. *The UCSD Network Telescope*. Online, April 2015. Last Accessed: 23/06/2015.
URL `http://www.caida.org/projects/network_telescope/`

[18] **Catanzaro, B.** *Trove Readme*. Online, March 2013. Last Accessed: 28/06/2015.
URL `https://github.com/BryanCatanzaro/trove/blob/master/README.md`

[19] **Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A.** *RAID: High-performance, Reliable Secondary Storage*. *ACM Computing Surveys (CSUR)*, 26(2):145–185, June 1994. ISSN 0360-0300. doi: 10.1145/176979.176981.

[20] **Crucial**. *Crucial MX100 256GB SATA 6Gb/s 2.5" Internal SSD Product Information*. Online. Last Accessed: 23/06/2015.
URL `http://www.crucial.com/usa/en/ct256mx100ssd1#productDetails`

[21] **Decasper, D., Dittia, Z., Parulkar, G., and Plattner, B.** *Router Plugins: A Software Architecture for Next Generation Routers*. *ACM SIGCOMM Computer Communication Review*, 28(4):229–240, October 1998. ISSN 0146-4833. doi:10.1145/285243.285285.

[22] **Demouth, J.** *Shuffle: Tips and Tricks*. Online, March 2013. Last Accessed: 28/06/2015.
URL `http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf`

[23] **Engler, D. R. and Kaashoek, M. F.** *DPF: Fast, Flexible Message Demultiplexing Using Dynamic Code Generation*. *ACM SIGCOMM Computer Communication Review*, 26(4):53–59, August 1996. ISSN 0146-4833. doi: 10.1145/248157.248162.

[24] **Fielding, R., Adobe, Reschke, J., and Greenbytes**. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230, Internet En-

gineering Task Force, June 2014.
URL https://www.ietf.org/rfc/rfc7230.txt

[25] **Foley, D.** *Nvlink, pascal and stacked memory: Feeding the appetite for big data*. Online., March 2014.
URL http://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data/

[26] **Fusco, F., Dimitropoulos, X., Vlachos, M., and Deri, L.** *pcapIndex: An Index for Network Packet Traces with Legacy Compatibility*. *ACM SIGCOMM Computer Communication Review*, 42(1):47–53, January 2012. ISSN 0146-4833. doi:10.1145/2096149.2096156.

[27] **Fusco, F., Vlachos, M., Dimitropoulos, X., and Deri, L.** *Indexing Million of Packets Per Second Using GPUs*. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, IMC '13, pages 327–332. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-1953-9. doi:10.1145/2504730.2504756.

[28] **Glaskowsky, P. N.** *Nvidia's Fermi: The First Complete GPU Computing Architecture*. Online, September 2009. Last accessed: 23/06/2015.
URL http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf

[29] **GPGPU.org**. *About gpgpu.org*. Online. Last accessed: 31/03/2011.
URL http://gpgpu.org/about

[30] **Gupta, P. and McKeown, N.** *Classifying Packets with Hierarchical Intelligent Cuttings*. *IEEE Micro*, 20(1):34–41, 2000. ISSN 0272-1732. doi:10.1109/40.820051.

[31] **Gupta, S.** *Nvidia Updates GPU Roadmap; Announces Pascal*. Online., March 2014. Last Accessed: 23/06/2015.
URL http://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal/

[32] **Harris, M.** *Unified Memory in CUDA 6*. Online, November 2013. Last Accessed: 23/05/2015.
URL http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[33] **Hintjens, P.** *ØMQ - The Guide*. Online, 2014. Last Accessed 23/06/2015.
URL `http://zguide.zeromq.org/page:all`

[34] **Hoberock, J.** *Thrust Documentation*. Online, May 2012. Last Accessed: 23/06/2015.
URL `https://github.com/thrust/thrust/wiki/Documentation`

[35] **Hogg, S.** *Security at 10Gbps*. Online, February 2009. Last Accessed: 23/06/2015.
URL `http://www.networkworld.com/community/node/39071`

[36] **Hsieh, C.-L. and Weng, N.** *High Performance Multi-field Packet Classification Using Bucket Filtering and GPU Processing*. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 233–234. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-2839-5. doi:10.1145/2658260.2661768.

[37] **Intel Corporation**. *Intel® Core i7-3930K Processor*. Online. Last Accessed: 24/06/2015.
URL `http://ark.intel.com/products/63697/Intel-Core-i7-3930K-Processor-12M-Cache-up-to-3_80-GHz`

[38] **Ioannidis, S., Anagnostakis, K., Ioannidis, J., and Keromytis, A.** *xPF: packet filtering for low-cost network monitoring*. In *Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR)*, pages 116–120. 2002. doi:10.1109/HPSR.2002.1024219.

[39] **Irwin, B. V. W.** *A Framework for the Application of Network Telescope Sensors in a Global IP Network*. Ph.D. thesis, Rhodes University, Grahamstown, South Africa, January 2011.

[40] **Jiang, W. and Prasanna, V. K.** *Field-split parallel architecture for high performance multi-match packet classification using FPGAs*. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 188–196. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-606-9. doi:10.1145/1583991.1584044.

[41] **Jiang, W. and Prasanna, V. K.** *Large-scale wire-speed packet classification on FPGAs*. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 219–228.

ACM, New York, NY, USA, 2009. ISBN 978-1-60558-410-2. doi:10.1145/1508128.1508162.

[42] **Kanter, D.** *NVIDIA's GT200: Inside a Parallel Processor*. Online, August 2008. Last Accessed: 23/06/2015.
URL http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=1

[43] **Kessenich, J., Baldwin, D., and Rost, R.** *The OpenGL® Shading Language*. Online., June 2014. Last Accessed: 22/06/2015.
URL http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf

[44] **Khronos Group**. *OpenCL Overview*. Online, 2015. Last Accessed: 23/06/2015.
URL http://www.khronos.org/opencl/

[45] **Khronos OpenCL Working Group**. *The OpenCL Specification, Version 2.0*. Online, October 2014. Last Accessed: 23/06/2015.
URL http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf

[46] **Kim, Y.-H., Konow, R., Dujovne, D., Turletti, T., Dabbous, W., and Navarro, G.** *PcapWT: An Efficient Packet Extraction Tool for Large Volume Network Traces*. *Computer Networks, Elsevier*, 79:12, March 2015.
URL https://hal.inria.fr/hal-00938264

[47] **L. Degioanni, G. V., F. Risso**. *PCAP Next Generation Dump File Format*. Online, March 2004. Last Accessed: 23/06/2015.
URL http://www.winpcap.org/ntar/draft/PCAP-DumpFileFormat.html

[48] **Lakshman, T. V. and Stiliadis, D.** *High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching*. *ACM SIGCOMM Computer Communication Review*, 28(4):203–214, October 1998. ISSN 0146-4833. doi:10.1145/285243.285283.

