Parallel packet classification using GPU co-processors

Alastair Nottingham
Security and Networks Research Group
Department of Computer Science
Rhodes University
Grahamstown, South Africa
anottingham@gmail.com

Barry Irwin
Security and Networks Research Group
Department of Computer Science
Rhodes University
Grahamstown, South Africa
b.irwin@ru.ac.za

ABSTRACT

In the domain of network security, packet filtering for classification purposes is of significant interest. Packet classification provides a mechanism for understanding the composition of packet streams arriving at distinct network interfaces, and is useful in diagnosing threats and uncovering vulnerabilities so as to maximise data integrity and system security. Traditional packet classifiers, such as PCAP, have utilised Control Flow Graphs (CFGs) in representing filter sets, due to both their amenability to optimisation, and their inherent structural applicability to the metaphor of decision-based classification. Unfortunately, CFGs do not map well to cooperative processing implementations, and single-threaded CPU-based implementations have proven too slow for real-time classification against multiple arbitrary filters on next generation networks. In this paper, we consider a novel multi-threaded classification algorithm, optimised for execution on GPU co-processors, intended to accelerate classification throughput and maximise processing efficiency in a highly parallel execution context.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network monitoring*

General Terms

Algorithms, Design, Performance

Keywords

CUDA, GP-GPU, General Packet Filter, PCAP, Network Telescope

1. INTRODUCTION

SAICSIT '10, October 11-13, Bela Bela, South Africa

The task of classifying a single packet is essentially a trivial operation. In an abstract sense, it may be considered as a boolean valued function which is applied to packet data in order to determine a classification result [19]. The function, termed a filter, may be simple, comprising only a single test, or more sophisticated, involving multiple comparisons evaluated within a predicate to return a result. A typical packet classifier evaluates a small, static set of one or more filters over large volumes of packet data [5, 13, 19], a repetitive procedure which is ideally suited to parallel processing. Unfortunately, due to the historically limited availability and affordability of parallel processing hardware, and the significant processing overhead inherent in filtering large packet sets, most packet filtering algorithms have been heavily optimised for sequential execution so as to reduce processing time as much as possible [5, 8, 19]. Despite numerous optimizations however, sequential software packet classifiers often take longer to classify a packet set captured off a giga-bit network interface than it took the set to arrive, making them infeasible for real-time traffic analysis.

By leveraging Nvidia CUDA [3] enabled low cost, commodity GPU co-processors to accelerate packet filtering throughput, packet classification could be utilised in high-resolution real-time network monitoring and long-term packet capture analysis. In order to achieve this, it was necessary to devise a suitable algorithm, specifically tailored to GPU architecture, and optimised to ensure maximum utilization of GPU resources. Unlike traditional sequential demultiplexing algorithms which search for the first matching filter [5, 8, 13, 23], the algorithm discussed, called gPF, is a multi-matching filter, evaluating all filters against every packet in order to collect a more accurate set of classifications. It is worth noting that while an OpenCL implementation will not be considered in this paper, the conceptual design presented is also applicable within an OpenCL implementation, with minimal modification required [1].

In Section 2, we introduce the packet filtering domain, describing the abstract mechanisms used, and detailing several distinct problem areas in which they are employed. Section 3 follows, providing a brief overview of GPU architecture and the CUDA Programming model, while Section 4 considers methods for improving kernel and memory performance. In Section 5 we apply this knowledge to demonstrate why current algorithms are ill suited to the problem, and detail the conceptual design of a CUDA based filter optimised to maximise packet classification throughput on GPU hardware, with preliminary results.

Finally, Section 6 concludes with a summary of the paper.

2. OVERVIEW OF PACKET FILTERING

This section provides a brief introduction to the domain of packet filtering, detailing the concept and providing some important background.

2.1 Abstract Packet Filtering

Data is transferred between network interfaces contained within binary arrays known as packets [13, 18]. Packets typically comprise a small segment of data, known as the payload, combined with a series of protocol headers used for transmitting, routing and receiving packets. Large payloads are typically divided over several packets, termed fragments, which may be reconstituted into a single payload by the receiving host, using information contained within the packet header [18]. Headers contain a collection of arbitrarily sized bit-ranges, which specify address and port information, protocol flags, and other information relevant to successful transmission [18, 19]. Packet headers are encoded as structured binary arrays, with arbitrary-length header fields located at protocol-specific indexes within the array. Headers contain vast amounts of useful network-related data which may be used to predict and protect against security threats, diagnose network anomalies, and aid in network related research. Since the packet payload is typically application specific [13, 18], packet filtering generally focuses on evaluating the data contained within the packet header, ignoring the payload entirely.

A filter is a boolean valued predicate function, operating over a collection of criteria, against which each arriving packet is compared [5, 13, 19]. A packet is said to be classified by a filter if the filter function returns a boolean result of true, indicating that the packet has met the specific criteria for that filter. Filter criteria are boolean valued comparisons, performed between values contained in discreet bit-ranges in the packet header and static protocol defined values [19]. For example, in the Ethernet II Frame Header, the Type field is a two-octet (16-bit) value, offset 12 bytes from the start of the header by two six-octet (48-bit) MAC addresses [18]. If the packet is an IP datagram, then this field will be set to a value of 0x800 (hexidecimal) [18], equivalent to 2048 in decimal. Thus, any filter targeting IP datagrams need only compare the 16-bit range offset 12 bytes from the beginning of the packet to the value 2048 in order to determine if the filter succeeds. In most cases however, a filter will contain multiple criteria, in multiple levels of the protocol stack, which must be met in order for the filter to succeed.

In general, packet filtering involves the comparison of each arriving packet against a set of one or more filters in order to determine important information about the packet; for example, its type, purpose and origin. Packet filters have been employed in many distinct areas of the network domain, including but not limited to Packet Demultiplexing, IP Routing and Packet Analysis. In the following subsection, we discuss some important observations regarding the abstract filtering mechanism which we can exploit in all of these application areas. We follow this with a brief overview of each area, including algorithm specific optimisations and limitations.

2.2 Important Observations

In this subsection we focus on the implications of the above abstract filter mechanism, in order to demonstrate two important properties of filter sets. A filter set is a collection of one or more unique filters, where each filter is a predicate function, operating over a set of one or more distinct criteria. Criteria are boolean valued comparisons against fields within the packet header, where each field is an array of one or more bits. As a field is typically comprised of only a few bits, the field has a limited number of possible values. A field n bits wide can have a maximum of 2^n possible values, and typically far fewer values are actually defined for larger fields. This leads to our first observation: the number of possible field values is typically quite small, and remains small independent of filter count. This is termed the Matching Set Confinement property [19].

