Software-Hardware Co-design for Energy Efficient Datacenter Computing

by

Tayler Hicklin Hetherington

B.A.Sc., The University of British Columbia, 2011

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

October 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

Software-Hardware Co-design for Energy Efficient Datacenter Computing

submitted by Tayler Hicklin Hetherington in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

in Electrical and Computer Engineering

Examining Committee:

Tor M. Aamodt, Electrical and Computer Engineering
Supervisor

Mieszko Lis, Electrical and Computer Engineering
Supervisory Committee Member

Margo Seltzer, Computer Science
University Examiner

Sudip Shekhar, Electrical and Computer Engineering
University Examiner

Emmett Witchel, The University of Texas at Austin, Computer Science
External Examiner

Additional Supervisory Committee Members:

Steve Wilton, Electrical and Computer Engineering
Supervisory Committee Member

Abstract

Datacenters have become commonplace computing environments used to offload applications from distributed local machines to centralized environments. Datacenters offer increased performance and efficiency, reliability and security guarantees, and reduced costs relative to independently operating the computing equipment. The growing trend over the last decade towards server-side (cloud) computing in the datacenter has resulted in increasingly higher demands for performance and efficiency. Graphics processing units (GPUs) are massively parallel, highly efficient accelerators, which can provide significant improvements to applications with ample parallelism and structured behavior. While server-based applications contain varying degrees of parallelism and are economically appealing for GPU acceleration, they often do not adhere to the specific properties expected of an application to obtain the benefits offered by the GPU.

This dissertation explores the potential for using GPUs as energy-efficient accelerators for traditional server-based applications in the datacenter through a software-hardware co-design. It first evaluates a popular key-value store server application, Memcached, demonstrating that the GPU can outperform the CPU by 7.5× for the core Memcached processing. However, the core processing of a networking application is only part of the end-to-end computation required at the server. This dissertation then proposes a GPU-accelerated software networking framework, GNoM, which offloads all of the network and application processing to the GPU. GNoM facilitates the design of MemcachedGPU, an end-to-end Memcached implementation on contemporary Ethernet and GPU hardware. MemcachedGPU achieves 10 Gbit line-rate processing at the smallest request size with 95-percentile latencies under 1.1 milliseconds and efficiencies under 12

microjoules per request. GNoM highlights limitations in the traditional GPU programming model, which relies on a CPU for managing GPU tasks. Consequently, the CPU may be unnecessarily involved on the critical path, affecting overall performance, efficiency, and the potential for CPU workload consolidation. To address these limitations, this dissertation proposes an event-driven GPU programming model and set of hardware modifications, EDGE, which enables any device in a heterogeneous system to directly manage the execution of pre-registered GPU tasks through interrupts. EDGE employs a fine-grained GPU preemption mechanism that reuses existing GPU compute resources to begin processing interrupts in under 50 GPU cycles.

Lay Summary

This dissertation explores the potential to improve the performance, efficiency, and cost of datacenter computing through the use of highly parallel and efficient hardware accelerators, specifically graphics processing units. However, the types of applications that typically reside in a datacenter are often not considered to be well suited for graphics processing units. This dissertation explores how a popular datacenter application performs on contemporary graphics processing units, highlighting sizable improvements over traditional datacenter hardware. This dissertation then proposes a general software framework for accelerating datacenter applications on graphics processing units, recognizing that all of the computation from receiving a request to sending the reply must be accounted for to achieve the full benefits of the efficient parallel computing hardware. Finally, this dissertation identifies limitations with contemporary graphics processing units and proposes hardware and software enhancements to further improve the usability, performance, and efficiency of graphics processing units in the datacenter.

Preface

This section lists my publications that were completed at The University of British Columbia, discusses how they are incorporated into this dissertation, and highlights my contributions to this dissertation.

The publications are listed in chronological order as follows:

[C1] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor, Tor M. Aamodt. Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems [69]. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 88-98, April 2012.

[C2] Tayler H. Hetherington, Mike O'Connor, Tor M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores [70]. In Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC), pp. 43-57, August, 2015.

Chapter 2. This chapter combines and expands on the material that was presented in [C1] and [C2] to describe the necessary background information for this dissertation.

Chapter 3. A version of this material has been published as [C1]. In [C1], I was the lead investigator responsible for conducting the research and writing the majority of the manuscript under the guidance of Dr. Tor M. Aamodt and with input from Mike O'Connor and Lisa Hsu. Timothy G. Rogers implemented an initial version of the Memcached GPU code, which was targeted towards the GPGPU-Sim software simulator, conducted the corresponding simulator experiments, and wrote the related results sections (Section 3.4.2 and Section 3.4.2). I was responsible for

implementing and evaluating the control-flow simulator, implementing a new version of Memcached targeted towards GPU hardware, conducting the Memcached GPU hardware experiments, analyzing the results, and writing the corresponding portions of the manuscript.

Chapter 4. A version of this material has been published as [C2]. In [C2], I was the lead investigator responsible for conducting the research, implementing the software frameworks, performing the experiments, collecting and analyzing the results, and writing the manuscript under the guidance of Dr. Tor M. Aamodt and with input from Mike O'Connor.

Chapter 5. In this chapter, I was the lead investigator responsible for conducting the research, implementing the proposed event-driven GPU execution programming model and corresponding GPU architectural enhancements in the evaluated software simulation frameworks, conducting the majority of experiments, analyzing the results, and writing the chapter under the guidance of Dr. Tor M. Aamodt. Maria Lubeznov evaluated the performance overheads of concurrent CPU applications and GPU networking workloads and collected the corresponding data. Following this dissertation, a modified version of the material presented in this chapter was published in the International Conference on Parallel Architectures and Compilation Techniques (PACT) 2019 [71].

Chapter 6. This chapter combines and expands on the related work sections that were presented in [C1] and [C2] with the related work in Chapter 5.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ALM Adaptive Logic Modules

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

CDP CUDA Dynamic Parallelism

CFG Control Flow Graph

CLB Configurable Logic Blocks

CMP Chip Multiprocessor

CPC Current Program Counter

CPU Central Processing Unit

CTA Cooperative Thread Array

CU Compute Units

CUDA Compute Unified Device Architecture

DLP Data-Level Parallelism

DMA Direct Memory Access

DNA Direct NIC Access

EDGE Event-Driven GPU Execution

FLOPS Floating-Point Operations Per Second

FPGA Field-Programmable Gate Array

GNOM GPU Network Offload Manager

GPGPU General-Purpose Graphics Processing Unit

GPU Graphics Processing Unit

HLS High-Level Synthesis

HPC High Performance Computing

IAAS Infrastructure-as-a-Service

ILP Instruction-Level Parallelism

IPDOM Immediate Post Dominator

KDU Kernel Dispatch Unit

KMD Kernel Metadata Structure

KMU Kernel Management Unit

LRU Least Recently Used

MSI-X Message Signalled Interrupts

NAPI New API

NIC Network Interface Card

PAAS Platform-as-a-Service

PC Program Counter

PCIE Peripheral Control Interface Express

PGW Privileged GPU Warp

PKQ Pending Kernel Queue

PT Persistent Threads

PTX Parallel Thread Execution

RDMA Remote Direct Memory Access

RLP Request-Level Parallelism

RPC Reconvergence Program Counter

RPS Requests per Second

RSS Receive Side Scaling

RTT Round-Trip Time

RX Receive

SAAS Software-as-a-Service

SIMD Single-Instruction Multiple-Data

SIMT Single-Instruction Multiple-Thread

SKB Socket Buffers

SM Streaming Multiprocessor

SMX Streaming Multiprocessor

TCO Total Cost of Ownership

TDP Thermal Design Power

TLP Thread-Level Parallelism

TMD Task Meta Data

TOS Top of Stack

TPU Tensor Processing Unit

TX Transmit

UTDP Ultra Thread Dispatch Processor

VM Virtual Machine

Acknowledgments

None of this work would have been possible without the support of countless people and organizations. First and foremost, I would like to thank my advisor, Professor Tor Aamodt, for the many years of guidance, support, wisdom, and motivation. Thank you for providing the opportunity to pursue my ideas and helping me to improve them, for your invaluable insights into our field, research, and teaching, for sending me around the world, and for always pushing me to do high-quality research. Your passion and dedication will always be an inspiration to me.

I would especially like to thank my family and friends for helping me throughout this journey. To my father – thank you for everything. You ignited my love for computers; you inspired me to become an Engineer; you taught me that a good scotch with friends is worth the extra weight to carry up a mountain; but most importantly, you are the reason I am the man I am today. To my mother – thank you for all of your love, support, patience, forced breaks, meals, hospital trips, ...the list goes on. You walked with me through the door of my first undergraduate classroom – I am excited to walk off campus with you in the end. To my brother – thank you for your levelheadedness, for always being someone that I can talk to, and for all of the great experiences we have shared. I love that we had the opportunity to work side-by-side throughout our time at UBC and look forward to our many adventures in the future. To my peanut – getting to know you will always be my best memory from UBC. Thank you for your love, for keeping me sane, and for all of your help along the way. I could not have done this without you. Merci. To my grandparents, aunts, uncles, and cousins – thank you for always being there to support me and for helping me to enjoy my other passions. It means so much to me. To my friends – thank you for all of the happiness that you bring to my life and for sticking with

me through the countless late nights and ”sorry, I can’t make it”s.

