Conceptual Design of a CUDA based packet classifier

Alastair Nottingham[1] and Barry Irwin[2]Security and Networks Research Group
Department of Computer Science
Rhodes University
E-Mail: [1]anottingham@gmail.com [2]b.irwin@ru.ac.za

***Abstract*—Packet classification and analysis is an important task in network security, which typically relies on a flexible packet filtering system to extrapolate important packet information from each processed packet. This task is computationally intensive, if highly parallelisable, and as such, classification of large packet sets, such as those collected by a network telescope, can require significant processing time in sequential environments. In order to accelerate packet classification to facilitate real-time classification of giga-bit network traffic, we aim to exploit the inherent parallelism of packet filtering through the use of CUDA enabled GPUs. In this paper, we introduce a variety of relevant optimizations and bottlenecks in CUDA architecture, and apply this knowledge to the conceptual design of a highly parallel GPU packet classifier, optimised for efficient execution on CUDA hardware. In particular, we focus on accelerating memory transfer between long term storage and the CUDA device, minimizing memory access latency on the device, and maximizing processing throughput within each CUDA kernel.**

***Index Terms*—Packet Classification, GP-GPU, CUDA**

I. Introduction

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 [1]. 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 [1], [2], [3], a repetitive procedure which is ideally suited to parallel processing. Unfortunately, due to the 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 [1], [3], [4]. Despite numerous optimizations however, sequential 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 CUDA (Compute Unified Device Architecture) enabled GPU (Graphics Processing Unit) coprocessors 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 is necessary to devise a suitable algorithm, tailored to GPU architecture, and optimised to ensure maximum utilization of GPU resources. Unlike sequential demultiplexing algorithms which search for the first matching filter [2], [3], [4], [5], the algorithm discussed is exhaustive, 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 [6].

In section II, we provide a brief introduction to GPU architecture, and discuss some useful observations and methods for improving kernel and memory performance. In section III we apply this knowledge to the conceptual design of a CUDA kernel, optimised to maximise filter throughput, thus minimizing execution time. Finally, in section IV, we provide a summary.

II. Overview of GPU Architecture

GPU architecture has a significant impact on the effectiveness of data processing strategies which could be leveraged to process packets efficiently [7]. Before addressing the limitations of GPU architecture with respect to individual algorithmic scenarios, it is first necessary to understand the broader benefits and weaknesses of GPU architecture which impose these limitations. This section briefly considers some important considerations in algorithm design.

A. Kernel Execution

A CUDA Kernel is a function which is executed by the host process on the CUDA device, and essentially encapsulates a CUDA program or procedure [8]. Simply put, 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. In order to manage thousands of independent threads effectively, kernel threads are divided into thread blocks, with each thread block being limited to a maximum of 512 threads [8]. Thread blocks are conceptually positioned in a Grid which may contain thousands of thread blocks. Each thread is aware of its own position within its Block, its Block's position within the Grid. Thus, each thread can calculate, through an

The authors would like to acknowledge the financial support of Telkom SA, Comverse, Stortech, Amatole Telecom Services, Bright Ideas 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

application specific algebraic formula, which elements of data to operate on, and which regions of memory to write output to [8].

Conceptually, kernels support a parallel execution model called SIMT [8], 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 multiprocessor 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 [8], [7]. 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 [8], can dramatically impair performance [7].

CUDA Kernels are expressed using C'99 syntax, and as such facilitate all requisite bit wise, algebraic, comparative and assignment operators [8]. Most operators perform relatively well, with the exception of integer division and modulo operations which are significantly more expensive [7]. 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 and operations respectively [7].

B. *Memory*

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.

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 [8], [7]. 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 [7]. This can greatly improve warp-level access latency, but unfortunately is not always possible to achieve. In a compute level 1.3 device, 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 [7]. In compute capability 1.0 - 1.2 devices, coalescing only occurs if sequential threads access sequential memory elements [7], and is thus even more difficult to achieve.

2) Constant Memory: Constant memory is a small read-only region of globally accessible memory which resides in device DRAM [8]. 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 [7]. 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 [7]. Due to its limited sise however, its use is significantly limited.