The second property is related to Matching Set Confinement. We note that the majority of defined protocols and header field values are for specialised, legacy or research-related applications, and most traffic is transmitted using a small subset of the available protocol field values. For instance, most Internet traffic uses the TCP/IP protocol suite for transmission, and thus the vast majority of filters operating on these transmissions require similar, if not identical, tests on a small subset of packet header fields. Similarly, while there are millions of possible source and destination hosts on the Internet, the vast majority of filter sets will only reference a small subset of these hosts. Combined with the Matching Set Confinement property, this leads to our second observation: the number of distinct criteria evaluations we perform on an individual packet is small, and remains small as packet filter sets increase in size. This is termed the Match Condition Redundancy property [5, 19].

2.3 IP Routing Algorithms

In order to facilitate communication between two hosts over the Internet, each individual packet must traverse a path between several routers. When a router receives a packet, it must either transmit the packet to the waiting host directly, or send the packet to another router closer to the destination network [19], called the next-hop router. In order for the router to be able to perform this task, it must be able to discern information about the packet's route from its header specification [18, 19]. Given the significant volume of IP traffic traveling through the Internet backbone every second, coupled with the copious number of possible destinations, IP routing algorithms need to be able to compare thousands of packets against hundreds of filters efficiently [19].

As their name suggests, IP Routing algorithms operate exclusively on a subset of the IP protocol, commonly referred to as the IP 5-tuple [19]. The IP 5-tuple is composed of: source address, destination address, source port, destination port, and protocol. Due to their performance requirements, IP routing algorithms are heavily optimised with

respect to both the IP protocol and the underlying hardware, so as to ensure maximum routing throughput[19]. This comes at the cost of flexibility and protocol independence, making routing algorithms difficult to re-target toward arbitrary protocols or complex match conditions. They do, however, provide significant insights into optimising classification speed.

There are numerous distinct approaches to IP routing algorithms, which utilise a wide array of data structures and procedural optimisations in order to improve performance. The most simple class of routing algorithm — known as Exhaustive Search Algorithms — compare packets against each and every filter in the filter set until such time as a suitable match is found [19, 16]. These algorithms are generally slow, and therefore not very useful. Other classes of algorithm include Decision Tree, Decomposition and Tuple Space approaches. Decision tree algorithms are diverse in design, but all leverage a sequential tree like traversal of a specialised data structure in order to narrow down the number of criteria against which the packet needs to be compared [5, 14, 17, 19]. Decision trees also form the foundation of most demultiplexing and analysis filters [5, 8, 13], as they are highly sequential, and thus map well to processing on sequentially optimised CPUs such as those found in modern desktop computers.

In contrast, Decomposition algorithms target parallel processing hardware such as FPGAs, typically breaking down filter classifications into smaller sub-classifications which can be performed in parallel[4, 12, 19]. A final classification step is performed by a single processor, which combines the output from each sub-classification and evaluates the result. Finally, Tuple Space algorithms are highly specialised, and exploit a variety of filter set properties in order to reduce processing time [19].

2.4 Packet Demultiplexing Algorithms

Packet demultiplexing algorithms filter traffic arriving at a particular host interface, so as to direct collected packets to the address space of the correct waiting application [18, 23]. In order to facilitate arbitrary applications, demultiplexing algorithms have to be both general and protocol independent [13]. This ensures compatibility with future protocols, and allows for flexibility in defining filters. All demultiplexing algorithms employ a decision tree approach, due to their efficiency at eliminating redundant computation on cached sequential processors.

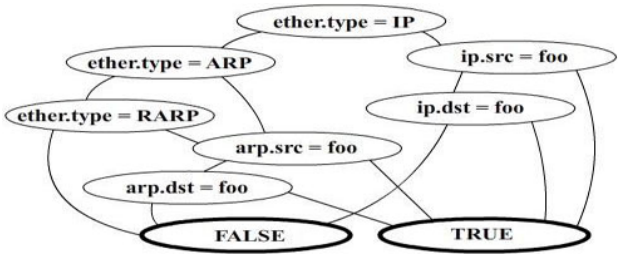

Figure 1: A BPF filter, depicted as a Control Flow Graph [13].

The CMU/Stanford Packet Filter (CSPF), a tree approach designed for stack-based machines, was the inspiration for the canonical Berkley Packet Filter (BPF). BPF employed Control Flow Graphs (CFGs) in an assembler style programmable pseudo-machine, so as to improve performance on register-based processors[13] (see Figure 1). While BPF was initially designed with Packet Analysis in mind, it forms the conceptual foundation for modern packet demultiplexing. Later packet filters such as the Mach Packet Filter (MPF) and the Dynamic Packet Filter (DPF) optimised the conceptual model for CFG filtering introduced in BPF[8, 23]. MPF was designed to extend and improve demultiplexing performance[23], while DPF focused on exploiting dynamic code generation in order to prune redundant instructions [8]. These filters led to the development of BPF+, which employed techniques such as Predicate Assertion Propagation and Partial Redundancy Elimination, in conjunction with Just-In-Time (JIT) processing and a variety of other optimisations, to dramatically improve processing speeds [5].

Subsequent packet filters included the Extensible Packet Filter (xPF), the Fairly Fast Packet Filter (FFPF) and SWIFT [6, 9, 22]. xPF was designed to reduce the context switching overhead prevalent in the BPF pseudo-machine [9], while FFPF was designed to facilitate high performance demultiplexing between multiple network monitoring applications [6]. Finally, SWIFT aimed at reducing filter update latency so as to support real-time filter updates, while providing an overall speed boost to classification throughput [22].

2.5 Packet Analysis

The domain of Packet Analysis provides the foundation for network monitoring applications, and is comprised of two relatively distinct fields; namely, header-based packet filtering and deep packet inspection. Whereas header-based filtering targets header fields exclusively and conforms with the abstract filtering mechanisms introduced in this section, deep packet inspection algorithms target payload data and generally rely heavily on string matching techniques [20]. Network Intrusion Detection Systems (NIDS) which employ deep packet inspection often utilise fast 5-tuple filters, such as those used in IP routing, to efficiently determine which packets to inspect [10]. We shall focus explicitly on general header-based filtering, as deep packet inspection algorithms do not map well to the problem space, and 5-tuple algorithms do not provide sufficient flexibility for general packet filtering.