Thank you to all of the amazing colleagues and co-authors I had the privilege of working with and learning from. Specifically, I would like to thank Mike O’Connor for his invaluable industry knowledge and support, which helped shape my research. To all of the UBC friends I had the pleasure of working with – Wilson Fung, Ali Bakhoda, Tim Rogers, Shadi Assadikhomami, Andrew Boktor, Ahmed ElTantawy, Hadi Jooybar, Ayub Gubran, Dave Evans, Rimon Tadros, Inderpreet Singh, Arun Ramamurthy, Dongdong Li, Amruth Sandhupatla, Maria Lubeznov, Deval Shah, Amin Ghasemazar, Bo Fang, Rohit Singla, Salma Kashani, Hassan Halawa, Jimmy Kwa, Johnny Kuan, Xiaowei Ren, Samer Al-Kiswany, Derick Hsieh, Ryan Jung, Alvin Lam, Dillon Yang, Jimmy Chao, Sheldon Sequeira, Oscar Hou, John Kang, and Chris Eng (and many others) – thank you for making this such a memorable experience. Thank you to all of my colleagues and friends at Oracle Labs who supported me while I finished my PhD. I would also like to thank the members of my qualifying, department, and final university PhD exams, Professor Steve Wilton, Professor Mieszko Lis, Professor Matei Ripeanu, Professor Sathish Gopalakrishnan, Professor Margo Seltzer, Professor Sudip Shekhar, and my external examiner, Professor Emmett Witchel, for their insightful feedback and contributions to improving the work in this dissertation. Additionally, I would like to thank all of the professors I had the opportunity to learn from over the years.

Lastly, I would like to acknowledge the funding sources that made it possible to pursue graduate school – the Natural Sciences and Engineering Research Council of Canada (NSERC) for providing the CGS-D3 and CGS-M, The University of British Columbia for providing multiple awards, and all others who contributed throughout my graduate and undergraduate studies.

To my mother, Roberta Hicklin, and father,
Darrin Hetherington.

Chapter 1

Introduction

Enter the age of computing. Over the past few decades, modern computing systems have rapidly integrated themselves deeply into many aspects of life, ranging from driving research, progressing modern science and medicine, exploring the universe, and managing the world's economy, to playing games, watching our favourite shows, and staying connected with family and friends. Regardless of the task, there is an ever increasing need for higher performance and efficiency.

Improvements in performance enable new classes of applications not previously before possible. Take machine learning and deep learning, for example. In 1957, Cornell Aeronautical Laboratory introduced the concepts of the perceptron and neural networks used for pattern recognition [151]. However, it was nearly 50 years later before deep learning for pattern recognition began its rise [72]. Fast-forward to today where large tech companies such as Google [90], Amazon [8], Oracle [138], Microsoft [120], and Facebook [51] employ machine learning and deep learning in many of their computing centers and products. A large contributor to this accelerated growth in deep learning is the improvements in microprocessor performance. However, these performance improvements must be met with increases in energy efficiency to remain feasible as computing systems continue to scale up and scale out [20].

1.1 Computing Trends

Traditionally, the rapid performance improvements of integrated-circuits (IC) in microprocessors has been driven by two main factors: Moore's law [121] and Dennard Scaling [43]. Moore's law states that the number of transistors on an IC will double roughly every 18-24 months, whereas Dennard states that voltage and current, and hence dynamic power, is proportional to the dimensions of the transistor. The combination of these factors enables the transistor switching frequency (clock frequency) on ICs to increase without increasing dynamic power and enables more transistors to fit on an IC within similar size and power constraints. As a result, general-purpose central processing units (CPUs) have enjoyed consistent improvements in single-threaded performance from generation to generation.

However, Dennard scaling has begun to break down over the last decade as we approach the physical limits of transistor sizes and other factors, such as the increasing contribution of transistor leakage current, reduce the ability to continue decreasing power proportionally with smaller transistor sizes. Consequently, powering and cooling the larger number and higher density of transistors on ICs are becoming prohibitively more expensive (referred to as the "power wall" [15, 122]) and has limited the opportunity to continue increasing the clock frequency as a means to improve performance. This has driven many architectural enhancements to utilize the additional transistors, such as exploiting instruction-level parallelism (ILP) through out-of-order processing and branch prediction, to further improve single-threaded performance. While effective, there is only so much parallelism that can be extracted from single-threaded programs, which, along with high memory latencies, limits the potential gains from ILP. This has motivated the microprocessor industry to transition towards parallel computing architectures, such as chip-multiprocessors (CMPs), to combat the diminishing returns in single-threaded optimizations [98].

CMPs consist of multiple independent CPU cores integrated on a single chip, typically sharing portions of the memory system and input/output (I/O) interfaces to communicate with each other and the outside world. CMPs enabled many new opportunities to improve performance over single core systems. Multiple programs can operate concurrently on different cores in a form of spatial mul-

titasking, instead of temporal multitasking on a single core, or individual programs can explicitly define parallel sections of the code to be handled by multiple threads across different cores using parallel application program interfaces (APIs) [19, 84, 94, 109, 133, 137]. Operating systems – privileged system software responsible for the control, management, and security of microprocessor systems – efficiently schedule programs and threads to multiple cores to improve overall system throughput relative to single core systems. Modern CMPs commonly have on the order of 10s of cores and may consist of multiple homogeneous cores, such as Intel's x86 Core-i architectures [167], or heterogeneous cores, such as ARM's big.LITTLE architecture [14]. The trend for adding more cores has continued to push forward as transistor sizes decrease. For example, Intel's Xeon E7 processors contain 24 cores and can be combined in multi-socket server systems for up to 192 cores [81].

However, recent concerns about "dark silicon", in which only a fraction of a chip can be actively utilized within a given power envelope [48, 66, 118], have introduced challenges with multicore scaling to improve performance. This has lead to an increasing focus on specialized accelerators and massively multi-core systems, such as graphics processing units (GPU), field-programmable gate-arrays (FPGA), and application-specific integrated-circuits (ASIC). Such architectures can provide very high levels of performance and efficiency for specific classes of applications, but may give up the flexibility and programmability inherent in general-purpose processors. ASICs are at the extreme end of this scale, providing dedicated hardware solutions to specific operations and applications at the cost of generality and programmability. In contrast to general-purpose processors, which often require multiple steps to perform a single operation to maintain a level of generality, ASICs can directly implement the operation efficiently in hardware. However, while ASICs may contain a level of programmability, they are tied to a specific class of applications and cannot easily evolve with the application. Additionally, ASICs require hardware design and implementation, which increase the complexity and cost relative to software-only solutions.

FPGAs, on the other hand, fall in the middle of the scale as reprogrammable hardware devices. Internally, FPGAs contain many reconfigurable hardware blocks capable of implementing any logic function, dedicated hardware blocks,

I/O blocks, and a reconfigurable interconnection fabric for connecting these components [7, 113, 114, 180]. The FPGA architecture enables high levels of performance and efficiency for certain applications, but the reprogrammability reduces the benefits relative to ASICS [100]. FPGAs are programmed using hardware design languages (HDL), such as Verilog or VHDL, and can be reprogrammed to evolve with changing applications. However, programming in an HDL is still considerably more difficult than programming in software [154] and reprogramming times can be on the order of milliseconds to seconds [142, 147], which imposes challenges when implementing multitasking on FPGAs. While the ease of programming has improved on recent FPGAs with support for higher-level software languages through high-level synthesis (HLS), such as OpenCL [36] and CUDA [142], current HLS solutions tend to achieve improvements in developer productivity by trading off the quality of results [12].

GPUs, the focus of this dissertation, are massively multi-threaded, many-core, throughput-oriented architectures traditionally designed to accelerate graphics applications. Graphics processing often involves performing multiple thousands of similar and independent computations on different pixels, resulting in large amounts of data-level parallelism (DLP). GPUs exploit this parallelism by concurrently executing multiple independent operations on a single-instruction, multiple-data (SIMD) architecture to provide significant gains in performance and efficiency. Fortunately, this property of high DLP is not exclusive to graphics applications. Over the past decade, GPUs have evolved into general-purpose GPUs (GPGPU), increasing the scope of applications that can benefit from the GPU's high-efficiency architecture to non-graphics applications with sufficient DLP. Contemporary GPGPUs are programmed in high-level, parallel software languages, such as CUDA or OpenCL.

At their core, GPGPUs consist of hundreds to thousands of small, low-frequency, in-order cores grouped together into SIMD processing engines, commonly referred to as streaming multiprocessors (SMs) or compute units (CU), and high-bandwidth memory (the GPGPU architecture and programming model are described in detail in Section 2.1). Unlike CPUs, which aim to improve performance through high clock frequencies and aggressive ILP optimizations, GPUs focus on exploiting fine-grained multi-threading (FGMT). Assuming

Table 1.1: Comparing GPU and CPU theoretical performance and efficiency over previous architecture generations. The GPU values were obtained from an NVIDIA whitepaper [131]. The CPU value were obtained from Intel processor specifications [83] and calculated using: FLOPS = number of cores $\times$ peak clock frequency $\times$ flops per cycle.

Architecture (Year)	IC fab.	TDP	GFLOPS	GFLOPS/W
GPU NVIDIA Volta ('17)	12nm FFN	300	15,700	52.3
GPU NVIDIA Pascal ('16)	16nm Fin-FET+	300	10,600	35.3
GPU NVIDIA Maxwell ('15)	28nm	250	6,800	27.2
GPU NVIDIA Kepler ('13)	28nm	235	5,000	21.3
CPU Intel 8th gen core i9 ('18)	14nm	45	921.6	20.5
CPU Intel 8th gen core i7 ('18)	14nm	45	825.6	18.3
CPU Intel 7th gen core i7 ('17)	14nm	45	524.8	11.7
CPU Intel 6th gen core i7 ('16)	14nm	45	473.6	10.5

ample amounts of structured parallelism in the application, FGMT can hide the effects of long latency operations by seamlessly switching between thousands of concurrently operating GPU thread contexts. Coupled with the lower clock frequency, FGMT trades off single-threaded performance with high throughput processing to improve overall performance and energy-efficiency.