[49] **Lakshminarayanan, K., Rangarajan, A., and Venkatachary, S.** *Algorithms for Advanced Packet Classification with Ternary CAMs*. *ACM SIGCOMM Computer Communication Review*, 35(4):193–204, August 2005. ISSN 0146-4833. doi:10.1145/1090191.1080115.

[50] **Lamping, U.** *Wireshark Developer's Guide: for Wireshark 1.99*. Online, 2014. Last Accessed: 23/06/2015.
URL http://www.wireshark.org/download/docs/developer-guide-us.pdf

[51] **Lamping, U., Harris, G., and Combs, G.** *Libpcap File Format*. Online, July 2013. Last Accessed: 23/06/2015.
URL http://wiki.wireshark.org/Development/LibpcapFileFormat

[52] **Lawlor, O.** *Message passing for GPGPU clusters: CudaMPI*. In *IEEE International Conference on Cluster Computing*, pages 1–8. Aug 2009. ISSN 1552-5244. doi:10.1109/CLUSTR.2009.5289129.

[53] **Lidl, K. J., Lidl, D. G., and Borman, P. R.** *Flexible Packet Filtering: Providing a Rich Toolbox*. In *Proceedings of the BSD Conference*, BSDC'02, pages 11–11. USENIX Association, Berkeley, CA, USA, 2002.

[54] **McCanne, S. and Jacobson, V.** *The BSD Packet Filter: A New Architecture for User-level Packet Capture*. In *Proceedings of the USENIX Winter Conference*, USENIX '93, pages 2–2. USENIX Association, Berkeley, CA, USA, 1993.

[55] **Micikevicius, P.** *GPU Performance Analysis and Optimization*. Online, 2012. Last Accessed: 23/06/2015.
URL http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf

[56] **Microsoft Corporation**. *LINQ (Language-Integrated Query)*. Online., . Last Accessed: 22/06/2015.
URL https://msdn.microsoft.com/en-us/library/bb397926.aspx

[57] **Microsoft Corporation**. *What do the Task Manager memory columns mean?* Online, . Last Accessed: 24/06/2015.
URL http://windows.microsoft.com/en-za/windows/what-task-manager-memory-columns-mean#1TC=windows-7

[58] **Microsoft Corporation**. *Explanation of Big Endian and Little Endian Architecture*. Online, 2006. Last Accessed: 23/06/2015.
URL https://support.microsoft.com/en-us/kb/102025

[59] **Microsoft Corporation**. *The OSI Model's Seven Layers Defined and Functions Explained*. Online, June 2014. Last Accessed: 23/06/2015.
URL `https://support.microsoft.com/en-za/kb/103884`

[60] **Mockapetris, P.** *Domain Names - Concepts and Facilities*. RFC 1034, Internet Engineering Task Force, November 1987.
URL `https://www.ietf.org/rfc/rfc1034.txt`

[61] **Mogul, J. and Deering, S.** *Path MTU Discovery*. RFC 1191, Internet Engineering Task Force, November 1990.
URL `https://www.ietf.org/rfc/rfc1191.txt`

[62] **Moore, D., Shannon, C., Voelker, G., and Savage, S.** *Network Telescopes: Technical Report*. Technical report, Cooperative Association for Internet Data Analysis (CAIDA), July 2004.
URL `http://www.caida.org/publications/papers/2004/tr-2004-04/`

[63] **Navarro, G.** *Wavelet Trees for All*. *Journal of Discrete Algorithms*, 25:2–20, 2014.
URL `http://www.dcc.uchile.cl/~gnavarro/ps/cpm12.pdf`

[64] **Nielsen, J.** *Nielsen's Law of Internet Bandwidth*. Online, 2010. Last Accessed: 23/06/2015.
URL `http://www.useit.com/alertbox/980405.html`

[65] **Nmap.Org**. *Nmap Reference Guide*. Online. Last Accessed: 30/05/2015.
URL `https://nmap.org/book/man-legal.html`

[66] **Nottingham, A.** *GPF: A framework for general packet classification on GPU co-processors*. Master's thesis, Rhodes University, Grahamstown, South Africa, February 2012.

[67] **Nottingham, A. and Irwin, B.** *gPF: A GPU Accelerated Packet Classification Tool*. In *Proceedings of the Southern Africa Telecommunication Networks and Applications Conference*, SATNAC '09. Swaziland, 2009.

[68] **Nottingham, A. and Irwin, B.** *GPU packet classification using OpenCL: a consideration of viable classification methods*. In *Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '09, pages 160–169. ACM,

New York, NY, USA, 2009. ISBN 978-1-60558-643-4. doi:10.1145/1632149.1632170.

[69] **Nottingham, A. and Irwin, B.** *Investigating the effect of genetic algorithms on filter optimisation within packet classifiers.* In *Proceedings of the ISSA 2009 Conference*, ISSA 2009, pages 99–116. Information Security South Africa (ISSA), University of Johannesburg, South Africa, July 2009.

[70] **Nottingham, A. and Irwin, B.** *Conceptual Design of a CUDA Based Packet Classifier*. In *Proceedings of the Southern Africa Telecommunication Networks and Applications Conference*, SATNAC '10. Cape Town, South Africa, 2010.

[71] **Nottingham, A. and Irwin, B.** *Parallel packet classification using GPU co-processors*. In *Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '10, pages 231–241. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-950-3. doi:10.1145/1899503.1899529.

[72] **Nottingham, A. and Irwin, B.** *A high-level architecture for efficient packet trace analysis on GPU co-processors*. In *Information Security for South Africa*, ISSA '13, pages 1 – 8. IEEE, Johannesburg, 2013. doi:10.1109/ISSA.2013.6641052.

[73] **Nottingham, A. and Irwin, B.** *Towards a GPU Accelerated Virtual Machine for Massively Parallel Packet Classification and Filtering*. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '13, pages 27–36. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2112-9. doi:10.1145/2513456.2513504.