Header based filters comprise those algorithms introduced in 2.4, with many demultiplexing algorithms — such as BPF and BPF+ — designed with packet monitoring in mind [5, 13]. As both fields depend heavily on filtering speed, utilise the same header-based processing abstraction, and operate on similar hardware, this overlap makes practical sense.

2.6 Hardware Considerations

The hardware environment on which a particular packet filter is intended to run typically provides the motivation for the structural design of the algorithm. Packet filters have been utilised in a variety of both general and specialised hardware contexts, most notably on CPUs (Central Processing Units), NPUs (Network Processing Units),

TCAM (Ternary Content-Addressable Memory) and FPGAs (Field Programmable Gate Arrays) [10, 11, 15, 16, 21]. Each platform provides distinct benefits and weaknesses, which are capitalised on and mitigated respectively within the algorithms design. Of these platforms, FPGAs are conceptually closest to that of discreet GPUs, as both are highly parallel co-processors which execute hardware specific kernels to accelerate computational tasks. These similarities are however largely cosmetic, as GPUs employ a fundamentally different programming abstraction and hardware design, with strengths and limitations inconsistent with FPGAs [7]. Furthermore, the majority of FPGA algorithms are routing algorithms, and do not map well to general header processing [4, 10, 12, 14].

For these reasons, we have opted to design an algorithm targeted specifically for GPU co-processors — so as to take full advantage of the hardware platform — rather than port an existing algorithm poorly matched to the target hardware. We have, however, taken inspiration from many diverse classifiers in order to ensure optimum performance. As discreet GPUs are cheap, massively parallel commodity hardware, they are well suited to general purpose packet classification. In the following section, we give a brief introduction to GPU programming in CUDA.

3. CUDA PROGRAMMING

In this section we consider the design of GPU hardware, and use this to provide an overview of the CUDA processing abstraction. We shall focus on the Nvidia Geforce 200 series, and refer explicitly to the GTX 280, as these cards are powerful, affordable, and far more common than Nvidia 400 series GPUs, which utilise the newer Fermi architecture .

3.1 GPU Hardware

Graphics Processing Units (GPUs) are massively parallel co-processors, and consist of hundreds of processing cores and substantial volumes of memory, which have traditionally been leveraged exclusively for real-time multimedia applications such as computer games and graphical simulations. Recently, due to the adoption of programmable shaders by device manufacturers, general processing on GPU hardware (referred to as GP-GPU) has become a viable alternative to sequential CPU processing.

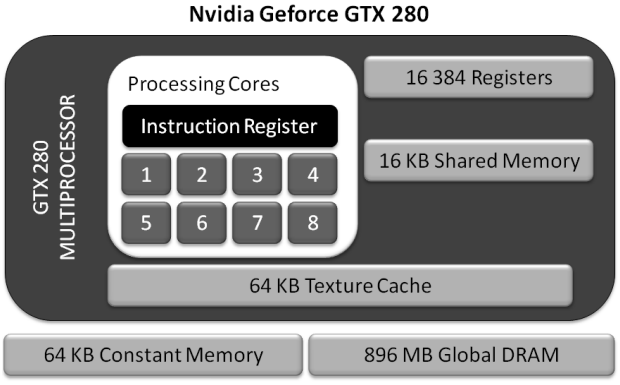

Figure 2: The Nvidia Geforce GTX 280 [3].

A GPU is essentially comprised of a collection of multiprocessors, each driving eight distinct processing cores, and a sizable volume of globally accessible DRAM [3]. The GTX 280 contains 30 multiprocessors driving 240 processing cores between them, and 896 MB GDDR3 device memory [3] (see Figure 2). Each multiprocessor provides a shared instruction register, 16KB of low latency shared memory (which acts as an explicit cache) as well as 16 384 32-bit registers stored in a register file [3]. Each multiprocessor also provides 64 KB of on-chip texture cache, which can be used to reduce device memory access latency in some situations [2]. Global memory is supplemented by 64KB of low latency constant memory which is readable by all multi-processors, but can only be written to by the host thread [3].

The GPU co-processor is controlled by a thread executing on the host through the CUDA Run-time API. The host thread can schedule the transfer of data to and from the device, bind device memory to texture cache, and schedule the execution of kernels [3].

3.2 Programming Model

A CUDA program is known as a kernel. Kernels encapsulate all CUDA device-side processing, in a similar manner to how the main method encapsulates a C++ application [3]. Kernels typically process data transferred to the device through the PCI-E bus, and write the results of processing to a device side output array. This output can then be transferred back to the host through a device-to-host memory transfer after kernel completion.

Kernels execute a collection of threads, typically over a region of device memory, with each thread computing a result for a small segment of data [3]. In order to manage thousands of independent threads effectively, kernels are partitioned into thread blocks, with each thread block being limited to a maximum of 512 threads [3]. Thread blocks are conceptually positioned within a one- or two-dimensional Grid which may contain thousands of thread blocks. Each thread is aware of its own position within its Block, and its Block's position within the Grid. Thus, each thread can calculate — through an application specific algebraic formula — which elements of data to operate on, and which regions of memory to write output to [3].

Each block is executed by a single multiprocessor, which allows all threads within the block to communicate through on-chip shared memory. While thread blocks can contain anywhere between 1 and 512 threads, compute capability 1.3 multiprocessors are capable of context switching between 1024 active threads at one time. Thus, a single multiprocessor can execute multiple blocks simultaneously, up to a maximum of $n = floor(1024/BlockSize)$ [2, 3]. 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.

In the following section we discuss several relevant performance considerations which must be addressed in order to achieve maximum classification throughput.

4. PERFORMANCE CONSIDERATIONS

CUDA kernels — while providing a massively parallel execution platform — are subject to a wide variety of bottlenecks which, if not avoided, result in significant processing slowdown. The purpose of this section is to introduce those bottlenecks relevant to packet filtering, and describe how they may be avoided.

