HARNESSING GPU COMPUTING IN SYSTEM-LEVEL SOFTWARE

by

Weibin Sun

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2014

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

The dissertation of Weibin Sun

has been approved by the following supervisory committee members:

Robert Ricci	, Chair	May 1, 2014 Date Approved
Jacobus van der Merwe	, Member	May 1, 2014 Date Approved
John Regehr	, Member	May 1, 2014 Date Approved
Mary Hall	, Member	May 1, 2014 Date Approved
Eddie Kohler	, Member	May 1, 2014 Date Approved

and by Ross Whitaker , Chair of the School of Computing

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

As the base of the software stack, system-level software is expected to provide efficient and scalable storage, communication, security and resource management functionalities. However, there are many computationally expensive functionalities at the system level, such as encryption, packet inspection, and error correction. All of these require substantial computing power.

What's more, today's application workloads have entered gigabyte and terabyte scales, which demand even more computing power. To solve the rapidly increased computing power demand at the system level, this dissertation proposes using parallel graphics processing units (GPUs) in system software. GPUs excel at parallel computing, and also have a much faster development trend in parallel performance than central processing units (CPUs). However, system-level software has been originally designed to be latency-oriented. GPUs are designed for long-running computation and large-scale data processing, which are throughput-oriented. Such mismatch makes it difficult to fit the system-level software with the GPUs.

This dissertation presents generic principles of system-level GPU computing developed during the process of creating our two general frameworks for integrating GPU computing in storage and network packet processing. The principles are generic design techniques and abstractions to deal with common system-level GPU computing challenges. Those principles have been evaluated in concrete cases including storage and network packet processing applications that have been augmented with GPU computing. The significant performance improvement found in the evaluation shows the effectiveness and efficiency of the proposed techniques and abstractions. This dissertation also presents a literature survey of the relatively young system-level GPU computing area, to introduce the state of the art in both applications and techniques, and also their future potentials.

CONTENTS

ABSTRACT . . . . . **iii**

LIST OF FIGURES . . . . . **vii**

LIST OF TABLES . . . . . **viii**

ACKNOWLEDGMENTS . . . . . **ix**

CHAPTERS

1. INTRODUCTION **1**

1.1 Dissertation Statement 3
1.2 Contributions 3
1.3 Findings 3
1.4 Dissertation Outline 5

2. GENERAL PURPOSE GPU COMPUTING **6**

2.1 Overview 6
2.2 Parallel GPGPU 7
2.3 GPU Memory Architecture 8
2.4 Advanced GPGPU Computing 9
2.5 Efficient GPGPU Computing Techniques 10

3. SYSTEM-LEVEL GPU COMPUTING PRINCIPLES **12**

3.1 Throughput-oriented Architecture 13
3.1.1 Parallel Workload Abstraction 13
3.1.2 Avoid Control Flow Divergence 14
3.2 Coalesce Global Memory Accesses 16
3.3 Overlapped GPU Operations 16
3.4 Asynchronous GPU Programming 17
3.5 Reduce Memory Copy Overhead 18
3.5.1 DMA Overhead 18
3.5.2 Avoid Double-buffering 19
3.5.3 Trade Off Memory Copy Costs 20
3.5.4 Discussion 20
3.6 Miscellaneous Challenges 21
3.6.1 Kernel-User Communication 21
3.6.2 Implicit Synchronization 21
3.7 Summary 22

4. GPUSTORE: GPU COMPUTING FOR STORAGE 23
4.1 Design 23
4.1.1 Overview 23
4.1.2 Memory Management 25
4.1.3 Request Management 26
4.1.4 Streams Management 28
4.2 Implementation 28
4.3 Evaluation 29
4.3.1 Framework Performance 29
4.3.2 Encrypted Device Mapper 30
4.3.3 Encrypted File System 31
4.3.4 Data Recovery 34
4.4 Summary and Future Work 36

5. SNAP: PACKET PROCESSING WITH CLICK AND GPUS 37
5.1 Click Background 38
5.2 Motivating Experiments 38
5.3 Design 41
5.3.1 Batched Packet Processing 41
5.3.2 Packet Processing Divergence 46
5.3.3 Asynchronous Processing 48
5.3.4 Packet I/O 49
5.4 Implementation 50
5.5 Evaluation 51
5.5.1 Packet I/O 52
5.5.2 Applications 52
5.5.3 Latency and Reordering 54
5.5.4 Packet Processing Divergence 55
5.5.5 Flexibility and Modularity 55
5.6 Summary and Future Work 58

6. A SURVEY OF SYSTEM-LEVEL GPU COMPUTING 60
6.1 Applications 61
6.1.1 Networking 61
6.1.2 Storage 63
6.1.3 Database Processing 64
6.1.4 Security 65
6.1.5 Program Analysis 65
6.1.6 Data Compression 66
6.2 Techniques 66
6.2.1 Batched Processing 67
6.2.2 Memory Copy Overhead 67
6.2.3 Warp Divergence 70
6.2.4 Improve GPU Utilization 70
6.2.5 Virtualization and Migration 71
6.2.6 Resource Management and Scheduling 71
6.3 Future 72
6.4 Summary 73

7. CONCLUSION **75**

REFERENCES **77**

LIST OF FIGURES

2.1 Architecture of a CUDA GPU. 7

4.1 The architecture of `GPUstore`. 24

4.2 The workflow of a `GPUstore`-based storage stack. 24

4.3 GPU AES cipher throughput with different optimizations compared with Linux kernel's CPU implementation. The experiments marked "w/o RB" use the techniques described in Section 4.1.2 to avoid double-buffering. 25

4.4 Throughput of a GPU kernel that copies data but performs no computation. . 30

4.5 `dm-crypt` throughput on an SSD-backed device. 31

4.6 `dm-crypt` throughput on a DRAM-backed device. 31

4.7 `eCryptfs` throughput on an SSD-backed file system. 32

4.8 `eCryptfs` throughput on a DRAM-backed file system. 32

4.9 `eCryptfs` concurrent write throughput on a DRAM disk for two block sizes. . 34

4.10 Throughput for the RAID recovery algorithm with and without optimizations to avoid redundant buffers. 34

4.11 RAID read bandwidth in degraded mode. 35

4.12 RAID read bandwidth in degraded mode on DRAM-backed devices. 35

5.1 Simplified Click configurations for motivating experiments. 39

5.2 GPU versus CPU performance on packet processing algorithms. Note that the axes are nonlinear. 40

5.3 The *PacketBatch* structure. 43

5.4 A slicing example. 44

5.5 Batching and debatching elements. Serial interfaces are shown as simple arrows, wide interfaces as double arrows. 45

5.6 Handling divergent packet paths. 47

5.7 Application performance. 53

5.8 Forwarding performance when using a `GPUSDNClassifier` that diverges to two `GPUIDSMatcher` elements. 56

5.9 Major changes of the standard Click router to build a GPU-based IDS router. 57

5.10 Fully functional IP router + IDS performance 57

LIST OF TABLES

4.1 Modified lines of code (LOC) to use `GPUstore`. 29
5.1 Relative throughputs of simple processing pipelines. 40
5.2 Base forwarding performance of `Snap` and Click. 53

ACKNOWLEDGMENTS

I want to thank my parents and my grandparents, who always give me the freedom to choose my life and always encourage me to pursue my dream.

Endless thanks to my wife Shuying, who always gives me a backup and cheers me up in any circumstances. She also listened to my practice talks every day during the week before the defense and gave me valuable comments and suggestions to improve my slides and presentation style. Thank you, for loving me!

I want to thank my daughter, Maggie. She is the main reason that drives me to graduate! She gives us endless laughter and tears.

Many thanks to my advisor, Robert Ricci. I cannot use a finite number of words to express my thanks to him. He is the best advisor and friend you can imagine! He listens to my strange ideas, supports me to study and do research on them, writes funding proposals for my project, modifies and refines every word of my awkward English paper drafts, gives me comments and suggestions on each single slide of my presentations... Thank you, for guiding me!

I also want to thank my friend and office-mate Xing. We discussed a lot on my research project, papers, dissertation and presentations, which helps me make this work. Thank you for your help in both study and life as my neighbor!

Special thanks to Mary Hall, who is also in my committee. I learned GPU programming and various techniques from Mary's class. Without her class, I would not be able to do this work at all. Mary also wrote a recommendation letter for me to help me get the NVIDIA fellowship, which funded part of my research. So, also thank NVIDIA!

I also want to thank my other committee members: John, Kobus and Eddie. John may be not aware of the fact that I started learning the low-level bus details because of his embedded class, where he showed us how easy the low-level hardware and principle is. Kobus gave me useful suggestions on how to improve my dissertation. He also helped me with my Snap paper's publication, which makes me qualified for graduation. Eddie developed the Click modular router, without which Snap cannot exist at all!

There are more people that I should thank. Mike, who was always helpful when I did experiments; Gary, who helped editing my papers; Eric, the LaTeX guru that helped my papers, and also my presentations; Ren, who helped me with English grammars and words, and many other people who helped me in Flux or in school of computing.

CHAPTER 1

INTRODUCTION

System-level software sits at the bottom of the software stack, including file systems, device drivers, network stacks, and many other operating system (OS) components. They provide basic but essential storage, communication and security functionalities for upper-level applications. It is vital for system software to make those functionalities efficient and scalable, otherwise the entire software stack will be slowed down. Yet, many system-level functionalities require substantial computing power. Examples include encryption for privacy, deep packet inspection for network protection, error correction code or erasure code for fault tolerance, and lookups in complex data structures (file systems, routing tables, or memory mapping structures). All of these may consume excessive processing power. What's more, today's application workloads are dramatically increasing: gigabytes or even terabytes of multimedia contents, high definition photos and videos, tens or even hundreds of gigabits per second network traffic and so on. Thus more and more computing power is needed to process those bulk-data workloads on modern rich functional software stacks.

A very common and important feature of many system-level computational functionalities is that they are inherently parallelizable. Independent processing can be done in parallel at different granularities: data blocks, memory pages, network packets, disk blocks, etc. Highly parallel processors, in the form of graphics processing units (GPUs), are now common in a wide range of systems: from tiny mobile devices up to large scale cloud server clusters [1] and super computers [2]. Modern GPUs provide far more parallel computing power than multicore or many-core central processing units (CPUs): while a CPU may have two to eight cores, a number that is creeping upwards, a modern GPU may have over two thousands [3], and the number of cores is roughly doubling each year [4]. As a result, exploiting parallelism in system-level functionalities to take advantage of today's parallel processor advancement is valuable to satisfy the excessive demand of computing power by modern bulk-data workloads.

GPUs are designed as a *throughput-oriented architecture* [5]: thousands of cores work

together to execute very large parallel workloads, attempting to maximize the total throughput, by sacrificing serial performance. Though each single GPU core is slower than a CPU core, when the computing task at hand is highly parallel, GPUs can provide dramatic improvements in throughput. This is especially efficient to process bulk-data workloads, which often bring more parallelism to fully utilize the thousands of GPU cores.

GPUs are leading the way in parallelism: compilers [6, 7, 8], algorithms [9, 10, 11], and computational models [12, 13, 14] for parallel code have made significant advances in recent years. High-level applications such as video processing, computer graphics, artificial intelligence, scientific computing and many other computationally expensive and large scale data processing tasks have benefited with significant performance speedups [15, 16] from the advancement of parallel GPUs. System-level software, however, have been largely left out of this revolution in parallel computing. The major factor in this absence is the lack of techniques for how system-level software should be mapped to and executed on GPUs.

For high-level software, GPU computing is fundamentally based on a computing model derived from the data parallel computation such as graphics processing and high performance computing (HPC). It is designed for long-running computation on large datasets, as seen in graphics and HPC. In contrast, system-level software is built, at the lowest level of the software stack, on sectors, pages, packets, blocks, and other relatively small structures despite modern bulk-data workloads. The scale of the computation required on each small structure is relatively modest. Apart from that, system-level software also has to deal with very low-level computing elements, such as memory management involving pages, caches, and page mappings, block input/output (I/O) scheduling, complex hardwares, device drivers, fine-grained performance tuning, memory optimizations and so on, which are often hidden from the high-level software. As a result, system-level software requires technologies to bridge the gap between the small building structures and the large datasets oriented GPU computing model; also to properly handle those low-level computing elements with careful design trade-offs and optimizations, to take advantage of the parallel throughput-oriented GPU architecture.

This dissertation describes the generic principles of system-level GPU computing, which are abstracted and learned from designing, implementing and evaluating general throughput-oriented GPU computing models for two representative categories of system-level software: storage applications and network packet processing applications. Both models are in the form of general frameworks that are designed for seamlessly and efficiently integrating parallel GPU computing into a large category of system-level software. The principles

include unique findings of system-level GPU computing features, and generic design techniques and abstractions to deal with common system-level GPU computing challenges we have identified. The significant performance improvements in storage and network packet processing applications brought by integrating GPU computing with our frameworks shows the effectiveness and efficiency of the proposed techniques and abstractions, and hence supporting the following statement:

1.1 Dissertation Statement

The throughput of system software with parallelizable, computationally expensive tasks can be improved by using GPUs and frameworks with memory-efficient and throughput-oriented designs.

1.2 Contributions

The contributions of this dissertation work include the following.

- Two general frameworks, `GPUstore` [17] and `Snap` [18], for integrating parallel GPU computing into storage and network packet processing software, as described in Chapter 4 and Chapter 5.
- Three high throughput GPU-accelerated Linux kernel storage components, including a filesystem and two storage device mappers, which are discussed in Chapter 4.
- A set of Click [19] elements for `Snap` to help build parallel packet processing pipelines with GPUs, which are described in Chapter 5.
- A modified fully functional Click Internet protocol (IP) router with fast parallel GPU acceleration built on top of `Snap` as described in Chapter 5.
- A literature survey of system-level GPU computing that covers existing work, potential applications, comparison of the useful techniques applied in surveyed work and the ones proposed in our work as presented in Chapter 6.

1.3 Findings

Besides the above contributions, we also have found and learned valuable facts and lessons from designing and implementing `GPUstore` and `Snap`. We believe they are common and applicable to other system-level GPU computing software, too. We will thoroughly discuss them in the later chapters. For now, they are listed below as an overview.

- **Batching improves GPU utilization.** System code is often latency-oriented. As a result, system software often works with small building blocks. However, GPUs' architecture is throughput-oriented. Small building blocks lead to constant overhead

and low GPU utilization. To bridge the gap between the latency-oriented system code and throughput-oriented GPUs, batching small blocks to process many at once amortizes the overhead and improves GPU utilization. Batching may increase the processing latency of a single block, but for systems dealing with I/O, adding relatively small latency is tolerable.

- **High throughput computing needs truly asynchronous GPU programming.** With the required synchronization, current GPU programming models are mismatched to asynchronous system components, which widely exist in file systems, block I/O, device drivers, and network stacks at system level. Synchronization stalls the typical GPU computing pipeline which consists of three common stages: (1) host-to-device direct memory access (DMA) copy; (2) GPU kernel execution; (3) device-to-host DMA copy. Event-based or callback-based asynchronous GPU programming is necessary for asynchronous systems code, allowing it to fully utilize the GPU pipeline and achieve high throughput.
- **System code has data usage patterns that are different from traditional GPU code.** Traditional GPU computing usually reads the data from a file or the network into host memory, copies the entire data buffer into GPU device memory, and does the computation based on all the data. However, the code in a system component often works as a stage of a long data processing pipeline, and may use either the entire dataset passed through the pipeline, or just a few small pieces. In order to improve the system performance, special care must be taken to provide different memory models to system code (both the CPU side code and the GPU side code) according to the code's data usage pattern.
  - **The computation needs all the data.** This data usage pattern is similar to traditional GPU computing. In this case, the GPU needs the entire dataset to be copied into GPU memory. To reduce the memory copy overhead, we should focus on reducing the copy within host memory. Different from traditional GPU computing, in the system-level context, processing stages often pass data buffers owned by a third party, such as the page-cache managed memory in the storage stack and network packet buffers in the network stack, through a long pipeline. Remapping the memory to make it GPU DMA-capable avoids redundant memory copy in host memory.
  - **The computation uses just a few small pieces of the large trunk of data.** In contrast to most GPU computing, some system-level computing tasks

such as packet processing use only a few small pieces of the entire dataset. In this case, copying the entire dataset into GPU memory wastes a large portion of the bus bandwidth. Considering the much higher host memory bandwidth than the bus bandwidth, it is worth copying the scattered small pieces into a consecutive host memory buffer, allowing one smaller DMA copy to GPU. Such trade-off takes advantage of the faster host memory to reduce the overall memory copy time. Gathering the scattered pieces together also benefits memory access coalescing on the GPU, taking advantage of the wide GPU memory interface.

1.4 Dissertation Outline

Chapter 2 gives a simple background introduction of concepts, techniques and features of general purpose GPU computing. This chapter tries to get readers without GPU computing background familiar with it, and also defines the terminology used in later chapters. Readers familiar with traditional GPU computing can skip this.

Chapter 3 describes the system-level GPU computing principles at the high level. It discusses the challenges and special requirements in system-level GPU computing and the proposed techniques and abstractions to deal with them.

Chapter 4 describes the storage framework `GPUstore`, including the design, implementation and the experimental evaluation. The GPU-accelerated file system and block device drivers built on top of `GPUstore` have achieved up to an order of magnitude performance improvements compared with the mature CPU-based implementations.

Chapter 5 describes the network packet processing framework `Snap`, similar to the previous `GPUstore`, including the design, implementation and the experiments. The demonstrated deep packet inspection router built with `Snap` has shown 40 Gbps (gigabits per second) line rate packet processing throughput at very small packet size.

Chapter 6 is the survey of system-level GPU computing, which includes both existing work and the identified possible application areas. The survey discusses techniques used by other system-level GPU systems to compare with what this dissertation has proposed and applied.

Chapter 7 reviews the dissertation and concludes.

CHAPTER 2

GENERAL PURPOSE GPU COMPUTING

This chapter describes essential general purpose GPU (GPGPU) computing background that is needed to help understand the rest of the dissertation. Currently, there are two most widely used GPGPU frameworks: compute unified device architecture (CUDA) [4] and open computing language (OpenCL) [20]. CUDA is a proprietary framework developed by NVIDIA corporation; OpenCL is a public framework designed by Khronos Group. Despite its proprietary feature, CUDA has several advanced features, such as concurrent streaming and flexible memory management which are helpful to system-level computing and OpenCL missed at the time of my dissertation work. As a result, CUDA is used in this dissertation to represent the lowest level GPGPU computing framework. And its terminology and concepts are used in this chapter to explain GPGPU computing. For a comprehensive description of CUDA-based modern GPGPU computing, readers may refer to NVIDIA's CUDA programming guide [4].

2.1 Overview

A GPU works as a coprocessor of the CPU. It has dedicated video memory on the card to save the computing data. The processor cores on GPU can only access the video memory, so any data to be processed by the GPU must be copied into the video memory. To utilize the GPU, a typical workflow includes three steps:

1. CPU code copies the data to be processed from main memory (also called "host memory" in CUDA) to the video memory (also called "device memory");
2. CPU code starts the GPU kernel, which is the program to be executed on the GPU, to process the copied data and produce the result in the device memory;
3. After the GPU kernel execution, the CPU code copies the result from device memory back to host memory.

Most GPUs sit in the peripheral component interconnect express (PCIe) slots, needing DMA over PCIe for the aforementioned memory copy. The GPU kernel is a program consisting

of GPU binary instructions. Using CUDA, programmers can write C/C++ code, then use `nvcc` to compile them into GPU instructions. CUDA also provides runtime libraries to allow CPU code to use special host memory, device memory, make DMA copy and manage GPU execution.

2.2 Parallel GPGPU

A GPGPU is a special single instruction multiple data (SIMD) processor with hundreds or thousands of cores and a variety of memory types (as shown in Figure 2.1). On recent CUDA GPUs, each 32 cores are grouped into a "warp." All 32 cores within a warp share the same program counter. A GPU is divided into several "stream multiprocessors," each of which contains several warps of cores and fast on-chip memory shared by all cores. A GPU kernel is executed by all the physical cores in terms of threads. So for a given GPU kernel, it becomes one thread on each GPU core when executing. High-end GPU models, such as GTX Titan Black [3], can have as many as 2880 cores. In that sense, up to 2880 threads can concurrently run on a GPU to execute a single GPU kernel. Threads on GPU cores may do hardware-assisted context switching, in case of memory operations or synchronization, which is determined and scheduled by the GPU hardware. Such zero-cost hardware context switching makes it possible to run millions of threads on a single GPU without any context switching overhead. Such a "single kernel executed by multiple threads" computing model is called single instruction multiple threads, or SIMT.

SIMT is a simple and efficient parallel processor design. But similar to other SIMD processors, the performance of SIMT may suffer from any control flow divergence because of the shared program counter within a warp. On CUDA GPUs, any control flow divergence

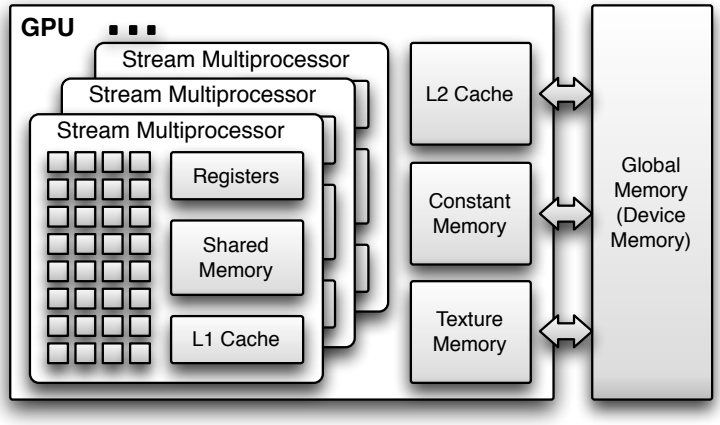


Figure 2.1. Architecture of a CUDA GPU.

within a warp may cause serialized control flow. When a core executes the control flow branch that its current thread should not jump into, it simply disables memory or register access to disable the execute effects. Different from the powerful sequential cores on CPUs that are armed with large cache, out-of-order execution and accurate branching prediction, a single GPU core is very weak. The performance speedup of many GPGPU programs comes from partitioning the workload into millions of small pieces processed by thousands of lame cores in parallel. As a result, a GPU kernel full of such control flow structures may severely slow down the GPU performance due to the control flow divergence.

SIMT architecture requires programmers to carefully tune their GPU kernels to avoid conditional structures and to reduce as many loops as possible. This can affect the data processing design in system software. For example, when using GPUs to process a batch of network packets through the network stack, packets in the batch may diverge to different paths because of their different protocols. Such path divergence will lead to warp divergence on GPUs because GPU kernel threads must execute different code for packets going to different paths. As we will see in `Snap`, we have proposed techniques and design principles to reduce the overhead caused by this problem.

2.3 GPU Memory Architecture

The original video memory on GPU board has been largely increased to up to 6GB (gigabyte) for a single GPU. Compared with the instruction execution on GPU cores, accessing GPU video memory, which is also called global memory, is much slower and may cost hundreds of cycles. So global memory access is a time consuming operation that should be minimized. Fortunately, the GPU memory bus width is much wider than normal CPUs, e.g., some model [3] has a 384 bits memory bus width which leads to 336 GB/s bandwidth. Though a single thread may not be able to fully utilize the 384 bits bus, a SIMT-oriented GPU kernel may take advantage of the wide bus by issuing memory access operations to consecutive locations from the same warp, which is called "memory access coalescing."