[74] **Nottingham, A., Richter, J., and Irwin, B.** *CaptureFoundry: a GPU accelerated packet capture analysis tool*. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '12, pages 343–352. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1308-7. doi:10.1145/2389836.2389877.

[75] **NVIDIA Corporation**. *Tesla Software Features*. Online. Last Accessed: 23/06/2015.
URL `http://www.nvidia.com/object/software-for-tesla-products.html`

[76] **NVIDIA Corporation**. *GeForce 256: The Worlds First GPU*. Online, 1999. Last Accessed: 23/06/2015.
URL http://www.nvidia.com/page/geforce256.html

[77] **NVIDIA Corporation**. *Geforce3: The Infinite Effects GPU*. Online, 2001. Last Accessed: 23/06/2015.
URL http://www.nvidia.com/page/geforce3.html

[78] **NVIDIA Corporation**. *DX11 Samples*. Online, 2009. Last Accessed: 23/06/2015.
URL https://developer.nvidia.com/dx11-samples

[79] **NVIDIA Corporation**. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. 2012. Whitepaper. Last Accessed 23/06/2015.
URL http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf

[80] **NVIDIA Corporation**. *GeForce GTX Titan*. Online, 2013. Last Accessed: 23/06/2015.
URL http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications

[81] **NVIDIA Corporation**. *Kepler Tuning Guide*. Online, July 2013. Last Accessed: 23/06/2015.
URL http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html

[82] **NVIDIA Corporation**. *GeForce GTX 750*. Online, 2014. Last Accessed: 23/06/2015.
URL http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-750/specifications

[83] **NVIDIA Corporation**. *Maxwell Tuning Guide*. Online, August 2014. Last Accessed: 23/06/2015.
URL http://docs.nvidia.com/cuda/maxwell-tuning-guide/index.html

[84] **NVIDIA Corporation**. *NVIDIA GeForce GTX 750 Ti*. 2014. Whitepaper. Last Accessed 23/06/2015.
URL http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf

[85] **NVIDIA Corporation**. *NVIDIA Nsight Visual Studio Edition 4.2 User Guide*. Online, 2014. Last Accessed: 23/06/2015.
URL http://docs.nvidia.com/nsight-visual-studio-edition/4.2/Nsight_Visual_Studio_Edition_User_Guide.htm

[86] **NVIDIA Corporation**. *NVIDIA CUDA C Best Practices Guide, Version 7.0*. Online, March 2015. Last Accessed: 23/06/2015.
URL http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

[87] **NVIDIA Corporation**. *NVIDIA CUDA C Programming Guide, Version 7.0*. Online, March 2015. Last Accessed: 23/06/2015.
URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[88] **NVIDIA Corporation**. *NVIDIA GRID: Virtual GPU Technology*. Online, 2015. Last accessed: 04/02/2016.
URL http://www.nvidia.com/object/grid-technology.html

[89] **NVIDIA Corporation**. *Tesla Accelerated Computing*. Online, 2015. Last Accessed: 23/06/2015.
URL http://www.nvidia.com/object/tesla-supercomputing-solutions.html

[90] **Open Networking Foundation**. *OpenFlow Switch Specifcation Version 1.0.0*. Online, December 2009. Last Accessed: 06/06/2015.
URL https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf

[91] **Open Networking Foundation**. *Software-Defined Networking: The New Norm for Networks*. Technical report, Open Networking Foundation, April 2012.
URL https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[92] **OpenTK**. *The Open Toolkit Manual*. Online. Last Accessed: 29 /06/2015.
URL http://www.opentk.com/doc

[93] **Pang, R., Yegneswaran, V., Barford, P., Paxson, V., and Peterson, L.** *Characteristics of internet background radiation*. In *Proceedings of the*

ACM SIGCOMM Conference on Internet Measurement, IMC '04, pages 27–40. ACM, New York, NY, USA, 2004. ISBN 1-58113-821-0. doi:10.1145/1028788.1028794.

[94] **Parr, T.** *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007. ISBN 0978739256.

[95] **Patterson, D. A., Gibson, G., and Katz, R. H.** *A Case for Redundant Arrays of Inexpensive Disks (RAID)*. *ACM SIGMOD Record*, 17(3):109–116, June 1988. ISSN 0163-5808. doi:10.1145/971701.50214.

[96] **PCI-SIG**. *Frequently Asked Questions*. Online, 2015. Last Accessed: 23/06/2015.
URL `https://www.pcisig.com/news_room/faqs/pcie3.0_faq`

[97] **Plummer, D.** *An Ethernet Address Resolution Protocol – or – Converting Network Protocol Addresses*. RFC 826, Internet Engineering Task Force, November 1982.
URL `https://www.ietf.org/rfc/rfc826.txt`

[98] **Postel, J.** *Internet Control Message Protocol*. RFC 792, Internet Engineering Task Force, September 1981. Last Accessed: 29/05/2015.
URL `https://www.ietf.org/rfc/rfc792.txt`

[99] **Postel, J. and Reynolds, J.** *File Transfer Protocol (FTP)*. RFC 959, Internet Engineering Task Force, October 1985.
URL `https://www.ietf.org/rfc/rfc959.txt`

[100] **Postel, J. B.** *User Datagram Protocol*. RFC 768, Internet Engineering Task Force, August 1980.
URL `https://www.ietf.org/rfc/rfc768.txt`

[101] **Postel, J. B.** *Internet Protocol*. RFC 791, Internet Engineering Task Force, September 1981.
URL `https://www.ietf.org/rfc/rfc791.txt`

[102] **Postel, J. B.** *Transmission Control Protocol*. RFC 793, Internet Engineering Task Force, September 1981.
URL `https://www.ietf.org/rfc/rfc793.txt`

[103] **Risso, F. and Degioanni, L.** *An architecture for high performance network analysis*. In *Proceedings of the IEEE Symposium on Computers and Communications*, pages 686–693. 2001. ISSN 1530-1346. doi:10.1109/ISCC.2001.935450.