Consider the comparison in Table 1.1, which presents the theoretical peak performance and energy-efficiency of different NVIDIA GPUs and Intel CPUs across multiple generations. The table also presents the IC fabrication process and year the processor was released. Performance is measured in billions of single-precision floating-point operations per second (GFLOPS) and energy-efficiency is measured in peak GFLOPS versus the thermal design power (TDP) (GFLOPS/W). As can be seen, the latest NVIDIA Volta GPU (GV100) provides over 17$\times$ the compute throughput and 2.6$\times$ higher energy-efficiency than the latest Intel CPU (Core i9-8950HK). While the Volta has a superior technology fabrication process, even the Maxwell and Kepler architectures with twice the transistor size are able to provide higher performance and energy-efficiency than the latest Intel CPU. However, there

are many limitations in the properties of applications that can actually benefit from GPU acceleration, which introduces challenges with exploiting the available parallelism offered by the GPU architecture. This dissertation argues that the perceived bar for the types of applications that can obtain benefits from the GPU is often too high and that GPUs should be considered as efficient accelerators for a broader class of applications. Specifically, this dissertation explores the potential for using GPUs to improve the performance and efficiency of datacenter applications containing ample request-level (packet-level) parallelism, using Memcached [115] (Section 2.2) as an example.

1.2 GPUs and the Datacenter

The initial applications to pioneer the road for GPGPU computing belonged to the domain of scientific and high-performance computing (HPC) and were able to attain large performance improvements using GPUs [59]. These types of applications are highly structured and well suited for the GPU's SIMD architecture. Furthermore, the GPU is able to match these high levels of performance with efficiency. In fact, as of June 2017, GPUs were used as accelerators in all ten of the top ten most efficient supercomputers (GFLOPS/W), as indicated by the Green500 list [165], while also appearing in two out of the top five supercomputers (TFLOPS) [166]. However, HPC represents a relatively small segment of the overall computing market. According to the IDC, in 2015 the overall server market, such as those found in a datacenter (described below), had revenues of $55.1 billion [78] compared to $11.4 billion for HPC servers [25, 53], which has been a consistent trend over the previous six years ($43.2 billion [76] for the overall server marked compared to $8.6 billion [77] for HPC in 2009). Consequently, improving the performance and efficiency of datacenter applications can have significant economic benefits.

Modern datacenters are massive buildings containing thousands of servers, memory, non-volatile storage, network hardware, and power and cooling systems. Server-side ("cloud") computing in datacenters has become an increasingly popular computing environment with the growth in Internet services and provides many benefits relative to independently managing custom computing resources [20]. Datacenters offer large amounts of computing potential, services, scalability,

security, and reliability guarantees, which enable vendors and customers to easily deploy, manage, and tailor the computing environment to their applications' needs. Services such as Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS) provide varying levels of control for, and support of, the software and hardware resources within the datacenter [6]. Additionally, computing resources can be shared across multiple applications, known as workload consolidation, which improves efficiency through higher utilization and reduces costs for both vendors and customers [20, 105].

Datacenters also simplify the development and maintenance of software. Software vendors can frequently and transparently distribute updates to applications running in the datacenter on known software and hardware configurations, instead of distributing updates to a variety of different types of client hardware and software systems [20]. Furthermore, existing applications tend to be frequently updated and new datacenter applications with varying processing requirements are rapidly deployed, referred to as workload churn, which places generality and flexibility requirements on the hardware resources to be able to support the continuously evolving applications.

Together, the hardware resources in the datacenter can consume tens of megawatts [20]. Reducing energy consumption is therefore a key concern for datacenter operators. At the same time, typical datacenter workloads often have strict performance requirements, which makes obtaining higher energy efficiency through "wimpy" nodes [11] that give up single-threaded performance nontrivial [73, 147]. Additionally, high workload churn introduces challenges with deploying high-efficiency ASICs to accelerate datacenter applications, as the hardware has been specifically designed for a certain class of applications. Consequently, datacenters have traditionally relied on general-purpose processors as the main computing resources. Recently, however, the use of specialized processors in the datacenter to address performance and efficiency limitations has been growing. For example, Facebook's Big-Basin [103] utilizes racks of tightly coupled discrete GPUs for improving machine learning performance; Google uses GPUs [39] and custom Tensor Processing Units (TPU) [90] for improving machine learning performance; and Microsoft uses FPGAs for Bing web search [147] and hardware microservices [27], such as encryption [29]. Along with an increase

in specialized accelerators, datacenters have begun to shift towards rack-scale computing [82], where communicating components in a disaggregated computing [107] system reside in separate racks to increase utilization and efficiency. In such an environment, applications reserve only the resources they require (e.g., computing resources, accelerators, memory, storage), as opposed to underutilizing over-provisioned servers. In order to be effective, the communication overhead between components in separate racks must be minimized.

As highlighted above, GPUs are becoming commonplace accelerators in datacenters. One of the key reasons for this is the GPU's ability to provide very high levels of performance and energy-efficiency. However, the classes of applications taking advantage of the GPUs are limited and, for example, frequently marketed towards machine learning and scientific computing [61]. While these applications have been the driving force for the inclusion of GPUs in the datacenter, there is a large fraction of more traditional datacenter applications, such as web services and databases, that are not often considered for GPU acceleration. This dissertation asks the question, can these types of network-centric applications also benefit from the high performance and efficiency provided by contemporary GPUs? These types of server applications typically contain large amounts of thread-level (TLP) or request-level (RLP) parallelism [98]. For example, Facebook uses an in-memory web caching service, Memcached [115], to alleviate network traffic to expensive backing databases, which is responsible for handling billions of network requests per second (RPS) across multiple servers [124]. This results in significant amounts of parallelism across network requests. However, there are many challenges with exploiting this available parallelism on contemporary GPUs due to the irregular behavior associated with the network-centric server applications.

First, while multiple network requests may perform the same high-level operation in parallel, such as retrieving a piece of data from the server, there may be drastically different types or amounts of operations performed within the request. For example, varying network packet sizes can result in a different number of iterations to process, or different packet contents may trigger additional levels of processing. This leads to decreased utilization and efficiency on the GPU's SIMD architecture (Chapter 2 and Chapter 3). Second, the data access patterns across multiple similar network requests can not be known a priori and may be highly distributed across

the GPU's memory. This decreases the memory bandwidth utilization, reducing both performance and efficiency (Chapter 2 and Chapter 3). Third, network-based server applications tend to have strict latency constraints [41], which is complicated by the GPU's high-throughput oriented architecture with relatively low single-threaded performance. Additionally, GPUs require large amounts of parallel computation to provide the high levels of performance and efficiency. This translates to network request batching, and hence increased latency, when accelerating network requests on the GPU (Chapter 4). Fourth, because network requests are coming off of the network, an efficient framework is required to manage the data and computation movement across communicating components within the heterogeneous environment (Chapter 4). Fifth, network packet processing in current operating systems can contribute to a large fraction of the total end-to-end network application latency. Without also considering the network processing, the benefits of using a GPU for the remaining application processing are limited by Amdahl's law (a measure of the total potential performance gains given the fraction of total computation able to be parallelized) (Chapter 4). Finally, contemporary GPUs are considered as offload accelerators, which traditionally rely on the CPU for managing the launching and completion of GPU tasks. As a result, even if the GPU is responsible for all of the network and application processing, the CPU must still be involved on the critical path using the standard GPU programming interfaces. This increases programming complexity, reduces performance and efficiency, and unnecessarily reduces the ability for the CPU to concurrently work on other tasks (Chapter 5). Each of these challenges must be addressed to obtain any benefits from GPU acceleration of datacenter applications.

1.3 Thesis Statement

This dissertation explores the potential to utilize GPUs as energy-efficient accelerators for server-based applications in the datacenter through a software-hardware co-design. Datacenters are important and ubiquitous computing environments with strict requirements for high performance, high efficiency, and generality. While general-purpose GPUs are capable of providing significant gains in both performance and efficiency for certain applications, traditional server-based applications

do not often adhere to the specific properties expected of an application to obtain the benefits offered by the GPU. This dissertation highlights that the GPU can be used to accelerate such applications through a top-down approach – evaluating the behavior of a popular datacenter application (Memcached) on contemporary GPU hardware and proposing a full end-to-end software stack for accelerating network services on contemporary GPUs – and a bottom-up approach – proposing a novel hardware mechanism and modifications to the GPGPU programming model to improve the independence, efficiency, and programmability of GPUs in a heterogeneous system, such as the datacenter.

This dissertation first performs a detailed characterization and evaluation of Memcached, a high-performance distributed key-value store application, on contemporary GPUs. Compared to traditional GPGPU applications, Memcached is highly irregular in terms of control flow and data access patterns. From an initial evaluation, it might reasonably appear that such an application would perform poorly on a GPU. This dissertation highlights that even in light of the irregular behavior, an application such as Memcached can be redesigned to take advantage of the GPU's high computational capacity and memory bandwidth to achieve improvements in request throughput. Additionally, this dissertation assists with understanding the potential SIMD utilization of an irregular application on a GPU, prior to actually spending the time to implement the application on a GPU, through the use of a custom control-flow simulator.

However, the actual server application processing is only part of the full end-to-end processing required to service a network request. For example, the network packet processing is performed in the operating system on the CPU prior to the server application. As a result, the gains achieved by offloading only the application processing to the GPU are limited by Amdahl's law. This dissertation proposes a complete end-to-end software framework, GPU Network offload Manager (*GNoM*), for offloading both the network and server application processing to the GPU. *GNoM* addresses many of the challenges with achieving high-throughput, low-latency, and energy-efficient processing on the GPU's throughput-oriented architecture, facilitating the development of server-based applications on contemporary GPU and Ethernet hardware.