4.1 Memory Bandwidth

The process of transferring data between host memory and device DRAM is a necessary requirement in all useful kernels. Without this functionality, a kernel would not be able to collect data to process, or communicate computational results to the waiting host process. Unfortunately, memory transfer is a relatively slow process, as it is limited by the bandwidth of the PCI-E bus, and can therefore significantly impact on total processing time if it is not optimised [2]. Page-locked memory performs better than pageable memory, and allows for a number of optimizations, but is a scarce resource and should not be overused [2]. For the purposes of this discussion, we shall ignore pageable memory due to its poor performance [2], and focus explicitly on Page-locked memory. This is possible due to the relative abundance of host memory in modern desktops.

Typically, the host process synchronously transfers data to the waiting device. In synchronous transfer, the host process only regains control after all memory has been transferred, and can thus only execute a kernel after the transfer completes [3]. Of course, as most kernels operate on only a subset of the input data (and could thus potentially begin executing select threads prior to transfer completion), synchronous transfer often results in wasted processing time [2]. By taking advantage of both asynchronous transfer of page-locked memory and Streams, however, it is possible to begin executing a kernel on a subset of the data prior to the completion of the entire transfer process [2]. When using asynchronous transfer, control returns to the host process immediately after a memory transfer is scheduled; this allows kernels to be scheduled (but not necessarily executed) prior to transfer completion, and opens the door to asynchronous kernel execution through the use of streams [2, 3].

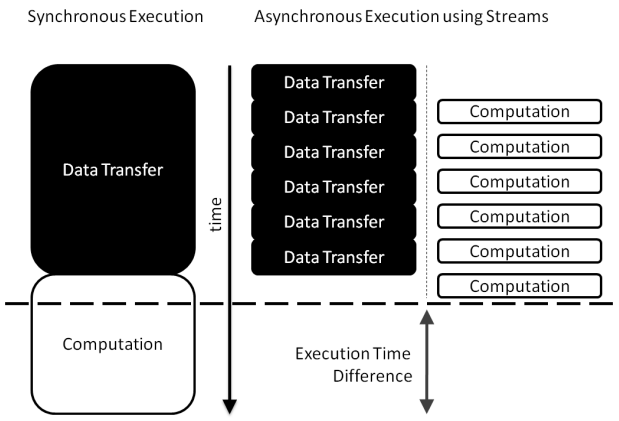

Figure 3: Synchronous and Asynchronous Transfer [2].

Kernels support multiple streams of execution, which effectively allow a single kernel to be invoked multiple times with separate input parameters [2]. Through streams, it is possible partition the data between each of the streams, and schedule each stream to execute the kernel when their prerequisite data has completed transfer. Thus, one stream can transfer data while another stream executes a kernel, allowing transfer and execution to overlap [2] (see Figure 3).

The rate at which data can be transferred to the device can also be improved by employing Write-Combined Memory [2]. In contrast to standard page-locked memory transfers, write-combined memory prevents host-side caching, effectively freeing up L1 and L2 resources, and transfers roughly 40% faster over PCI-E [2]. This performance improvement comes at the expense of host-side read and write speed, which is slightly reduced due to the lack of caching.

Alternatively, Memory transfer can be eliminated all together through the use of Mapped Memory [2]. Memory declared as mapped is read directly from host memory, and as such, removes the necessity to explicitly transfer data to device memory [2]. Mapped memory is most useful on integrated GPUs, since both host and device share the same memory, and as such transfer becomes redundant. On discreet GPUs, mapped memory is transferred through the PCI-E bus, introducing a significant bottleneck [2]. Given its usefulness in integrated GPUs it is, however, worth mentioning.

4.2 Memory Access Latency

CUDA devices provide access to several memory regions, each with their own benefits and limitations. Globally accessible memory regions reside in device DRAM, while local variants reside on the multiprocessor chip. This section describes relevant performance considerations regarding these memory variants.

4.2.1 Global Memory

Global memory is the most abundant memory region available on CUDA devices, and is capable of storing hundreds of megabytes of data. Unfortunately, while global memory provides abundant data storage capacity, this comes at the expense of access latency, with individual requests requiring between of 200 and 1000 clock cycles to succeed [2, 3]. As is evident, this introduces a critical bottleneck in kernel execution, which can significantly impoverish the processing throughput in data intensive applications. Fortunately, CUDA devices support Memory Access Coalescing, which effectively combines small global memory requests from multiple threads in a thread warp into a single, multi-thread request [2]. This can greatly improve warp-level access latency, but unfortunately is not always possible to achieve. In a compute level 1.3 device such as the GTX 280, threads in a half-warp will coalesce their memory access if and only if they request data from the same small segment of global memory [2]. In compute capability 1.0 - 1.2 devices, coalescing only occurs if sequential threads access sequential memory elements [2], and is thus even more difficult to achieve.

4.2.2 Constant Memory

Constant memory is a small read-only region of globally accessible memory which resides in device DRAM [3]. 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 [2]. 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 [2]. Due to its limited size however, its use is significantly limited.

4.2.3 Texture Memory

Texture memory essentially provides a compromise between global and constant memory. Each multi-processor on the CUDA device contains a 64KB texture cache which can be bound to one or more arbitrarily sized region of global memory [2, 3]. As a result, texture bound memory performs consistently, with roughly the same performance of fully coalesced global memory, making it ideal for accelerating data access [2]. Texture memory, like constant memory, is read only, and thus only provides performance benefits with regard to memory reads, and cannot be leveraged to accelerate global memory writes [3].

4.2.4 Registers

Registers are contained within a register file on each multi-processor [3], and provide fast thread-local storage during kernel execution. In compute capability 1.3 devices, each multi-processor contains 16 384 registers [3], which are shared between all threads in the executing thread block. Registers are typically accessed with zero added clock cycle overhead, but may incur a slight performance penalty due to read-after-write dependencies and register bank conflicts [2]. Executing threads have no direct control over register allocation, and hence have little control in avoiding register bank conflicts. By ensuring that thread blocks contain a multiple of 64 threads however, it is possible to improve the chances of avoiding register bank conflicts and minimizing access latency [2]. Read-after-write dependencies, on the other hand, have a latency of 24 clock cycles per occurrence, but this overhead is completely hidden in blocks containing more than 192 threads [2]. Thus, registers perform best when the executing block has a thread count that is both greater than or equal to 192, and is a multiple of 64.

While each multiprocessor has only 16 384 registers available, kernels do not fail to execute when the blocks executing on a multiprocessor exceed this limit [3]. Once the register file is exhausted, the multiprocessor allocates register storage on device DRAM. As such, kernels requiring more than 16 384 registers per multiprocessor will execute correctly, but will incur a significant performance penalty due to the high latency of DRAM access [3]. As such, register utilization should be minimised in order to ensure maximum execution speed.