[104] **Robert Watson, C. P.** *Zero-Copy BPF*. Online, March 2007. Last Accessed: 23/06/2015.
URL `http://www.watson.org/~robert/freebsd/2007asiabsdcon/20070309-devsummit-zerocopybpf.pdf`

[105] **S. Deering, R. H.** *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460, Internet Engineering Task Force, December 1998.
URL `https://www.ietf.org/rfc/rfc2460.txt`

[106] **Segal, M. and Akeley, K.** *The OpenGL Graphics System: A Specification, version 4.5*. Online, May 2015. Last Accessed: 29/06/2015.
URL `https://www.opengl.org/registry/doc/glspec45.core.pdf`

[107] **Singh, S., Baboescu, F., Varghese, G., and Wang, J.** *Packet Classification Using Multidimensional Cutting*. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 213–224. ACM, New York, NY, USA, 2003. ISBN 1-58113-735-4. doi:10.1145/863955.863980.

[108] **Song, H. and Lockwood, J. W.** *Efficient Packet Classification for Network Intrusion Detection Using FPGA*. In *Proceedings of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 238–245. ACM, New York, NY, USA, 2005. ISBN 1-59593-029-9. doi:10.1145/1046192.1046223.
URL `http://doi.acm.org/10.1145/1046192.1046223`

[109] **Spitznagel, E., Taylor, D., and Turner, J.** *Packet classification using extended tcams*. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, page 120. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-2024-3.

[110] **Srinivasan, V., Suri, S., and Varghese, G.** *Packet Classification Using Tuple Space Search*. *ACM SIGCOMM Computer Communication Review*, 29(4):135–146, August 1999. ISSN 0146-4833. doi:10.1145/316194.316216.

[111] **Srinivasan, V., Varghese, G., Suri, S., and Waldvogel, M.** *Fast and Scalable Layer Four Switching*. *ACM SIGCOMM Computer Communication Review*, 28(4):191–202, October 1998. ISSN 0146-4833. doi:10.1145/285243.285282.

[112] **Stevens, W. R.** *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993. ISBN 0-201-63346-9.

[113] **Taylor, D. E.** *Survey and taxonomy of packet classification techniques*. *ACM Computing Surveys (CSUR)*, 37(3):238–275, September 2005. ISSN 0360-0300. doi:10.1145/1108956.1108958.

[114] **Taylor, S.** *CUDA Pro Tip: Kepler Texture Objects Improve Performance and Flexibility*. Online, February 2013. Last Accessed: 23/06/2015.
URL `http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-texture-objects-improve-performance-and-flexibility/`

[115] **Tcpdump.org**. *Tcpdump manpage*. Online, March 2009. Last Accessed: 23/06/2015.
URL `http://www.tcpdump.org/tcpdump_man.html`

[116] **Tcpdump.org**. *Link Layer Header Types*. Online, 2014. Last Accessed: 23/06/2015.
URL `http://www.tcpdump.org/linktypes.html`

[117] **Terry, P.** *Compiling with C# and JAVA*. Addison Wesley, 2005.

[118] **The Snort Project**. *SNORT Users Manual*. Sourcefire, Inc, 2.9.6 edition, April 2014.
URL `http://s3.amazonaws.com/snort-org/www/assets/166/snort_manual.pdf`

[119] **Tongaonkar, A. S.** *Fast Pattern-Matching Techniques for Packet Filtering*. Master's thesis, Stony Brook University, May 2004. Last Accessed: 23/06/2015.
URL `http://seclab.cs.sunysb.edu/seclab/pubs/theses/alok.pdf`

[120] **University of California, Berkeley**. *BOINC: Open-source software for volunteer computing*. Online, January 2016. Last accessed: 04/02/2016.
URL `https://boinc.berkeley.edu/`

[121] **van Lunteren, J. and Engbersen, T.** *Fast and scalable packet classification*. *IEEE Journal on Selected Areas in Communications*, 21(4):560–571, May 2003. ISSN 0733-8716. doi:10.1109/JSAC.2003.810527.

[122] **Varvello, M., Laufer, R., Zhang, F., and Lakshman, T.** *Multi-Layer Packet Classification with Graphics Processing Units*. In *Proceedings of the ACM International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 109–120. ACM, New York, NY, USA, 2014. ISBN 978-1-4503-3279-8. doi:10.1145/2674005.2674990.

[123] **Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E. P., and Ioannidis, S.** *Gnort: High Performance Network Intrusion Detection Using Graphics Processors*. In *Proceedings of the Iinternational Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 116–134. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-87402-7. doi:10.1007/978-3-540-87403-4_7.

[124] **Vasiliadis, G., Koromilas, L., Polychronakis, M., and Ioannidis, S.** *GASPP: A GPU-accelerated Stateful Packet Processing Framework*. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'14, pages 321–332. USENIX Association, Berkeley, CA, USA, 2014. ISBN 978-1-931971-10-2.

[125] **Volkov, V.** *Better Performance at Lower Occupancy*. Online, September 2010. Last Accessed: 23/06/2015.
URL `http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf`

[126] **West, J.** *NVIDIA releases CUDA 1.0*. Online, July 2007. Last Accessed: 23/06/2015.
URL `http://insidehpc.com/2007/07/14/nvidia-releases-cuda-10/`

[127] **WinPcap.org**. *NPF driver internals manual*. Online, 2002. Last Accessed: 23/06/2015.
URL `http://www.winpcap.org/docs/docs_412/html/group__NPF.html`

[128] **WinPcap.org**. *WinPcap Documentation*. Online, 2002. Last Accessed: 23/06/2015.
URL `http://www.winpcap.org/docs/docs_412/html/main.html`

[129] **Wireshark Foundation**. *Display Filter Reference*. Online. Last Accessed: 23/06/2015.
URL `https://www.wireshark.org/docs/dfref/`

[130] **Wireshark Foundation**. *Linux cooked-mode capture (SLL)*. Online, 2013. Last Accessed: 23/06/2015.
URL `https://wiki.wireshark.org/SLL`

[131] **Wireshark Foundation**. *Capture File Format Reference*. Online, 2014.
URL `http://wiki.wireshark.org/FileFormatReference`

[132] **Wireshark Foundation**. *PcapNg*. Online, 2015. Last Accessed: 23/06/2015.
URL `http://wiki.wireshark.org/Development/PcapNg`

[133] **Wu, Z., Xie, M., and Wang, H.** *Design and Implementation of a Fast Dynamic Packet Filter*. *IEEE/ACM Transactions on Networking (TON)*, 19(5):1405–1419, October 2011. ISSN 1063-6692. doi:10.1109/TNET.2011.2111381.