Besides the global memory, there are some special memory types on GPUs for fast memory access in certain applications (as shown in Figure 2.1). Constant memory is a small region of memory that supports fast read-only operations. Texture memory is similar to constant memory but has very fast special access patterns. For example, according to the Kargus [21] network intrusion detection system (NIDS), using texture memory for deterministic finite atomata (DFA) transition table improves the pattern matching

performance by 20%. Shared memory can be viewed as a program-controlled cache for a block of threads. Its access latency can be 100x lower than the uncached global memory. In fact, current CUDA GPUs use a single region of 64KB (kilobyte) on-chip memory per stream multiprocessor to implement both layer one (L1) cache and shared memory, allowing either 48 KB cache and 16 KB shared memory or 16 KB cache and 48 KB shared memory split that is configurable by programmers. Besides program-controlled cache for fast frequent data access, shared memory can also be used to achieve global memory access coalescing, e.g., when using four threads to encrypt a single 16-byte advanced encryption standard (AES) data block, each thread can issue a four-byte read operation to be coalesced into a single sixteen-byte transaction.

As a result, to achieve high performance GPU computing, a GPU program should be designed and executed with the following memory related considerations: consecutive threads should issue coalescable memory accesses; memory access latency can be hidden by launching more threads than cores in order to switch out the threads that wait for memory transaction; read-only data should go into constant or texture memory; frequently accessed data should go into shared memory.

2.4 Advanced GPGPU Computing

GPUs have been designed to be more than simple coprocessors. Many advanced techniques have been applied to improve the GPU computing environment. In the hardware perspective: GPUs are capable of having multiple GPU kernels executed concurrently on their cores; some high-end GPUs have more than one DMA engine on a single GPU board; the rapidly development of dynamically random access memory (DRAM) technology has significantly reduced the cost of large volume of DRAM in modern computers, so using large trunks of nonpageable dedicated physical memory for DMA is feasible. To utilize these hardware features to improve the GPU computing performance, CUDA has implemented and exposed software interfaces for programmers to access them.

CUDA provides the "stream" abstraction, which represents a GPU computing context consisting of a DMA engine for memory copy, and GPU cores for GPU kernel execution. CUDA GPUs support multiple concurrent streams, which essentially enables concurrent execution of multiple GPU kernels on a single GPU, and utilizes multiple DMA engines for concurrently bidirectional PCIe transfers. Each stream is independent of the other: operations (memory copy and GPU kernel execution) in the same stream are sequentially performed, but operations in different streams are totally independent. This provides an

asynchronous streaming GPU computing model, which may fully utilize the execution and copy capabilities of GPUs.

DMA memory copy requires nonpageable memory pages that are locked in physical memory. So if a program wants to copy the data in pageable memory with GPU DMA, the GPU driver has to firstly allocate an extra locked memory buffer, then copy the data from pageable memory to locked memory, and finally copy the data from locked memory to the GPU with DMA. This causes a double-buffering problem that causes an extra copy in host memory and also wastes extra memory space. As a result, a program should use locked memory directly to store its data in order to avoid this double-buffering problem in GPU DMA.

2.5 Efficient GPGPU Computing Techniques

To achieve efficient GPGPU computing, the aforementioned GPU features must be considered to design GPU kernels and to design the host side software.

- SIMT architecture requires as few branches as possible in GPU kernel's control flow to avoid warp divergence. This is more an algorithmic requirement than a system one. However, as the packet processing example shows in Section 2.2, sometimes the system-level processing control flow can cause the GPU side divergence, hence it requires the system design to be GPU-aware, to either avoid processing flow divergence, or apply techniques to reduce the divergence overhead like in `Snap`.
- The wide memory bus requires coalescable memory accesses in consecutive threads. This not only requires SIMT GPU kernel to guarantee coalescable memory accesses from consecutive threads, but also needs designing coalescing-friendly data structures or data layout in system-level software.
- The variety of different GPU memory requires GPU kernels to be carefully optimized and tuned to make use of the faster read only memory regions and shared memory. This is mostly a general algorithmic requirement, not a very "system" need. But as said in Section 2.3, system code still needs finely tuned system-level algorithms such as Kargus' DFA table placement in texture memory that improves 20% pattern matching performance.
- The CUDA stream technology requires programmers to make use of the concurrency and bidirectional PCIe bus with careful design. This needs system designers to take advantage of concurrent workloads or partitioning large single-threaded workload, properly abstracting the computations on GPUs to utilize the overlapped GPU com-

putation and memory copy pipeline enabled by CUDA stream.

- The page-locked memory requires the programmers to design their memory management mechanism with the consideration of GPU resources at the very beginning. Because memory management in system-level, especially in OS components can be very complex. System components can either have their own memory allocators, or more often, just work as stages in a pipeline that don't own the memory of the processed data buffers at all.

Traditional GPGPU computing considers mostly the GPU side optimization techniques such as the SIMT architecture, wide memory bus and different types of device memory. Their workflows are often very simple, as shown in Section 2.1. For these kind of traditional GPGPU programs, all the complex low-level problems such as memory management, memory copy efficiency, bus bandwidth, device synchronization overhead, interaction with other system components, are hidden and handled efficiently by the system-level code. Now when it comes to the system-level software targeted in this dissertation, all of them must be considered in the system design when integrating GPU computing. Special system-level data usage patterns may cause large PCIe bus bandwidth waste when copying memory, and may also cause difficulty to issue coalescable memory accesses from GPU threads. Using page-locked memory may be not as simple as calling the CUDA memory allocator; it may lead to complex memory page remapping in system code. Some inherently asynchronous system codes may not be able to afford the host-device synchronization because that will change their efficient ways of working. All of these system-level issues must be handled in this dissertation work, as will be shown in the next chapter.

CHAPTER 3

SYSTEM-LEVEL GPU COMPUTING PRINCIPLES

A lot of system-level workloads are good candidates of the throughput-oriented GPU accelerations: file systems and storage device layers use encryption, hashing, compression, erasure coding; high bandwidth network traffic processing tasks need routing, forwarding, encryption, error correction coding, intrusion detection, packet classification; a variety of other computing tasks perform virus/malware detection, code verification, program control flow analysis, garbage collection and so on. Not every system-level operation is capable of throughput improvement with GPUs. Those inherently sequential or latency sensitive tasks such as process scheduling, acquiring current date-time, getting process identifier (ID), setting up I/O devices, allocating memory and so on definitely can't afford the relatively high-latency GPU communication that often crosses external buses.

Some early system-level work has demonstrated the amazing performance speedup using GPUs: PacketShader [22], a GPU-accelerated software router, is capable of running IP routing lookup at most 4x faster than the CPU-based mode. SSLShader [23], a GPU-accelerated secure sockets layer (SSL) implementation, runs four times faster than an equivalent CPU version. Gnort [24], as an GPU-accelerated NIDS, has showed 2x speedup over the original CPU-based system. EigenCFA [25], a static program control flow analyzer, has reached as much as 72x speedup when transforming the flow analysis into matrix operations on GPUs.

However, all of them are very specialized systems that have been done in an ad hoc way to deal with performance obstacles such as memory copy overhead, parallel processing design and those mentioned in the previous chapter, without considering any generic systematic design for a wide range of related applications. A general system-level GPU computing model needs to provide generic design principles. It requires a thorough study of system-level behaviors and GPU computing features to identify the common challenges when integrating GPUs into low-level system components, and then come up with generic techniques and

abstractions to decompose the system, trade off design choices according to contexts, and finally solve the problems.

This chapter discusses the generic challenges faced by system-level GPU computing and our proposed design principles. Similar to traditional general purpose GPU computing, tuning GPU kernel to make parallel algorithms fully utilize the SIMT cores is important to system-level code, too. However, a unique feature of system-level computing tasks is that most of them are much simpler than the application-level tasks in scientific computing and graphics processing. If we take a look at the surveyed computing tasks in system-level in Section 6.1, they are mostly one-round deterministic algorithms without any iterative scheme and convergence requirement such as in partial differential equation (PDE) algorithms and randomized simulations. In that case, other noncomputational costs may become major overhead: for example, as we evaluated for `GPUstore`, the significant AES performance improvements with a variety of memory copy optimization techniques show that the memory copy, rather than the cipher computation, is the bottleneck. As a result, system-level GPU computing is faced with challenges from not only traditional device side GPU kernel optimization, but also the host side GPU computing related elements. The following sections discuss each of the major challenges we've identified in the `GPUstore` and `Snap` projects and the generic techniques to deal with them.

3.1 Throughput-oriented Architecture

The SIMT GPU architecture enables simple but highly parallel computing for large throughput processing. But the SIMT architecture is also a big challenge to GPU kernel design and GPU computing workload. As for the GPU kernel design, it means avoiding control flow divergence via tuned GPU code or SIMT-friendly data structures to process. To fit the SIMT architecture, the workload must be able to provide enough parallelism to fully utilize the thousands of parallel SIMT cores. The parallelism may come from the parallelized algorithms in terms of GPU kernels. More importantly, it comes from the amount of data to be processed in one shot, which often requires batched processing in system environments. When trying to provide a generic design for a variety of system-level applications, a right workload abstraction that covers the various processing tasks is needed to guarantee the parallelism, and also to make it easy to design divergenceless GPU kernels.

3.1.1 Parallel Workload Abstraction

Considering the thousands of cores on a modern GPU that processes one basic data unit in one thread, at least the same number of basic units is required to fully utilize all the

cores. This is actually the key to achieve high throughput in GPU computing. Taking the GTX Titan Black [3] as an example: even with the parallelized AES algorithm that uses four threads to process a single 16-byte block, its 2880 cores still need $2880 \times 4 = 11KB$ data to process in one GPU kernel launch. If we consider the context switching due to global memory access, even more data are required to keep GPU cores busy.

Unfortunately, system-level software is often designed to process data sequentially due to the latency-oriented system design philosophy through the ages. However, the performance of parallel and sequential processors have been significantly improved to be easily achieve very low latency, and also today's "big data" workloads focus more on high throughput processing, even at the system-level. Batching is a simple yet effective technique to accumulate enough parallelism in one-shot workload. Instead of a single data unit to be processed at one time, a one-shot workload with batching now includes enough units to fully occupy the thousands of GPU cores. "Enough" is ambiguous in the high-level workload abstraction. An intuitive but effective policy for concrete design is to ensure that processing the batched workload on GPUs is at least faster than on CPUs.

An obvious drawback of batching is the increased latency. This may cause a serious performance problem in some latency sensitive software, especially the network applications such as video conference, video streaming, etc. As we will see in `Snap`, the packet processing latency introduced by batching is eight to nine times larger than the nonbatching latency. So batching is not a pervasive solution to adapt sequential software with the parallel GPUs. This also reveals the fact that not every system-level software can benefit from parallel GPUs, such as those latency sensitive computing tasks whose algorithms can't be efficiently parallelized.

3.1.2 Avoid Control Flow Divergence

Many research projects [26, 27, 28, 29, 30, 31, 32] have been focused on designing efficient SIMT style parallel implementation of particular algorithms on the GPU. Besides those well-studied techniques, how to efficiently partition the workload into SIMT threads and how to choose SIMT-friendly algorithms and data structures (if possible) are also very important.

The workload partitioning is often very straightforward: each GPU thread processes one single basic data unit. Many system-level computing tasks can use such simple partitioning scheme such as computing erasure code of one disk sector per thread, encrypting a single sixteen-byte AES block per thread, looking up next hop for one IP address per thread, etc. However, exceptions always exist. The AES algorithm is an example, which has been

parallelized by exploiting the intrablock parallelism [33]. `GPUstore` applied this technique in its AES implementation to have four threads to encrypt/decrypt a single AES block.

Designing the data structures used by processing logic to make them SIMT-friendly is another technique to avoid control flow divergence. One example is the pattern matching algorithm in `Snap` that is based on DFA. The DFA matching is a very simple loop: get the current symbol; find the next state indexed by the symbol in current state's transition table; use the found state as the current state. The algorithm itself can be implemented in SIMT style because all input strings use the same three-step matching. But for the second step, some transition table data structures cannot ensure same code execution when finding the next state. For example, a tree-based map may need different lookup steps for different symbols to find the mapped states, which means different GPU threads may have to execute different numbers of lookup loops to find the next states, and hence is not SIMT-friendly. In the meanwhile, an array-based transition table, which assigns state for every symbol in the DFA alphabet, can guarantee equal steps to find the next state of any given symbol, and hence is SIMT-friendly. That's because we can use the symbol as the array index to fetch the mapped state, and one state lookup becomes a single array access. So although the tree-based map is more memory-efficient and the array-based map is very memory-consuming because even invalid symbols in the alphabet have transition states, sometimes the memory may be sacrificed to achieve SIMT-friendly GPU code.

3.1.2.1 When It Is Unavoidable

An effective but not efficient way that can always solve the control flow divergence is to partition the computing tasks into multiple GPU kernels, each GPU kernel works for a particular control flow branch of the computing task. This needs some information available at the host side to indicate how many threads a GPU kernel should be executed by, and which data items should be processed by a particular GPU kernel. Such host side information implies a device-to-host memory copy, the necessary host-device synchronization for its completion and probably a data structure reorganization, which may cost more than admitting divergence at the GPU side. So it is not always unacceptable to introduce divergence into a GPU kernel. The batched packet processing in different GPU elements in `Snap` is a very representative example.

However, allowing divergence in GPU kernel doesn't mean writing arbitrary code; we still need to minimize the effects. We have proposed and implemented the predicated execution to minimize the divergence affections. More details of the predicated execution are in Section 5.3.2.

3.2 Coalesce Global Memory Accesses

The throughput-oriented GPU architecture is armed with not only high computation throughput, but also high memory access throughput. To fully utilize the 384 bits global memory bus in GTX Titan Black [3], coalescable memory accesses must be issued from neighbor threads in a thread block. Compared with noncoalesced access, coalesced memory access can achieve orders of magnitude faster [26]. So the GPU kernel must be designed to access consecutive memory locations from threads within the same thread block or warp. Besides the GPU kernel code, the data structures to be processed must be carefully organized to make them coalescing-friendly. Some kinds of workloads are quite easy to satisfy the coalescing requirement, for example, data read from a block device are naturally consecutively stored, hence when encrypting them, neighbor GPU threads can always coalesce their memory accesses. However, some workloads are not that coalescing-friendly, needing a coalescing-aware abstraction to achieve a generic solution. A simple but very good example is the IP routing lookup. The lookup GPU kernel needs only the destination IP address of a packet. If we put the entire packet into the global memory for each thread to access, the memory reads for IP addresses issued from neighbor threads will be scattered. One 384 bits memory transaction may only read 32 bits effective data, which wastes more than 90% memory bandwidth. But if we organize the destination IP addresses into a separate buffer, then up to 12 memory reads for the IP addresses issued from neighbor threads can be coalesced into a single memory transaction, which significantly reduces the number of global memory transactions. The "region-of-interest"-based (ROI) slicing technique applied in `Snap` is an effective abstraction of the workload data to build such coalescing-friendly data structures for a variety of GPU-accelerated computating tasks (see Chapter 5).

3.3 Overlapped GPU Operations

The typical GPU computing workflow mentioned in Section 2.1 can be pipelined to improve the utilization of the two major GPU components: SIMT cores and DMA engines. The pipelined model needs multiple CUDA streams, each of which carries proper-size workloads for certain computing tasks. This may require the workload to be split into multiple trunks in order to fill into multiple streams. The host code will be responsible to do such workload splitting, which needs the task-specific knowledge to ensure the splitting is correct. For a generic GPU computing framework such as `GPUstore`, it is impractical to put those splitting knowledge of every task into the generic framework. The computing tasks

need a right abstraction to describe both the task-specific logic and the task management (such as partitioning) logic. For example, `GPUstore` provides the modular GPU services, which are the abstraction of computing tasks. Each GPU service not only processes computing requests issued from `GPUstore` clients, but also does service-specific request scheduling including workload splitting and merging.

This requirement seems totally contrast to the parallelism one in Section 3.1.1. However, it is not a paradox. An application usually needs to find the balance point of these two requirements to decide the optimal size of the workload to be processed on GPUs in one shot, so as to achieve the best performance. This can be done either dynamically or statically. A static approach finds the balance point offline by benchmarking the computing tasks and uses predefined sizes at run time. Both `GPUstore` and `Snap` simply use this approach in their prototypes. The dynamic way would do microbenchmarking at run time to find optimal sizes for the specific execution hardware and software environment. It can adjust workload sizes according to the hardware and system load, and hence is more accurate.

3.4 Asynchronous GPU Programming

CUDA stream requires asynchronous DMA memory copy and GPU kernel launching operations so that a single CPU thread can launch multiple streams. However, current GPU programming model needs the synchronization between the host and the device to detect the completion of the device operations in each stream. This may not be a problem at all for traditional GPGPU computing that just focuses on a particular computing task. However, many system-level components are designed to exploit asynchrony in order to achieve high performance.

For example, filesystems work with the virtual file system (VFS) layer, and often rely on the OS kernel page-cache for reading and writing. By its nature, the page-cache makes all I/O operations asynchronous: read and write requests to the page cache are not synchronous unless an explicit `sync` operation is called or a *sync* flag is set when opening a file.

Other examples include the virtual block device drivers, which work with the OS kernel's block I/O layer. This layer is an asynchronous request processing system. Once submitted, block I/O requests are maintained in queues. Device drivers, such as small computer system interface (SCSI) drivers, are responsible for processing the queue and invoking callbacks when the I/O is complete.

Some filesystems and block devices, such as network file system (NFS), common Internet file system (CIFS), and iSCSI (Internet SCSI), depend on the network stack to provide their

functionality. Because of the high and unpredictable latency on a network, these subsystems are asynchronous by necessity.

When it goes to the network packet processing, it is totally asynchronous workflow. Although there does exist totally synchronous network stacks such as the uIP [34] and lwIP [35], they are designed for memory constraint embedded systems, not for performance. As a result, the synchronization CUDA call is totally unacceptable in those asynchronous environment because it will block the normal workflow and may cause performance slow down.

`GPUstore` and `Snap` solve this problem by implementing asynchronous CUDA stream callback mechanism to enable completely asynchronous GPU program control flow. Such callback mechanism can be implemented in two different approaches.

Polling-based. this method has a thread keep polling stream state change and invoke callbacks. It is obvious that this approach can get low latency response but will keep a CPU core in busy waiting loop.

Signal-based. this is implemented with the events invoked by GPU interrupts at the low level. The signal-based response latency is definitely higher than the polling-based one, but its advantage is also obvious: CPU cores can be freed to process other work without busy waiting.

At the time of writing this dissertation, the latest CUDA release has provided similar signal-based callback for streams after `GPUstore` did that for more than two years. This further confirms the effectiveness of the techniques we've proposed.

3.5 Reduce Memory Copy Overhead

Having said at the beginning of this chapter, the memory copy is often the major overhead compared with the computation. Many different aspects are related to the performance of memory copy. Since we are discussing system-level GPU computing, the memory copy is not only the DMA over PCIe bus. We also need to consider what happens in the host main memory as an entire system. We will start from the obvious challenges and problems in GPU related memory copy, then gradually introduce other vital issues and techniques to deal with them.

3.5.1 DMA Overhead

The overhead of DMA comes from its special requirement: the memory pages in main memory must not be swapped out (paging) during DMA. To satisfy such requirement when copying data in pageable memory to GPUs, CUDA GPU drivers use the aforementioned

double-buffering approach as described in Section 2.4. According to the evaluations [36], double-buffering DMA can be two times slower than using page-locked memory directly. Compared with pageable memory, using page-locked memory may lock too many physical pages and reduce the available memory for other applications. However, considering today's cheap DRAM and widely available tens of gigabytes DRAM on a single machine, locking even 6GB memory (for a high end GPU [3]) is totally acceptable.

3.5.2 Avoid Double-buffering

The aforementioned solution with CUDA page-locked memory requires the system code to use CUDA-allocated memory as its data buffer. This seems trivial: traditional GPGPU computing usually reads the data from a file or the network into host memory, copies the entire data buffer into GPU device memory, and does the computation. It is seldom that the host memory used in this scenario may have any other users or complicated dependencies, so it is easy to replace it with CUDA-allocated page-locked memory. However, the code in a system component often works as a stage of a long data processing pipeline. In that case, it may be impractical to modify the entire system from the beginning of the pipeline to use CUDA's memory, especially in a large complex system such as the operating system kernel. One way to deal with this is allocating a separate page-locked buffer with CUDA, and copy the data to be processed into this CUDA buffer before DMA (and similar approach for the processing result). This leads to double-buffering, which is similar to the aforementioned early stage CUDA DMA implementation for pageable memory: introducing extra copy in host memory. To avoid the double-buffering problem, we've proposed and implemented the page remapping technique, which can remap external page-locked memory pages into CUDA GPU driver's memory area, and make them DMA-capable just like CUDA page-locked memory (refer to Section 4.1.2.) This allows minimum invasive GPU computing integration into an existing complex system: only the component containing the computing tasks needs a few modifications.

There do exist some special cases where the component we'd like to put GPU-accelerated computing tasks into may be the beginning of the data processing pipeline. It may also be a pipeline stage that allocates memory for later use. In those cases, replacing previous `malloc` or similar memory allocators with CUDA's memory allocation functions is feasible and efficient. Introducing remapping in those cases doesn't make any sense due to the added complexity of the extra page mappings.

The techniques in this section try to avoid memory copy in host memory because such copy is not necessary. However, there is an implicit assumption that the data in the memory

buffer are all useful to the computation. Otherwise it may be unwise to copy the entire buffer to device since it wastes the relatively slow PCIe bandwidth, as we shall discuss in the following section.

3.5.3 Trade Off Memory Copy Costs

Some computing tasks in a system component may need only small pieces of a large data unit it receives. Typical examples are the packet processing tasks: IP routing just needs the destination IP address, layer 2 forwarding just needs the destination's media access control (MAC) address, packet classification just needs five small fields, etc. Copying the entire data unit into GPU memory can waste a large portion of the PCIe bus bandwidth: considering the 4 bytes IP address versus the minimum 64 bytes packet. At the same time, the host side memory bandwidth is much faster than the PCIe bus. Take a look at the machine I used for `Snap` evaluation: the main memory has a 38.4GB/s bandwidth, while the maximum throughput of the PCIe 2.0 16x slot is 8GB/s in each direction, which is a 4.8 times difference. As a result, for computing tasks with the data usage patterns discussed here, copying the needed data pieces into a much smaller page-locked memory buffer and then launching a PCIe DMA for this small buffer may lead to a much faster total memory copy than copying the entire data unit into GPU memory through PCIe bus. The aforementioned `Snap`'s the ROI-based slicing technique is based on this idea. It takes advantage of the much faster host memory to achieve fast host-device memory copy. As we mentioned in Section 3.2, it also makes coalescing-friendly data structures for GPUs (refer to Chapter 5 for details).