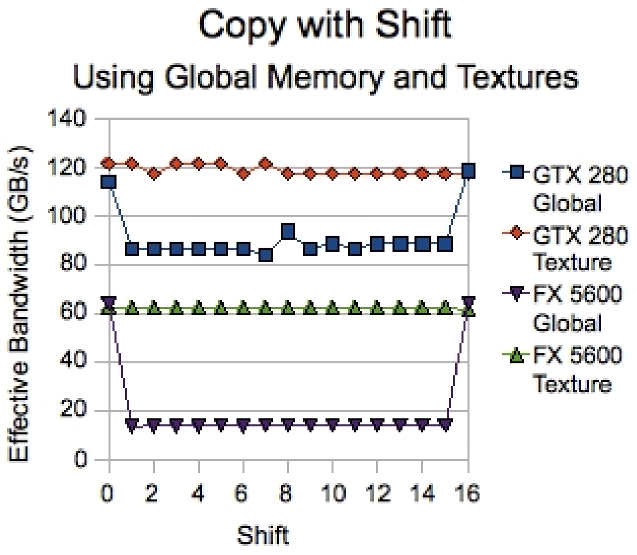

Figure 1. Avoiding coalescing constraints using texture memory [7].

3) Texture Memory: Texture memory essentially provides a compromise between global and constant memory. Each multiprocessor on the CUDA device contains a 64KB texture cache which can be bound to an arbitrarily sised region of global memory [8], [7]. As a result, texture bound memory performs consistently, and faster than coalesced global memory (see figure 1), making it ideal for accelerating data access [7]. 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 [8].

4) Registers: Registers are contained within a register file on each multi-processor [8], and provide fast thread-local storage during kernel execution. In compute capability 1.3 devices, each multi-processor contains 16 384 registers [8], 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 [7]. Executing threads have no direct control over register allocation, and as such 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 a register bank conflict [7]. 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 [7]. Thus, registers perform best when the executing block has a thread count that is both greater than 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 [8]. 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 [8]. As such, register utilization should be minimised in order to ensure maximum execution speed.

5) Shared Memory: Unlike register memory, shared memory is block-local, facilitating cooperation between multiple threads in an executing block [8]. Shared memory is limited to 16KB of storage per multi-processor on compute capability 1.3 devices [8] and, as multiple blocks may be executing on a single multi-processor, is a severely limited resource. Never the less, as long as no shared memory bank conflicts arise [7], 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 [7]. 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 [7]. 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 avoided.

6) Memory Transfer: 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 PCIe bus, and can therefor significantly impact on total processing time if it is not optimised [7]. Page-locked memory performs better than pageable memory, and allows for a number of optimizations, but is scarce resource and should not be overused [7]. For the purposes of this discussion, we shall ignore pageable memory due to its poor performance [7], and focus explicitly on Page-locked memory.

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 [8]. 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 [7]. 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 [7]. When using Asynchronous transfer, control returns to the host process immediately after a memory transfer is scheduled, which allows kernels to be scheduled (but not executed) prior to transfer completion, and opens the door to asynchronous kernel execution through the use of streams [8], [7].

Kernels support multiple streams of execution, which effectively allow a single kernel to be invoked multiple times with separate input parameters [7]. Through streams, it is possible to 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 [7].

The rate at which data can be transferred to the device can also be improved, by employing Write-Combined Memory [7]. 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 PCIe [7]. 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 [7]. Memory declared as mapped is read directly from host memory, and as such, removes the necessity to explicitly transfer data to device memory [7]. 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 PCIe bus, and as such, can introduce significant bottlenecks [7]. Given its usefulness in integrated GPUs it is however worth mentioning.

III. Classifier Design

The architectural specifics of CUDA devices imposes certain constraints on the design of a GPU based packet classification tool. In this section we discuss the limitations of the canonical control flow graph approach with respect to a CUDA enabled implementation, and suggest an alternative processing strategy to maximise performance.

A. Contemporary Packet Filters

In modern network environments, packet filtering tasks have become relatively numerous, and typically fall into one of several categories. These include IP Routing algorithms [9], [10], [11], [12], which operate exclusively over the IP 5-tuple, Deep Packet Inspection algorithms [13], [14], which operate over packet payloads, and Demultiplexing algorithms, used in both end-point demultiplexing and packet header analysis.