[134] **Wustrow, E., Karir, M., Bailey, M., Jahanian, F., and Huston, G.** *Internet background radiation revisited*. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 62–74. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0483-2. doi:10.1145/1879141.1879149.

[135] **Yuhara, M., Bershad, B. N., Maeda, C., Eliot, J., and Moss, B.** *Efficient packet demultiplexing for multiple endpoints and large messages*. In *Proceedings of the USENIX Winter Technical Conference*, WTEC'94, pages 153–165. 1994.

[136] **Zhou, S., Singapura, S., and Prasanna, V.** *High-performance packet classification on GPU*. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, Sept 2014. doi:10.1109/HPEC.2014.7041005.

Appendices

A

Grammar Syntax

This appendix provides complete EBNF listings for the high-level grammar (see Section 7.2) used in the DSL, as well as the instruction syntax used for the gather and filter programs used in GPF+ (see Sections 6.6 and 6.8).

A.1 EBNF for High-Level Grammar

```
program = protocol library, kernel function ;

(* protocol library *)
protocol library = protocol, { protocol } ;

(* Protocol *)
protocol = "protocol", id, [ "[", number, "]" ], ( ";" | protocol body ) ;
protocol body = "{", { field }, [ switch ], "}" ;

(* Field *)
field = "field" , id , "[", number, ":", number, "]" , ( ";" | field body )
    ;
```

```
12 field body = "{", { field filter }, [ field expression ], "}" ;
13
14 (* Filter *)
15 field filter = ".", id, comparison operator, number, ";" ;
16 comparison operator = "==" | "!=" | "<=" | ">=" | "<" | ">" ;
17
18 (* Length Field Processing *)
19 field expression = "$length", "=", integral expression, ";" ;
20 integral expression = multiply operation, { "+", multiply operation } ;
21 multiply operation = expression atom, { "*", expression } ;
22 expression atom = "$value" | number | ( "(", integral expression, ")" ) ;
23
24 (* Switch Statements *)
25 switch = "switch", "(", id, ")", "{", { switch case }, "}" ;
26 switch case = "case", id, ":", "goto", id, ";" ;
27
28 (* kernel function *)
29 kernel function = "main", "(",")", "{", { ( filter predicate | field
       extraction ), ";" }, "}" ;
30
31 (* Filter Predicates *)
32 filter predicate = and operation, { "||", and operation } ;
33 and operation = not operation, { "&&", not operation } ;
34 not operation = [ "!" ], filter atom ;
35 filter atom = field comparison | protocol | adhoc comparison | ( "(",
       filter predicate, ")" ) ;
36
37 (* Field Comparisons *)
38 field comparison = id, ".", id, ".", id ;
39 adhoc comparison = id, ".", id, comparison operator, number ;
40 protocol = id ;
41
42 (* Field Extraction *)
43 field extraction = "int", id, "=", id, ".", id ;
44
45 (* Types *)
46 number = ( ip address | integer | hex value ) ;
47 ip address = integer, ".", integer, ".", integer, ".", integer ;
48 id = ( letter | caps | "_" ), { letter | caps | "_" | digit } ;
49 integer = digit, { digit } ;
50 hex value = "0x", hex, { hex } ;
51
52 digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

```
53 letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" |
       "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" |
       "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
       "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" |
       "V" | "W" | "X" | "Y" | "Z" ;
54 hex = digit | "a" | "b" | "c" | "d" | "e" | "f" | "A" | "B" | "C" | "D" | "
       E" | "F" ;
```

A.2 EBNF for Gather Program

```
1  gather program = { stack layer } ;
2
3  (* Layer *)
4  stack layer = layer header , cache chunk count , { cache chunk } ;
5  layer header = protocol count , { layer protocol } , skip offset ;
6  layer protocol = protocol id , protocol length ;
7
8  (* Cache *)
9  cache chunk = local offset , protocol count , { protocol set } ;
10
11 (* Protocol Set *)
12 protocol set = protocol id , skip offset , field count , { field set };
13
14 (* Field *)
15 field set = field offset , field length , store index , filter count, {
       filter comparison } , expressions ;
16
17 (* Filter and Switch *)
18 filter comparison = comparison operator , lookup index , switch id ,
       working index ;
19 comparison operator = "0" | "1" | "2" | "3" | "4" | "5" ;
20
21 (* Expression *)
22 expression = expression count , { sum } ;
23 sum = product count , { product } , value ;
24 product = value count , { value } ;
25 value = type id , index ;
26
27 (* Types *)
28 protocol count = number ;
29 protocol id = number ;
30 protocol length = number ;
31 skip offset = number ;
```

```
cache set count = number ;
local offset = number ;
protocol count = number ;
field count = number ;
field offset = number ;
field length = number ;
store index = number ;
filter count = number ;
lookup index = number ;
switch id = number ;
working index = number ;
expression count = number ;
product count = number ;
store index = number ;
value count = number ;
read type = number ;
read index = number ;
store type = "0" | "1" ;