4.2.5 Shared Memory

Unlike register memory, shared memory is block-local, facilitating cooperation between multiple threads in an executing block [3]. Shared memory is limited to 16KB of storage per multi-processor on compute capability 1.3 devices [3] and, as multiple blocks may be executing on a single multi-processor, is a severely limited resource. Nevertheless, as long as no shared memory bank conflicts arise [2], access latency is equivalent to that of register memory.

In compute capability 1.3 devices, each multi-processor's shared memory is divided between 16 separate 1KB memory banks [2]. A bank conflict arises when two separate threads in a half-warp access the same memory bank at the same time, in which case the request is split into as many conflict free memory requests as possible [2]. As a result, if care is not taken, shared memory performance can be significantly impoverished. As long as each thread in a half-warp only accesses a single consecutive shared memory address however, bank conflicts are entirely avoidable.

4.3 Warp-level Thread Divergence

Conceptually, kernels support a parallel execution model called SIMT [3], or Single Instruction, Multiple Thread. This model allows threads to execute independent and divergent instruction streams, facilitating decision based execution which is not provided for by the more common SIMD (Single Instruction Multiple Data) execution model. SIMT is limited however, as each physical multi-processor contains only a single instruction register which drives eight independent processing cores simultaneously. Thus, any divergence between threads executing on the same multiprocessor forces the instruction register to issue instructions for all thread paths sequentially whilst non-participating threads sleep [2, 3]. Furthermore, each processing core can issue a single instruction to four distinct threads in the time between each instruction register update, giving a total of 32 threads executing a single instruction. Thus, significant thread divergence within the cluster of the 32 threads executing on a multiprocessor, termed a Thread Warp [3], can dramatically impair performance [2].

4.4 Operator Performance

CUDA Kernels are expressed using C'99 syntax extended for parallelism, and as such facilitate all requisite bit wise, algebraic, comparative and assignment operators [3]. Most operators perform relatively well, with the exception of integer division and modulo operations which are significantly more expensive [2]. The cost of these operations can be avoided in cases where the divisor or modulus is power of two, as the operations can easily be translated into efficient bit shift and bit wise operations respectively [2]. For instance, if $k = 2^n$ where $n \geq 1$, then $x/k = x \gg log_2 k$, and $x\%k = x\&(k-1)$, where $x \epsilon \mathbb{Z}$ [2]. As $log_2 k$ and $k-1$ can often be calculated at design or compile time, both these methods generally require execution of only a single bit-shift or bit-wise AND in order to determine a result.

Now that we have considered the CUDA bottlenecks relevant to packet filtering, we shall describe our parallel algorithm.

5. PARALLEL PACKET FILTERING

Packet filtering is, in its simplest form, a highly parallelisable task which repeatedly compares the same filter set against millions, if not billions, of packets[19]. Thus, at least conceptually, a GPU based packet classification implementation should improve packet classification throughput by several orders of magnitude. Unfortunately, the optimisations employed by previous packet filters are optimised for an entirely different hardware context [5, 8], and do not map well to GPU implementation. In the following subsection we consider this problem, so as to better illustrate the need for an alternative approach. We follow this discussion with a high-level overview of the gPF classification algorithm, and detail the primary structural optimisations employed in each of its components.

5.1 Limitations of Control Flow Graphs

General host-side packet filters — such as those employed in demultiplexing and analysis — have traditionally employed variations of CFGs to direct packet header processing, due to their optimisation potential within a sequential context [5, 8, 13]. This optimisation potential results from the iterative redundant comparison pruning that CFGs provide. Each comparison performed against a header field affects the filter execution path, allowing a filter to direct thread execution down an approximate minimal path, culling unnecessary comparisons at each step. However, given the severe performance penalties incurred from thread divergence in CUDA kernels [2], coupled with the necessity for such divergence in a CFG implementation, it follows that a CFG based-algorithm is ill-adapted to GPU processing.

To illustrate the problem, we shall consider two alternative high-level CFG approaches to GPU based packet filtering. The first and most obvious approach is to allow each thread to traverse the filter set CFG in order to classify a single packet. The flaws in this approach are instantly apparent. As we cannot predict in what order packets will arrive, or which filters they will match, individual threads in a thread warp cannot be guaranteed to follow the same execution path. As divergent execution is serialised within a thread warp [2], it follows that any significant divergence within a thread warp dramatically reduces classification rates, as a significant proportion of executing threads are forced to sleep while traversing divergent instruction streams.

An alternative solution, inspired by the Grid-of-Tries IP routing algorithm [17], is to process the CFG in multiple phases by dividing it into sub-trees (see figure 4). An initial CFG performs the first phase of classification and divides packets into one of several preliminary types. The packet's preliminary type indicates which CFG it should traverse in the next phase of processing, which allows us to cluster similar packets into the same executing thread block when the next phase begins. This has the effect of reducing the number of possible divergent paths within each warp and thus improves overall performance. The more classification phases employed by a filter, the less divergence potential each phase presents. Unfortunately, such a technique only serves to reduce divergence, and its effectiveness is less apparent when employing large filter sets with hundreds of possible execution paths. Decomposition of this nature is certainly preferable to a single CFG approach, but fails to mitigate divergence overhead entirely, particularly when dealing with large filter sets.

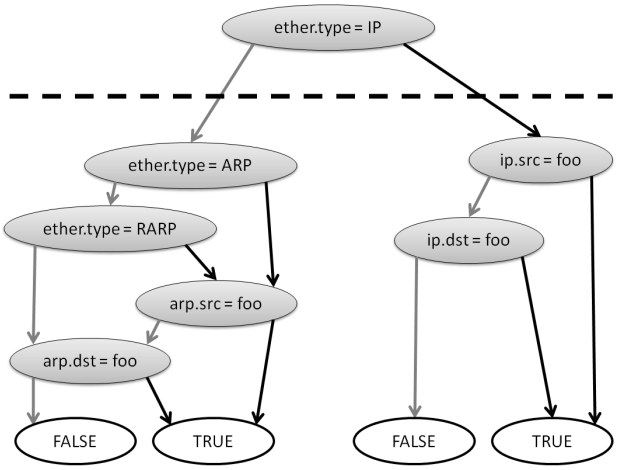

Figure 4: The CFG in figure 1 split over two phases.