Using *GNoM*, this dissertation proposes MemcachedGPU, an end-to-end im-

plementation of Memcached on a GPU. Multiple components in Memcached are redesigned to better fit the GPU's architecture and communicating components in a heterogeneous environment. MemcachedGPU is evaluated on both high-performance and lower power GPUs and is capable of reaching 10 Gbps line-rate processing with the smallest Memcached request size (over 13 million requests per second (MRPS)) at efficiencies under 12 *u*J per request. Furthermore, MemcachedGPU provides a 95-percentile round-trip time (RTT) latency under 1.1ms at peak throughputs. Together, *GNoM* and MemcachedGPU highlight the GPU's potential for accelerating such server-based applications.

GNoM aims to offload all of the network and application processing to the GPU. However, contemporary GPUs are often considered as second-class computing resources, which require interactions with the host CPU to manage the launching and completion of tasks on the GPU. As a result, even if all of the required end-to-end processing can be performed on the GPU, the CPU is still required to handle I/O and control between GPU and other third-party devices, such as the network interface. This dissertation proposes an event-driven GPU programming model and corresponding hardware modifications, *EDGE*, to enable any device in a heterogeneous system to manage the execution of GPU tasks. *EDGE* pre-registers tasks on the GPU and utilizes fine-grained preemption to execute privileged threads capable of triggering the execution of these tasks. *EDGE* exposes the GPU's interrupt interface to completely bypass the CPU, which improves performance and efficiency, reduces system complexity, and frees up the CPU to work on other tasks. This dissertation also proposes a new GPU barrier instruction, the *wait-release barrier*, which blocks GPU threads indefinitely until being released by the privileged GPU threads in response to an event. The wait-release barriers can help to reduce the overheads of persistently running GPU software frameworks, which continuously poll in-memory work queues for new tasks.

1.4 Contributions

This dissertation makes the following contributions:

1. It argues that GPU's should be considered as accelerators for datacenter network services with ample request-level parallelism by contrasting a pro-

grammer's intuition of an application's potential execution behavior on a GPU with the actual behavior, highlighting that the appearance of irregular control-flow and data-access patterns do not necessarily result in negative performance on a GPU.

2. It describes the methodology used to port Memcached to run on integrated CPU-GPU and discrete GPU architectures, focussing on GPU-only performance with minimal modifications to Memcached's internal implementation and data structures.

3. It characterizes and evaluates Memcached on both integrated CPU-GPU and discrete GPU architectures. To provide deeper insights, this dissertation evaluates the behavior of Memcached on a cycle-accurate GPGPU simulator [1].

4. It presents the initial design of a control flow simulator, CFG-Sim, which can assist GPGPU developers in understanding the potential GPU SIMD utilization of an application prior to actually porting the application to a GPU.

5. It presents *GNoM* (GPU Network Offload Manager), a software system for efficient UDP network and application processing on GPUs, and evaluates the feasibility of achieving low-latency, high-throughput (10 GbE line-rate), and energy-efficient processing at any request size on commodity Ethernet and GPU hardware.

6. It describes the design of MemcachedGPU, an accelerated key-value store that leverages *GNoM* to run efficiently on a GPU, and addresses the challenges associated with partitioning a key-value store across heterogeneous processors. Compared to the initial GPU version of Memcached, MemcachedGPU optimizes for both throughput and latency in a full end-to-end design. Additionally, this dissertation compares MemcachedGPU against prior accelerated Memcached implementations.

7. It explores the potential for workload consolidation on GPUs during varying client demands while maintaining a level of QoS for a higher priority GPU network-based application.

8. It highlights the limitations with contemporary GPUs being considered as second-class computing resources and discusses the need for increased independence of such accelerators to improve performance and efficiency in the datacenter. To this end, this dissertation proposes *EDGE*, an event-driven programming model, API, and corresponding GPU hardware modifications to enable increased GPU independence for applications that primarily use the GPU.

9. It proposes and evaluates a fine-grained, warp-level (Section 2.1.1) GPU interrupt and preemption mechanism, which triggers a set of privileged GPU warps (*PGW*s) from any device in a heterogeneous system through *EDGE* for initiating and managing tasks internally on the GPU.

10. It proposes a new GPU barrier instruction, the wait-release barrier, which halts the execution of specific GPU threads indefinitely until being released by an event, and highlights the benefits of the wait-release barrier to reduce the polling overheads of a persistent GPU thread style of programming.

11. It evaluates *EDGE* in a multiprogrammed environment and highlights the ability to achieve the performance and simplicity of the baseline CUDA programming model with the flexibility of software-only workarounds aimed to increase the independence of GPUs.

1.5 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 discusses the relevant background information for this dissertation, such as the GPU architecture and programming models evaluated in this study, the Memcached key-value store application, networking, and the event-driven programming model.

- Chapter 3 presents the initial evaluation into porting Memcached, an irregular key-value store datacenter application, to both integrated and discrete GPU hardware, and provides deeper insights into the behavior of Memcached on a GPU via a GPGPU simulator.

- Chapter 4 tackles the challenges with implementing Memcached on a GPU in a complete end-to-end system. This chapter proposes *GNoM*, a software framework for accelerating both network and application processing for network-based applications on contemporary GPU hardware. This chapter then presents the end-to-end design and implementation of MemcachedGPU, which utilizes *GNoM* to achieve 10 GbE line-rate processing for any Memcached packet size on both high-performance and low-power discrete GPUs. This chapter also highlights the potential for workload consolidation on GPUs in the datacenter while maintaining a level of QoS for high-priority network applications.

- Chapter 5 identifies limitations with the current system and architectural support for considering GPUs as first-class computing resources in a heterogeneous environment, which rely on the CPU to act as the middleman for control and task management. This chapter proposes *EDGE*, an event-driven programming model and corresponding modifications to the GPU architecture, to enable third-party devices in a heterogeneous environment to directly manage tasks on the GPU.

- Chapter 6 discusses the related work for this dissertation.

- Chapter 7 concludes this dissertation and discusses directions for future work.

Chapter 2

Background

This chapter presents the relevant background information for this dissertation. It first describes the GPU programming model, details two GPU architectures evaluated throughout this research, and discusses irregularities that arise in the GPU thread control-flow and memory systems. This chapter then discusses GPU system-level frameworks and current support for interrupts on GPUs. Next, this chapter details a key-value store application, Memcached, which is evaluated throughout this dissertation. Finally, this chapter provides an overview of the relevant networking and event-driven programming background.

2.1 Graphics Processing Units (GPUs)

This section details the contemporary GPU programming model and GPU architectures assumed in this dissertation.

2.1.1 GPU Programming Model

GPUs are high throughput-oriented offload accelerators traditionally designed for graphics. GPUs have since evolved into general-purpose processors, namely general-purpose graphics processing units (GPGPUs), capable of providing high-throughput, energy-efficient processing for data parallel software, such as high-performance computing (HPC). In this dissertation, the terms GPUs and GPGPUs are used interchangeably.

Non-graphics applications are written in C-like language, such as CUDA [133] or OpenCL [94]. In this dissertation, both NVIDIA and AMD GPUs are evaluated. The terminology is defined for both vendors in the form *NVIDIA term* [*AMD term*]. GPU applications consist of two main components, a host (CPU) component and a device (GPU) component. In this dissertation, both host and CPU, and device and GPU, are used interchangeably. The CPU is responsible for communicating data and control with the GPU (through a GPU driver running on the CPU) via the application programming interfaces (APIs) provided by CUDA and OpenCL. Asynchronous operations enable overlapping data communication and (parallel) task execution through multiple *streams* [*command queues*], which are mapped to physical hardware queues on the GPU. Data communication, which is discussed in more detail below, can occur explicitly through memory copies or implicitly through direct memory access (DMA) from either the CPU or GPU. The GPU is responsible for executing user-defined parallel sections of code, called *kernels*, which perform the actual application processing on the GPU. In this dissertation, kernel and compute kernel are used interchangeably.

GPUs support a single-instruction, multiple-thread (SIMT) execution model, in which groups of scalar threads execute instructions in lock-step. SIMT architectures improve energy efficiency by amortizing the instruction fetch and decode logic, and memory operations, across multiple threads. Similar to single-instruction, multiple-data (SIMD), SIMT processes the same instructions concurrently over multiple different data elements. Unlike SIMD, SIMT enables threads to take different paths through the application's control-flow graph (CFG), which is described more below. In this dissertation, SIMD and SIMT are both used to refer to the GPU's SIMT architecture, which contains a SIMD pipeline capable of executing subsets of SIMD lanes.

GPU kernels contain a hierarchy of threads, which define boundaries for communication and levels of task scheduling. GPU *threads* [*work items*] are grouped into *warps* [*wavefronts*], which execute instructions in a lock-step SIMT fashion. Typical warp sizes are 32 or 64 threads. Warps are further grouped into *cooperative thread arrays* (CTAs) [*work groups*]. CTAs are also referred to as thread blocks. CTAs are the main schedulable unit of work on the GPU. CTAs are then grouped into *grids* [*NDRanges*], which form the main work for the kernel. CTAs

are dispatched as a unit to a *streaming multiprocessor* (SM or SMX) [*Stream core*], whereas individual warps within the CTA are scheduled independently on the corresponding SM. When a kernel is launched, the size and dimension of the kernel is defined. Threads within a CTA can communicate via fast on-chip scratch pad memory, *shared* [*local*] memory, whereas threads in different CTAs must communicate through slower off-chip *global* memory. Additionally, warps within a CTA can perform efficient synchronization through GPU hardware barriers, whereas warps in different CTAs must implement their own form of synchronization via global memory [1].

2.1.2 GPU Architecture

This section presents an overview of the AMD (Chapter 3) and NVIDIA (Chapter 4 and Chapter 5) GPU architectures evaluated in this dissertation.