number = digit , { digit } ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

A.3 ENBF for Filter Program

```
filter program program = filter count , { filter } ;

(* Filter Predicate *)
filter = or count , { group } , store location;
group = and count , { element } ;
element = invert , read index ;
store location = store type, write index;

(* Types *)
filter count = number ;
or count = number ;
and count = number ;
invert = "0" | "1" ;
store type = "0" | "1" ;
read index = number ;
write index = number ;

number = digit , { digit } ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

B

Common TCP/IP Protocols

This appendix shows the structure of a small selection of common protocols from the TCP/IP suite that were discussed frequently in this research.

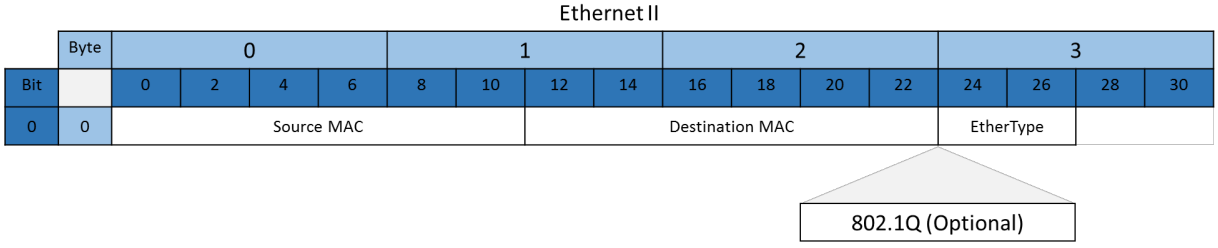


Figure B.1: Ethernet II Header Format

Internet Protocol Version 4

	Byte	0				1				2				3			
Bit		0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
0	0	Version		IHL		DSCP			ECN	Total Length							
32	4	Identification								Flags	Fragment Offset						
64	8	Time To Live				Protocol				Header Checksum							
96	12	Source IP Address															
128	16	Destination IP Address															
160	20	Options															

Figure B.2: IPv4 Header Format

Internet Protocol Version 6

	Byte	0				1				2				3			
Bit		0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
0	0	Version			Traffic Class				Flow Label								
32	4	Payload Length								Next Header				Hop Limit			
64	8	Source Address															
96	12																
128	16																
160	20																
192	24	Destination Address															
224	28																
256	32																
288	36																

Figure B.3: IPv6 Header Format

Transmission Control Protocol

	Byte	0				1				2				3			
Bit		0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
0	0	Source Port								Destination Port							
32	4	Sequence Number															
64	8	Acknowledgement Number															
96	12	Data Offset		Reserved		Flags				Window Size							
128	16	Checksum								Urgent pointer							
160	20	Options															

Figure B.4: TCP Header Format

User Datagram Protocol

	Byte	0				1				2				3			
Bit		0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
0	0	Source Port								Destination Port							
32	4	Length								Checksum							

Figure B.5: UDP Header Format

ICMP

	Byte	0				1				2				3			
Bit		0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
0	0	Type				Code				Checksum							
32	4	Message Body															

Figure B.6: ICMP Header Format

C

Summary of Testing Configuration

This appendix provides a summary of the testing configurations used during testing.

Table C.1: Host Configuration

CPU	Make	Intel
	Model	Core i7-3930K
	Cores	6
	Base Frequency	3.2 GHz
RAM	Type	DDR3 1600
	Total Memory	32 GB
	System Memory	24 GB
	RAM Disk	8 GB
Operating System		Windows 7 x64
PCIe Interface		PCIe 2

Table C.2: Overview of GPUs used in testing.

GPUs Tested		
	GTX 750	GTX Titan
Manufacturer	Gigabyte	MSI
Version	GV-N750OC-1GI	06G-P4-2790-KR
Micro-architecture	Maxwell	Kepler
Compute Capability	5.0	3.5
CUDA Cores	512	2688
Device Memory (MB)	1024	6144
Memory Type	GDDR5	GDDR5
Core Base Clock (MHz)	1020	837
Memory Bandwidth (GB/s)	80	288
Release Price	$119	$999
Release Date	February 18^{th}, 2014	February 21^{st}, 2013

Table C.3: Storage devices used during testing.

Storage Devices		
	HDD	SSD
Interface	SATA II	SATA III
Drive	Various	Crucial MX100
Model No.	Various	CT256MX100SSD1
Drive Count	4	2
Drive Capacity	1.5TB - 2TB	256 GB
Peak Read Speed	$\pm$120 MB/s	$\pm$500 MB/s

Table C.4: Packet sets used in testing.

Packet Set	A	B	C
Total Packets	26,334,066	260,046,507	338,202,452
Average Size	70 bytes	155 bytes	480 bytes
Average Rate	0.9 /s	15 /s	12,150 /s
File Size	2.1 GB	41.4 GB	156 GB
Duration	11 months	6 months	8 hours

D

Filter Programs

This appendix contains the full source for all programs used during testing.

D.1 Linux CookedCapture Header

This code fragment shows the link-layer header used when processing capture C.

```
protocol LinuxCookedCapture
{
    field EtherType [112:16]
    {
        .IPv4 == 0x800;
        .IPv6 == 0x86DD;
        ...
    }
    switch(EtherType)
    {
        case IPv4: goto IPv4;
        case IPv6: goto IPv6;
```

```
13          ...
14      }
15  }
```

D.2 Program Set A

This section provides the full source for programs A1, A2 and A3, which perform filtering operations only.

D.2.1 Program A1

```
1  protocol Ethernet
2  {
3      field EtherType [96:16]
4      {
5          .IP == 0x800;
6      }
7      switch(EtherType)
8      {
9          case IP: goto IP;
10     }
11 }
12 protocol IP {}
13
14 main()
15 {
16     filter ip = IP;
17 }
```

D.2.2 Program A2

```
1  protocol Ethernet
2  {
3      field EtherType [96:16]
4      {
5          .IPv4 == 0x800;
6          .IPv6 == 0x86DD;
7          .ARP == 0x806;
8      }
```

```
9       switch(EtherType)
10      {
11          case IPv4: goto IPv4;
12          case IPv6: goto IPv6;
13          case ARP: goto ARP;
14      }
15  }
16  protocol ARP {}
17  protocol IPv4
18  {
19      field IHL [4:4]
20      {
21          $length = $value * 4;
22      }
23      field Protocol [72:8]
24      {
25          .ICMP == 1;
26          .TCP == 6;
27          .UDP == 17;
28      }
29      switch (Protocol)
30      {
31          case ICMP: goto ICMP;
32          case TCP : goto TCP;
33          case UDP : goto UDP;
34      }
35  }
36  protocol IPv6
37  {
38      field PayloadLength [32:16]
39      {
40          $length = $value;
41      }
42      field NextHeader [48:8]
43      {
44          .TCP == 6;
45          .UDP == 17;
46          .ICMP == 58;
47      }
48      switch (NextHeader)
49      {
50          case TCP : goto TCP;
51          case UDP : goto UDP;
52          case ICMP: goto ICMP;
```

```
53      }
54  }
55  protocol TCP {}
56  protocol UDP {}
57  protocol ICMP {}
58
59  main()
60  {
61      filter tcp = TCP;
62      filter udp = UDP;
63      filter icmp = ICMP;
64      filter arp = ARP;
65  }
```

D.2.3 Program A3

```
1   protocol Ethernet
2   {
3       field EtherType [96:16]
4       {
5           .IPv4 == 0x800;
6           .IPv6 == 0x86DD;
7       }
8       switch(EtherType)
9       {
10          case IPv4: goto IPv4;
11          case IPv6: goto IPv6;
12      }
13  }
14  protocol IPv4
15  {
16      field IHL [4:4]
17      {
18          $length = $value * 4;
19      }
20      field Protocol [72:8]
21      {
22          .ICMP == 1;
23          .TCP == 6;
24          .UDP == 17;
25      }
26      switch (Protocol)
27      {
28          case TCP : goto TCP;
```