On a final note, while CFG based filters have proven useful in packet analysis due to their superior throughput on CPUs, they are not ideally suited to this task, as they can only match a packet to a single filter [5, 8, 13, 19, 23]. While sufficient in IP routing and Demultiplexing domains, which aim only to direct packets to their final destination[18, 19], a best matching filter introduces a fairly significant limitation when considering analysis. This limitation arises when a packet matches two distinct filters. In such an instance, the classification which is performed first identifies the packet, whilst all other possible matches are essentially lost. As filter hiding can be mitigated through careful filter construction, has relatively minor impact on small filter sets, and is difficult to avoid in CFG approaches, this limitation is largely ignored. However, when comparing packets against large filter sets, mitigating filter hiding becomes more difficult, and any minor errors made by the filter designer, or any instances of unavoidable non-trivial overlap, may result in lost information. As the intention of packet analysis is to illuminate network traffic composition, it follows that a thorough analysis should be exhaustive, so that the results of comparison are not skewed or hidden by poor filter set construction or any assumptions made by the filter set designer.

As such, an alternative, exhaustive approach is warranted, which minimises thread divergence in each thread warp and capitalises on CUDA optimisation strategies in order to maximise classification throughput. In the following subsections, we will discuss the design of a CUDA based packet classifier which leverages this knowledge to maximise classification performance.

5.2 Parallel Classification Algorithm

As has been noted, CFG based classification is ill-suited to GPU processing. In this section, we propose an alternative algorithm, called gPF, which is optimised specifically for

execution on CUDA devices. Before considering specific optimisations, we shall provide a brief overview of, and provide motivation for, the structure of the algorithm.

One of the primary design goals of our parallel classification algorithm was to avoid warp-level thread divergence all together and thus eliminate all divergence overhead [2]. Unfortunately, entirely eliminating all thread divergence requires abandoning all optimisations relying on run-time observations, and thus cannot exploit any criteria pruning techniques used by earlier header harvesting filters. Due to the Match Condition Redundancy property of filter sets however, we note that the set of possible classification criteria is small, and remains small even when filter sets grow extremely large [19]. We may therefore perform an exhaustive evaluation of each packet against the set of all unique criteria without incurring significant penalties, and subsequently eliminate redundant criteria evaluation. Such an approach allows us to eliminate thread divergence entirely, and provides the added benefit of supporting exhaustive filter evaluations. This step is performed by a separate CUDA kernel, called the Rule kernel.

After a packet has been compared against all criteria, the boolean results are used to evaluate the filter predicates. To this end, we observe that in some instances, several filters may share one or more common sub predicates (such as Type = IP AND Protocol = TCP). In order to avoid redundant computation wherever possible, the filter may optionally perform an intermediate Subfilter evaluation, and store the results with those collected from the previous criteria comparison step, for use during the final filter predicate evaluation. This step is optional. It is only applicable when such a redundancy is apparent and only provides significant benefit when that redundancy is relatively prominent, and occurs in several filters. In cases where several filters evaluate a common sub-predicate, performing an intermediate evaluation of the subfilter ensures the predicate is only evaluated once, and all instances of the sub-predicate are replaced by a single memory access in the classifying filters. The final filter evaluation is exhaustive, outputting a boolean array of per packet filter classification results which may be retrieved by the host.

In the following subsections, we consider the filter algorithm in detail, with specific reference to the hardware optimisations leveraged.

5.3 Mitigating Data Starvation

Before classification can be carried out, packets must first be transferred to the device. Packets may arrive for processing via two distinct mediums: either they are captured from a network interface (a live capture), or they are read from a packet dump file held in long term storage. We shall focus on packet dump file processing, while noting that the techniques discussed may easily be extended to support live packet captures.

First, it is worth addressing the most notable bottleneck in packet transfer; Disk I/O. SATA II hard disks are the dominant storage medium in modern desktop systems, and offer a maximum read speed of 3 Gbps; they typically average around 100MBps — around 30% of the maximum value.. In contrast, a PCI-E 2 bus can transfer between host and device memory at up to 8Gbps [2]. We shall discuss briefly three potential solutions to mitigating this bottleneck, as well as their negative implications. Firstly, utilizing a faster long-term storage medium, such as a SCSI disk, could potentially alleviate much of this performance gap, but the necessity for special hardware reduces its effectiveness in the general case. Striping data over multiple hard drives provides a potentially less expensive alternative, but requires multiple, appropriately formatted drives as a prerequisite. Finally, RAM Disks, which reside in host memory and may be created on any modern PC, can provide extremely fast access speeds, but are dependent on host memory for storage, thus limiting their capacity and making them infeasible in systems with limited Host RAM. Due to the flaws inherent in each of these solutions, selection should be based on particular circumstance. The medium selected has no structural impact on the algorithm, and thus can be left to the user.

We shall now consider a mechanism for minimizing the volume of data to be transferred over the PCI-E bus, so as to reduce transfer times and minimise memory requirements on the device. When classifying packets, a subset of the packet data, contained within the packet header, are compared to set of target values to produce a boolean result [5, 19]. These boolean values are combined through boolean algebra to define a filter, which succeeds in classification if the filter predicate returns true. As filters typically comprise relatively few comparisons, and noting that the number of distinct comparisons in a filter typically far outnumber the number of distinct data elements they collectively test [19], it is possible to reduce packets by cropping unnecessary data during collection from storage. By determining which bytes in a packet are necessary for classification during the filter compilation stage, unused bytes may be skipped. The filter compiler need then only adjust the byte indexes to be tested in the packet data array, ensuring that the correct values are referenced. This mechanism should also improve packet collection performance, as less data need be communicated over disk I/O.