Figure 2.1 presents a high-level view of an AMD-like GPU architecture assumed in this research [9] [2]. The GPU consists of multiple compute units, each containing one or more stream cores. The stream cores contain multiple processing elements, which perform the actual SIMD computations, branch units to manage thread control flow, general purpose registers, and a memory interface to the caches and the global memory controller. The number of processing elements is typically equal to, or a factor of, the warp/wavefront size (32 or 64). An Ultra-Threaded Dispatch Processor (UTDP) manages the scheduling of kernels and work groups to the compute units. Each compute unit is also connected to on-chip, read-only L1 instruction and data caches, on-chip local data stores (LDS) for scratchpad memory and intra work item communication, and on-chip, read-only L2 caches for images and constant data.

The compute units are connected to a memory controller to service requests to off-chip constant and global memory. AMD provides both discrete GPUs and integrated GPUs. Discrete GPUs are physically separate units connected to the host machine via a connection bus, such as the peripheral component interconnect express (PCIe) bus. Integrated GPUs are collocated on the same physical

[1] The latest NVIDIA GPUs enable synchronization across multiple CTAs [131].

[2] The architecture presented here corresponds to the time that the study in Chapter 3 that was performed (2012).

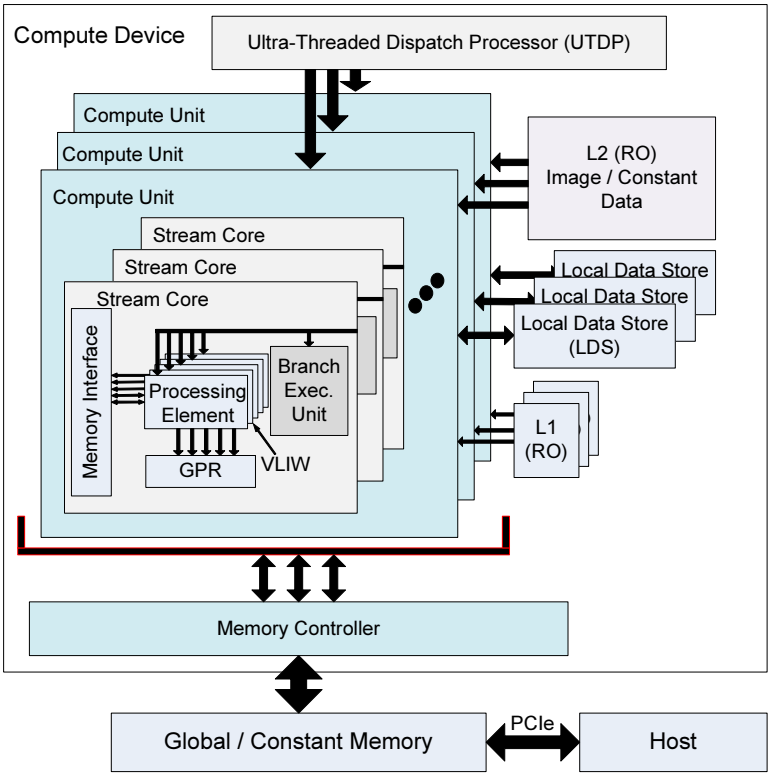


Figure 2.1: High-level view of an AMD-like GPU architecture assumed in this dissertation.

die as the CPU, sharing a common global memory. Each type of GPU, discrete and integrated, have benefits and drawbacks. Discrete GPUs tend to have significantly higher processing capabilities, have their own high-bandwidth physical GPU global memory, and have their own power and cooling systems. However, the separate memory requires data to be explicitly copied between the CPU and GPU or directly accessed across the PCIe bus, which increases complexity and reduces potential performance benefits. On the other hand, integrated GPUs tend to trade off lower processing capabilities with lower power consumption, and remove the need for data to be copied between physical memories over the PCIe bus, since both the CPU and GPU share the same physical memory. Chapter 3 evaluates both discrete and integrated GPUs.

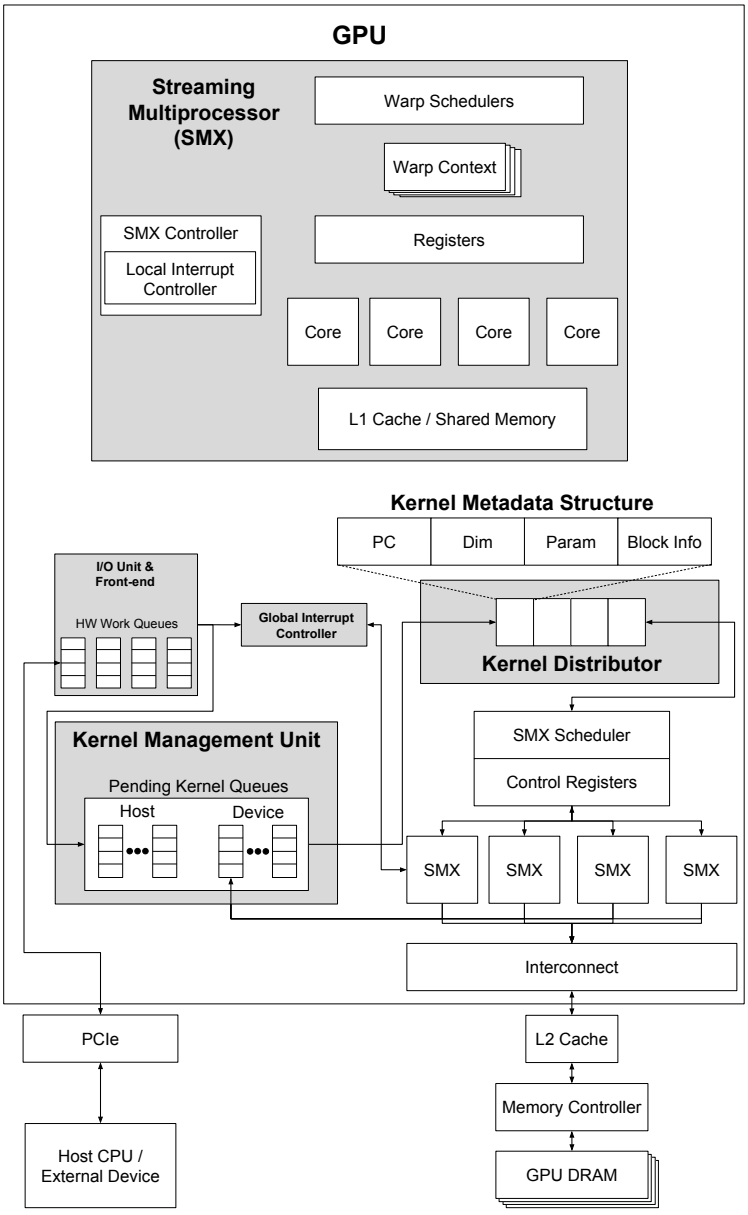


Figure 2.2: High-level view of an NVIDIA-like GPU architecture assumed in this dissertation.

Figure 2.2 presents a high-level view of an NVIDIA-like GPU architecture in-

fluenced by two NVIDIA Patents [87, 155] and academic research [17, 174]. The illustrated GPU contains multiple streaming multiprocessors (SMXs), each with their own control logic, warp schedulers, SIMD processing cores, on-chip L1 instruction and data caches, on-chip shared scratchpad memory for intra-CTA communication, large register files, and thread/warp contexts. Each SMX is connected to an off-chip shared L2 cache via an interconnect. The L2 cache is connected to a memory controller for managing requests to high-bandwidth, off-chip global GPU memory.

The host communicates data and kernels with the GPU over PCIe through a set of hardware managed CUDA *streams*, allowing for concurrently operating asynchronous tasks. NVIDIA's Hyper-Q [134] enables up to 32 independent hardware streams, which depending on the level of resource contention, can perform up to 32 independent concurrent operations. These hardware queues may be stored in a front-end I/O unit, which receives PCIe packets from the host.

The functionality of the Kernel Metadata Structure (KMD), Kernel Management Unit (KMU), and Kernel Distributor Unit (KDU) (Figure 2.2) are best described through an example of launching a kernel on the GPU. The CPU first passes the kernel parameters, kernel metadata (kernel grid and CTA dimensions, shared memory requirements, and stream identifier), and a function pointer to the actual kernel code to the GPU driver running on the CPU. The GPU driver then configures the kernel parameter memory and KMD (structure storing the kernel metadata), and launches the task into the hardware queue corresponding to the specified stream in the GPU's front-end. Internally, the KMU stores pointers to the KMDs waiting to execute in a pending kernel queue (PKQ). The KDU stores the KMDs for the actively running kernels. KMDs are loaded into the KDU from the KMU when a free spot is available. Finally, the SMX scheduler configures and distributes CTAs from the KDU to the SMXs based on available resources for the CTA contexts. The CTA resources consist of thread/warp/CTA contexts, registers, and shared memory. The first resource to be depleted dictates the maximum number of CTAs able to run concurrently on an SMX. Note that a CTA from another kernel, which has different resource requirements, may be able to concurrently run on the same SMX if it does not require more resources than are available. Current NVIDIA GPUs support up to 32 concurrently running kernels and 1024 pending

kernels.

2.1.3 GPU Memory Transfers

Discrete GPUs have physically separate, high-bandwidth GPU memory. Traditional GPU programs perform explicit data transfers between the CPU and GPU memory through the CUDA API and stream interface. More recent GPUs, such as NVIDIA's Pascal and Volta GPUs, support *unified memory*, which enables implicit memory copies via virtual memory demand paging [126, 131]. However, explicitly managing memory is still beneficial from a performance standpoint. Significant losses in end-to-end performance can occur with large data transfers or a large number of small data transfers between the host and device [62], because the time required to transfer data increases linearly with the amount of data needing to be transferred [32]. PCIe 2.x and 3.x with an x16 connection can transfer data at 8 GB/s and 16 GB/s respectively [145], which can be one to two orders of magnitude lower than current GPU DRAM bandwidths (e.g., 900 GB/s on the NVIDIA Volta architecture [131]). On the other hand, integrated GPUs avoid the necessity for explicit memory copies to GPU memory since they share the physical DRAM with the CPU [3]. However, this comes at the cost of lower GPU bandwidth due to sharing resources with the CPU and a lower bandwidth memory architecture.