```
29          case UDP : goto UDP;
30          case ICMP: goto ICMP;
31      }
32  }
33  protocol IPv6
34  {
35      field PayloadLength [32:16]
36      {
37          $length = $value;
38      }
39      field NextHeader [48:8]
40      {
41          .TCP == 6;
42          .UDP == 17;
43          .ICMP == 58;
44      }
45      switch (NextHeader)
46      {
47          case TCP : goto TCP;
48          case UDP : goto UDP;
49          case ICMP: goto ICMP;
50      }
51  }
52  protocol TCP
53  {
54      field SourcePort [0:16];
55      field DestinationPort [16:16];
56  }
57  protocol UDP
58  {
59      field SourcePort [0:16];
60      field DestinationPort [16:16];
61  }
62  protocol ICMP {}
63
64  main()
65  {
66      filter icmp = ICMP;
67      filter tcp = TCP;
68      filter udp = UDP;
69
70      filter ftp_src = TCP.SourcePort == 21 || TCP.DestinationPort == 21;
71      filter smtp = TCP.SourcePort == 25 || TCP.DestinationPort == 25;
72      filter http = TCP.SourcePort == 80 || TCP.DestinationPort == 80;
```

```
73
74      filter ssh = TCP.SourcePort == 22 || TCP.DestinationPort == 22 || UDP.
            SourcePort == 22 || UDP.DestinationPort == 22;
75      filter dns = TCP.SourcePort == 53 || TCP.DestinationPort == 53 || UDP.
            SourcePort == 53 || UDP.DestinationPort == 53;
76      filter dhcp = UDP.SourcePort == 68 && UDP.DestinationPort == 67 || UDP.
            SourcePort == 67 && UDP.DestinationPort == 68;
77
78      filter dhcp6 = UDP.SourcePort == 546 && UDP.DestinationPort == 547 ||
            UDP.SourcePort == 547 && UDP.DestinationPort == 546 || TCP.
            SourcePort == 546 && TCP.DestinationPort == 547 || TCP.SourcePort ==
             547 && TCP.DestinationPort == 546;
79  }
```

D.3 Program Set B

This section provides the full source for programs B1, B2 and B3, which perform field extraction operations only.

D.3.1 Program B1

```
1   protocol Ethernet
2   {
3       field EtherType [96:16]
4       {
5           .IP == 0x800;
6       }
7       switch(EtherType)
8       {
9           case IP: goto IP;
10      }
11  }
12  protocol IP
13  {
14      field Protocol [72:8];
15  }
16  main()
17  {
18      int proto = IP.Protocol;
19  }
```

D.3.2 Program B2

```
protocol Ethernet
{
    field EtherType [96:16]
    {
        .IP == 0x800;
    }
    switch(EtherType)
    {
        case IP: goto IP;
    }
}
protocol IP
{
    field IHL [4:4]
    {
        $length = $value * 4;
    }
    field Protocol [72:8]
    {
        .TCP == 6;
        .UDP == 17;
    }
    field SourceAddress [96:32];
    field DestinationAddress [128:32];
    switch (Protocol)
    {
        case TCP : goto Ports;
        case UDP : goto Ports;
    }
}
protocol Ports
{
    field SourcePort[0:16];
    field DestinationPort[16:16];
}

main()
{
    int proto = IP.Protocol;
    int srcaddr = IP.SourceAddress;
    int destaddr = IP.DestinationAddress;
    int srcport = Ports.SourcePort;
```

```
43      int destport = Ports.DestinationPort;
44  }
```

D.3.3 Program B3

```
1   protocol Ethernet
2   {
3       field EtherType [96:16]
4       {
5           .IPv4 == 0x800;
6           .IPv6 == 0x86DD;
7       }
8       switch(EtherType)
9       {
10          case IPv4: goto IPv4;
11          case IPv6: goto IPv6;
12      }
13  }
14  protocol IPv4
15  {
16      field IHL [4:4]
17      {
18          $length = $value * 4;
19      }
20      field Protocol [72:8]
21      {
22          .ICMP == 1;
23          .TCP == 6;
24          .UDP == 17;
25      }
26      switch (Protocol)
27      {
28          case ICMP: goto ICMP;
29          case TCP : goto TCP;
30          case UDP : goto UDP;
31      }
32  }
33  protocol IPv6
34  {
35      field PayloadLength [32:16]
36      {
37          $length = $value;
38      }
39      field NextHeader [48:8]
```

```
40      {
41          .TCP == 6;
42          .UDP == 17;
43          .ICMP == 58;
44      }
45      switch (NextHeader)
46      {
47          case TCP : goto TCP;
48          case UDP : goto UDP;
49          case ICMP: goto ICMP;
50      }
51  }
52  protocol TCP
53  {
54      field SourcePort [0:16];
55      field DestinationPort [16:16];
56      field SequenceNumber [32:32];
57      field AcknowledgmentNumber [64:32];
58  }
59  protocol UDP
60  {
61      field SourcePort[0:16];
62      field DestinationPort[16:16];
63      field Length [32:16];
64  }
65  protocol ICMP
66  {
67      field Type [0:8];
68      field Code [8:8];
69  }
70  main()
71  {
72      int ethertype = Ethernet.EtherType;
73
74      int tcpsrcport = TCP.SourcePort;
75      int tcpdestport = TCP.DestinationPort;
76      int tcpseqno = TCP.SequenceNumber;
77      int tcpackno = TCP.AcknowledgmentNumber;
78
79      int udpsrcport = UDP.SourcePort;
80      int udpdestport = UDP.DestinationPort;
81      int udplen = UDP.Length;
82
83      int icmptype = ICMP.Type;
```

```
84      int icmpcode = ICMP.Code;
85  }
```

D.4 Program Set C

This section provides the full source for programs C1, C2 and C3, which perform filtering and field extraction operations simultaneously.

D.4.1 Program C1