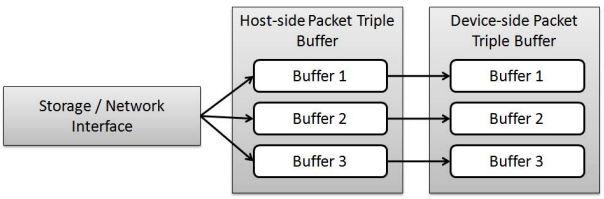

Figure 5: Packet collection triple buffering

While packet reduction improves the rate at which packets can be transferred between long term storage and the CUDA device, it does not provide a mechanism to ensure that the device is not starved of packet data while packets are collected from disk. Typically, CUDA memory transfers are synchronous, and prevent the host from executing any operations until such time as the transfer is complete [3]. An acceptable strategy in dealing with this problem involves intelligent utilization of asynchronous transfer and packet buffers (see figure 5). Firstly, packet data is copied from long-term storage into a triple buffered staging area executing in a separate host-side processing thread. The child thread continuously fills empty buffers, while the parent process responsible for kernel execution transfers full buffers onto the device through asynchronous write-combined memory transfers, using a

separate processing stream for each buffer, or if necessary, multiple streams per buffer. Furthermore, as only the first phase of our classification algorithm actually accesses this packet data, buffers may begin refilling as soon as this step completes, in parallel with the remainder of classification. This should ensure that periods of data starvation are minimised [2, 3].

5.4 Eliminating Divergence Overhead

As discussed in section 5.1, classification methods which depend on divergent execution paths (such as CFGs) are not particularly useful in the context of GPUs [2]. As a result, it is difficult to efficiently exclude certain classifications from processing based on run-time observations of data. We can however exploit the Match Condition Redundancy property of filter sets, which states that, given a set of filters, the total number of unique match conditions in a filter set is significantly less than the number of distinct filters in that set. We can thus pre-compute all match criteria for each packet in a separate kernel, called the rule kernel, and store the results of these comparisons in device memory. As all packets are exhaustively compared against all match conditions, no threads need diverge, providing for efficient processing [2].

After the rule kernel completes execution, an intermediate subfilter kernel is optionally launched, which operates on the boolean results of the previous classification step. Each subfilter is a predicate whose result is calculated by a single thread, and stored as a boolean value in device memory. As with the rule kernel, subfilters are exhaustively calculated for each packet, with each thread classifying multiple subfilters. This ensures that each thread evaluates the same predicate as its warp-level siblings, eliminating divergence all together. The results of both the rule kernel and the subfilter kernel are then used in a final filter kernel, which operates similarly to the subfilter kernel, but the results of which constitute the final classification of each packet with respect to each filter.

While it is possible to perform all these steps within a single kernel, such a design would not allow for kernel grid dimensions to be optimised for each individual step, and would also prohibit binding intermediate results to texture references to improve memory access speeds [3], prevent refilling an exhausted buffer midway through kernel execution, which is important in mitigating starvation. Thus, a multi kernel approach provides for greater flexibility in optimizing classification speeds, making it more attractive than a single kernel implementation.

5.5 Reducing Memory Access Latency

In order to approach optimal classification efficiency, the algorithm has been optimised to promote minimal access latency when threads access multiple memory regions. In this subsection we describe the memory utilisation of each kernel, and the performance benefits they provide.

The Rule kernel essentially compares a set of bit ranges within a packet to a set of target values, and stores the results in device memory. As the incoming packet data is read only once, and is never modified, it may be bound to a texture reference in order to exploit the texture cache. Each block in the rule kernel contains exactly 256 threads, thus mitigating register bank conflicts as well as hiding register read after write dependencies [2]. Each thread is responsible for classifying up to 16 criteria, or *rules*, where the set of criteria, or *rule set*, to be evaluated is stored in constant memory so as to ensure fast access. If the number of rules contained in the rule set exceeds 16, then rules are divided over multiple threads, partitioned appropriately into distinct thread warps so as to prevent any thread divergence. Packet data is read iteratively through texture fetching, with each distinct data element being extracted into a temporary register and compared to its associated target. The results of each comparison are stored in shared memory, which provides 16 bytes of storage per thread. Thus, a single block will contain 256 threads, which collectively consume 4 KB of shared memory. In compute capability 1.3 devices, each multiprocessor supports a maximum of 1024 threads, 16 KB of shared memory, and 16 384 registers. Thus, four blocks may execute simultaneously on a multi-processor at once, and each block can utilise a maximum of 16 registers without incurring a performance penalty [2].

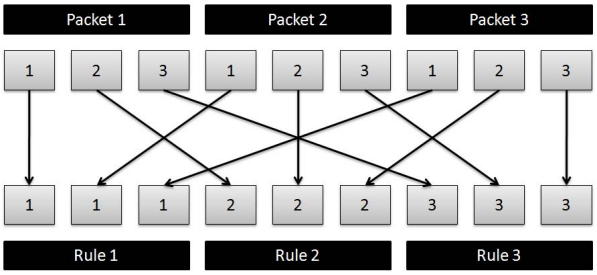

Figure 6: Optimised Memory Layout for coalescing.

Once all rules have been compared, results stored in shared memory are copied into device memory. In order to optimise for coalescing, the results are grouped by rule rather than by packet, with the first n bytes containing the result of the first rule comparison for all n packets, the second n bytes containing the results of the second rule comparison for all n packets etc. (see figure 6). As all threads will write the results of rules sequentially, this memory layout ensures that threads can coalesce while reading and writing to global memory.

Both the subfilter and filter kernels read in boolean values from device memory and compute the result of a predicate equation. Boolean values arrive optimised for coalescing, thus facilitating fast memory reads. For improved performance, these regions may be bound to a texture reference prior to the kernels execution, so as to exploit the texture cache [2]. Again, kernels execute in blocks of 256 threads, with each thread allocated 16 bytes of shared memory to store the results of 16 filter or subfilter predicates. If more than 16 predicates need to be evaluated, they are divided between multiple threads in a manner similar to the rule kernel. This ensures that no thread diverges within a warp, as each packet will be compared against a constant set of predicates, which are stored as commands in constant memory.

The architecture described is optimised for execution on Compute Capability 1.2-1.3 devices, but must only consume 8 registers per thread to achieve full thread occu-

pancy on 1.0-1.1 devices. This is due to only 8192 registers being available per multiprocessor in these GPUs.

5.6 Preliminary Results

The packet filter described is currently under development, and thus far shows promise. While it is impossible at this stage to provide scientifically valid comparative performance metrics between gPF and other contemporary general packet filters, we have successfully measured the kernel throughput of several bulk packet classifications and verified results. As contemporary filters typically interleave classification with expensive disk I/O operations, classification throughput results alone are not sufficient for a true comparison, but merely illustrate potential.