The proposed packet filter implements a demultiplexing algorithm, as they provide for sufficient generality through protocol indepence, providing the necessary flexibility to support classification of arbitrary protocols. Unfortunately, the majority of modern demultiplexing filters derive their architecture from BPF or BPF+ [2], [3], [4], [5], [15], [16], [17], [18], which were developed to maximise classification speeds on sequential desktop processors (see Section III-B). Furthermore, due to the high traffic volumes transferred over modern networks, demultiplexing algorithms have become increasingly reliant on fairly low-level optimisations in order to maximise throughput [15], [16], [18]. Due to the significant architectural differences and divergent performance considerations of GPUs, CPUs and

FPGAs (Field Programmable Gate Arrays), such specialisation significantly reduces algorithm portability.

As result of these limitations and the numerous restrictions imposed when optimising architecture for GPU based execution, existing algorithms prove inadequate in a highly optimised CUDA context. In the following subsection, we detail the primary limitation of modern demultiplexing algorithms executing on GPU coprocessors.

B. Limitations of Control Flow Graphs

Control Flow Graphs, or CFG's, are a decisional tree structure used as the conceptual foundation of packet filters such as BPF, MPF, DPF, and BPF+ [2], [3], [4], [5], as well as xPF, FFPF and SWIFT [15], [16], [18]. Their effectiveness is due to their susceptibility to optimization techniques such as partial redundancy elimination and predicate assertion propagation, which greatly reduce redundant processing and allow a sequential processor to classify packets in a minimal time [3]. While a CFG approach is certainly valid, if not desirable, in the context of a sequential packet processor, the severe performance penalties incurred from thread divergence in CUDA kernels [7], coupled with the necessity for such divergence in a CFG implementation, make such a technique infeasible to use in a CUDA context.

As such, an alternative approach is needed which minimises thread divergence in each thread warp and capitalises on CUDA optimization 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.

C. Packet Transfer

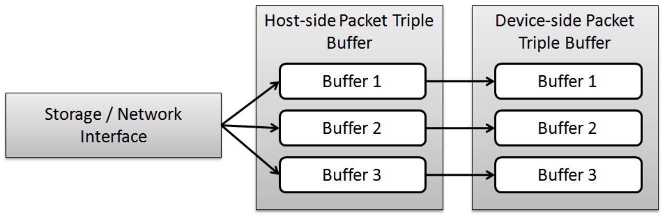
Host-side Packet Triple Buffer
Device-side Packet Triple Buffer
Storage / Network Interface
Buffer 1
Buffer 2
Buffer 3
Buffer 1
Buffer 2
Buffer 3

Figure 2. Packet collection triple buffering

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 help 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, namely Disk I/O. SATA II hard disks are the dominant storage medium in modern desktop systems, and offer 300 Mbps read speeds. In contrast, a PCIe 2 bus can transfer between host and device memory at up to 8Gbps [7]. We shall briefly discuss three potential solutions to mitigating this bottleneck, as well as their negative implications. first, utilizing a faster storage medium 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.

We shall now consider a mechanism for minimizing the volume of data to be transferred over the PCIe 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 a set of target values to produce a boolean result [1], [3]. 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 [1], 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.

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 [8]. An acceptable strategy in dealing with this problem involves intelligent utilization of packet buffers. First, packet data is copied from long term storage into a triple buffered staging area executing in a separate 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. This should ensure that periods of data starvation are minimised [8], [7].

D. Multi-phase Classification Method

As discussed in section III-B, classification methods which depend on divergent execution paths, such as CFGs, are not particularly useful in the context of GPUs [7]. 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 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 [1]. We can thus pre-compute all match conditions 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 [7].

After the rule kernel completes execution, a second subfilter kernel is 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. 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 whose results constitute the final classification of each packet with respect to each filter. This two-step filter evaluation algorithm allows for sub-predicates which occur in multiple filters to be pre-computed to reduce redundant computation.

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 [8]. Thus, a multi kernel approach provides for greater flexibility in optimizing classification speeds, making it the more attractive than a single kernel implementation.

We shall now briefly consider the structure of each of these kernels, with respect to thread counts and memory utilization.

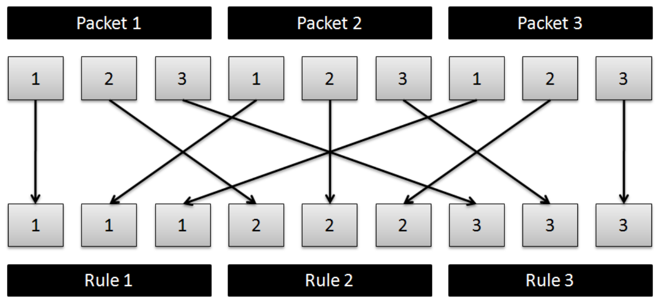

Figure 3. Optimised Memory Layout for coalescing.