Furthermore, the separate GPU memory traditionally requires that an external device, such as a network interface controller (NIC) or field-programmable gate array (FPGA), first copy the data to CPU memory and then from CPU memory to GPU memory. GPUDirect [128] removes this requirement by enabling remote direct memory access (RDMA) from a third-party device directly to GPU memory, completely bypassing CPU memory. This is achieved by exposing the GPU's page table and mapping an RDMA-able portion of the GPU's memory into the external device's memory space. However, while GPUDirect solves the issue of the data-path bypassing the CPU, the control-path is still required to go through the GPU driver running on the CPU.

Chapter 4 and Chapter 5 discuss the challenges with control-path dependence on the CPU in more detail.

[3] Mapping and un-mapping operations may be required to ensure the data the GPU is accessing is coherent with any CPU updates.

2.1.4 CUDA Dynamic Parallelism

CUDA Dynamic Parallelism (CDP) [127] enables GPU threads to launch sub-kernels directly on the GPU. CDP exploits nested irregular parallelism in GPU applications, where there may be a varying degree of parallelism throughout the kernel. For example, graph search algorithms may have a varying and unknown number of nodes to visit at each stage, requiring a different number of threads at each stage. CDP can also be used to avoid round trip kernel launches to/from the CPU by allowing GPU threads to internally launch and synchronize kernels. CDP exposes an API to manage the dynamic resource allocation, launching, and synchronization of children kernels in a parent kernel. Similar to host-launched kernels, GPU-launched children kernels are inserted into hardware queues in the KMU for managing pending kernels. However, the flexibility in CDP for any GPU thread to configure and manage sub-kernels has been shown to have high overheads [174, 175], which can significantly limit the performance benefits compared to the baseline where the CPU launches tasks on the GPU.

2.1.5 Kernel Priority and Kernel Preemption

Based on an NVIDIA patent [155], the GPU maintains multiple different queues/lists for storing pending kernels (KMD pointers) to be executed on the GPU. Both host-launched and device-launched kernels can specify a priority associated with the kernel, which may be implemented by assigning different queues different priorities in the KMU. Children kernels in CDP inherit their parent kernel's priority. The KDU can then select a KMD to launch based on a given priority selection algorithm.

Kernel priority can aid the choice of deciding which kernel to schedule next when enough free resources are available. However, while GPUs have long supported preemption for graphics applications, GPUs have traditionally only supported spatial multitasking for GPGPU applications, not temporal multitasking via preemption and context switching. The lack of temporal multitasking results in large, long running GPU applications that consume all GPU resources to block other kernels from running. This occurs even if a pending kernel has a higher priority than the large long running kernel. NVIDIA has recently proposed fine-grained,

instruction-level preemption for GPGPU applications in their newest GPU architectures [126, 131]. However, unlike CPUs, context switching a full GPU kernel can be very expensive, in terms of computing cycles, due to the large amounts of state to save (e.g., register files and shared memory). Additionally, recent research has proposed multiple optimizations to reduce preemption overheads to better support GPU preemption and multitasking [33, 92, 143, 157, 162, 176].

GPU preemption is discussed in more detail in Chapter 5.

2.1.6 Current GPU Interrupt Support

An NVIDIA patent [155] presents an exception handling mechanism that enables the GPU to handle internal exceptions (e.g., arithmetic errors), traps (e.g., threads hitting a breakpoint when debugging) or external interrupts (e.g., cuda-gdb interacting with the GPU or a kill signal from the host CPU). The patent describes a global interrupt mechanism, which minimizes the design and verification complexity – when an interrupt or exception occurs, all warps running on an SM are interrupted and transition from their current code to an interrupt handler. Only the warp(s) responsible for the exception/interrupt actually processes the interrupt, while all other warps immediately return to their original execution. However, all warps from all active CTAs on an SM must temporarily pause their execution to test the exception, which can reduce performance if only a subset of warps are required to service the interrupt/exception. An AMD Graphics Cores Next (GCN) Architecture white paper [10] also describes scalar cores within the Compute Units that are responsible for handling GPU interrupts, which is useful for supporting GPU debugging.

As previously described, discrete GPUs are typically connected to the host CPU via a PCIe bus and contain an I/O and front-end unit to communicate over the PCIe connection (Figure 2.2). As of PCI 3.0, the CPU and external devices can interact with PCIe devices through Message Signaled Interrupts (MSI-X), which support up to 2048 different interrupts. Unlike traditional interrupts that rely on specific interrupt wires, MSI-X treats interrupts as special PCIe packets. As such, any device that can communicate over PCIe is (in theory) able to send and trigger GPU interrupts. To the best of our knowledge, no GPU manufacturer has published

an API for enabling programmers to take advantage of the ability to send interrupts to the GPU directly from a user-space application.

GPU interrupts are discussed more in Chapter 5.

2.1.7 GPU Persistent Threads

Persistent threads (*PT*) are an alternative technique for programming and launching tasks on the GPU. *PT* pre-configures and launches a large number of continuously running persistent CTAs (pCTAs) on the GPU, which poll for new tasks from in-memory work queues to perform the application processing [63]. This increases the application's control of the task and thread scheduling, effectively replacing the CUDA driver and the GPU's hardware kernel and CTA schedulers with a user-level software GPU task scheduler that is independent from the CPU. As such, *PT* also enables any device in the heterogeneous system to initiate a task on the GPU by simply writing into a work queue in the GPU's memory. The polling pCTAs can then identify the arrival of a new task and begin the actual kernel processing for the corresponding task. For this to work, each pCTA must be sized appropriately to accommodate the maximum task size that may be scheduled, which results in underutilized resources for smaller tasks.

There are multiple different techniques for implementing *PT* on GPUs [96, 182]. For example, there can be a single global work queue, which requires synchronization between all pCTAs, or local per-pCTA work queues, which limits contention at the cost of load imbalances. Additionally, all pCTAs may be responsible for both polling the work queue and performing the actual kernel processing, or separate pCTAs may be responsible for polling the work queues and passing the tasks to a separate set of pCTAs for the actual kernel processing (a form of prefetching to overlap reading the work queues and processing the tasks). Each design provides a trade-off in the number of threads synchronizing on the shared work queues and the amount of resources available for performing the kernel processing.

Persistent threads are evaluated further in Chapter 5.

2.1.8 GPU Architectural Irregularities

This section describes two key performance irregularities that are inherent to the GPU architecture: *branch divergence* and *memory divergence*.

Branch Divergence

As previously described, scalar GPU threads are grouped together into warps. In an application without any conditional control flow operations, such as conditional branches, the warp is able to make forward progress through the code while executing instructions for each thread in parallel. This follows the normal SIMD execution model. However, if conditional branches are introduced into the code, it is possible for a subset of the threads in a warp to take the branch while the remaining threads do not. While all threads in a warp previously executed the same instructions, now only a subset of threads in the warp execute the same instructions. This is known as *branch divergence* [110, 119, 172]. Branch divergence can occur any number of times and, in the worst case, each thread in a warp executes a different instruction. Consequently, threads that are not executing a given instruction result in idle lanes in the SIMD hardware, which lowers the SIMD utilization. To handle branch divergence, GPUs contain a hardware component similar to the SIMT stack described by Fung et al. [172, 173]. An example highlighting the SIMT stack is shown in Figure 2.3 and is described below. The SIMT stack can be used to track the active threads at various points throughout the program's execution. An active thread refers to a GPU thread within a warp that is currently executing instructions. A thread becomes inactive if it takes a branch that diverges away from the other threads in the warp.

Each entry on the SIMT stack contains a bit-mask representing the active threads (work items) in a warp (wavefront), with the top element in the stack (TOS) signifying the subset of threads to execute. The SIMT stack also records the current program counter (CPC) and a re-convergence program counter (RPC). The CPC specifies the instruction that the active threads on a corresponding stack entry will execute once it becomes the TOS. As the threads in the TOS entry execute the instructions, the CPC is incremented accordingly. The other counter, the RPC, specifies the immediate post-dominator (IPDOM) instruction. The IPDOM is de-

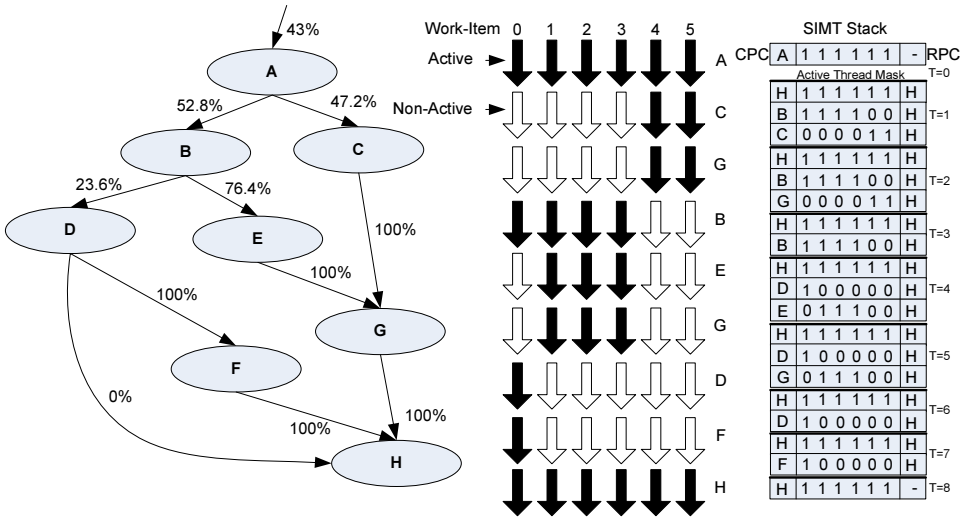


Figure 2.3: SIMT execution example.

fined as the closest instruction in the program that all paths leaving the branch must go through before exiting the function. As such, the IPDOM is the earliest instruction that all threads in the warp are guaranteed to execute the same instruction. The SIMT stack uses the RPC to specify where the active and inactive threads can rejoin. Once the active subset of threads reaches the RPC, the TOS is popped from the stack and the GPU starts executing the new TOS.