```
1   protocol Ethernet
2   {
3       field EtherType [96:16]
4       {
5           .IPv4 == 0x800;
6           .IPv6 == 0x86DD;
7       }
8       switch(EtherType)
9       {
10          case IPv4: goto IPv4;
11          case IPv6: goto IPv6;
12      }
13  }
14  protocol IPv4 { field Protocol [72:8]; }
15  protocol IPv6 { field NextHeader [48:8]; }
16
17  main()
18  {
19      filter ipv4 = IPv4;
20      filter ipv6 = IPv6;
21
22      int proto = IPv4.Protocol;
23      int nextHeader = IPv6.NextHeader;
24  }
```

D.4.2 Program C2

```
1   protocol Ethernet
2   {
3       field EtherType [96:16]
```

```
4       {
5           .IPv4 == 0x800;
6           .IPv6 == 0x86DD;
7           .ARP == 0x806;
8       }
9       switch(EtherType)
10      {
11          case IPv4: goto IPv4;
12          case IPv6: goto IPv6;
13          case ARP: goto ARP;
14      }
15  }
16  protocol ARP {}
17  protocol IPv4
18  {
19      field IHL [4:4]
20      {
21          $length = $value * 4;
22      }
23      field Protocol [72:8]
24      {
25          .ICMP == 1;
26          .TCP == 6;
27          .UDP == 17;
28      }
29      switch (Protocol)
30      {
31          case ICMP: goto ICMP;
32          case TCP : goto ServicePorts;
33          case UDP : goto ServicePorts;
34      }
35  }
36  protocol IPv6
37  {
38      field PayloadLength [32:16]
39      {
40          $length = $value;
41      }
42      field NextHeader [48:8]
43      {
44          .TCP == 6;
45          .UDP == 17;
46          .ICMP == 58;
47      }
```

```
48      switch (NextHeader)
49      {
50          case TCP : goto ServicePorts;
51          case UDP : goto ServicePorts;
52          case ICMP: goto ICMP;
53      }
54  }
55  protocol ServicePorts
56  {
57      field SourcePort [0:16]
58      {
59          .SSH == 22;
60          .DNS == 53;
61      }
62      field DestinationPort [16:16]
63      {
64          .SSH == 22;
65          .DNS == 53;
66      }
67  }
68  protocol ICMP
69  {
70      field Type [0:8];
71      field Code [8:8];
72  }
73  main()
74  {
75      filter tcp = IPv4.Protocol.TCP || IPv6.NextHeader.TCP;
76      filter udp = IPv4.Protocol.UDP || IPv6.NextHeader.UDP;
77      filter icmp = ICMP;
78      filter arp = ARP;
79
80      filter ssh = ServicePorts.SourcePort.SSH || ServicePorts.DestinationPort
            .SSH;
81      filter dns = ServicePorts.SourcePort.DNS || ServicePorts.DestinationPort
            .DNS;
82
83      int srcport = ServicePorts.SourcePort;
84      int dstport = ServicePorts.DestinationPort;
85
86      int icmptype = ICMP.Type;
87      int icmpcode = ICMP.Code;
88  }
```

D.4.3 Program C3

```
protocol Ethernet
{
    field EtherType [96:16]
    {
        .IPv4 == 0x800;
        .IPv6 == 0x86DD;
    }
    switch(EtherType)
    {
        case IPv4: goto IPv4;
        case IPv6: goto IPv6;
    }
}
protocol IPv4
{
    field IHL [4:4]
    {
        $length = $value * 4;
    }
    field Protocol [72:8]
    {
        .ICMP == 1;
        .TCP == 6;
        .UDP == 17;
    }
    switch (Protocol)
    {
        case TCP : goto TCP;
        case UDP : goto UDP;
        case ICMP: goto ICMP;
    }
}
protocol IPv6
{
    field PayloadLength [32:16]
    {
        $length = $value;
    }Linux CookedCapture Header


    field NextHeader [48:8]
    {
```

```
43          .TCP == 6;
44          .UDP == 17;
45          .ICMP == 58;
46      }
47      switch (NextHeader)
48      {
49          case TCP : goto TCP;
50          case UDP : goto UDP;
51          case ICMP: goto ICMP;
52      }
53  }
54  protocol TCP
55  {
56      field SourcePort [0:16];
57      field DestinationPort [16:16];
58      field SequenceNumber [32:32];
59      field AcknowledgmentNumber [64:32];
60  }
61  protocol UDP
62  {
63      field SourcePort[0:16];
64      field DestinationPort[16:16];
65      field Length [32:16];
66  }
67  protocol ICMP
68  {
69      field Type [0:8];
70      field Code [8:8];
71  }
72
73  main()
74  {
75      filter icmp = ICMP;
76      filter tcp = TCP;
77      filter udp = UDP;
78
79      filter ftp_src = TCP.SourcePort == 21 || TCP.DestinationPort == 21;
80      filter smtp = TCP.SourcePort == 25 || TCP.DestinationPort == 25;
81      filter http = TCP.SourcePort == 80 || TCP.DestinationPort == 80;
82
83      filter ssh = TCP.SourcePort == 22 || TCP.DestinationPort == 22 || UDP.
            SourcePort == 22 || UDP.DestinationPort == 22 ;
84      filter dns = TCP.SourcePort == 53 || TCP.DestinationPort == 53 || UDP.
            SourcePort == 53 || UDP.DestinationPort == 53 ;
```

```
85      filter dhcp = UDP.SourcePort == 68 && UDP.DestinationPort == 67 || UDP.
            SourcePort == 67 && UDP.DestinationPort == 68;
86
87      filter dhcpv6 = UDP.SourcePort == 546 && UDP.DestinationPort == 547 ||
            UDP.SourcePort == 547 && UDP.DestinationPort == 546 || TCP.
            SourcePort == 546 && TCP.DestinationPort == 547 || TCP.SourcePort ==
             547 && TCP.DestinationPort == 546;
88
89      int ethertype = Ethernet.EtherType;
90
91      int tcpsrcport = TCP.SourcePort;
92      int tcpdestport = TCP.DestinationPort;
93      int tcpseqno = TCP.SequenceNumber;
94      int tcpackno = TCP.AcknowledgmentNumber;
95
96      int udpsrcport = UDP.SourcePort;
97      int udpdestport = UDP.DestinationPort;
98      int udplen = UDP.Length;
99
100     int icmptype = ICMP.Type;
101     int icmpcode = ICMP.Code;
102 }
```

E

Program Source

The complete program source for components discussed in this research may be found at `https://github.com/anottingham/PhdResearch`.