3.5.4 Discussion

Section 3.5.3 advocates a technique that is totally opposite to the one in its previous section (Section 3.5.2). But they are not paradoxical. It is the data usage pattern of the computing task that decides which technique to use. On the one hand, for a computing task that needs the entire data unit received by the enclosing system for its computation, it should avoid extra buffers in host memory for the same data unit content with either the remapping technique or using CUDA page-locked memory directly, depending on the memory management role of the system as we discussed in Section 3.5.2. On the other hand, a computing task needing only small pieces of the entire data unit should trade off the main memory bandwidth and the host-device memory copy bandwidth, and may use techniques similar to `Snap`'s ROI-based slicing for faster total memory copy performance. `Snap`'s ROI-based slicing is a very flexible data abstraction to deal with both kinds of usage

patterns: it allows computing tasks to define their own pieces to be processed in a single data unit, for a "use all" usage pattern, the computing task can specify the entire data unit as its interested region.

3.6 Miscellaneous Challenges

There are some challenges caused by the limitations in current GPU computing libraries or GPU hardware. Though we can believe that with the development of GPU technology and the GPU market, these obstacles may disappear in future, it is still worth describing them here for readers who are trying to apply GPU computing into system components at this time. The challenges discussed in this section are only related to our GPU computing experiences learned from `GPUstore` and `Snap`, the survey chapter (in Chapter 6) discusses more such challenges and solutions provided by other system-level GPU computing works.

3.6.1 Kernel-User Communication

Currently, in almost all GPU computing libraries, drivers are closed source, not to mention the even more closed GPU hardware. This leads to a big performance problem when applying GPU computing into an operating system: the OS kernel mode code has to rely on the userspace GPU computing library to use GPUs. So now in OS kernel mode, using GPUs is not as efficient as a function call in userspace, but a cross context communication. Such a system needs an efficient kernel-user communication mechanism for invoking GPU computing library functions, and also memory sharing between two modes for computing data. `GPUstore` got this problem on Linux kernel when using CUDA GPU library. Current open source GPU drivers such as nouvea [37] and open source CUDA implementation such as Gdev [38] still can't reach the proprietary software's performance. `GPUstore` uses a userspace helper to deal with requests from OS kernel and invoke CUDA calls. The userspace helper is based on polling-based file event mechanism to achieve fast kernel-user communication. The details are in Chapter 4. The Barracuda [39] GPU microdriver has evaluated different approaches to implement efficient kernel-user communication for GPU computing.

3.6.2 Implicit Synchronization

The host-device synchronization happens not only when the host side explicitly calls `cudaDeviceSynchronize` function, but also when some GPU resource management operations are performed [4]. Such operations include CUDA host or device memory allocation, GPU information query, etc. The host and device memory allocation is the main trouble

maker because it is almost unavoidable. This may stall the aforementioned asynchronous GPU programming in Section 3.4 even when there is no explicit stream synchronization. As a result, as we will see in Chapter 6, a common technique used by many system-level GPU computing works including our `GPUstore` and `Snap` frameworks is to preallocate CUDA page-locked memory, and manage the memory allocation on their own. This can easily consume a lot of memory, but due to the current GPU limitations, it is a must to achieve asynchronous GPU computing.

3.7 Summary

In this chapter, we discussed generic system-level challenges and also proposed high-level principles and techniques to deal with them. The proposed principles and techniques are designed to make system code efficiently work with the throughput-oriented GPU architecture, take advantage of the wide GPU memory interface, do nonblocking host and device communication and reduce unnecessary overheads during GPU DMA. These principles are mainly about batching to provide parallel workloads, truly asynchronously programming GPUs with callbacks or status polling at the CPU side, compacting workload data to reduce unnecessary PCIe transfer, and using locked memory directly to avoid double-buffering DMA. In the next two chapters, we will discuss our two concrete frameworks: `GPUstore` and `Snap`, to explain how we apply these generic principles and techniques in practice to deal with their specific problems.

CHAPTER 4

GPUstore: GPU COMPUTING FOR STORAGE

This chapter covers the design, implementation and evaluation of GPUstore. GPUstore is a framework for integrating GPU computing into storage systems in Linux kernel. Different from the systems surveyed in Section 6.1.2, GPUstore is a generic framework, not for a particular storage application or subsystem. It has been designed to collaborate with the storage subsystem in Linux kernel in order to use the GPU as a coprocessor. We try to minimize the source code change for a storage component to use GPU computing. So GPUstore follows the OS working style and utilizes existing resource management mechanisms to avoid any fundamental change in the OS kernel. GPUstore has been evaluated with three storage system case studies, showing its efficiency and effectiveness.

4.1 Design

This section will go into the details of how GPUstore has been designed to apply the generic technical principles discussed in Chapter 3 in practice. The following text first takes an overview of the architecture, then discusses specific aspects including memory management, request scheduling and stream management.

4.1.1 Overview

GPUstore has three main functional blocks: memory management, request management and streams management, as shown in Figure 4.1. There are also GPU "services" that don't belong to the framework, but are managed by GPUstore and essential to provide GPU computing tasks. GPUstore abstracts the computing tasks into "services." So storage components request for services to get computational functionalities. Services are modular libraries that are dynamically linked with GPUstore. There is a generic service interface provided by GPUstore to use and manage concrete services. Due to the closed source GPU driver and library, GPUstore has to use a userspace helper to interact with the userspace

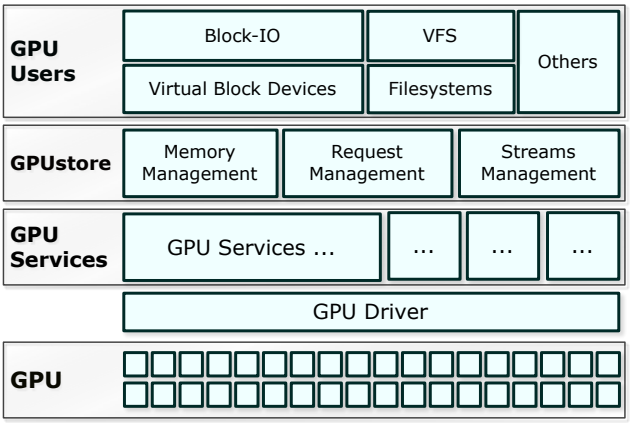


Figure 4.1. The architecture of `GPUstore`.

CUDA GPU library. A GPU service also has to be split into two parts: the kernel part and the userspace part. The kernel part mainly hides the `GPUstore` application programming interface (API) calls to turn a function call invoked from storage components into a GPU service request. The userspace part of a service deals with the necessary host-device memory copies and GPU kernel launching to process a request. The workflow of the three example GPU-accelerated storage systems built with `GPUstore` are illustrated in Figure 4.2.

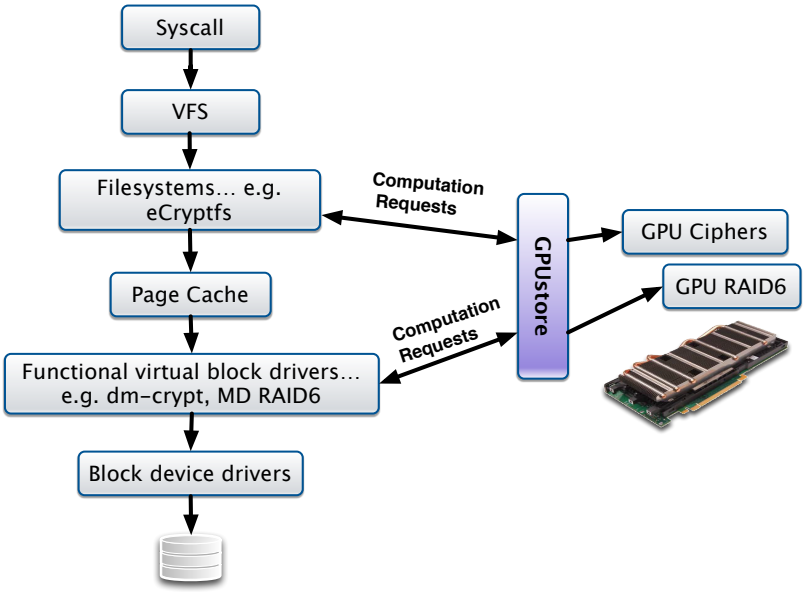


Figure 4.2. The workflow of a `GPUstore`-based storage stack.

4.1.2 Memory Management

`GPUstore` manages both host memory and device memory that are used for GPU computing. All the host memory is CUDA page-locked memory to achieve best memory copy performance, and also preallocated to avoid stalling the asynchronous GPU computing, as discussed in Section 3.6.2. `GPUstore` simplifies the device memory management by maintaining a one-to-one mapping between the host memory and the device one. The downside of such one-to-one mapping is that it makes suboptimal use of host memory: buffers must remain allocated even when not in active use for copies.

The Linux kernel storage workflow can be treated as a long pipeline, as illustrated in Figure 4.2. Each layer may either allocate memory buffer for new data processing, or accept data buffers from neighbor layers to process. Most computing tasks in the storage systems work on the entire data rather than just small portions of them. That said in Section 3.5.4, such data usage pattern requires avoiding double-buffering when integrating GPU computing. According to our AES cipher evaluation shown in Figure 4.3, avoiding double-buffering adds almost 3x speedup to the cipher performance.

`GPUstore` provides API to do memory remapping for components processing data buffers received from others, and also in-kernel CUDA memory allocation for components creating its own buffers for data processing. For example, `eCryptfs` in our case studies is a good candidate to use remapping because all the memory pages it uses are allocated and managed by kernel page-cache, while `dm-crypt` is a good fit for allocating its own buffers capable of GPU DMA because its code creates and manages buffers for the encrypted or decrypted

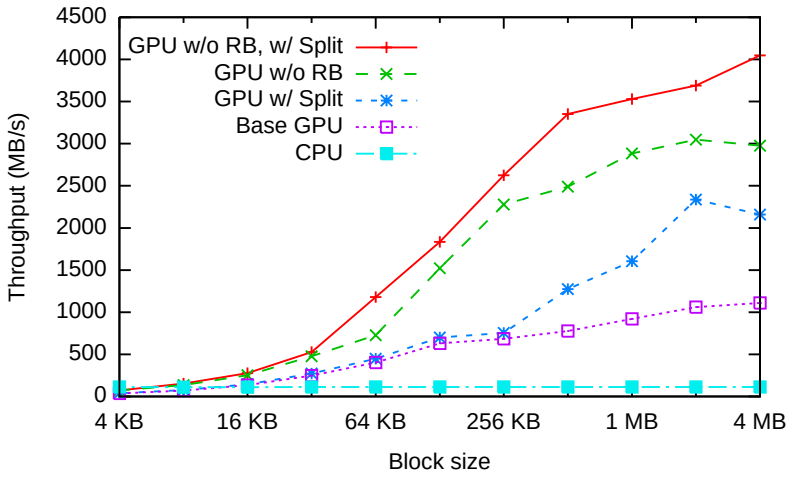


Figure 4.3. GPU AES cipher throughput with different optimizations compared with Linux kernel's CPU implementation. The experiments marked "w/o RB" use the techniques described in Section 4.1.2 to avoid double-buffering.

data.

"Remapping" means mapping normal memory pages into GPU driver's memory area to let it treat them the same as its page-locked memory that is capable of DMA. Remapping is not a safe way in some cases: some memory pages such as the `mmap` ones may not be allowed to do further remapping in Linux kernel. We "rudely" manipulate the page tables for remapping regardless of the potential problem. Although we haven't got any errors for our case studies, it may cause bugs in some circumstances.

Allocating CUDA page-locked memory in Linux kernel is achieved with the `GPUstore` userspace helper. The helper is responsible for all the userspace CUDA calls, and the kernel part takes care of necessary address translation for kernel code use and manages the memory in a memory pool.

4.1.3 Request Management

4.1.3.1 Request Processing

GPU service requests are submitted to `GPUstore` by storage components (via the kernel part GPU service), passed across the kernel-user boundary to the userspace helper, then processed by the requested services, and finally get the response. `GPUstore` processes service requests asynchronously to avoid breaking the normal workflow of storage components as we discussed in Section 3.4. So each request has a callback that is invoked after it has been processed.

`GPUstore` exposes a Linux character device file for the kernel-user communication. The userspace helper `read`s submitted requests, and after a request has been processed, it `write`s to that file for completion notification. The userspace helper works completely asynchronously to achieve low latency. It does nonblocking `read` on the `GPUstore` device file, and also relies on the nonblocking `write` implemented by the device file. `GPUstore` implements such nonblocking `write` using a separate kernel thread that is blocked on a request completion queue, which is filled by the device file's `write`. The kernel thread is waked up on available elements in the completion queue and responsible for invoking the request callbacks.

`GPUstore` defines a generic service interface, which includes the following three common steps and operations to process a request.

- `PrepareProcessing` allows a service to do some preparation before processing a request, such as calculating CUDA kernel execution parameters, launching asynchronous host-to-device memory copy, etc.

- `LaunchKernel` does the CUDA kernel launching operation, which is an asynchronous operation.
- `PostProcessing` performs asynchronous device-to-host memory copy and other necessary operations in specific services.

A service only does asynchronous GPU operations in each step to avoid blocking the userspace helper's workflow. One service doesn't have to follow the typical host-to-device copy, GPU kernel execution and device-to-host copy GPU computing steps. For example, two services can collaborate in this way: the first service produces an intermediate result, without copying it back to the host, and then the second service processes the result on GPU directly, no host-to-device copy needed. This allows `GPUstore` users to do efficient data flow control by saving unnecessary memory copies, which is similar to the pipe data flow model in PTask [40].

4.1.3.2 Request Scheduling

In the userspace helper, request processing is started as first-come-first-serve. However, due to the totally asynchronous CUDA calls, the completion order of requests is not guaranteed. `GPUstore` doesn't try to maintain the order of requests to allow fully asynchronous processing. Users who needs such processing order must maintain it via the request callbacks or special logic in GPU services.

Having said in Section 3.1.1 and Section 3.3: GPUs need enough parallelism in one-shot computing workload which is often accumulated via batching; GPUs also need to pipeline computation and memory copy to improve the utilization of GPU components. But most kernel storage code is unaware of GPUs, and hence can hardly issue requests with GPU-friendly workload sizes. Changing exist code to make GPU-friendly workloads may lead to significant amount of code refactoring. `GPUstore` tries to reduce the amount of such refactoring by merging or splitting accepted requests.

The *merge* operation is performed on small requests (for the same service), which is an analogy of batching as said in Section 3.1.1. The *split* operation is on large requests to utilize the overlapped GPU computation and memory copy technique (see Section 3.3). Both merging and splitting requires the service-specific knowledge to guarantee the correctness of the result requests, so `GPUstore` doesn't perform the actual merging and splitting. Instead, it depends on each GPU service to do them. So GPU services can optionally implement merging or splitting logic, which will be utilized by `GPUstore` for request scheduling. In the current `GPUstore` system prototype, all GPU services use predefined constant values to decide their optimal request sizes for best performance, which may be affected by runtime

system load and other factors. Implementing dynamically adjustable merging or splitting parameters is an interesting future work.

merge is not the "ultimate" solution to produce GPU-friendly workloads with enough parallelism. *merge* can only process requests already submitted to `GPUstore`. If a storage component explicitly splits a large workload to process one much smaller unit at a time in a loop, it causes unnecessarily sequential constraints on requests: the loop must wait until completion of the previous request in order to do the next round. In that case, *merge* can't find enough requests in request queue to merge at all. The existing code may have to be refactored to produce bulk-data requests.

4.1.4 Streams Management

CUDA streams are assigned to service requests in `GPUstore` to achieve sequentially executed operations on the same request. CUDA stream is also a GPU computing resource abstraction, which is the analogy of a CPU thread or process. So managing the allocation is analogous to process scheduling on CPUs. Due to the functionally limit, GPUs are not capable of preemptive execution, so `GPUstore` uses first-come-first-serve policy to allocate streams to requests, and the request processing is unpreemptable (once started). Even with current GPU functional constraints, it is still possible to enable prioritized request execution with techniques similar to TimeGraph [41] and Gdev [38], though it may need significant engineering work to make use of the immature open source GPU drivers and CUDA libraries they depend on.

4.2 Implementation

`GPUstore` has been prototyped on Linux kernel to accelerate three existing kernel storage components. We enhanced encrypted storage with `dm-crypt` and `eCryptfs`, and the software RAID (redundant array of inexpensive disks) driver `md`. We chose these three subsystems because they interact with the kernel in different ways: `md` and `dm-crypt` implement the block I/O interface, and `eCryptfs` works with the VFS layer.

The design of `GPUstore` ensures that client subsystems need only minor modifications to call GPU services. Table 4.1 gives the approximate numbers of lines of code that we had to modify for our example subsystems. The lines of code reported in this table are those in the subsystems that are modified to call `GPUstore`, and do not include the lines of code used to implement the GPU services. Linux storage subsystems typically call other reusable kernel components to perform common operations such as encryption. Essentially, we replace these with calls to `GPUstore` and make minor changes to memory management.

Table 4.1. Modified lines of code (LOC) to use `GPUstore`.

Subsystem	Total LOC	Modified LOC	Percent
`dm-crypt`	1,800	50	3%
`eCryptfs`	11,000	200	2%
`md`	6,000	20	0.3%

4.3 Evaluation

We benchmarked the `GPUstore` framework itself as well as the three storage subsystems that we adapted to use it. We used two machine configurations, *S1* and *S2* for our evaluation. *S1* is used for file system and block device tests. *S2* is used for the RAID benchmarks.

All benchmarks were run without use of *hybrid mode* in `GPUstore`, that is, GPU services were not allowed to fall back to the CPU for small requests. This has the effect of clearly illustrating the points where the GPU implementation, by itself, underperforms the CPU, as well as the points where their performance crosses. With *hybrid mode* enabled, `GPUstore` would use the CPU for small requests, and the CPU performance can thus be considered an approximate lower bound for `GPUstore`'s hybrid mode performance.

In many cases, our GPU-accelerated systems are capable of out-performing the physical storage devices in our systems; in those cases, we also evaluate them on DRAM-backed storage in order to understand their limits. These DRAM-based results suggest that some `GPUstore` accelerated subsystems will be capable of keeping up with multiple fast storage devices in the same system, or PCIe-attached flash storage, which is much faster than the drives available for our benchmarks.

4.3.1 Framework Performance

Our first microbenchmark examines the effect of block sizes on `GPUstore`'s performance and compares synchronous operation with asynchronous. On *S1*, we called a GPU service which performs no computation: it merely copies data back and forth between host memory and GPU device memory. Note that total data transfer is double the block size, since the data block is first copied to GPU memory and then back to host memory. In Figure 4.4, we can see that at small block sizes, the PCIe bus overheads dominate, limiting throughput. Performance steadily increases along with block size, and reaches approximately 4 GB/s on our system. This benchmark reveals three things. First, it demonstrates the value to be gained from our *merge* operation, which increases block sizes. Second, it shows a performance boost of 30% when using asynchronous, rather than synchronous, requests to

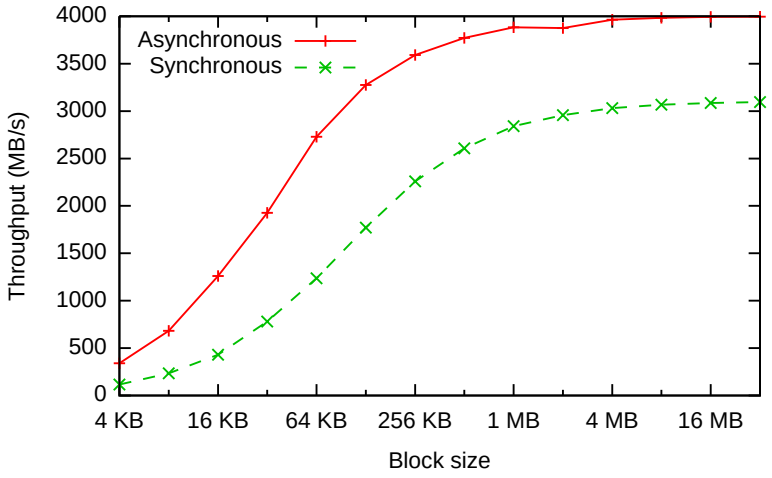


Figure 4.4. Throughput of a GPU kernel that copies data but performs no computation.

the GPU. Finally, it serves as an upper bound for performance of GPU services, since our test service performs no computation.

Our second microbenchmark shows the effects of our optimization to remove redundant buffering and the *split* operation. This benchmark, also run on *S1*, uses the AES cipher service on the GPU, and the results can be seen in Figure 4.3. The baseline GPU result shows a speedup over the CPU cipher, demonstrating the feasibility of GPU acceleration for such computation. Our *split* operation doubles performance at large block sizes, and eliminating redundant buffering triples performance at sizes of 256 KB or larger. Together, these two optimizations give a speedup of approximately four times, and with them, the GPU-accelerated AES cipher achieves a speedup of 36 times over the CPU AES implementation in the Linux kernel. The performance levels approach those seen in Figure 4.4, implying that the memory copy, rather than the AES cipher computation, is the bottleneck.

4.3.2 Encrypted Device Mapper

Next, we use the `dd` tool to measure raw sequential I/O speed in `dm-crypt`. The results shown in Figure 4.5 indicate that with read and write sizes of about 1MB or larger, the GPU-accelerated `dm-crypt` easily reaches maximum throughput of our solid state disk (SSD): 250MB/s read and 170MB/s write. The CPU version is 60% slower; while it would be fast enough to keep up with a mechanical hard disk, it is unable reach the full potential of the SSD. Substituting a DRAM disk for the SSD (Figure 4.6), we see that the GPU-accelerated `dm-crypt` was limited by the speed of the drive: it is able to achieve a maximum read throughput of 1.4 GB/s, more than six times as fast as the CPU implementation.

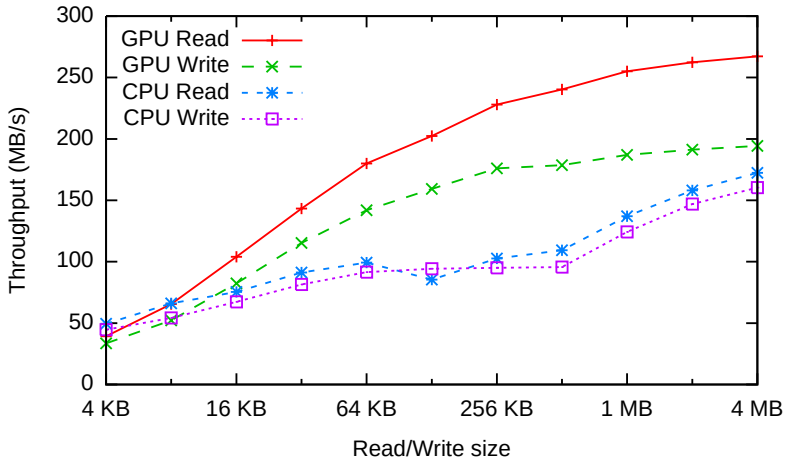


Figure 4.5. `dm-crypt` throughput on an SSD-backed device.