1) Rule Kernel: 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 [7]. Each thread is responsible for classifying up to 16 rules, which are stored in constant memory 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 [7].

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 3). As all threads will write the the results of rules sequentially, this memory layout ensures that threads can coalesce while reading and writing to global memory.

2) Subfilter and Filter Kernels: Both the subfilter and filter kernels read in boolean values from device memory and 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 [7]. 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.

E. Preliminary Results

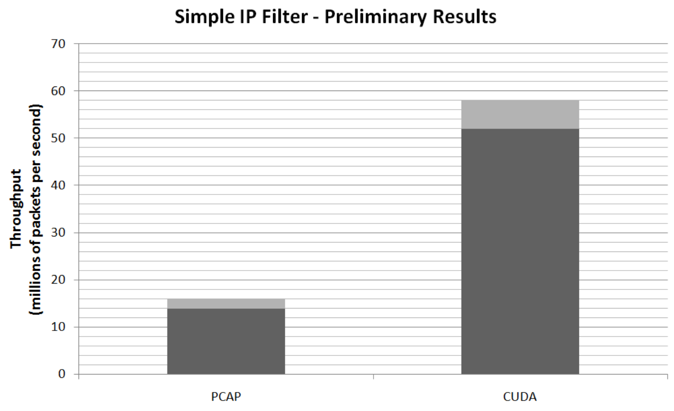

Figure 4. A Simple IP filter: Preliminary kernel classification speed comparison.

Preliminary results show promise, with early CUDA kernels averaging a throughput between 52 million and 58 million packets per second when performing a simple IP filter on a single NVidia Geforce GTX 275. In contrast, the same filter implemented using the PCAP library in C++ resulted in between 14 million and 16 million packets per second throughput on an Intel Q9550 Quad-core processor (see figure 4). We expect throughput to increase as development continues.

IV. Summary

In this paper, we have attempted to devise a parallel filtering algorithm which capitalises on CUDA architecture to accelerate classification. After briefly introducing the problem space, we focused on the architecture of CUDA devices and kernels, 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 [2], [3], when implemented on GPUs. This brought us to the conceptual design of a CUDA classifier, 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 early implementation results.

References

[1] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005. http://doi.acm.org/10.1145/1108956.1108958.

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

[3] A. Begel, S. McCanne, and S. L. Graham, "Bpf+: exploiting global data-flow optimization in a generalized packet filter architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 123–134, 1999.

[4] 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*, (New York, NY, USA), pp. 53–59, ACM, 1996.

[5] 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*, pp. 153–165, 1994.

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

[7] "Cuda best practices guide, version 2.3." Online, July 2009.

[8] "Cuda programming guide, version 2.3." Online, July 2009.

[9] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 191–202, 1998.

[10] T. Y. C. Woo, "A modular approach to packet classification: Algorithms and results," in *In IEEE Infocom*, pp. 1213–1222, 2000.

[11] F. Baboescu and G. Varghese, "Scalable packet classification," *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 199–210, 2001.

[12] 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*, (New York, NY, USA), pp. 213–224, ACM, 2003.

[13] 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*, (Berlin, Heidelberg), pp. 116–134, Springer-Verlag, 2008.

[14] Y. H. Cho and W. H. Mangione-Smith, "Deep network packet filter design for reconfigurable devices," *Trans. on Embedded Computing Sys.*, vol. 7, no. 2, pp. 1–26, 2008.

[15] 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*, pp. 121–126, 2002.

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

[17] A. S. Tongaonkar, "Fast pattern-matching techniques for packet filtering," tech. rep., 2004.

[18] 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*, (Berkeley, CA, USA), pp. 279–292, USENIX Association, 2008.

Mr Alastair Nottingham completed his BSc (Hons) in Computer Science in 2008 . His research interests include high-performance parallel processing, computer security, and artificial intelligence, as well as computer graphics and data visualisation. Alastair is currently in his second year of MSc Study focusing on High performance packet classifiers at Rhodes University, within the Security and Networks Research Group (SNRG).