Figure 2.3 shows an example of the SIMT execution flow when executing a piece of code taken from a hash function in a popular key-value store application, Memcached. Memcached is discussed in detail in Section 2.2. The corresponding section of Memcached's CFG is shown on the left. The CFG is also annotated with the actual branch probabilities extracted from multiple Memcached runs. For simplicity, in this example there are six threads (work items) per warp (wavefront). Snapshots of the SIMT stack are shown at different points throughout execution on the right.

A GPU scalar thread is represented by a vertical column in the middle of Figure 2.3 (labeled 0 through 5). Active threads are indicated by black arrows and inactive threads are indicated by white arrows. The ovals represent basic blocks in the CFG, which signify a portion of code with single entry and exit points that ensure no

further branch divergence. Hence, once execution starts at the beginning of a basic block, it will always continue until the end of the basic block. Without branch divergence, the minimum number of basic blocks required to reach H from A is four ($A \rightarrow C \rightarrow G \rightarrow H$) and the maximum is five ($A \rightarrow B \rightarrow D/E \rightarrow F/G \rightarrow H$).

At the beginning, all of the threads in the warp are set to active and execute the instructions at block A. At the first branch, threads 0-3 go to block B while threads 4-5 go to block C; the corresponding entries are pushed onto the SIMT stack. The re-convergence point for each thread split is set to the IPDOM block H, which is the first point where all threads must pass through, regardless of the branches taken or not taken in this CFG. Additionally, the previous TOS's CPC is set to H, which indicates that all threads will resume concurrent execution at H. The next step is to execute a subset of the original warp until reaching a re-convergence point. This is achieved by executing the threads at the new TOS. Threads 4-5 execute basic block C (T=1) and G (T=2) before reaching the re-convergence point H. The TOS is popped off and execution switches to threads 0-3 at B (T=3). Here, the threads split again, removing its current entry from the SIMT stack and pushing two new entries onto the stack for basic blocks D and E (T=4). Threads 1-3 execute basic blocks E (T=4) and G (T=5) until reaching the re-convergence point H (T=6), which pops the TOS and switches execution to thread 0 (T=7) to execute basic block D prior to reaching H (T=8). At this point (T=8) all of the threads in the warp are at the re-convergence point H and resume concurrent execution.

Assuming all the basic blocks have the same number of instructions, the SIMD efficiency in this example is approximately 46% and executes nine blocks in total (four more than the maximum number of basic blocks under no branch divergence). Also, block G is executed twice, resulting in more instructions being issued than necessary.

Thus, to achieve the highest performance from the GPU, it is desirable to have as little branch divergence as possible.

Memory Divergence

Memory accesses are another property of GPUs directly affected by the grouping of threads in a warp. If the instruction being executed by all threads is a memory-

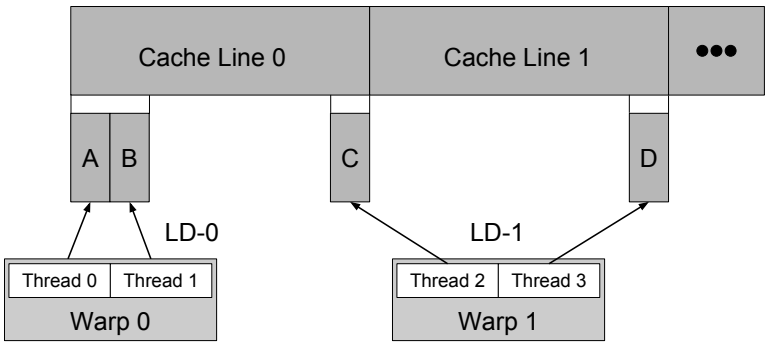


Figure 2.4: GPU memory request coalescing.

access operation, such as a load or a store, each thread will generate a memory request to be handled by the memory system. Assuming each memory request is for a data object in a different region of memory, all requests will be handled separately. This phenomenon is referred to as *memory divergence*. However, if the requested data from threads in a warp lies within a given range, such as the size of a cache line or the size of data returned from a memory request (i.e. high data locality between threads in a warp), the memory requests falling into one of these common regions can be coalesced into a single request. This can significantly reduce the amount of traffic on the memory system.

Consider the example in Figure 2.4. In this example, there are two warps, Warp 0 and Warp 1, each containing two threads, Thread 0-1 and Thread 2-3 respectively. Each thread in Warp 0 executes a load instruction LD-0 and each thread in Warp 1 executes a load instruction LD-1. On the first load instruction, LD-0, Thread 0 loads in *A* and Thread 1 loads in *B*. Because both data objects lay in a single cache line, the two memory requests can be coalesced into a single memory request. As a result, if the request misses in the cache, only a single memory request will be sent to the lower level cache or memory system. However, on the second load instruction, LD-1, Thread 3 sends a memory request for *D*, which falls into a separate cache line from Thread 2's memory request to *C*. Thus, two physical memory requests will be generated. If one of the memory requests hits in the cache while the other memory request misses in the cache, all threads in the warp will be stalled until the last memory request is serviced. Extending this to

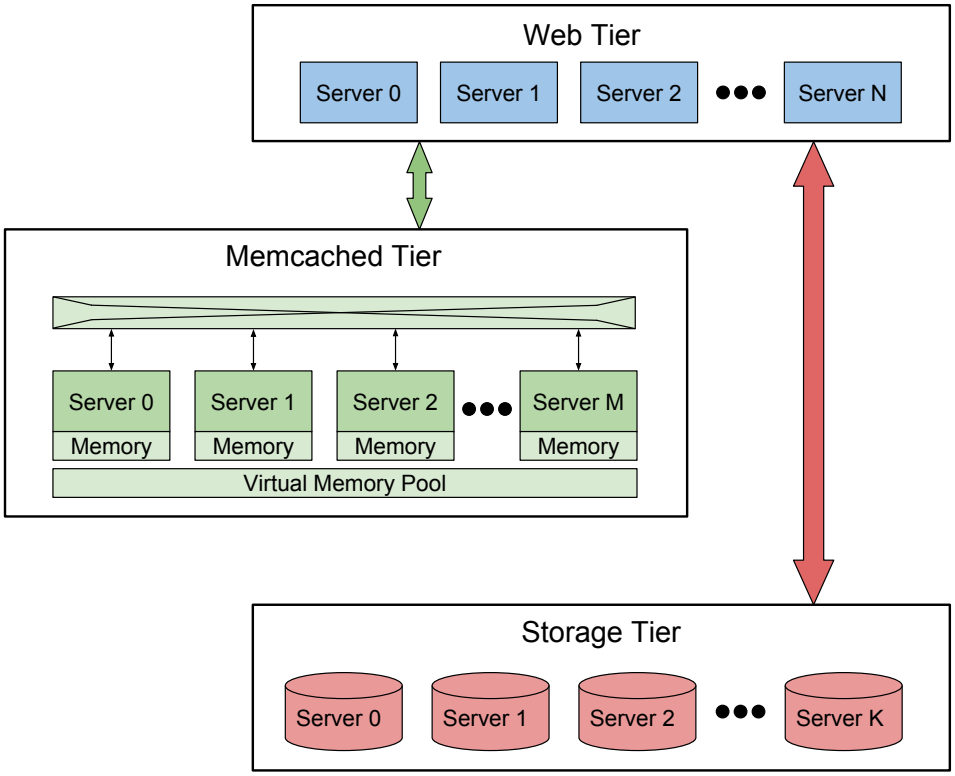


Figure 2.5: Memcached.

real applications, typically containing 32 threads per warp (or 64 work-items per wavefront) and hundreds of warps per kernel, un-coalesced memory requests can generate large amounts of memory traffic to the memory system and significantly affect performance.

Thus, to achieve the highest performance from the GPU, it is desirable to limit the amount of memory divergence.

2.2 Memcached

This section details the main application evaluated in this dissertation, Memcached [115]. Memcached is a general-purpose, scale-out, high-performance, in-memory, key-value caching system used to improve the performance of distributed databases in server applications by caching recently queried data in

main memory. This alleviates the amount of traffic required to be serviced by the back-end databases, which access non-volatile storage or other external sources. These expensive I/O operations can significantly reduce overall performance and increase power consumption. Memcached is used by many popular network services such as Facebook, YouTube, Twitter, Wikipedia, Flickr, and others [115].

A high-level view of how Memcached fits into the existing web and storage tiers in a datacenter is shown in Figure 2.5. Memcached acts as a look-aside cache. Requests are first sent to the Memcached servers and either (1) the data is available and is returned to the requesting server or (2) the requesting server is notified of the cache miss and is responsible for querying the back-end database for the missing data, which is then stored into the Memcached system.