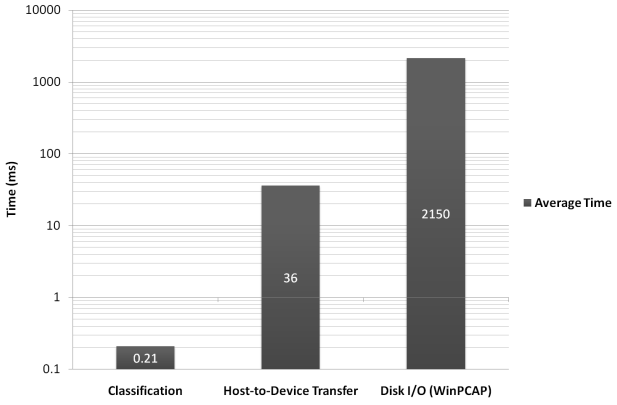

Figure 7: Time taken to classify 3.6 million IP packets using a GTX 275.

Figure 7 shows the amount of time taken to correctly classify 3.6 million packets from a packet dump file, presented using a log base 10 scale. The classification was performed on an Nvidia Geforce GTX 275, using a 1333Mhz DDR3 Ram Disk virtual drive for dumpfile storage — accessed from within the classifier using the WinPCAP C++ API. The figure differentiates between three distinct stages, namely: classification, host-to-device transfer, and dumpfile disk I/O. At this point, the CUDA classification kernel is relatively complete, and as such performs relatively well, classifying 3.6 million packets in roughly 0.21 ms. While mostly implemented, host-to-device transfer optimisations have not yet been incorporated into the classification system. Thus the average transfer timing reported, 36 ms, does not reflect the performance of the final system, and we expect timings for this process to improve by up to an order of magnitude once packet reduction and asynchronous transfers, among other optimisations, are incorporated. The most expensive component with regard to latency at this point, taking around 2150 ms, is the disk I/O necessary to load the packet into host memory. We have thus far been using the WinPCAP API for collecting packet data from disk, which implicitly and transparently adds significant parsing overhead to collection operations. It is our intention to move to a more efficient collection mechanism at a later stage.

These results serve to illustrate the importance of mitigating data starvation. Without due consideration for optimising transfer between long-term storage and the discreet GPU, a significant proportion of time (in this case about 99.99 percent) is spent simply transferring the data to GPU memory for processing. This overhead can mean the difference between minutes and days when processing extremely large packet dump files, such as those collected at network telescopes.

It is worth noting that while this test has been performed using a single GTX 275, classification can easily be extended over multiple GPUs, providing near linear speedup. As CUDA enabled graphics cards are commodity hardware, and cost significantly less than even the most basic FPGA, a multi-GPU classification system provides an inexpensive alternative to specialised hardware implementations.

6. SUMMARY

In this paper, we have detailed a parallel filtering algorithm which capitalises on CUDA architecture to accelerate classification. After briefly introducing the packet filtering domain, we focused on the architecture of CUDA devices and kernels, providing a high level overview of their structure, and detailing relevant performance bottlenecks and optimization opportunities. In particular, we considered the performance of the various memory mediums available to CUDA kernels, and how such performance may be optimised. After considering GPU architecture, we discussed the performance penalties incurred by Control Flow Graphs, leveraged by most desktop packet filters [5, 13], when implemented on GPUs. This brought us to the conceptual design of a CUDA classifier, gPF, which applied the optimization knowledge previously detailed so as to maximise performance. The section enumerated several design choices regarding packet collection, buffering and host-to-device memory transfer acceleration, as well as device-side storage and execution optimization. We concluded by providing some preliminary results, demonstrating the potential for the algorithm.

By harnessing the power of modern GPUs to perform packet classification, we hope to make real-time packet composition analysis viable in the domains of network research, security and fault diagnosis. In particular, we aim to apply GPU based filtering to real-time giga-bit network visualisation and fast network telescope data analysis, applications which we believe to be both highly desirable and within reach.

7. REFERENCES

[1] The opencl specification, version 1.0. Online, April 2009.

[2] Nvidia cuda c best practices guide, version 3.1. Online, May 2010.

[3] Nvidia cuda c programming guide, version 3.1. Online, May 2010.

[4] F. Baboescu and G. Varghese. Scalable packet classification. *SIGCOMM Comput. Commun. Rev.*, 31(4):199–210, 2001.

[5] A. Begel, S. McCanne, and S. L. Graham. Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture. *SIGCOMM Comput. Commun. Rev.*, 29(4):123–134, 1999.

[6] H. Bos, W. D. Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. Ffpf: Fairly fast packet filters. In *In Proceedings of OSDI04*, pages 347–363, 2004.

[7] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. Online, 2008.

[8] D. R. Engler and M. F. Kaashoek. Dpf: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 53–59, New York, NY, USA, 1996. ACM.

[9] S. Ioannidis and K. G. Anagnostakis. Xpf: Packet filtering for low-cost network monitoring. In *In Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR*, pages 121–126, 2002.

[10] W. Jiang and V. K. Prasanna. Field-split parallel architecture for high performance multi-match packet classification using fpgas. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 188–196, New York, NY, USA, 2009. ACM.

[11] W. Jiang and V. K. Prasanna. Large-scale wire-speed packet classification on fpgas. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 219–228, New York, NY, USA, 2009. ACM.

[12] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev.*, 28(4):203–214, 1998.

[13] S. Mccanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. pages 259–269, 1993.

[14] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 213–224, New York, NY, USA, 2003. ACM.

[15] H. Song and J. W. Lockwood. Efficient packet classification for network intrusion detection using fpga. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 238–245, New York, NY, USA, 2005. ACM.

[16] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended tcams. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, page 120, Washington, DC, USA, 2003. IEEE Computer Society.

[17] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28(4):191–202, 1998.

[18] W. R. Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[19] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, 2005. http://doi.acm.org/10.1145/1108956.1108958.

[20] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] K. Vlaeminck, T. Stevens, W. Van de Meerssche, F. De Turck, B. Dhoedt, and P. Demeester. Efficient packet classification on network processors. *Int. J. Commun. Syst.*, 21(1):51–72, 2008.

[22] Z. Wu, M. Xie, and H. Wang. Swift: a fast dynamic packet filter. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 279–292, Berkeley, CA, USA, 2008. USENIX Association.

[23] M. Yuhara, B. N. Bershad, C. Maeda, J. Eliot, and B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *In Proceedings of the 1994 Winter USENIX Conference*, pages 153–165, 1994.