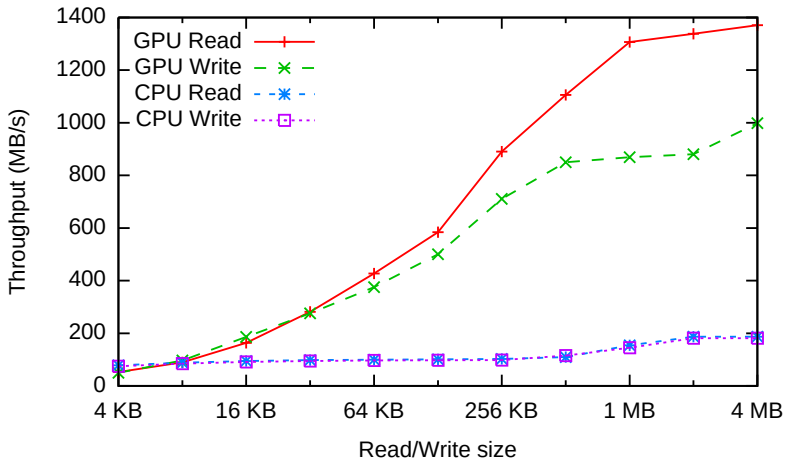


Figure 4.6. `dm-crypt` throughput on a DRAM-backed device.

This is almost exactly the rated read speed for the ioDrive Duo, the third fastest SSD in production [42] at the time of developing `GPUstore` (in 2012). As the throughput of storage systems rises, GPUs present a promising way to place computation into those systems while taking full advantage of the speed of the underlying storage devices.

4.3.3 Encrypted File System

We evaluated both sequential performance and the concurrent performance of `eCryptfs`, as shown in the following sections.

4.3.3.1 Sequential Throughput

Figure 4.7 and Figure 4.8 compare the sequential performance for the CPU and GPU implementation of `eCryptfs`. We used the `iozone` tool to do sequential reads and writes using varying block sizes and measured the resulting throughput. Because `eCryptfs` does not support direct I/O, effects from kernel features such as the page cache and readahead affect our results. To minimize (but not completely eliminate) these effects, we cleared the page-cache before running read-only benchmarks, and all writes were done synchronously.

Figure 4.7 shows that on the SSD, the GPU achieves 250 MBps when reading, compared with about 150 MBps for the CPU, a 70% speed increase. Unlike our earlier benchmarks, read speeds remain nearly constant across all block sizes. This is explained by the Linux

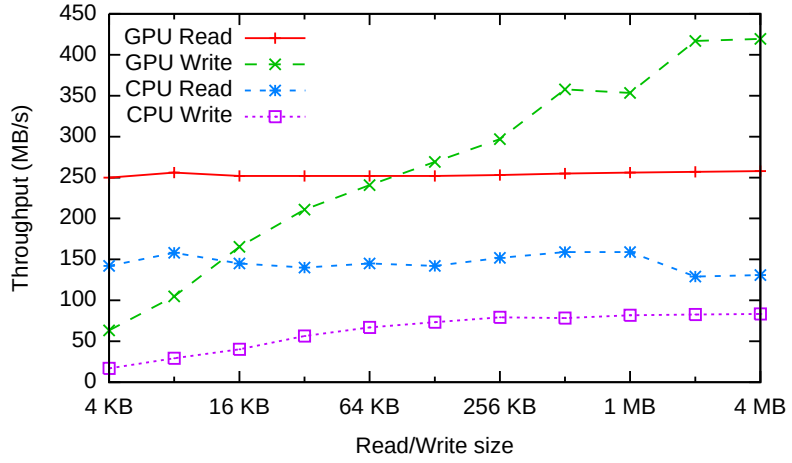


Figure 4.7. `eCryptfs` throughput on an SSD-backed file system.

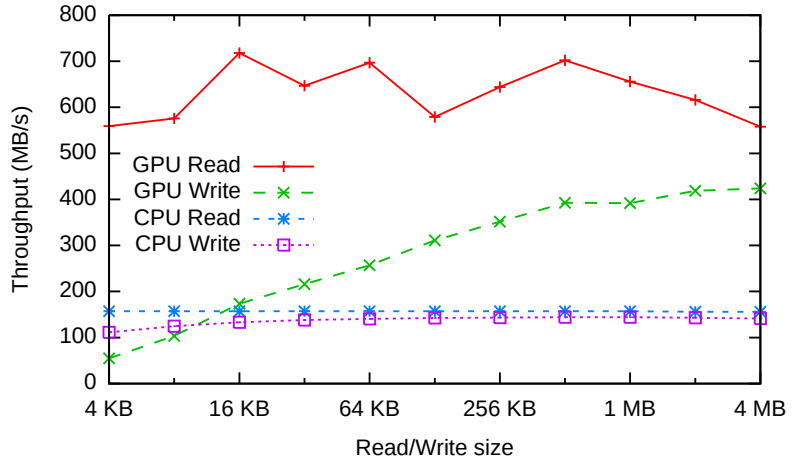


Figure 4.8. `eCryptfs` throughput on a DRAM-backed file system.

page-cache's readahead behavior: when small reads were performed by `iozone`, the page-cache chose to issue larger reads to the filesystem in anticipation of future reads. The default readahead size of 128 KB is large enough to reach the SSD's full read speed of 250MB/s. This illustrates an important point: by designing `GPUstore` to fit naturally into existing storage subsystems, we enable it to work smoothly with the rest of the kernel. Thus, by simply implementing the multipage `readpages` interface for `eCryptfs`, we enabled existing I/O optimizations in the Linux kernel to kick in, maximizing performance even though they are unaware of `GPUstore`.

Another surprising result in Figure 4.7 is that the GPU write speed exceeds the write speed of the SSD, and even its read speed, when block size increases beyond 128 KB. This happens because `eCryptfs` is, by design, "stacked" on top of another file system. Even though we take care to `sync` writes to `eCryptfs`, the underlying file system still operates asynchronously and caches the writes, returning before the actual disk operation has completed. This demonstrates another important property of `GPUstore`: it does not change the behavior of the storage stack with respect to caching, so client subsystems still get the full effect of these caches without any special effort.

We tested the throughput limits of our GPU `eCryptfs` implementation by repeating the previous experiment on a DRAM disk, as shown in Figure 4.8. Our GPU-accelerated `eCryptfs` achieves more than 700 MBps when reading and 420 Mbps when writing. Compared to the CPU, which does not perform much better than it did on the SSD, this is a speed increase of nearly five times for reads and close to three times for writes. It is worth noting that Linux's readahead mechanism not only "rounds up" read requests to 128 KB, it "rounds down" larger ones as well, preventing `eCryptfs` from reaching even higher levels of performance.

4.3.3.2 Concurrent Throughput

We also used `FileBench` to evaluate `eCryptfs` under concurrent workloads. We varied the number of concurrent writers from one to one hundred, and used the DRAM-backed file system. Each client writes sequentially to a separate file. The effects of `GPUstore`'s *merge* operation are clearly visible in Figure 4.9: with a single client, performance is low, because we use relatively small block sizes (128 KB and 16 KB) for this test. But with ten clients, `GPUstore` is able to *merge* enough requests to get performance on par with `dm-crypt` at a 1 MB blocksize. This demonstrates that `GPUstore` is useful not only for storage systems with heavy single-threaded workloads, but also for workloads with many simultaneous clients. While block size still has a significant effect on performance, `GPUstore`

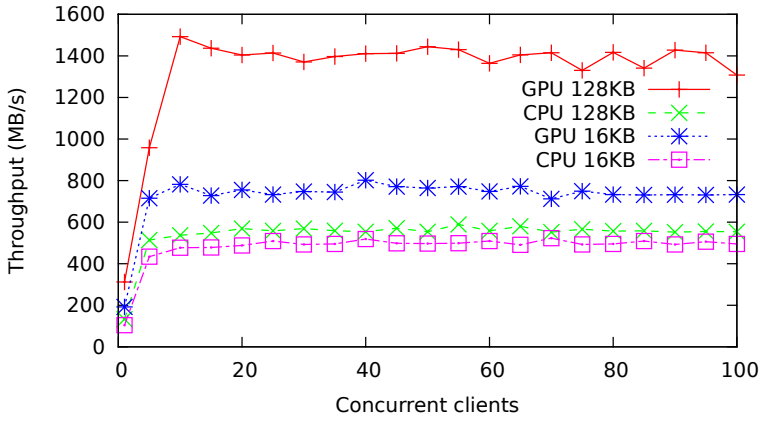


Figure 4.9. `eCryptfs` concurrent write throughput on a DRAM disk for two block sizes.

is able to amortize overheads across concurrent access streams to achieve high performance even for relatively small I/O sizes.

4.3.4 Data Recovery

Similar to encryption, the performance of our GPU-based RAID recovery algorithm increases with larger block sizes, eventually reaching six times the CPU's performance, as seen in Figure 4.10.

We measured the sequential bandwidth of a degraded RAID 6 array consisting of 32 disks in our *S2* experiment environment. The results are shown in Figure 4.11. We find that GPU accelerated RAID 6 data recovery does not achieve significant speedup unless the array is configured with a large chunk size, or strip size. Interestingly, the speedup is

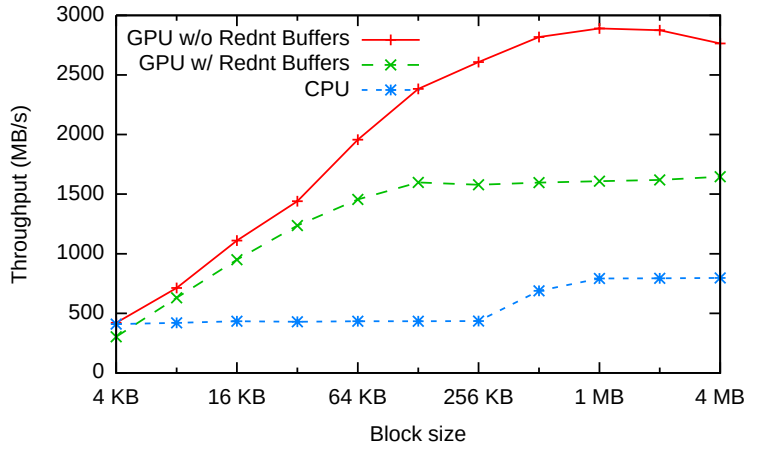


Figure 4.10. Throughput for the RAID recovery algorithm with and without optimizations to avoid redundant buffers.

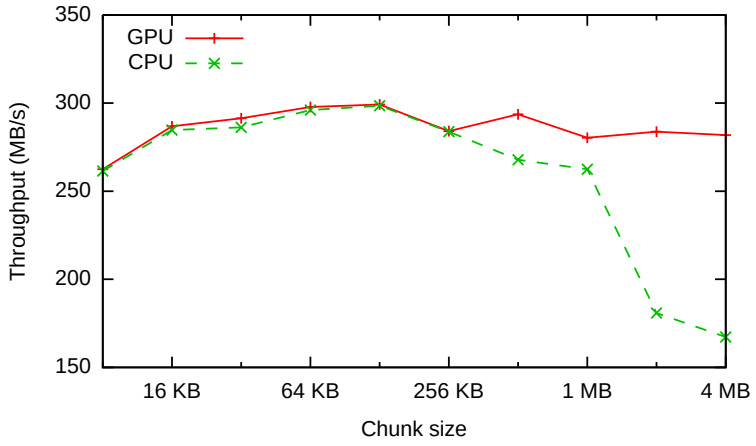


Figure 4.11. RAID read bandwidth in degraded mode.

caused by decreasing CPU performance. We believe the decrease is caused by the limit-sized design of `md`'s I/O request memory pool, which does not efficiently handle large numbers of requests on large stripes. Because `GPUstore` *merge*s these requests into larger ones, it avoids suffering from the same problem.

We also measured the degraded mode performance of a RAID array in the *S1* system using 6 DRAM disks. The results are shown in Figure 4.12. We find that our previous recovery experiment was limited by the speed of the hard disks, and both CPU and GPU implementations would be capable of faster performance given faster disks. With DRAM disks, the CPU based recovery reaches the maximum throughput we saw in Figure 4.10, while the GPU version is still far from its own maximum in the figure.

However, with chunk sizes below 16KB, the throughputs on DRAM disk arrays are

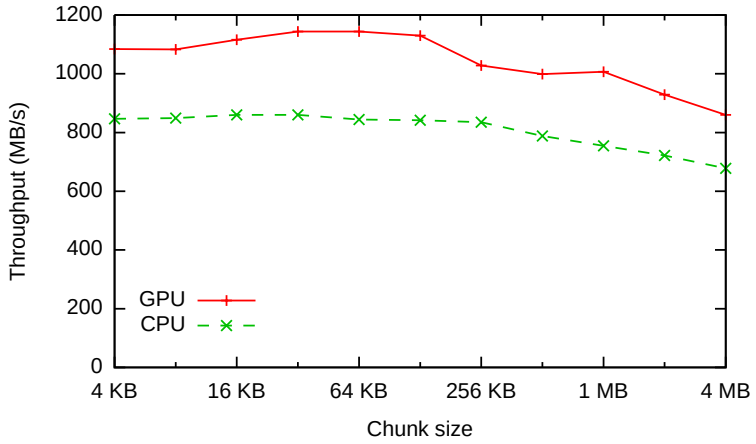


Figure 4.12. RAID read bandwidth in degraded mode on DRAM-backed devices.

actually much higher than we saw for the raw algorithm in Figure 4.10. This result demonstrates the effectiveness of the request *merge* operation in `GPUstore`. *merge* was in use for recovery benchmark, but not the raw algorithm test, and the former therefore saw larger effective block sizes.

4.4 Summary and Future Work

`GPUstore` is a general-purpose framework for using GPU computing power in storage systems within the Linux kernel. By designing `GPUstore` around common storage paradigms, we have made it simple to use from existing code, and have enabled a number of optimizations that are transparent to the calling system. We modified several standard Linux subsystems to use `GPUstore`, and were able to achieve substantial improvements in performance by moving parts of the systems' computation on to the GPU. Our benchmark results also demonstrate the effectiveness of the optimizations adopted by `GPUstore` for matching the storage subsystems requirements. The source code of `GPUstore` is released at http://code.google.com/p/kgpu/.

Because of its in-kernel target applications, `GPUstore` has to suffer from the kernel-user switching overhead and technical workarounds to implement in-kernel CUDA memory management. A promising future work will be migrating `GPUstore` onto open source GPU drivers such as nouvea [37] and the in-kernel CUDA runtime Gdev [38]. Both of them eliminate the overhead due to kernel-user switching and resource management and provide direct GPU computing primitive access inside Linux kernel. Authors of Gdev [38] have actually done a simple, `eCryptfs` only migration of `GPUstore` to Gdev, and still got pretty large speedup over CPU implementations. We believe that with future open specification GPUs and their open source drivers and computing libraries, `GPUstore` will be more efficient on top of them.

CHAPTER 5

Snap: PACKET PROCESSING WITH CLICK AND GPUS

This chapter describes Snap, the GPU-accelerated network packet processing framework. As networks advance, the need for high-performance packet processing in the network increases for two reasons: first, networks get faster, and second, we expect more functionality from them [43, 44, 45, 46, 47, 48]. The nature of packet data naturally lends itself to parallel processing [49], and as we shall see in the survey chapter (Chapter 6), a wide variety of network functionalities are capable of parallel GPU accelerations. However, a software router is made up of more than just these heavyweight processing and lookup elements. A range of other elements are needed to build a fully functional router, including "fast path" elements such as time-to-live (TTL) decrement, checksum recalculation, and broadcast management, and "slow path" elements such as handling of IP options, Internet control and management protocol (ICMP), and address resolution protocol (ARP). Building new features not present in today's routers adds even more complexity. To take full advantage of the GPU in a packet processor, what is needed is a flexible, modular framework for building complete processing pipelines by composing GPU programs with each other and with CPU code.

We have designed and implemented Snap to address this need. It extends the architecture of the Click modular router [19] to support offloading parts of a packet processor onto the GPU. Snap enables individual elements, the building blocks of a Click processing pipeline, to be implemented as GPU code. It extends Click with "wide" ports that pass batches of packets, suitable for processing in parallel, between elements. Snap also provides elements that act as adapters between serial portions of the processing pipeline and parallel ones, handling the details of batching up packets, efficiently copying between main memory and the GPU, scheduling GPU execution, and directing the outputs from elements into different paths on the processing pipeline. In addition to these user-visible changes to Click, Snap also makes a number of "under the hood" changes to the way that Click

manages memory and optimizes its packet I/O mechanisms to support multiple 10 Gbps network interface card (NIC) rates.

The following sections start from introducing simple Click background, and several motivating experiments we have done for `Snap` work to convince you of the need for `Snap`. Then we shall discuss in detail the design and implementation of `Snap` to describe how it extends Click to efficiently use parallel GPUs to accelerate packet processing. And last we evaluate `Snap` from a variety of aspects to demonstrate its performance and also the flexibility and modularity derived from Click.

5.1 Click Background

Click is a modular software router that provides an efficient pipeline-like abstraction for packet processing on the hardware. A packet processor is constructed by connecting small software modules called "elements" into a graph called a "configuration." Click elements have two kinds of "ports": input ports and output ports, and a single element may have more than one of each. A connection between two elements is made by connecting an output port of one element to an input port of another. Packets move along these connections when they are pushed or pulled; an element at the head of the pipeline can push packets downstream, or packets can be pulled from upstream by elements at the tail of the pipeline. Packets typically enter Click at a `FromDevice` element, which receives them from a physical NIC and pushes them downstream as they arrive. Unless dropped, packets leave through a `ToDevice` element, which pulls them from upstream and transmits them as fast as the outgoing NIC allows. Queues are used to buffer packets between push and pull sections of the configuration.

At the C++ source-code level, elements are written as subclasses of a base `Element` class, ports are instances of a `Port` class, and network packets, which are represented by instances of the `Packet` class, are passed one at a time between `Element`s by calling the elements' `push`() or `pull`() methods. We run Click at user level—although Click can run directly in the kernel, with the Netmap [50] zero-copy packet I/O engine, user-level Click has a higher forwarding rate than the kernel version [50].

5.2 Motivating Experiments

Our work on `Snap` is motivated by two facts: (1) rich packet processing functionality can represent a major bottleneck in processing pipelines; and (2) by offloading that functionality onto a GPU, large performance improvements are possible, speeding up the entire pipeline.

We demonstrate these facts with two motivating experiments. (Our experiment setup and methodologies are described in more detail in Section 5.5.)

Our first experiment starts with the simplest possible Click forwarder, shown at the top of Figure 5.1. It does no processing on packets. It simply forwards them from one interface to another. We then add, one at a time, elements that do IP route lookup, classification based on header bits as in most software-defined network (SDN) designs, and string matching that is used in many intrusion detection systems (IDS) and deep-packet-inspection (DPI) firewalls. The relative throughputs of these four configurations, normalized to the throughput of the "Simple Forwarder," are compared in Table 5.1. We can clearly see from this table that the addition of even a single and ordinary processing task into the forwarding pipeline can significantly impact performance, cutting throughput by as much as 43%. In short, processing does represent a bottleneck, and if we can speed it up, we can improve router throughput.

Our next experiments compare the performance of these three processing element when run on the CPU and on the GPU. These experiments involve no packet I/O—we are simply interested in discovering whether the raw performance of the GPU algorithms offers enough of a speedup to make offloading attractive. We process packets in batches, which is necessary to get parallel speedup on the GPU. The results are shown in Figure 5.2. Two things become clear from these graphs. First, GPUs do indeed offer impressive speedups for these tasks: we see a 16x speedup for IP route lookup: 559 Mpps (million packets per second) on the GPU versus 34.7 Mpps on the CPU. Second, fairly large batches of packets are needed to achieve this speedup. These results are in line with findings from earlier studies [24, 22].

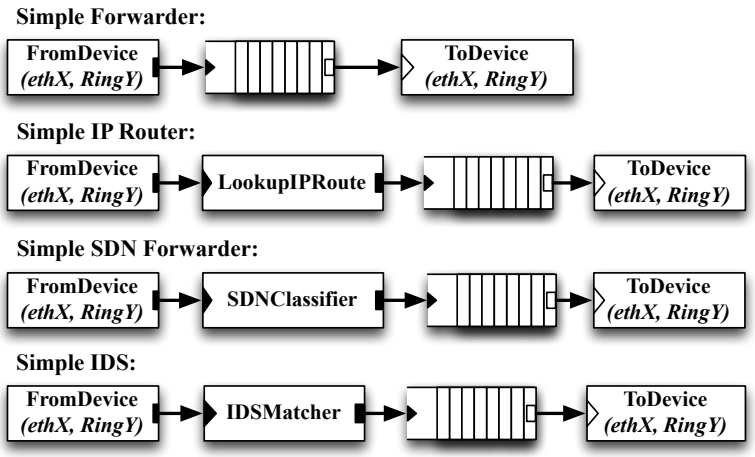
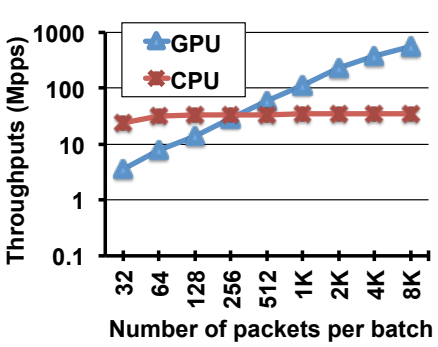


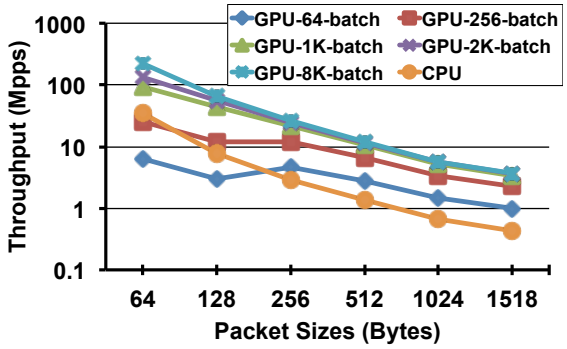
Figure 5.1. Simplified Click configurations for motivating experiments.

Table 5.1. Relative throughputs of simple processing pipelines.

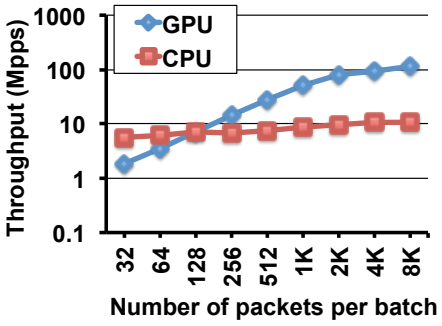
Configuration	Throughput	Relative Throughput
Simple Forwarder	30.97 Gbps	100%
IP Router	19.4 Gbps	62.7%
IDS	17.7 Gbps	57.3%
SDN Forwarder	18.8 Gbps	60.7%



(a) IP lookup algorithm (radix tree).



(b) Pattern matching algorithm.



(c) Classifier algorithm.

Figure 5.2. GPU versus CPU performance on packet processing algorithms. Note that the axes are nonlinear.

GPUs are not appropriate for every type of packet processing element. In particular, elements that require a guarantee that they see every packet in a flow in order, or that have heavy state synchronization requirements, are not well-suited to massively parallel processing. Our challenge in `Snap` is to make it possible to take advantage of GPU parallelism in a practical way that preserves the inherent composability and flexibility of Click, including incorporation of CPU elements into the processing pipeline.

5.3 Design

We designed `Snap` with two goals in mind: enabling *fast* packet processors through GPU offloading while preserving the *flexibility* in Click that allows users to construct complex pipelines from simple elements. `Snap` is designed to offload specific elements to the GPU: parts of the pipeline can continue to be handled by existing elements that run on the CPU, with only those elements that present computational bottlenecks reimplemented on the GPU. From the perspective of a developer or a user, `Snap` appears very similar to regular Click, with the addition of a new type of "batch" element that can be implemented on the GPU and a set of adapter elements that move packets to and from batch elements. Internally, `Snap` makes several changes to Click in order to make this pipeline work well at high speeds. Several themes appear in our design choices. In many cases, we find that if we do "extra" work, such as making copies of only the necessary parts of a packet in main memory, or passing along packets that we know will be discarded, we can decrease the need for synchronization and reduce our use of the relatively slow PCIe bus. We also find that scheduling parts of the pipeline asynchronously works well, and fits naturally with Click's native push/pull scheduling. In this section, we walk through the design and implementation of `Snap`, starting at a high level with the user-visible changes, and progressing through the low-level changes that stem from these high-level decisions.

5.3.1 Batched Packet Processing

GPUs need parallel workloads to fully utilize their large number of cores, as discussed in Section 3.1.1. To provide and to process parallel workloads in `Snap`, we designed a batched processing mechanism to batch a large amount of packets and process them on GPUs by one time GPU kernel execution. We extended Click's single packet processing pipeline to support multiple packets processing in a batch. We also designed special elements and efficient data structures to do the batching and to store the batched packets. The following sections will describe the details of our batched processing design in `Snap`.

5.3.1.1 Wider Pipeline

In standard Click, the connection between elements is a single packet wide: the `push`() and `pull`() methods that pass packets between elements yield one packet each time they are invoked. To efficiently use a GPU in the pipeline, we added wider versions of the `push`() and `pull`() interfaces, `bpush`() and `bpull`(). These methods exchange a new structure called a *PacketBatch*, which will be described in more detail in the following section. We also made Click's `Port` class aware of these wider interfaces so that it can correctly pass *PacketBatch*es

between elements. `bpush`() and `bpull`() belong to a new base class, `BElement`, which derives from Click's standard `Element` class.

In standard Click, to implement an element, the programmer creates a new class derived from `Element` and overloads the `push`() and `pull`() methods. This is still supported in `Snap`; in fact, most of our pipelines contain many unmodified elements from the standard Click distribution, which we refer to as "serial" elements. To implement a parallel element in `Snap` the programmer simply derives it from `BElement` and overrides the `bpush`() and `bpull`() methods.

A GPU-based parallel element is comprised of two parts: a GPU side, which consists of GPU kernel code, and a CPU side, which receives *PacketBatch*es from upstream elements and sends commands to the GPU to invoke the GPU kernel. `Snap` provides a `GPURuntime` object to help Click code interact with the GPU, which is programmed and controlled using CUDA [4]. GPU-based elements interact with `GPURuntime` to request GPU resources such as memory. The GPU kernel is written in CUDA's variant of C or C++, and is wrapped in an external library that is linked with the element sources when compiling `Snap`. Typically, each packet is processed by its own thread on the GPU.

5.3.1.2 Batching

The `BElement` class leaves us with a design question: how should we collect packets to form *PacketBatch*es, and how should we manage copies of *PacketBatch*es between host and GPU memory? Our answer to this question takes its cue from the functioning of Click's `Queue` elements. Parts of a Click configuration operate in a push mode, with packets arriving from a source NIC; other parts of the configuration operate in pull mode, with packets being pulled along towards output NICs. At some point in the configuration, an adapter must be provided between these two modes of operation. The family of `Queue` elements plays this role. In practice, the way a packet is processed in Click is that it is pushed from the source NIC through a series of elements until it reaches a `Queue`, at which point it is deposited there and Click returns to the input NIC to process the next packet. On the output side of the `Queue`, the packet is dequeued and processed until it reaches the output NIC.

In an analogous manner, we have created a new element, `Batcher`, which collects packets one at a time from a sequential `Element` on one side and pushes them in batches to a `BElement` on the other side. A `Debatcher` element performs the inverse function. `Batcher` can be configured to produce *PacketBatch*es with specified *batch-size* packets, or fewer if a specified *timeout* period passed. Implementing this functionality as a new element, rather than changing Click's infrastructure code, has three advantages. First, it minimizes the

changes within Click itself. Second, it makes transitions between CPU and GPU code explicit; since there are overheads associated with packet batching, it is undesirable for it to be completely invisible to the user. Third, and most important, it means that batching and offloading are fully under the control of the creator of the `Snap` configuration—while we provide carefully-tuned implementations of `Batcher` and `Debatcher` elements, it is possible to provide alternate implementations designed for specific uses (such as `BElement`s running on devices other than GPUs) without modifying `Snap`.

5.3.1.3 Data Structure

The *PacketBatch* data structure (shown in Figure 5.3) that represents a batch of packets has been carefully designed specially for offloading computation to GPUs. A *PacketBatch* is associated not only with a collection of packets (represented by Click's `Packet` objects), but also with allocations of host and GPU device memory. Large consecutive buffers are used in host and GPU memory in order to enable efficient DMA transfers, minimizing the overhead of setting up multiple transfers across the PCIe bus.

The large buffers of a *PacketBatch* are split into small buffers, which contain the slices of packets (such as the headers) that are needed by the `BElement`(s). `Snap` does such slicing and extra host memory buffer because of the special data usage pattern of most packet processing tasks. As we have discussed in Section 3.5.3 and Section 3.5.4, network packet processing is a representative task that only needs small pieces of a given data unit. Such

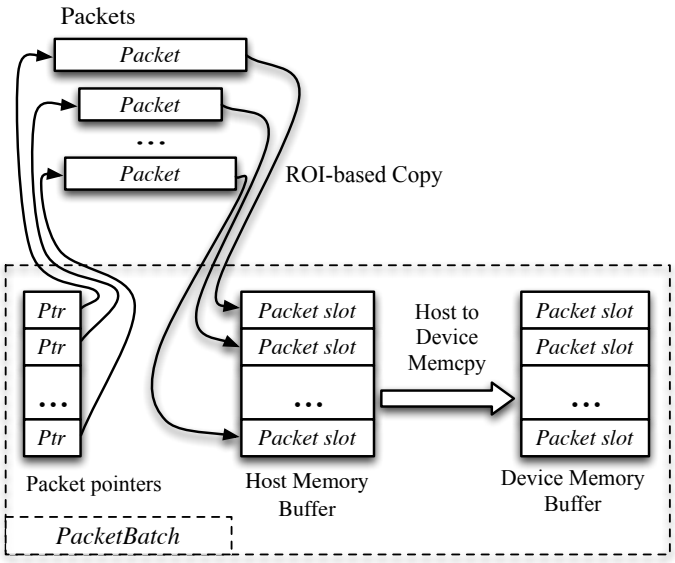


Figure 5.3. The *PacketBatch* structure.

data usage pattern needs a trade-off of the relatively large host memory bandwidth and the relatively small host-device memory bandwidth in order to effectively reduce the total memory copy overhead.

We designed packet slicing (illustrated in Figure 5.4) for `Snap` to deal with this problem. Slicing allows GPU processing elements to specify the regions of packet data that they operate on, called regions of interest (ROIs). An ROI is a consecutive range in the packet data buffer, and a GPU processing element can have multiple ROIs spread throughout the packet. `Batcher` accepts ROI requests from its downstream `BElement`s and takes their union to determine which parts of the packet must be copied to the GPU. It allocates only enough host and device memory to hold these ROIs, and during batching, `Batcher` only copies data in these regions into a packet's host memory buffer. This reduces both the memory requirements for *PacketBatch*es and the overhead associated with copying them to and from the GPU. During debatching, ROIs are selectively copied back into the Click `Packet` structure. Slicing is one reason that we chose not to use a zero-copy approach for *PacketBatch*es.

`Batcher` contains optimizations to avoid redundant copies in the case of ROIs that overlap and to combine `memcpy()` calls for consecutive ROIs to reduce function call overhead. For element developers' convenience, we have provided helper API for `BElement`s that allow the element to address packet data relative to its ROIs—the true offsets within the *PacketBatch* are computed transparently.

One problem associated with ROIs is that it may be difficult to describe the exact range

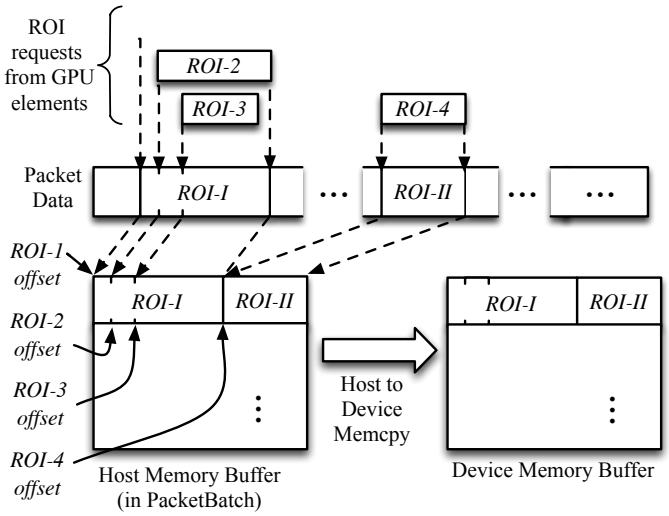


Figure 5.4. A slicing example.

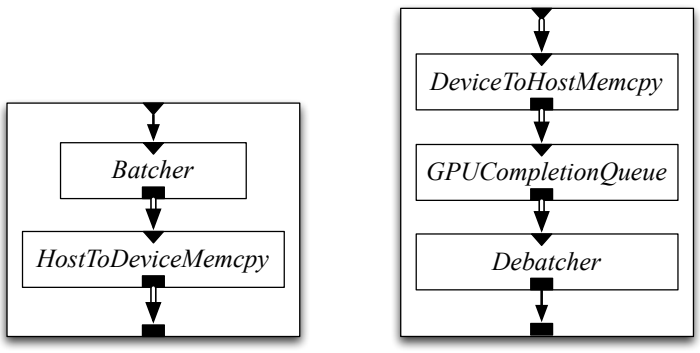
in numeric values. For example, a `Classifier` element may need transmission control protocol (TCP) port numbers, but the offset of this data within a packet is not constant due to the presence of IP option headers. To support this case, `Batcher` provides some special values to indicate variable offsets, such as the beginning of the TCP header or the end of the IP header. This also enables some special ROIs, such as ROIs that request the entire IP header including all IP options, or ROIs that cover the payload of the packet.

5.3.1.4 Flexible Memory Copy

GPU code requires both input data and output results to reside in GPU memory, making it necessary to copy packets back and forth between main memory and the GPU across the PCIe bus. `Snap` factors this task out of the `BElement`s that contain processing code: a `HostToDeviceMemcpy` element (provided as part of `Snap`) is placed between the `Batcher` and the first element that runs on the GPU. An analogous `DeviceToHostMemcpy` element is placed before the `Debatcher`. Multiple GPU elements can be placed between a `HostToDeviceMemcpy`/`DeviceToHostMemcpy` pair, allowing the output ports of one to feed into the input ports of another without incurring a copy back to host memory. This design reduces the host-device packet copy times, reducing the overall memory copy overhead. These memory copy elements, along with the batching and debatching elements, can be seen in Figure 5.5.

5.3.1.5 Avoid Packet Reordering

Previous work on parallel packet processing often causes reordering among packets due to techniques such as load balancing and parallel dispatch across multiple cores [51, 49].



(a) Batching packets. (b) Completion and debatching.

Figure 5.5. Batching and debatching elements. Serial interfaces are shown as simple arrows, wide interfaces as double arrows.

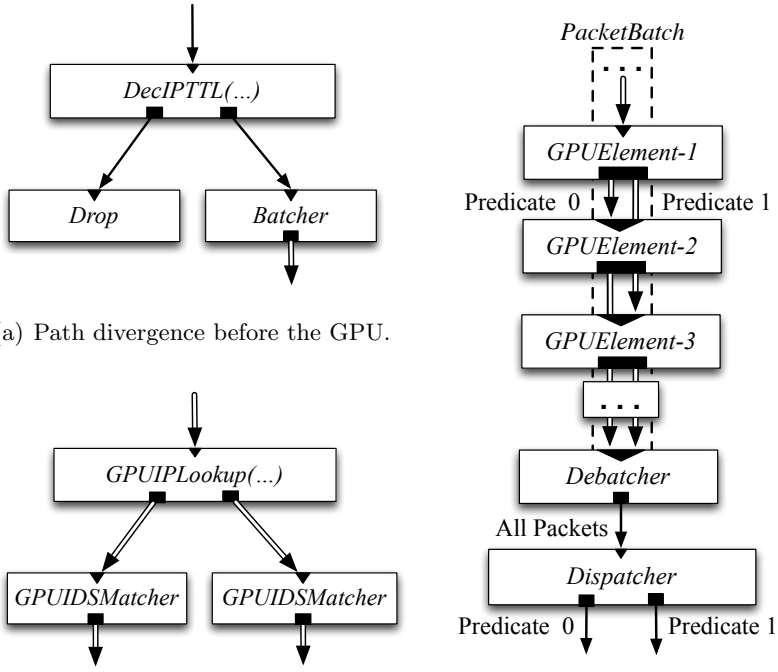
This can hurt TCP or streaming performance [52]. `Snap`, however, does not suffer from this problem: it waits for all threads in a GPU `BElement` to complete before passing the batch to the next element, so `Snap` does not reorder packets within a *PacketBatch*. Because asynchronous scheduling on the GPU (discussed in more detail in Section 5.3.3) may cause reordering of the *PacketBatch*es themselves, we add a `GPUCompletetionQueue` element between the `DeviceToHostMemcpy` and `Debatcher` elements. `GPUCompletetionQueue` keeps a first-in-first-out (FIFO) queue of outstanding GPU operations, and does not release a *PacketBatch* downstream to the `Debatcher` until all previous *PacketBatch*es have been released, keeping them in order. Because `GPUCompletetionQueue` is simply an `Element` in the configuration graph, a configuration that is not concerned about reordering could simply provide an alternate element that releases batches as soon as they are ready.

5.3.2 Packet Processing Divergence

`Snap` faces a problem not encountered by other GPU processing frameworks [22, 21, 24], namely the fact that packets in Click do not all follow the same path through the `Element` graph. `Element`s may have multiple output `Port`s, reflecting the fact that different packets get routed to different destination NICs, or that different sets of processing elements may be applied depending on decisions made by earlier elements. This means that packets that have been grouped together into a *PacketBatch* may be split up while on the GPU or after being returned to host memory. We encounter two main classes of packet divergence. In *routing or classification divergence*, the number of packets exiting on each port is relatively balanced; with *exception-path divergence* most packets remain on a "fast path" and relatively few are diverted for special processing. Packet divergence may also appear in two places: before the packets are sent to the GPU, or on the GPU, as a result of the decisions made by `BElement`s.

Figure 5.6(a) shows an example of exception-path divergence before reaching the GPU: packets may be dropped after the TTL decrement if their TTLs reach zero. Figure 5.6(b) shows an example of routing divergence on the GPU. In this example, different IDS elements (likely applying different sets of rules) are used to process a packet depending on its next hop, as determined by IP routing lookup.

Divergence before reaching the GPU is another reason that we do not attempt to implement zero-copy in the *PacketBatch* structure. The effect of divergence early in the pipeline is memory fragmentation, giving us regions of memory in which only some packets need to be copied to the GPU. This problem is particularly pronounced in the case of routing/classification divergence. Copying all packets, even unnecessary ones, to the GPU



(a) Path divergence before the GPU.

(b) Routing divergence on the GPU.

(c) Predicated execution.

Figure 5.6. Handling divergent packet paths.

would waste time and scarce PCIe bandwidth. Alternately, we could set up a number of small DMA transfers, covering only the necessary packets, but this results in high DMA setup overhead. Instead, we use the relatively plentiful memory bandwidth in host RAM to copy the necessary packets to one continuous region, which can be sent to the GPU with a single DMA transfer.

For divergence that occurs on the GPU, our experiments show that the overheads associated with splitting up batches and copying them into separate, smaller *PacketBatch*es are prohibitive, especially in the case of exception-path divergence. Assembling output *PacketBatch*es from selected input packets is also not concurrency-friendly: determining each packet's place in the output buffer requires knowledge of how many packets precede it, which in turn requires global serialization or synchronization. Having discussed in Section 3.1.2.1, `Snap` should embrace the warp control flow divergence on GPU. But instead of suffering from the slow performance caused by the divergence, `Snap` uses the predicated execution to minimize the performance impact.

To do the predicated execution, `Snap` attaches a set of *predicate bits* to each packet in a *PacketBatch*—these bits are used to indicate which downstream `BElement`(s) should process the packet. Predicates are stored as annotations in Click's `Packet` structure. The thread

processing each packet is responsible for checking and setting its own packet's predicate; this removes the need for coordination between threads in preparing the output. With predicate bits, a thread simply decides whether to process a given packet or not by checking the packet's predicate at the very beginning of the GPU kernel, hence it just introduces two branches of the control flow, and one of them is totally "empty," which follow the guidelines presented in Section 3.1.2.1. Because they are only used to mark divergence that occurs on the GPU, not before it, we save PCIe bus bandwidth by not copying predicate bits *to* GPU memory; we do, however, copy them *from* the GPU once a chain of `BElement`s is finished, since they are needed to determine the packets' next destination `Element`s on the CPU.

Figure 5.6(c) shows how this predicated execution works. The `GPUElement-1` element marks packets with either Predicate 0 or Predicate 1, depending on which downstream element should process them. The packets remain together in a single *PacketBatch* as they move through the element graph, but `GPUElement-2` only processes packets with Predicate 0 set and `GPUElement-3` only processes those with Predicate 1. Eventually, once they have left the GPU, the packets encounter a `Dispatcher` element, which sends them to different downstream destinations depending on their predicate bits. This arrangement can be extended to any number of predicate bits to build arbitrarily complicated paths on the GPU.

We have experimented with two strategies for using predicate bits: scanning all packets' predicates, and only launching GPU threads for the appropriate packets; and launching threads for all packets, and returning immediately from threads that find that their packet has the wrong predicate. We found it more efficient to launch threads for all packets: scanning packets for the correct predicates in order to count the number of threads adds to the startup overhead for the `BElement`. The savings in execution time that come from launching fewer threads are typically smaller than the overhead of scanning for the correct threads to launch, and it is faster to simply launch all threads. Because we run many threads per core, the threads that exit early do not necessarily waste a core. A further experiment to evaluate these two strategies on `Snap` applications is discussed in Section 5.5.4.

5.3.3 Asynchronous Processing

The host uses two main operations to control the GPU: initiating copies between host and device memory and launching kernels on the GPU. Both can be done asynchronously, with the CPU receiving an interrupt when the copy or code execution completes. Multiple GPU operations can be in flight at once. Based on this, most GPU elements can work asynchronously on the CPU side: when a GPU element is scheduled by `Snap`, it issues

appropriate commands to the GPU and schedules its downstream element by passing the *PacketBatch* immediately, without waiting for completion on the GPU. As a result, a push path or a pull path with GPU elements can keep pushing or pulling *PacketBatch*es without blocking on the GPU. This allows us to achieve very low latency at the beginning of the path, which is critical when packet rates are high—for example when receiving minimum-sized packets from a 10Gbps interface. The Click `FromDevice` element that receives packets from the NIC must disable interrupts while it pushes packets downstream to first `Queue` or `Batcher` element that they encounter; it is thus critical to have this path run as quickly as possible to avoid lost packets.

`Snap` uses CUDA stream to achieve overlapped asynchronous kernel execution and asynchronous memory copy. Each stream has a queue of operations which is run in FIFO order. Operations from different streams run concurrently and may complete in any order. We associate each *PacketBatch* with a unique stream. When the *PacketBatch* is first passed to a GPU element (typically, `HostToDeviceMemcpy`), it gets a stream assignment. Each subsequent `BElement` along the path asynchronously queues execution of its operation within the stream and passes control to the next `BElement` immediately, without waiting for the GPU. This sequence of events continues until control reaches a `GPUCompletetionQueue`, which is a *push-to-pull* element, much like a `Queue`. When a *PacketBatch* is pushed into the `GPUCompletetionQueue`, it simply adds the batch's stream to its FIFO queue and returns. When the `GPUCompletetionQueue`'s `bpull`() method is called, usually by a downstream `Debatcher`, it checks the status of the stream at the head of the FIFO by calling a nonblocking CUDA stream checking function. If the stream has finished, `bpull`() returns the *PacketBatch*; if not, it indicates to the caller that it has no packets ready.

5.3.4 Packet I/O

Click includes existing support for integration with Netmap [50] for fast, zero-copy packet I/O from userspace. We found, however, that Click's design for this integration did not perform well enough to handle the packet rates enabled by `Snap`.

Netmap uses the multiqueue support in recent NICs to enable efficient dispatch of packets to multiple threads or CPU cores. The queues maintained by Netmap in DRAM are mapped to hardware queues maintained by the NIC; the NICs we use for our prototype fix each receive and transmit queue at 512 packets. When a packet arrives on the NIC, a free slot is found in a queue, and the NIC places the packet in a buffer pointed to by the queue slot. When the packet is passed to Click, the buffer, and thus the queue slot, remains unavailable until Click is either finishes with the packet (by transmitting it on another port

or discarding it) or copies it out into another buffer. Since packets may take quite some time to be processed, they tie up these scarce queue slots, which can lead to drops. This problem is exacerbated in `Snap`, which needs to wait for suitably large batches of packets to arrive before sending them to the GPU. Click's solution is to copy packets out when it notices the Netmap queues getting full. It uses a single global memory pool for all threads, leading to concurrency problems. We found that at the high packet rates supported by `Snap`, these copies occurred for nearly every packet, adding up to high overhead incurred for memory allocation and copying. Note that unlike our *PacketBatch* structure, which copies only regions of interest, the copies discussed here must copy the entire packet.

Unmodified Netmap gives the userspace application a number of packet buffers equal to the number of slots in the hardware queues; while kernel code can request more buffers, userspace code cannot. We added a simple system call that enables applications to request more packet buffers from Netmap. Though the size of the queues themselves remain fixed, `Snap` can now manipulate the queue slots to point to these additional buffers, allowing it to maintain a large number of in-process packets without resorting to copying. `Snap` maintains a pool of available packet buffers—when it receives a packet from the NIC, it changes the queue to point to a free packet buffer, and packet buffers are added back to the free pool when the packets they hold are transmitted or dropped. This eliminates packet buffer copying and overhead from complex memory allocation (`kmalloc()`), and we use multithreaded packet buffer pools to avoid overhead from locking.

We also modified Click to pin packet I/O threads to specific cores. This is a well-known technique that improves cache behavior and interrupt routing when used with multiqueue NICs. Combined, these two optimizations give `Snap` the ability to handle up to 2.4 times as many packets per second as Click's I/O code—this improvement was critical for small packet sizes, where the unmodified packet I/O path was unable to pull enough packets from the NIC to keep the processing elements busy.

5.4 Implementation

`Snap` has been implemented on top of the `9200a74` commit in the Click source repository [53]. We used the Netmap release from August 13, 2012 and Linux kernel 3.2.16. `Snap` makes 2,179 lines of changes to Click itself, plus includes 4,636 lines of code for new elements and a 3,815 line library for interacting with the GPU. We modified only 180 lines of code in Netmap. The source for `Snap`, including our modifications to Netmap, can be downloaded from https://github.com/wbsun/snap.

The new packet processing elements we have implemented are for three kinds of packet processing tasks. Each has a CPU and a GPU version.

- `GPUIPLookup`**:** this GPU-based IP lookup element implements Click's `IPRouteTable` class using a radix tree. Its CPU counterpart is Click's `RadixIPLookup` element. For evaluation, we used a routing table dump from routeview.org [54] that has 167,000 entries.
- `SDNClassifier`**,** `GPUSDNClassifier`**:** these two elements classify packets using seven fields from the ethernet, IP, and TCP headers. Each entry assigns an action by forwarding the packet out of a specific outbound `Port` on the element. This is roughly analogous to the flow space matching used by many SDN forwarding schemes. `SDNClassifier` is the CPU version. The classification rule set is ClassBench [55]'s "ACL1_10K" filter set. We randomly assigned an action number to each rule.
- `IDSMatcher`**,** `GPUIDSMatcher`**:** these two elements implement the Aho-Corasick [56] string matching on packet payloads. The Aho-Corasick algorithm can match multiple patterns simultaneously by scanning the entire packet payload once. We used Snort's [57] rules for an Internet application server.

We combine these elements to build three kinds of `Snap` configurations, each of which has both a GPU and a CPU version:

- **SDN Forwarder:** This configuration includes only the `SDNClassifier` or its GPU counterpart. It simulates an SDN switch.
- **DPI Router:** This configuration includes an IP lookup element (`RadixIPLookup` or `GPUIPLookup`) and a string matching element (`IDSMatcher` or `GPUIDSMatcher`) as the major processing elements. The intent is to simulate a router with a simple deep packet inspection firewall.
- **IDS Router:** This configurations includes all three elements (IP lookup, IDS matcher, and SDN classifier) to simulate a more sophisticated router with complicated forwarding rules and intrusion detection.

5.5 Evaluation

All experiments were performed on the "gpunode" machine in the Emulab testbed [58]. Packets were generated at full line rate using a modified version of the packet generator that comes with the Netmap distribution, using a separate set of hosts in Emulab. Forwarding tables were designed such that all packets were forwarded back out the interface they arrived on. This ensured that all outgoing traffic was perfectly balanced so that any drops we

observed were due to effects within the Snap host, rather than congestion on unbalanced outbound links.

5.5.1 Packet I/O

Our first set of experiments are simple microbenchmarks that evaluate the packet I/O optimizations described in Section 5.3.4. We measured the forwarding rate for minimum-sized (64 bytes) packets using Click's Netmap packet I/O engine and Snap's improvements to that engine. These experiments use the simplest possible forwarder, which simply passes packets between interfaces with no additional processing. We test both a one-path arrangement, which passes packets from a single input NIC to a single output, and a four-path arrangement that uses all four NICs in our test machine. Click's existing Netmap support is not thread-safe, allowing only one packet I/O thread to be run. We added multithreading support to standard Click's Netmap code, and also report performance for four threads, one per NIC. Snap adds support for multiple threads per NIC, each using a different NIC queue, so we use sixteen threads for the Snap configuration.

The performance numbers are found in Table 5.2. Snap's improvements to the I/O engine introduce a 1.89x speedup for single path forwarding and 2.38x speedup for four-path forwarding. One interesting result is that Snap's four-path performance is not quite four times that of its single-path performance. This suggests that there may be room to improve the forwarding performance of Snap using more cores; our test CPU has four physical cores and hyperthreading, meaning that there are two I/O threads mapped to each hyperthreaded core. A recent (at the time of writing this dissertation) evaluation of Snap's forwarding performance confirms this guess. We use a recent six-core high-end CPU: Intel Core i7-3930K, and six 10Gb ports to do the basic forwarding, and get 49.62 Gbps forwarding rate, which is a little bit higher than the quad-core machine when considering the per-port rate. That makes sense because the six-core machine still assigns two I/O threads to a single hyperthreaded core. When we use only four 10GB ports on the six-core machine so that in theory each two I/O threads can get one-third more core, the forwarding rate approximately reaches 40 Gbps line rate.

5.5.2 Applications

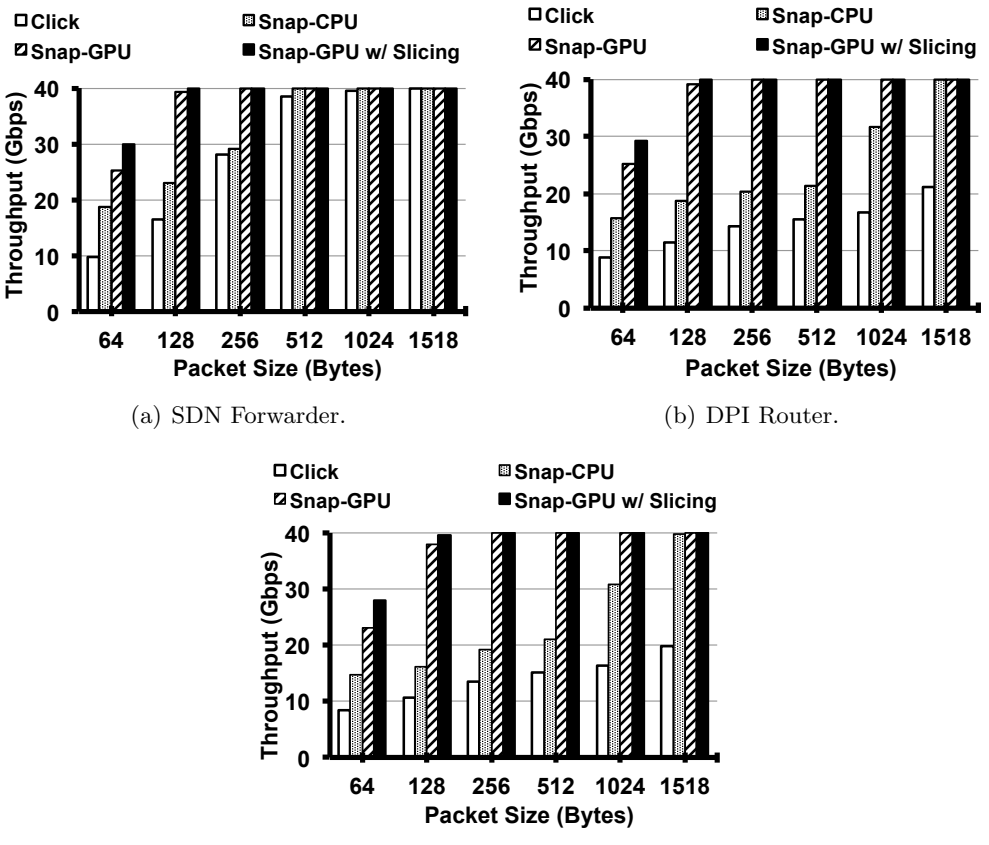
Using the implemented applications, we compared the performance of four configurations: standard Click; Snap with only CPU Elements; Snap with GPU elements, but with packet slicing disabled; and Snap with all optimizations enabled. We experimented with a variety of packet sizes. Each experiment lasted at least one minute, and the numbers

Table 5.2. Base forwarding performance of `Snap` and Click.

Configuration	Throughput	
Click 1 Path	4.55 Gbps	6.5 Mpps
Click 4 Paths (1 thread)	8.28 Gbps	11.8 Mpps
Click 4 Paths (4 threads)	13.02 Gbps	18.5 Mpps
`Snap` 1 Path	8.59 Gbps	12.2 Mpps
`Snap` 4 Paths	30.97 Gbps	44.0 Mpps

reported are the average of three runs. The results are shown in Figure 5.7.

The results show that `Snap` gets significant performance improvements over Click, particularly for small packet sizes. A significant fraction of this speedup comes from our I/O optimizations, which can be seen by comparing the bars for "Click" and "Snap-CPU:" 63% of the 3.1x speedup seen by the SDN forwarder on 64-byte packets comes from this source. Another jump comes from moving the processing-heavy elements to the GPU, with another modest increase with the addition of packet slicing. `Snap` is able to drive all four NICs at



(a) SDN Forwarder.

(b) DPI Router.

(c) IDS Router.

Figure 5.7. Application performance.

full line rate for all but the smallest packets: at and above 128 bytes, it gets full line rate for all configurations (with the exception of the IDS Router, which gets 39.6 GBps at 128 bytes). `Snap` is limited by the availability of PCIe slots in our test machine, which has 32 PCIe lanes: 16 are used by the GPU, and each dual-port NIC uses 8 lanes, meaning that we cannot add any more NICs. The results strongly suggest that `Snap` would be capable of higher bandwidth on a machine with more or faster PCIe lanes, but such hardware was not available for our tests. We thus leave exploration of `Snap`'s full limits to future work and the availability of suitable hardware, but we are optimistic that it will be capable of exceeding 40 Gbps for large packets. Conversely, this result also means that there is headroom available to do more processing per packet than is performed by our example applications.

For minimum sized packets, `Snap` reaches 29.9 Gbps (75% of the full line rate) for the SDN forwarder; the primary cause of this limitation appears to be due to packet I/O, as the throughput seen in this experiment is very close to the trivial forwarder from Section 5.5.1. The other two GPU applications reach almost the same performance: 29.2 Gbps for the DPI router and 28.0 Gbps for the IDS Router. This matches our intuition, because with all of the complex processing done on the GPU, the CPU only needs to perform simple operations and can spend most of its time on packet I/O. When we use the CPU elements, both packet I/O and the processing algorithms need the CPU, and all three applications slow down significantly: the IDS Router gets 14.73 Gbps, a slowdown of 52% from the trivial forwarder. With the DPI and IDS Router configurations, standard Click is unable to reach much more than 20 Gbps, even for large packet sizes.

The ROI-based slicing mechanism makes a modest improvement in forwarding throughput. For example, the SDN forwarder sees a 13.7% increase in throughput for 64-byte packets. Slicing enables `Snap` to reach nearly the full rate supported by the packet I/O engine for small packets. At larger packet sizes, the improvement disappears because we have reached full line rate on the NICs.

5.5.3 Latency and Reordering

The most obvious drawback of batched processing is an increase in latency, since packets arriving at the beginning of a batch must wait for the batch to fill. To find out how much latency the batching mechanism adds to `Snap`, we measured round-trip time for 64-byte packets using both a CPU and GPU configuration. For the CPU-only configuration, we saw a mean latency of 57.5μs (min: 31.4μs, max: 320μs, σ: 25.7μs). For the GPU-based configuration with batched processing (*batch-size*: 1024), the mean latency was 508μs (min:

292μs, max: 606μs, σ: 53.0μs); this represents an increase of less than one quarter of a millisecond in each direction. Reducing the *batch-size* from 1024 to 512, the latency reduces to 380.4μs on average, but throughput also drops from 28 Gbps to 24 Gbps. The additional latency added by batching for GPU elements is likely to be noticeable, but tolerable, for many local area network (LAN) applications. On wide area network (WAN) links, this delay will be negligible compared to propagation delay. As part of this experiment, we also checked for packet reordering. We define the multiqueue dispatching rules of our router NICs to send packets in the same traffic flow into the same NIC transmit queue, and in the `Snap` configuration, we connected each `FromDevice` element to a `ToDevice` with the same transmit and receive queue IDs. With these settings, we found no reordering in the packet stream.

5.5.4 Packet Processing Divergence

To evaluate whether our design for handling divergent paths is effective, we built an IDS configuration that connects an `GPUSDNClassifier` element with two `GPUIDSMatcher` elements. The classifier marks each packet with a predicate indicating which of the two IDS elements is to process it; this simulates a scenario in which packets are to be handled by different IDSes depending on some property such as source, destination, or port number. In both this configuration and the IDS router configuration from our earlier experiments, each packet is processed by one IDS element; the difference is that in the diverging configuration, there are two IDS elements, each of which processes half of the packets. Thus, we can expect that, if the overhead of our divergence handling strategy is low, the configuration with two `GPUIDSMatcher`s should achieve similar throughput to the configuration with a single one. We evaluated this diverging configuration with different packet sizes and measured the throughput, which is shown in Figure 5.8. The performance under divergence is very similar to the IDS router result shown in Figure 5.7(c). It is only slightly slower at small packet sizes: the diverging configuration achieves 26.8 Gbps versus the IDS router's 28.0 Gbps for 64-byte packets, 39.4 Gbps versus 39.6 Gbps for 128-byte packets, and 39.9 Gbps versus 40.0 Gbps for 256-byte packets. At and above 512-byte packets, both achieve a full 40.0 Gbps. We conclude that the launch of extra GPU threads that have no work to do causes a slight slowdown, but the effects are minimal.

5.5.5 Flexibility and Modularity

Finally, we demonstrate that `Snap` can be used to build not only highly specialized forwarders, but also a complete standards compliant IP router. This task is simple, because

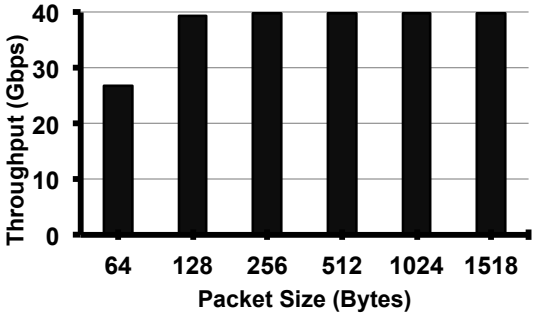


Figure 5.8. Forwarding performance when using a `GPUSDNClassifier` that diverges to two `GPUIDSMatcher` elements.

such configurations already exist for Click. Specifically, we base our IP router off of the configuration shown in Figure 8 of the Click paper [19], which includes support for error checking of headers, fragmentation, ICMP redirects, TTL expiration, and ARP. We replace the `LookupIPRoute` element with our `GPUIPLookup` element (and the accompanying `Batcher`, etc.), and add an IDS element to both the CPU and GPU configurations.

Due to the complexity of this router, we do not attempt to illustrate the entire `Snap` configuration here. Instead, we illustrate the major changes that we made to the standard Click router configuration in Figure 5.9. The left part of the figure shows our GPU processing path, and the right part is the original CPU route lookup path plus an `IDSMatcher` and its auxiliary alert element. This figure also shows a strategy for handling divergence on the GPU: the `GPUIDSMatcher` sets predicates on packets depending on whether they should raise an alert, then pushes entire the *PacketBatch* downstream. The `GPUIPLookup` is assigned a *CHK_ANNO* argument, which is the predicate controlling processing of each packet. `GPUIPLookup` thus ignores packets flagged for alerts by the IDS, and divergence on the actual element graph is delayed until after the `Debatcher`, using a `Dispatcher` element.

The performance of the CPU-based and GPU-based full router configurations are shown in Figure 5.10. This fully-functional router with built-in IDS is able to achieve 2/3 of the performance of a trivial forwarder for minimum-sized packets, and almost full line rate (38.8 Gbps) for 512-byte and larger packets. This demonstrates the feasibility of composing complex graphs of CPU and GPU code, and shows that existing CPU Click elements can be easily used in `Snap` configurations without modification. The bottleneck in performance appears to be the large number of CPU elements in this configuration—there are fifteen types of elements, some of which are duplicated sixteen times, once for each thread. As future work, we believe that the throughput can be significantly improved by moving some of

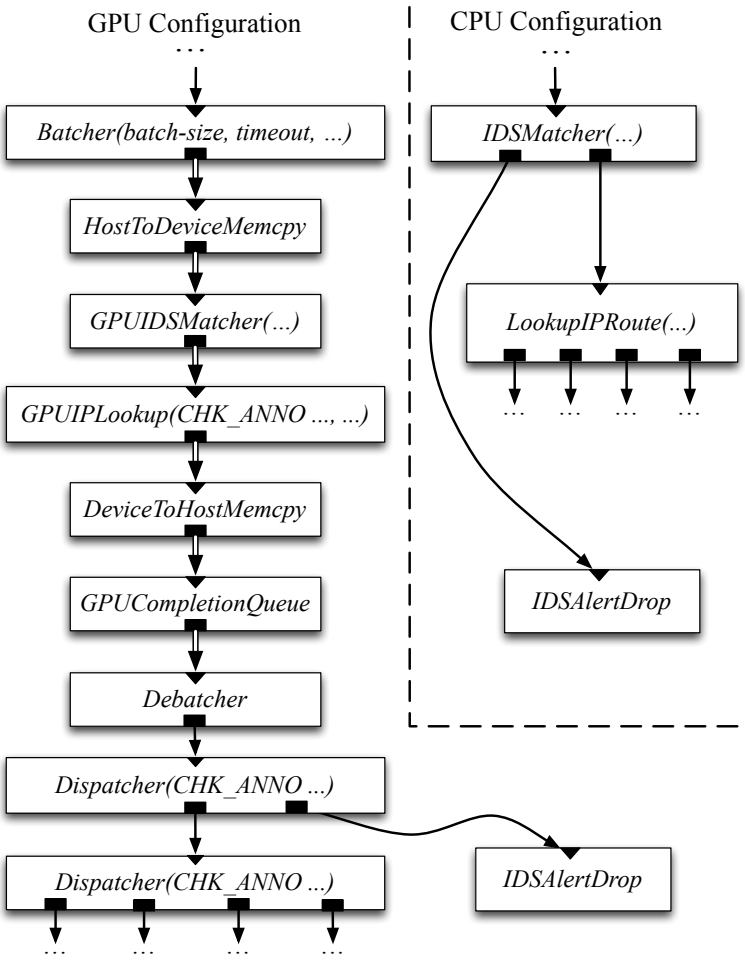


Figure 5.9. Major changes of the standard Click router to build a GPU-based IDS router.

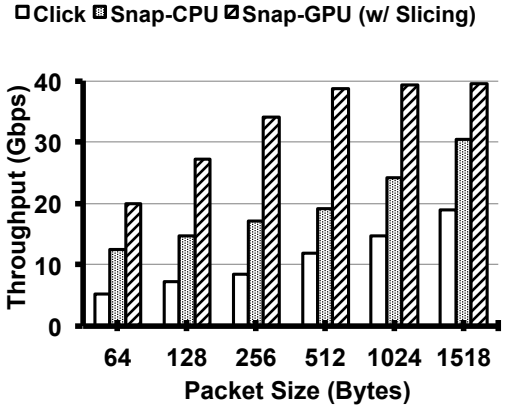


Figure 5.10. Fully functional IP router + IDS performance

these to the GPU and applying the techniques from the work [59] to optimize the remaining CPU portions of the configuration.

5.6 Summary and Future Work

`Snap` expands Click's composable element structure, adding support for batch processing and offloading of computation. At small packet sizes (128 bytes), `Snap` increases the performance of a combined IP router, SDN forwarder, and IDS on commodity hardware from 10.6 Gbps to 39.6 Gbps. This performance increase comes primarily from two sources: an improved packet I/O engine for Click that takes advantage of multiqueue NICs, and moving computationally expensive processing tasks to the GPU. A trivial forwarder created with `Snap` can forward at a rate of 44.0 Mpps, while the complex SDN/IDS router reaches 90% of this rate (39.8 Mpps). These results suggest that there is likely potential for elements that are even more computationally complex than the ones we investigated, pointing to future work in complex packet processing. The fact that we are able to saturate all NICs in our test platform with such small packets suggests that it will be possible to reach even higher throughputs when PCIe 3.0 devices are available for testing, allowing us to double the number of NICs on a bus.

The elements and techniques proposed and implemented in `Snap` are not GPU-specific only. While some of the new `Element`s implemented for `Snap`, such as `HostToDeviceMemcpy` and `GPUCompletetionQueue`, are GPU-specific, the extensions we made to the Click architecture should be applicable to other parallel offload engines (such as network processors and programmable NICs) as well.

Besides possible throughput improvement with techniques from the similar work [59], recent CUDA releases and GPU hardware advancement have provided useful techniques for `Snap` to do fast packet processing. The "dynamic parallelism" technique [60] on the latest CUDA GPUs allows a kernel thread on GPU to invoke a new kernel launching, exactly the same as the host side code can do. This indicates an alternative approach to implement predicated execution: packet pointers can be grouped on GPU by their predicates, and then only necessary threads are launched for a following GPU element's kernel to process the grouped packets only. One problem with this approach is that it requires on-GPU synchronization, which is an expensive operation, and hence is impractical on GPUs used for evaluating `Snap`. However, dynamic parallelism also makes it possible to only launch a one-thread scheduling kernel at the beginning of the first GPU element in a batched processing path. Such a scheduling kernel is responsible to launch the kernels for the

following GPU elements according to the configuration and predicates as explained before. In that case, on-GPU synchronization won't be a problem any more because the scheduling and kernel launching only happen in the single-threaded scheduling kernel.

CHAPTER 6

A SURVEY OF SYSTEM-LEVEL GPU COMPUTING

The motivations of this survey are two-fold.

1. To survey the existing work of different system-level GPU computing to demonstrate the wide applications of the GPU in system areas.
2. To analyze techniques used by the related work for efficient GPU integration in system software, comparing them with our techniques proposed in this dissertation for pros and cons, as a reference to guide later system-level GPU research.

This survey tries to cover major system-level work that targets the GPU. Most of the papers surveyed are related to various system-level applications with GPU acceleration. However, some work covered in this survey is on system-level support for GPU computing, such as virtualization, resource management, and migration.

This survey has two major sections. The first section is focused on applications of GPUs in system-level software. It discusses the computationally expensive operations or algorithms in different system areas which GPUs are expected to accelerate, with representative work in each area. Similar to our `KGPU` white paper [61], not only existing work but also potential applications are discussed. For each application area, the attacking targets, which are computationally expensive operations or algorithms, are listed first. Then the related work is described. This section serves as a traditional related work survey for this dissertation. The second section is focused on techniques proposed and used by applications in the first section to improve the performance of or ease the GPU computing integration into existing system software. Since Chapter 3 has described our own techniques for system-level GPU computing, this section will also compare them with the surveyed techniques. It is not surprising to find a large portion of common techniques during the comparison, which can further confirm the feasibility and effectiveness of techniques proposed in this dissertation. Besides those two major sections, there is also a small section at the end of this chapter,

which discusses the potential future GPU development and useful techniques to solve current problems.

6.1 Applications

The system-level work categories listed in the following sections are not orthogonal to each other. Many applications actually belong to two or more categories. For example SSLShader [23] is both networking and security work. Some categories such as networking and storage have much more existing work surveyed due to comprehensive studies done by the research community. Some have just attracted a few efforts, such as program analysis and data compression. However, that doesn't mean they are not important at all. On the contrary, less existing work often means that area has not been thoroughly studied and hence may contain more potential opportunities for future research.

6.1.1 Networking

Currently, the main computing tasks in network systems are the operations and algorithms used to process network packets, providing packet forwarding, security, and data integrity functionalities. The parallelism of packet processing tasks often requires independent processing on a per-packet and per-flow basis. So stateful processing that needs either high-level protocol data or particular packet orders is not a good candidate for parallel acceleration. As we will see in the following text, most computing tasks are packet classification, IP routing, pattern matching, hashing, error correction coding, encryption, name lookup, etc. They are all very packet-independent processing logic.

As for concrete applications, Gnort [24, 62] is a very early attempt at a GPU-accelerated NIDS. It's built on top of Snort [57], which is a widely used open source NIDS. Gnort attacks the pattern matching problem in intrusion detection. It matches multiple patterns in parallel using a DFA built by the Aho-Corasick [56] algorithm, getting 3.2x speedup over the CPU algorithm implementation, and 2x speedup in the macrobenchmark.

Kargus [21] is similar to Gnort. It is also a GPU-accelerated NIDS based on Snort. Different from Gnort, Kargus includes more pattern matching algorithms for different regular expression dialects. Kargus also balances the workload between the CPU and the GPU to trade off between the throughput and latency. The load-balanced processing model in Kargus helps it reach a factor of 1.9 to 4.3 performance improvement over the conventional Snort.

PacketShader [22] is a software router with GPU-accelerated IP routing. PacketShader considers the entire network stack from packet I/O on NICs to the routing table lookup,

to improve the IP routing throughput. It brings a factor of four performance improvement over the existing software router when doing minimum size IP forwarding. As the pioneer work of GPU-accelerated network systems, PacketShader demonstrates a great opportunity in this area.

SSLShader [23] is an SSL proxy that utilizes both CPU and GPU power. SSLShader uses GPUs to attack the cryptography operations in the SSL protocol, such as hashing, asymmetric public key encryption, AES private key encryption and other simple encryption algorithms. By combining the CPU and the GPU together to process SSL flows, SSLShader demonstrates a comparable performance to high-end commercial SSL appliances at a much lower price.

Our `Snap` framework is also focused on network packet processing. The main difference between `Snap` and the above work is `Snap`'s generality as a packet processing platform that is capable of building a variety of system-level network applications, including all of the above ones. Different from `Snap`, which decomposes the abstract GPU-accelerated packet processing task into modular elements for flexible configurable control in different processing tasks, each surveyed system is built as a single box for a particular task only, not suitable for other kinds of packet processing.

There are several algorithm-only efforts. An algorithmic work [63] has achieved scalable IP lookup under frequent routing table updates. Another project [64] did name lookup for the content-centric network (CCN) [65] with GPU-accelerated longest prefix matching on strings. The low-density parity-check (LDPC) coding algorithm that is used in wireless communication for error correction is also attacked by a project [66] with parallel GPUs. Last, the classic packet classification problem has also benefited from parallel GPU acceleration as done in [67].

One potential GPU computing application in network systems is the WAN optimization [43, 48]. Candidate computing tasks include computing data signatures for packet deduplication, compressing payload to reduce transfer overhead, computing error correction code and so on. The CCN is definitely another good target for GPU acceleration. The aforementioned name lookup work [64] only did algorithm parallelization with GPUs, the underlying system infrastructure still needs careful optimization to build a high performance CCN. Our `Snap` work provides such complementary infrastructure-level optimization, making it possible to build such a CCN on top of `Snap` with the optimized name lookup algorithm.

6.1.2 Storage

Storage systems also contain many computational tasks. There are content addressable storage systems that require hashing for signatures, secure storage systems that require encryption algorithms, reliable storage systems that require erasure code for error detection and recovery, archival or space-efficient storage systems that compress the data, and high performance storage systems that use complex index structures for fast lookup. Similar to network processing, storage tasks can often be parallelized per data block, per disk sector, per I/O request or per memory page.

Content addressable storage attracts a lot attention due to its expensive hashing operations. Shredder [68] and CrystalGPU [69] are representative work on incremental storage and deduplication systems with GPU-accelerated content addressing functionality. Shredder works as a content-based chunking framework, which has shown 5x chunking bandwidth speedup over the CPU counterpart, and also large performance improvements in real world applications. CrystalGPU is actually a GPU framework for building distributed content addressable storage systems with two other GPU libraries, HashGPU [69] and MesaStore [69], which has demonstrated up to 2x speedup.

Erasure coding, or error correction coding, is another hot topic: Gibraltar [70] and Barracuda [39] accelerated RAID [71] parity computation with GPUs. Gibraltar is built on top of the Linux kernel's SCSI target in userspace. It implements a flexible Reed-Solomon coding [72] on GPUs, so it supports broader RAID configurations compared to its counterpart, the `md`'s only $n + 2$ RAID 6 scheme in the Linux kernel. Barracuda is similar to `GPUstore`'s `md` work; they both accelerate the coding tasks in `md`. Similar to Gibraltar, Barracuda implements more schemes than `md`'s default two-failure tolerable one. Different from `GPUstore`'s `md` acceleration, it didn't see speedup at the two-failure case, but got a 72x speedup at the eight-failure case.

Besides content addressable storage and erasure coding, secure storage with data encryption can also benefit from GPUs with GPU-accelerated cryptography: `GPUstore` includes two secure storage systems, the encrypted filesystem and the disk encryption driver.

A potential GPU application in storage is for file system integrity, such as the Featherstitch [73], which exposes the dependencies among writes in a reliable file system. One of the most expensive parts of Featherstitch is analysis of dependencies in its *patch graph*, a task we believe could be done efficiently on the GPU with parallel graph algorithms.

There exists other work that tries to ease the storage access in GPU programming. GPUfs [74] is a file system API available to CUDA GPU kernels. With GPUfs, GPU kernel

programmers can access file content directly from within the kernel, no need to go back to the host side any more, which leads to complex interleaved host code and device code.

6.1.3 Database Processing

For databases, the major computing tasks are in data query, which may require large scale linear scanning, sorting and hashing. Generic large scale data processing, which often involves the MapReduce [75] model, requires not only sorting during key shuffling, but also a GPU computing friendly platform for potential GPU-accelerated mappers or reducers. Note that we won't survey the specific GPU-accelerated MapReduce applications for particular tasks such as stock analysis, because they are not system-oriented. Instead, we focus on the system support by means of a framework or runtime to build GPU-accelerated MapReduce applications.

The main algorithmic operations in relational database queries—scan, join and sorting—have been studied to take advantage of parallel GPUs by a lot of existing work [76, 77, 78, 79, 80, 81]. However, only a few have built practical database systems with GPU accelerations. The most practical work is the GPU query engine [82], which accelerates the linear scan of massive records in the industrial-level PostgreSQL [83] database management system with GPUs. The in-GPU-memory column-oriented database [84] is designed for processing analytical workloads. The main idea, or motivation of its in-memory design is to address the low bandwidth memory copy between host and device. By keeping data in GPU device memory, memory copy through PCIe bus is unnecessary. Key-value store databases also benefit from the GPU-accelerated lookup algorithms, such as the GPU-accelerated Memcached [85] which has got up to 7.5x performance increase with integrated CPU+GPU key-value lookup.

For large scale data processing, GPUTeraSort [86] is a pipelined disk-based external sorting system to process large scale data, and shows a good price-performance in practice. As for the GPU-accelerated MapReduce frameworks, Mars [87] is the pioneer work that eases the GPU-accelerated MapReduce programming. Mars works on a single machine, so its performance is limited compared with the other CPU-based distributed MapReduce systems. Mars also uses a simple workload mapping mechanism: it maps a single data item to one GPU thread. Such single mapping prevents programmers from exploiting the interthread collaboration to improve performance. Recent GPU-accelerated MapReduce frameworks[88, 89] relax such mapping constraint, allowing flexible one-to-many or many-to-one mappings. Moreover, they have been extended to multiple GPU clusters to embrace more computing power.

6.1.4 Security

Cryptography algorithms are a main class of computing tasks for security. Another computing task is pattern matching. We have gone through the NIDS that heavily relies on pattern matching for network security. In this section, we mainly focus on general security systems.

The cryptography algorithms fit quite well on GPUs because of their simple control flow logic: symmetric cryptography [90], asymmetric cryptography [91, 92] and hashing [23]. All have been exploited to use parallel GPUs to accelerate. The pattern matching based security systems are not only network intrusion detection systems as we have discussed in previous sections, but also antivirus software that requires heavy signature matching. There is a parallel antivirus engine [93] with GPU-accelerated signature matching via the Boyer-Moore algorithm [94] and the Aho-Corasick algorithm [56]. It is built on top of ClamAV[1], a popular open source antivirus software, to achieve 20 Gb/s, 100 times faster than the CPU-only performance. Kaspersky, which is a very famous antivirus software, also tried using GPUs to match virus signatures and got two orders of magnitude speedup [95]. This is the most practical GPU-accelerated antivirus solution right now, though as far as we know, Kaspersky hasn't released any product with such GPU acceleration available to end users.

A very useful potential GPU application in security is for the trusted computing in virtualized environment based on trusted platform module (TPM). A TPM is traditionally hardware, but recent software implementations of the TPM specification, such as vTPM [96], are developed for hypervisors to provide trusted computing in virtualized environments where virtual machines cannot access the host TPM directly. Because TPM operations are cryptography-heavy, they can also potentially be accelerated with GPUs.

6.1.5 Program Analysis

A very interesting work is the EigenCFA [25] static program analyzer, which maps flow analysis onto matrix multiplication accelerated by GPUs to get up to 72x speedup. Though EigenCFA just does the basic 0CFA [97] and can't fundamentally solve the exponential complexity control flow analysis problem, it makes the flow analysis further practical with much faster GPUs. One of our side projects, the GodelHash [98], also reveals another potential computation target for GPUs to attack: the time-consuming flow entry subsumption operations in kCFA [97]. GodelHash does perfect hashing that maps a flow entry to a

[1] http://www.clamav.net

prime number, then uses the product of the prime number to represent a flow set. Hence the subset and membership, which are actually the subsumption operation in control flow analysis, are done by the divisibility. It requires multiple precision arithmetic to deal with the huge integers produced by multiplying a set of prime numbers, which has demonstrated great speedup with GPU acceleration [23]. Hence GodelHash-based flow analysis is a good candidate for parallel acceleration.

6.1.6 Data Compression

Some data compression algorithms can also be parallelized. For general compression scheme, the LZSS [99] has been exploited in a recent work [100] to enable pipelined parallel compression. They accelerate the two main stages in LZSS, substring matching and encoding, to get a 34x speedup over a serial CPU implementation, and a 2.21x speedup compared with a parallel CPU implementation. The aforementioned GPU-accelerated database compression [101] uses a specialized compression scheme for column-oriented databases and gets very high performance.

Another opportunity comes from the recent migratory compression [102] work. Migratory compression does three steps to achieve better compression than traditional methods: it firstly deduplicates the data, then relocates the similar data chunks to put them together, and last does the traditional compression on the relocated data. The chunk relocation improves the data similarity, and helps migratory compression achieve 44-157% compression ratio improvement on archival data. The relocation step needs to find similar data chunks, which relies on the computation intensive "super-feature" generating and matching. The feature generating is totally chunk-independent, hence a good parallel target for GPUs. The matching needs hashing for hash table lookup and large scale sorting for similarity comparison, which are parallelizable.

6.2 Techniques

In this section, we will discuss the techniques used by a variety of existing system-level GPU work. As said at the beginning of the chapter, we focus on the system-level techniques rather than algorithm parallelization. The general goal of the techniques is to improve the system performance. However, there are some technologies that have been exploited to make GPU computing available in more environments, such as Gdev for in-kernel GPU computing, and the virtualized GPU solutions discussed at the end of this section that make GPGPU available to virtualization environments and cloud computing. The techniques are categorized by the challenges or problems they solved. In the categorized techniques, we

will see a large portion of common categories shared with the ones discussed in Chapter 3. For those categories, readers should refer to Chapter 3 for the description or origins of the challenges and problems.

6.2.1 Batched Processing

All the surveyed papers need some kind of batching to ensure enough parallelism in the workload to be processed on GPUs. However, there are two different kinds of batching approaches: static batch size and dynamic batch size.

The static batch size approach is easier, as mentioned in most surveyed papers [22, 70, 39, 69, 86] and in our own `GPUstore` and `Snap` work, the number of data units in a single batch is constant. Such constant value is decided by performing algorithmic benchmarking, finding the optimal size of the batch, and then using it forever. The optimal batch size can be the size where GPU performance goes down, or the first size that the GPU outperforms CPU. The drawback of such an approach is obvious: it can be easily affected by the system load and any other runtime overheads. Because of its simplicity, this has been applied by most system-level work.

The dynamic batch size is complex: it still needs some prebenchmarking, but instead of finding a single size, it builds a relationship between the batch size and the performance. During the run time, it dynamically adjusts the batch size, or disables GPU offloading according to the built relationship, system load and other factors. SSLShader [23] and Kargus [21] use this approach to implement flexible GPU offloading control to balance the latency and throughput. They also implement dynamic load balancing between the CPU and the GPU with the help of the built relationship: small packets, low packet rates and high GPU load all lead to CPU processing.

The dynamic approach is for sure the better way to finely tune the overall system performance. But it requires a comprehensive study and evaluation of the GPU-accelerated algorithms under a variety of different environments, and also possibly complex runtime workload balancer. So the static approach is still useful to demonstrate GPU accelerations without considering the combined CPU+GPU power and external environment conditions.

6.2.2 Memory Copy Overhead

Memory copy is the main bottleneck for most system-level GPU computing. Taking the GTX Titan as an example, compared with the 336 GB/s GPU device memory bandwidth, even the maximum PCIe 3.0 x16 can only run at 16 GB/s, which is 31 times slower! Besides that, Section 3.5.2 also discusses the double-buffering problem caused by inefficient

GPU computing integration into system software in practice. Several techniques have been applied by previous work to reduce the memory copy overhead. Here they are categorized into the following approaches.

6.2.2.1 Use Page-Locked Memory

Although there are some systems, such as the GPUTeraSort [86], Barracuda [39], the database compression mechanism on MonetDB [101], EigenCFA [25] and Mars [87], that haven't used page-locked memory to improve PCIe DMA performance, most of the aforementioned work applies this simple but effective memory copy optimization. However, not all of them have efficiently used page-locked memory in practice. Many systems just use page-locked memory as a separate memory area dedicated for GPU computing, rather than treating it as normal host memory that is used for any host side computation, which leads to the inefficient double-buffering problem.

6.2.2.2 Avoid Double-Buffering

Many systems surveyed perform a "use all" data usage pattern as discussed in Section 3.5.4. However, even when using page-locked memory to improve memory copy, most of them still failed to efficiently integrate the memory into the host side computation as normal host memory. Some are totally unaware of this problem such as the CrypstalGPU content addressable storage [69] and deduplication storage [103]. Some of them are due to the technical obstacle when getting data from the operating system kernel [70, 23, 68, 39, 85]. Such kernel-user memory sharing obstacles can be solved by allocating the GPU computing host side memory directly in the OS kernel as the kernel-space GPU driver Gdev [38] does, or supporting such in-kernel allocation with pre-allocated page-locked memory passed into the kernel from userspace, like `GPUstore` does. Only a few of the surveyed papers have carefully designed their systems to deal with the double-buffering problem. For example, the GPU antivirus engine [93] has modified the memory buffers used in the mature antivirus software, ClamAV, to efficiently use CUDA page-locked memory for virus pattern detection. Another example of such engineering work is the GPU query engine for PostgreSQL [82], which has efficiently integrated both page-locked memory and asynchronous overlapped GPU computation and memory copy into a large complex existing industrial-level database system.

6.2.2.3 Trade Off Different Bandwidths

Having discussed in Section 3.5.3, packet processing is the typical task that follows a "use part" of the entire data unit usage pattern. But not all the network systems in Section 6.1.1 have considered the potential performance improvement by taking advantage of the much larger host memory bandwidth. SSLShader [23] is such an example that is totally unaware of such potential speedup. Although the rest, such as those NIDS systems [24, 104, 21] that only copy packet payloads into device memory and the PacketShader [22] that copies only an IP address and a packet identifier to GPU, still do not use the more generic ROI-based slicing technique as `Snap` has proposed in Section 5.3.1.3. Actually, such copy in most of those papers is not described as a memory copy optimization, but instead, an overhead that must be suffered due to the technical difficulty of sharing DMA memory between the NIC driver and the GPU driver.

Interestingly, some papers other than network systems are aware of this issue, and have carefully traded off not only the host memory bandwidth and PCIe's, but also considering the CPU performance in some cases, such as in EigenCFA [25], which does not use page-locked memory, though authors try to significantly reduce the size of the flow data representation to be copied into device memory with preconverted bit-packed matrix at the CPU side. Another example is the GPUTeraSort [86] system, which is similar to EigenCFA, using pageable memory only, but still, authors do CPU side preprocessing on the data entries read from the disks to produce a much smaller `(key, pointer)` tuple for each entry. Although the GPUTeraSort case is very straightforward, it still represents the application of an effective memory copy optimization technique, and actually has inspired our generic ROI-based slicing design in `Snap`.

6.2.2.4 Exploit GPU Drivers

There are some papers that try to improve the underlying blackbox GPU driver in order to get better PCIe DMA performance. Although we can optimistically expect that future GPUs will be completely open, it is still unclear how long the "future" can be. Hence those papers studying the close source GPU drivers make sense to the relatively long "present." Some recent papers [105, 106] have exploited the microcontrollers on GPUs that manages the CPU-GPU communication and GPU command queues. By modifying the firmware for the GPU microcontroller using an open source GPU driver [37], they are able to achieve up to 10x speedup when copying small size data (ranging from 16 bytes to 256 bytes) controlled by the microcontroller rather than the dedicated DMA engines. Gdev [38] implements a CUDA runtime inside the Linux kernel based on the same open source GPU

driver, and has implemented DMA memory pages sharing for arbitrary kernel components, which is supposed to be provided by GPUDirect [107], but is rarely available in practice. Gdev is complementary to all the in-kernel GPU computing work such as `GPUstore` and Barracuda [39] because of its efficient in-kernel GPU driver access without the need of a userspace helper.

6.2.3 Warp Divergence

Besides the algorithmic design techniques that are used to eliminate as many control flow branches as possible, most system work doesn't encounter the on-GPU data divergence problem that `Snap` has solved in Section 5.3.2. Even in packet processing work, systems such as those NIDSes [24, 21, 104] do preprocessing on CPUs to classify packets into different "port groups" according to their transport layer protocol ports. This preprocessing is acceptable in those systems because they just have a one-stage (the pattern matching only) processing control flow, rather than a multistage one with multiple layers of branching as in `Snap`.

Some nonsystem work such as the GPU-accelerated packet classification algorithms [67] and CPU work such as the batched processing in Click [59] have identified and dealt with this problem. The classification paper uses a similar approach to `Snap`'s predicated execution: it uses a boolean value to notify the following classification rule kernel whether to process a particular packet or not. The batched processing Click paper encountered this problem when doing the CPU side batching. They simply split the batch into multiple small batches, which has not got any evaluation result due to being in the early stage of their work.

Another similar work, PTask [40], which tries to build a data flow programming model for GPU tasks by providing a UNIX pipe-like abstraction, simply does not support such flow divergence, or conditional data flow, in the flow graph, though it supports the opposite one: merging of multiple outputs into a single input. Besides that, PTask is a more application-level abstraction that is designed to deal with interprocess data flow. It is a heavy-weighted abstraction compared with data sharing mechanisms in system code, such as memory page sharing in an OS kernel.

6.2.4 Improve GPU Utilization

Most of the recent papers use CUDA's overlapped GPU computation and memory copy, and asynchronous GPU kernel execution features to improve their performance by fully utilizing the GPU cores and DMA engines. Those that failed to benefit from them are mostly very old papers before NVIDIA released its CUDA 4.0, which first introduced the

technique. However, there are exceptions such as PacketShader, which claims, though without any further explanation, a performance slow down when using such techniques [22]. Despite the fact that many systems use such overlapped scheme and asynchronous execution, none of them have studied the synchronization problem in GPU programming even with CUDA stream like technique, as discussed in Section 3.4.

6.2.5 Virtualization and Migration

The virtualization and application migration techniques are mainly for enabling GPU computing in cloud environments, Cloud providers such as Amazon[2] have provided GPU devices in their cloud instances [1]. However, those GPUs are completely exposed to users rather than through virtualization. This not only brings security issues, but also prevents resource consolidation and instance migration, which are the most important features of a cloud platform.

Several GPU virtualization solutions have been proposed by both industry and academia. The VMWare's GPU virtualization [108] aims at the graphics processing functionality. It is based on VMWare's hosted I/O architecture to enable GPU command queue isolation, GPU memory isolation and I/O space isolation so as to achieve independent virtualized GPUs. There are some recent projects [109, 110] that are very simple virtualization layers just for CUDA computing. They provide an intercept layer for CUDA library calls to forward them to the actual CUDA runtime in the host. Gdev [38] supports both graphics processing virtualization and the CUDA computing virtualization. Gdev can achieve the fully functional virtualization because it is built on top of an open source GPU driver, and hence it can access and control very fine-grained and low-level GPU functionality.

Migration of GPGPU computing programs is in a very early stage at this time. The two currently available solutions [111, 112] both use an interception layer for CUDA calls, which is similar to the aforementioned simple virtualization solutions. The proprietary GPU driver and closed GPU specifications prevent efficient GPU virtualization and migration. As a result, the obstacle is not any technical issue, but some marketing policies of GPU vendors.

6.2.6 Resource Management and Scheduling

These are actually no techniques to improve system-level GPU computing performance, but system-level work that improves all GPGPU computing applications.

---

[2] http://aws.amazon.com/ec2/

TimeGraph [41] is a GPU resource scheduling system based on an open source GPU driver. TimeGraph schedules GPU commands issued from different tasks based on their priorities to achieve resource isolation and reservation for real-time environments. A similar work is PTask [40], which also deals with the scheduling problem of GPU tasks. Different from TimeGraph, PTask considers the priorities of both GPU tasks and CPU tasks to achieve fair scheduling by counting GPU execution time into a task's total execution time. As said in Section 6.2.3, PTask also provides a UNIX pipe-like model to synthesize an efficient data flow of multiple GPU tasks with dependencies. The aforementioned Gdev [38] is also an OS-level GPU resource management and scheduling solution. Gdev uses the same open source GPU driver as TimeGraph, so it can provide similar scheduling functionality, though not for real-time environments.

6.3 Future

GPUs have been developed much faster than CPUs: the number of cores is doubled almost in only one year [4] instead of CPU's 18 months following Moore's Law.[3] Thus we can confidently believe that future GPUs will come with tens or hundreds of thousands of cores. Such amazingly large number of cores are ideal to deal with current "big data" trend in both application level and the system level.

Another technical improvement should be the CPU-GPU communication. Today's fastest PCIe 3.0 bus only reaches 16 GB/s peak bandwidth, compared with the more than 300 GB/s GPU device memory bandwidth, it easily becomes the bottleneck in many GPU computing as we have seen in previous sections and chapters. Besides dramatically increasing PCIe bandwidth, another potentially useful technique is the integrated GPUs that sit on the same chip of the CPU. Integrated GPUs are expected to eliminate the PCIe bus overhead. However, most current integrated GPUs are limited to only access their own dedicated main memory region. This can still cause possible data copy in main memory due to the dedicated GPU region. What's more, the host side memory bus interface is much narrower than the dedicated GPU's, for example, DRAM's 64 bits width versus GTX Titan Black's 384 bits. As a result, integrated GPUs suffer more memory copy issues. The recent Kaveri accelerated processing unit (APU)[4] has been designed to provide a better technique: heterogeneous uniform memory access (hUMA) [113]. hUMA allows integrated GPUs to directly access any memory locations, and even further, the CPU cache and GPU

[3] http://en.wikipedia.org/wiki/Moore's_law

[4] http://www.amd.com/en-us/products/processors/desktop/a-series-apu

cache benefit from hardware-based cache coherence. Though hUMA still can't provide the wide device memory interface, at least it eliminates any possible copy in main memory. As a result, the main memory techniques should try to provide wider and faster memory for integrated GPUs, too.

Another system-level GPU computing area that is lacking attention is with mobile devices. Traditional GPGPU computing has already explored mobile GPUs for mainly visual computing [114, 115, 116], but none of the system-level software has tried such small but ubiquitous GPUs. The main problem in current mobile GPU computing is their relatively low performance compared with the desktop GPUs. According to a mobile GPGPU computing evaluation [117], mobile GPU performance is still not on par with their desktop CPU competitors. Fortunately, mobile GPUs are getting more powerful. A recently released mobile processor has significantly increased its GPU cores to up to 192 CUDA cores.[5] This makes it possible to get much faster GPU computing than the CPU on mobile devices. Mobile GPUs are more closely combined with CPUs than the desktop ones. It may be possible for mobile GPUs to share and control more integrated components such as memory, DMA, and I/O on the highly integrated system-on-chip (SoC) for mobile devices. But mobile devices also suffer from a common energy consumption problem. So future system-level GPU computing on mobile devices will definitely need new techniques to achieve high performance yet power efficiency.

6.4 Summary

This literature survey covers recent applications of and techniques for improving the performance and functionality of system-level GPU computing. It has been done with strong system flavor, from the applications to the techniques, which are very different from traditional GPU computing surveys such as the work done in [15]. Besides existing work, it tries to identify several potential system-level tasks that can be improved with GPU computing. The system techniques surveyed in this chapter share a large portion with the ones applied in our work, which confirms the effectiveness of each other.

As showed in this survey, some techniques such as batched processing, overlapped GPU computation and memory copy, and using locked memory directly to avoid double-buffering GPU DMA, are commonly used by the surveyed system work. However, among the generic principles and techniques described in Chapter 3, the truly asynchronous GPU programming, SIMT-friendly data structures and using locked memory according to different data

[5] http://www.nvidia.com/object/tegra-k1-processor.html

usage patterns are newly proposed in this study. Previous system designs either did not study the related problems, or used very specific approaches. For example, as shown in Section 6.2.2.3, PacketShader avoids wasting PCIe bandwidth by copying only an IP address and packet identifier, but it is an IP routing specific design compared with `Snap`'s generic ROI-based packet slicing that is available to all the packet processing tasks. The surveyed papers are actually origins of most techniques proposed in this dissertation work, after careful trade-off, abstracting, generalization and many rounds of design considerations. Those techniques have been categorized by the problems and challenges they've solved in order to make it convenient for readers to refer to in practice.

CHAPTER 7

CONCLUSION

Because of its low-level position in the software stack, efficient system-level software is vital to high performance computer systems. However, computationally expensive tasks in system-level software require more and more computing power due to the rapidly increased bulk-data workloads. They may consume excessive processing power, slowing down the entire software stack.

We deal with this problem using parallel GPUs. The highly parallel GPU processors provide a throughput-oriented architecture, which is designed for parallel bulk-data processing. Integrating GPU computing into originally latency-oriented low-level system software has faced many technical challenges that prevent high performance GPU computing in system level. We have proposed generic principles for designing and implementing GPU-accelerated system-level applications. The two specific instances, `GPUstore` and `Snap` frameworks for two representative kinds of system software, have successfully enabled efficient GPU computing in typical system-level tasks, and both have got significant performance improvement with their efficient memory and throughput-oriented design guided by our proposed principles. The success of these two not only validates the feasibility of system-level GPU computing, but also provides a variety of effective generic techniques and abstractions that are designed to deal with common challenges encountered during the GPU computing integration, achieving better performance speedup. We then surveyed the current system-level GPU computing area, identifying potential applications, discussing useful techniques applied in existing work, comparing them with our techniques, and giving interested readers and researchers literature to get a picture of the state of the art and more importantly, a technical reference for their own system-level GPU computing work.

Although the techniques and principles described in this work are mainly for parallel GPUs, they may also be applied to other similar computing platforms. For example, the wide range of coprocessors includes cryptography coprocessors network processors, digital

signal processing (DSP) coprocessors, computing coprocessors such as Xeon Phi [1] and the field programmable gate array (FPGA). These coprocessors share a lot of common architecture features with GPUs: SIMD style throughput-oriented computing, PCIe-based high latency communication, indirect or limited host memory access capability, and dedicated on-device memory. All of these common features indicate that similar challenges also exist when using those coprocessors in system code. As a result, our generic system-level GPU computing principles and techniques that are designed to deal with those challenges may also apply to those platforms. Systems working with the coprocessors may also benefit from the study in this work. Besides the coprocessors, today's multicore CPU platform may benefit from this study as well. A recent study [59] that did batched processing in Click on CPUs shows that the CPU-based packet processing can also get speedup from batching packets. The similar techniques we implemented in `Snap`, such as batching elements and packet batch data structures, are the candidate techniques applicable to multicore CPUs. This shows the much broader impact of our work, which is not limited to GPUs only, but also valuable to a wide range of computing platforms.

[1] https://software.intel.com/en-us/mic-developer

REFERENCES

[1] Amazon, "Amazon Elastic Compute Cloud: User Guide," 2014. Amazon Web Services Inc.

[2] W. Wade, "How GRID Brings Amazing Graphics to the Enterprise Cloud," in *GPU Technology Conference*, GTC'13, 2013.

[3] NVIDIA, "GeForce GTX Titan Black Gaming GPU." http://www.nvidia.com/gtx-700-graphics-cards/gtx-titan-black.

[4] NVIDIA, "CUDA C Programming Guide: v6.0," 2014. NVIDIA Inc.

[5] M. Garland and D. B. Kirk, "Understanding Throughput-oriented Architectures," *Comm. ACM*, vol. 53, pp. 58–66, 2010.

[6] K. Yang, B. He, Q. Luo, P. V. Sander, and J. Shi, "Stack-based Parallel Recursion On Graphics Processors," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pp. 299–300, ACM, 2009.

[7] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 86–97, ACM, 2010.

[8] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU Communication Management and Optimization," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pp. 142–151, ACM, 2011.

[9] J. Soman, K. Kishore, and P. Narayanan, "A Fast GPU Algorithm for Graph Connectivity," in *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pp. 1 –8, April 2010.

[10] R. R. Amossen and R. Pagh, "A New Data Layout for Set Intersection on GPUs," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pp. 698–708, IEEE Computer Society, 2011.

[11] M. S. Rehman, K. Kothapalli, and P. J. Narayanan, "Fast and Scalable List Ranking On The GPU," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pp. 235–243, ACM, 2009.

[12] G. Capannini, F. Silvestri, and R. Baraglia, "K-Model: A New Computational Model for Stream Processors," in *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pp. 239 –246, Sept. 2010.

[13] B. Domonkos and G. Jakab, "A Programming Model for GPU-based Parallel Computing With Scalability and Abstraction," in *Proceedings of the 2009 Spring Conference on Computer Graphics*, SCCG '09, pp. 103–111, ACM, 2009.

[14] S. Hong and H. Kim, "An Analytical Model for A GPU Architecture With Memory-level and Thread-level Parallelism Wwareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 152–163, ACM, 2009.

[15] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[16] J. D. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[17] W. Sun, R. Ricci, and M. J. Curry, "GPUstore: Harnessing GPU Computing for Storage Systems In The OS Kernel," in *Proceedings of the Fifth ACM International Systems and Storage Conference (SYSTOR)*, June 2012.

[18] W. Sun and R. Ricci, "Fast and Flexible: Parallel Packet Processing With GPUs and Click," in *Proceedings of the Ninth ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, Oct. 2013.

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18, pp. 263–297, Aug. 2000.

[20] Khronos Group, "OpenCL Specification 2.0." https://www.khronos.org/opencl.

[21] M. Jamshed, J. Lee, S. Moon, D. Kim, S. Lee, and K. Park, "Kargus : A Highly-scalable Software-based Intrusion Detection System Categories and Subject Descriptors," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS'12, pp. 317–328, 2012.

[22] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader : A GPU-Accelerated Software Router," in *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM'10, pp. 195–206, ACM, Oct. 2010.

[23] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader : Cheap SSL Acceleration with Commodity Processors," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, 2011.

[24] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort : High Performance Network Intrusion Detection Using Graphics Processors," in *Proceedings of the 11th International Symposium On Recent Advances In Intrusion Detection*, RAID'08, Sept. 2008.

[25] T. Prabhu, S. Ramalingam, M. Might, and M. Hall, "EigenCFA: Accelerating Flow Analysis with GPUs," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pp. 511–522, ACM, 2011.

[26] NVIDIA, "CUDA C Best Practice Guide: v6.0." http://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[27] P. Kipfer and R. Westermann, "Improved GPU Sorting," *GPU Gems*, Apr. 2005.

[28] P. Harish and P. J. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pp. 197–208, Springer-Verlag, 2007.

[29] NVIDIA, "CUDA Basic Linear Algebra Subroutines (cuBLAS)." https://developer.nvidia.com/cublas.

[30] M. Harris, S. Sengupta, and J. D. Owens, "Parallel Prefix Sum (Scan) with CUDA," *GPU Gems*, 2007.

[31] D. Roger, U. Assarsson, and N. Holzschuch, "Efficient Stream Reduction on the GPU," in *Workshop on General Purpose Processing on Graphics Processing Units*, Oct 2007.

[32] M. Billeter, O. Olsson, and U. Assarsson, "Efficient Stream Compaction on Wide SIMD Many-core Architectures," in *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pp. 159–166, 2009.

[33] D. Le, J. Chang, X. Gou, A. Zhang, and C. Lu, "Parallel AES Algorithm for Fast Data Encryption on GPU," in *Proceedings of the 2nd International Conference on Computer Engineering and Technology*, ICCET'10, pp. 1–6, IEEE, Apr. 2010.

[34] A. Dunkels, "uIP (micro IP): A micro TCP/IP stack." http://en.wikipedia.org/wiki/UIP_(micro_IP).

[35] A. Dunkels, "lwIP: A very light-weighted TCP/IP stack." http://savannah.nongnu.org/projects/lwip.

[36] M. Harris, "How to Optimize Data Transfers in CUDA C/C++," Dec. 2012. http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc.

[37] freedesktop.org, "Nouveau: Accelerated Open Source driver for nVidia Cards." http://nouveau.freedesktop.org/wiki.

[38] S. Kato, M. Mcthrow, C. Maltzahn, and S. Brandt, "Gdev : First-Class GPU Resource Management in the Operating System," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, ATC'12, 2012.

[39] A. Brinkmann and D. Eschweiler, "A Microdriver Architecture for Error Correcting Codes inside the Linux Kernel," in *Proceedings of the ACM/IEEE Conference on High Performance Computing*, SC'09, (Portland, OR), Nov. 2009.

[40] C. J. Rossbach, J. Currey, and B. Ray, "PTask : Operating System Abstractions To Manage GPUs as Compute Devices," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, 2011.

[41] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa, "TimeGraph : GPU Scheduling for Real-Time Multi-Tasking Environments," in *Proceedings of the 2011 USENIX Annual Technical Conference*, ATC'11, 2011.

[42] FastestSSD.com, "SSD Ranking: The Fastest Solid State Drives," Apr. 2012. http://www.fastestssd.com/featured/ssd-rankings-the-fastest-solid-state-drives.

[43] Wikipedia, "WAN optimization." http://en.wikipedia.org/wiki/WAN-optimization.

[44] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[45] R. Heady, G. Luger, A. Maccabe, and M. Servilla, "The Architecture of a Network Level Intrusion Detection System," tech. rep., Department of Computer Science, University of New Mexico, 1990.

[46] T. Anderson, T. Roscoe, and D. Wetherall, "Preventing Internet Denial-of-Service with Capabilities," in *Proceedings of the HotNets Workshop*, 2003.

[47] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste, "XIA: Efficient Support for Evolvable Internetworking," in *Proceedings of NSDI*, 2012.

[48] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet Caches on Routers: the Implications of Universal Redundant Traffic Elimination," in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 219–230, ACM, 2008.

[49] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting Parallelism to Scale Software Routers," in *Proceedings of the 22nd SOSP*, 2009.

[50] L. Rizzo, "Netmap: A Novel Framework For Fast Packet I/O," in *Proceedings of USENIX ATC*, 2012.

[51] J. C. R. Bennett, C. Partridge, and N. Shectman, "Packet Reordering Is Not Pathological Network Behavior," *IEEE/ACM Trans. Networking*, vol. 7, pp. 789–798, Dec. 1999.

[52] M. Laor and L. Gendel, "The Effect of Packet Reordering in a Backbone Link on Application Throughput," *Network, IEEE*, vol. 16, pp. 28 – 36, Sep/Oct 2002.

[53] E. Kohler, "Click Source Code Repository." https://github.com/kohler/click.

[54] RouteView, "Routing Table Dump of RouteView Network." http://www.read.cs.ucla.edu/click/routetabletest-167k.click.gz.

[55] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, pp. 499–511, June 2007.

[56] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Commun. ACM*, vol. 18, pp. 333–340, June 1975.

[57] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *Proceedings of the 13th USENIX LISA*, 1999.

[58] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," in *Proceedings of OSDI*, 2002.

[59] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon, "The Power of Batching In The Click Modular Router," in *Proceedings of the ACM Asia-Pacific Workshop on Systems*, 2012.

[60] NVIDIA, "CUDA Dynamic Parallelism Programming Guide." http://docs.nvidia.com/cuda/cuda-dynamic-parallelism.

[61] W. Sun and R. Ricci, "Augmenting Operating Systems With the GPU," *Arxiv Preprint*, vol. abs/1305.3345, 2013.

[62] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular Expression Matching on Graphics Hardware for Intrusion Detection," in *Proceedings of the 12th International Symposium On Recent Advances In Intrusion Detection*, RAID'09, Sept. 2009.

[63] Y. Li, D. Zhang, A. X. Liu, and J. Zheng, "GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers," in *Proccedings of the 9th Architectures for Networking and Communications Systems*, ANCS'13, pp. 1–12, IEEE, Oct. 2013.

[64] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, and B. Liu, "Wire Speed Name Lookup : A GPU-based Approach," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13, pp. 199–212, 2013.

[65] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking Named Content," in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pp. 1–12, 2009.

[66] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, "High Throughput Low Latency LDPC Decoding on GPU for SDR Systems," in *Proceedings of the 1st IEEE Global Conference on Signal and Information Processing*, no. December in GlobalSIP'13, (Austin, Texas USA), pp. 3–6, IEEE, Dec. 2013.

[67] A. Nottingham and B. Irwin, "Parallel Packet Classification Using GPU Coprocessors," in *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '10, 2010.

[68] P. Bhatotia and R. Rodrigues, "Shredder : GPU-Accelerated Incremental Storage and Computation," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, 2012.

[69] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu, "A GPU Accelerated Storage System," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, p. 167, ACM Press, 2010.

[70] M. L. Curry, H. L. Ward, A. Skjellum, and R. Brightwell, "A Lightweight, GPU-Based Software RAID System," in *Proceedings of the 39th International Conference on Parallel Processing*, ICPP'10, (San Diego, CA), pp. 565–572, IEEE, Sept. 2010.

[71] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, 1988.

[72] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[73] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang, "Generalized File System Dependencies," SOSP 2007, ACM.

[74] M. Silberstein, B. Ford, and E. Witchel, "GPUfs : Integrating a File System with GPUs," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pp. 485–497, 2013.

[75] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, 2004.

[76] M. D. Lieberman, J. Sankaranarayanan, and H. Samet, "A Fast Similarity Join Algorithm Using Graphics Processing Units," in *Proceedings of the 24th IEEE International Conference on Data Engineering*, no. April in ICDE'08, pp. 1111–1120, Apr. 2008.

[77] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational Joins on Graphics Processors," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD'08, (New York, USA), ACM Press, 2008.

[78] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast Computation of Database Operations using Graphics Processors," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD'04, June 2004.

[79] P. B. Volk, D. Habich, and W. Lehner, "GPU-Based Speculative Query Processing for Database Operations," in *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, ADMS'10, Sept. 2010.

[80] G. Diamos, A. Lele, J. Wang, and S. Yalamanchili, "Relational Algorithms for Multi-Bulk-Synchronous Processors," tech. rep., NVIDIA, June 2012.

[81] C.-F. Lin and S.-M. Yuan, "The Design and Evaluation of GPU Based Memory Database," in *Proceedings of the 2011 Fifth International Conference on Genetic and Evolutionary Computing*, ICGEC '11, pp. 224–231, 2011.

[82] K. Kohei, "PG-Strom: A FDW Module of PostgreSQL using GPU for Asynchronous Super-Parallel Query Execution." https://github.com/kaigai/pg_strom.

[83] "PostgreSQL Database." http://www.postgresql.org.

[84] P. Ghodsnia, "An In-GPU-Memory Column-Oriented Database for Processing Analytical Workloads," in *The VLDB 2012 PhD Workshop*, (Istanbul, Turkey), pp. 54–59, Aug. 2012.

[85] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O. Connor, and T. M. Aamodt, "Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems," in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'12, pp. 88–98, Apr. 2012.

[86] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort : High Performance Graphics Co-processor Sorting for Large Database Management," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD'06, pp. 325–336, June 2006.

[87] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars : A MapReduce Framework on Graphics Processors," in *Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques*, PACT'08, (Toronto, Canada), Oct. 2008.

[88] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU Clusters," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, IPDPS'11, IEEE, May 2011.

[89] Y. Chen, Z. Qiao, H. Jiang, K.-C. Li, and W. W. Ro, "MGMR: Multi-GPU Based MapReduce," in *Proceedings of the 8th International Conference on Grid and Pervasive Computing*, GPC '13, pp. 433–442, Springer-Verlag, May 2013.

[90] O. Harrison and J. Waldron, "Practical Symmetric Key Cryptography on Modern Graphics Hardware," in *Proceedings of the 17th USENIX Security Symposium*, pp. 195–209, 2008.

[91] O. Harrison and J. Waldron, "Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware," in *Proceedings of the 2nd International Conference on Cryptology in Africa: Progress in Cryptology*, AfricaCrypt'09, pp. 350–367, Springer-Verlag, June 2009.

[92] A. Moss, D. Page, and N. P. Smart, "Toward Acceleration of RSA Using 3D Graphics Hardware," in *Proceedings of the 11th IMA International Conference on Cryptography and Coding*, pp. 369–388, Springer-Verlag, Dec. 2007.

[93] G. Vasiliadis and S. Ioannidis, "GrAVity : A Massively Parallel Antivirus Engine," in *Proceedings of the 13th International Symposium On Recent Advances In Intrusion Detection*, RAID'10, Sept. 2010.

[94] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Commun. ACM*, vol. 20, pp. 762–772, Oct. 1977.

[95] Kaspersky Lab, "Kaspersky Lab Utilizes NVIDIA Technologies to Enhance Protection." http://www.kaspersky.com/news?id=207575979.

[96] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," USENIX Security 2006.

[97] O. Shivers, "Control-Flow Analysis of Higher-Order Languages," May 1991. PhD Thesis, Carnegie Mellon University.

[98] S. Liang, M. Might, and W. Sun, "Godel Hashes: Fast, Compact, Monotonic, Dynamic, Incremental and Perfect," in *Submission*, 2014.

[99] J. A. Storer and T. G. Szymanski, "Data Compression via Textual Substitution," *Journal of ACM*, vol. 29, pp. 928–951, Oct. 1982.

[100] A. Ozsoy, M. Swany, and A. Chauhan, "Pipelined Parallel LZSS for Streaming Data Compression on GPGPUs," in *Proceedings of the IEEE 18th International Conference on Parallel and Distributed Systems*, pp. 37–44, IEEE, Dec. 2012.

[101] W. Fang, B. He, and Q. Luo, "Database Compression on Graphics Processors," *Proceedings of the VLDB Endowment*, vol. 3, pp. 670–680, Sept. 2010.

[102] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace, "Migratory Compression: Coarse-grained Data Reordering to Improve Compressibility," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2014.

[103] C. Kim, K.-w. Park, and K. H. Park, "GHOST : GPGPU-Offloaded High Performance Storage I/O Deduplication for Primary Storage System Categories and Subject Descriptors," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'12, pp. 17–26, ACM, 2012.

[104] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "MIDeA : A Multi-Parallel Intrusion Detection Architecture," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS'11, Oct. 2011.

[105] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data Transfer Matters for GPU Computing," in *Proceedings of the 19th IEEE International Conference on Parallel and Distributed System*, 2013.

[106] Y. Fujii, T. Azumi, N. Nishio, and S. Kato, "Exploring Microcontrollers in GPUs," in *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pp. 1–6, ACM Press, 2013.

[107] NVIDIA, "RDMA for GPUDirect." http://docs.nvidia.com/cuda/gpudirect-rdma.

[108] M. Dowty and J. Sugerman, "GPU Virtualization on VMware s Hosted I/O Architecture," *ACM SIGOPS Operating System Review*, vol. 43, pp. 73–82, July 2009.

[109] V. Gupta, A. Gavrilovska, and P. Ranganathan, "GViM : GPU-accelerated Virtual Machines," in *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt'09, (Nuremberg, Germany), ACM, Mar. 2009.

[110] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds," in *Proceedings of the 16th International Euro-Par Conference on Parallel Processing*, EuroPar'10, pp. 379–391, Springer-Verlag, 2010.

[111] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi, "CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS'11, pp. 864–876, IEEE, May 2011.

[112] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A Checkpoint/Restart Tool for CUDA Applications," in *Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT'09, pp. 408–413, IEEE, Dec. 2009.

[113] P. Bright, "AMDs "Heterogeneous Uniform Memory Access" Coming This Year in Kaveri," Apr. 2013. http://arstechnica.com/information-technology/2013/04/amds-heterogeneous-uniform-memory-access-coming-this-year-in-kaveri.

[114] K.-T. Cheng and Y.-C. Wang, "Using Mobile GPU for General-purpose Computing, A Case Study of Face Recognition on Smartphones," in *Proceedings of the 2011 International Symposium on VLSI Design, Automation and Test*, pp. 1–4, Apr 2011.

[115] G. Wang, Y. Xiong, J. Yun, and J. Cavallaro, "Accelerating Computer Vision Algorithms Using OpenCL Framework on the Mobile GPU - A Case Study," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2629–2633, May 2013.

[116] M. Huang and C. Lai, "Accelerating Applications using GPUs in Embedded Systems and Mobile Devices," in *Proceedings of the 15th IEEE International Conference on High Peeformance Computing and Communication (HPCC)*, pp. 1031–1038, Nov. 2013.

[117] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng, "General Purpose Computing on Low-power Embedded GPUs: Has It Come of Age?," in *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, pp. 1–10, July 2013.