Internally, Memcached implements the key-value store as a hash table. Memcached uses an asynchronous networking event notification library, Libevent [112], which removes the loop-based events in event-driven networks to increase performance. All of the Memcached data resides in volatile system memory, which results in fast accesses compared to storing on disk. Memcached implements a scale-out, distributed architecture by combining main memory from individual servers into a large pool of virtual memory. This aggregation of memory effectively provides a much larger memory space that scales linearly with the number of servers in the system. Each server is fully disconnected from the other servers in the pool, meaning that no communication takes places between the servers. All communication is done between the client and one or more servers, which greatly simplifies the overall system design. Furthermore, Memcached has no reproduction of data, logging, or protection from failure. As the data is stored in memory, a system failure results in the loss of data. Thus, it is the responsibility of the application to implement any failure recovery mechanisms.

Memcached provides a simple key-value store interface to store (*SET*), modify (*DELETE*, *UPDATE*), and retrieve (*GET*) data from the hash table. The key-value pair and corresponding metadata (e.g., size of key/value, last access time, expiry time, flags) is referred to as an *item*. All of the Memcached operations require two hashes of the keys. The first hash selects which server the corresponding request should be directed to (based on a pre-configured mapping of keys to servers) and the second hash selects the appropriate entry in the hash table. Hash-chaining

is used in the event of collisions on write operations and a linear traversal of the linked-list chain is used on read operations when necessary. To avoid expensive memory allocations for every write request, Memcached uses a custom memory allocator from pre-allocated *memory slabs* to allocate memory for items. To reduce memory fragmentation, Memcached allocates multiple memory slabs with different fixed-size entries. The hash table stores pointers to items stored in the memory slabs. Memcached uses a global Least Recently Used (LRU) eviction protocol to keep the most up-to-date data cached in main memory. Items are evicted on write operations if the write exceeds the available memory limit. A time-out period can be added to an item to specify when this item's lifetime is over, regardless of its current usage. This time-out period has precedence over the LRU eviction protocol; items whose timers have expired are first evicted, followed by the LRU items.

Facebook Memcached deployments typically perform modify operations over TCP connections, where it is a requirement that the data be successfully stored in the cache, whereas retrieve operations can use the UDP protocol [124]. Since Memcached acts as a look-aside cache, dropped *GET* requests can be classified as cache misses (requiring queries to the back-end database) or the client application can replay the Memcached request. However, excessive packet drops mitigate the benefits of using the caching layer, requiring a certain level of reliability of the underlying network for UDP to be effective.

2.3 Network Interfaces and Linux Networking

A network interface controller (NIC) is hardware component in a computer system responsible for receiving and sending network packets with other external computer systems. The NIC is connected to the host (CPU) either through an integrated bus on the motherboard or via a connection bus, such as PCIe. Network packets enter hardware RX (receive) queues at the NIC, which are copied to pre-allocated, DMA-able RX ring buffers (driver packet queues), typically residing in CPU memory. The NIC then sends interrupts to notify the CPU of one or more pending RX packets, or the CPU can poll the NIC to check for pending packets. To mitigate high interrupt rates when receiving packets, the Linux kernel uses a hybrid interrupt and polling approach, NAPI (New API). In NAPI, the NIC notifies

the Linux kernel when a packet arrives via an interrupt. The Linux kernel registers the notification, disables future interrupts from the NIC, and schedules a polling routine to run at a later time. The polling routine then services multiple received packets from the NIC up to some pre-defined threshold. The NIC driver copies the packets from the RX ring buffers to Linux Socket Buffers (SKBs) to be processed by the host Operating System (OS), returns the corresponding RX buffers back to the NIC to receive future packets, and re-enables interrupts. When transmitting (TX) packets, the Linux kernel copies packets into DMA-able TX ring buffers and notifies the NIC of the packet to send. The NIC copies the packet into internal TX queues and then transmits the packet on an outgoing link.

Optimizations such as direct NIC access (DNA) can reduce memory copies by allowing user-space applications to directly access the RX and TX ring buffers. This removes the requirement for the Linux kernel to perform additional memory copies to SKBs to process the packet. The user-space application is presented with the raw packet and is responsible for performing any packet processing. In Chapter 4, we expand on this technique by using NVIDIA GPUDirect [135] to directly copy the packet data from the NIC to RX buffers stored in GPU memory, which removes the requirement for the CPU to explicitly copy the packet from CPU memory to GPU memory. Another common optimization in modern NICs is receive side scaling (RSS), which enables the NIC to install packet filters to direct specific packets to specific CPU cores. Packet filters can be installed through the NIC's driver API. We also expand on this technique in Chapter 4 to filter certain packets to the GPU, while all other packets still go through the standard Linux kernel network flow on the CPU.

2.4 Event and Interrupt-Driven Programming

Event or interrupt-driven programming is an alternative style of programming compared to threads. Threads execute a program and can perform synchronous, asynchronous, blocking, and non-blocking operations. Synchronous blocking operations stall the thread until some condition is satisfied and the thread can resume execution. Asynchronous non-blocking operations immediately return, regardless of whether the operation completed successfully or not, which requires ad-

ditional logic to retry an operation if required. Threads require careful partitioning of resources or synchronization on shared resources. In event-driven programming, programs are broken down into fine-grained pieces of code, callbacks, or event handlers, responsible for performing specific operations in response to I/O events [37, 54, 140]. The event handlers can be called either in a continuously running event loop or triggered via interrupts. Event-driven programming can reduce the challenges with concurrency and synchronization of thread-based programming, while also achieving high performance. In Chapter 5, we explore interrupt-driven GPU events, which enables any device in a heterogeneous system to directly manage the execution of GPU tasks independently from the CPU.

Chapter 3

Evaluating a Key-Value Store Application on GPUs

This chapter performs an initial evaluation of a widely used key-value store datacenter application, Memcached (Section 2.2), on discrete and integrated AMD GPUs. GPUs have consistently proven to deliver positive results in scientific and high-performance computing (HPC), as demonstrated by their use in several top supercomputers [165, 166]. This chapter argues that the GPU's highly-parallel and efficient architecture also makes them strong candidates for non-HPC applications with ample parallelism as well. The family of applications considered in this dissertation are network-based datacenter workloads. Many datacenter workloads contain large amounts of thread-level or request-level parallelism [98]. This parallelism stems from the fact that the server is required to process multiple different network requests, potentially at high request rates. If the requests are independent, they can be processed concurrently. However, at first glance these workloads may not seem suitable to run on GPUs due to the existence of irregular control-flow and memory access patterns, since the application does not dictate the type, ordering, or rate of requests that arrive at the server. The main goal of this chapter is to quantify and evaluate the potential for GPUs to accelerate such irregular applications. To this end, this chapter evaluates the discrepancies between a programmer's reasonable intuition on how an irregular application may perform on a GPU and the actual achievable performance. This chapter then explores the challenges in

porting a popular datacenter application, Memcached, to OpenCL and provides a detailed analysis into Memcached's behaviour on discrete and integrated GPUs. To gain greater insight, this chapter also evaluates Memcached's performance on a cycle-accurate GPU simulator. On the integrated CPU+GPU system, we observe up to 7.5× increase in throughput relative to a single CPU core when executing the key-value look-up handler on the GPU. Section 3.4.1 discusses a multi-core CPU comparison.

HPC applications are known to efficiently utilize the underlying GPU hardware to achieve high performance [59]. As discussed in Section 2.1, high GPU performance is possible with ample data-level parallelism to enable a large number of threads to execute in parallel, structured control-flow to maximize the SIMD efficiency of the GPU pipeline, and structured memory access patterns to maximize the utilization of the GPU's high memory bandwidth. Additionally, the time to transfer application data to the GPU should either be short relative to the application's GPU processing time, or the application should be well structured to overlap the communication of data with the computation for another GPU kernel. When performing an initial analysis of an application to accelerate on a GPU, many programmers may look for similar characteristics to existing applications with proven high performance on GPUs. This can lead a programmer to disregard applications that appear to deviate from these characteristics, eliminating some applications from consideration that actually have the potential to perform well on a GPU.

Datacenter (server) applications are a family of highly parallel and economically appealing applications that fall under this category of not appearing to be strong candidates for GPU acceleration. Server applications represent a larger class of applications than HPC, but one that is unstructured. For example, a web server is responsible for receiving incoming network requests, processing the request, and sending the response back out to the network. Assuming each network request performs a similar task on different data, there may be large amounts of available *request-level* parallelism. We define request-level parallelism as the ability to operate on multiple independent requests in parallel. However, the specific operations to perform may differ slightly between network request types or based on the data within the network request. Additionally, the data access patterns may be dependent on the data within the network request, which limits the ability to ex-

ploit memory coalescing on GPUs. For these reasons, it is challenging to quickly determine how such an application would perform on a GPU.

We focus on Memcached (Section 2.2) as a representative example of a datacenter application with high performance requirements. Memcached is a highly memory-intensive application, with the main purpose of storing and retrieving data objects in memory to service a network request. A typical Memcached server may be responsible for handling hundreds of thousands to tens of millions of requests per second (RPS). As a result, there may be a large number of Memcached requests needing to be processed at a given time. Assuming Memcached's requests are independent and that each request can be handled concurrently by a separate thread, a reasonable expectation might be that each thread exhibits independent and irregular behaviour. Specifically, depending on the type of request (*GET*, *SET*, *UPDATE*, *DELETE*), each request may perform a completely different set of operations, or within the same type of request, different operations may be performed depending on the data which is being operated on. Additionally, since the access patterns of each request can not be known a priori, the data accessed in the request packet, hash table, or corresponding memory slab can certainly not be expected to exhibit high locality, leading to memory divergence. Furthermore, on a discrete GPU with isolated memory spaces, special care must be taken to ensure that the CPU and GPU maintain a coherent view of memory. For example, either the Memcached data structures must be partitioned between the devices, such that only a single device accesses each data structure, or potentially large data transfers are required between the CPU and GPU memory on every kernel launch. As such, from an initial evaluation of Memcached, a programmer may conclude that there is little potential for any benefits from GPU acceleration.

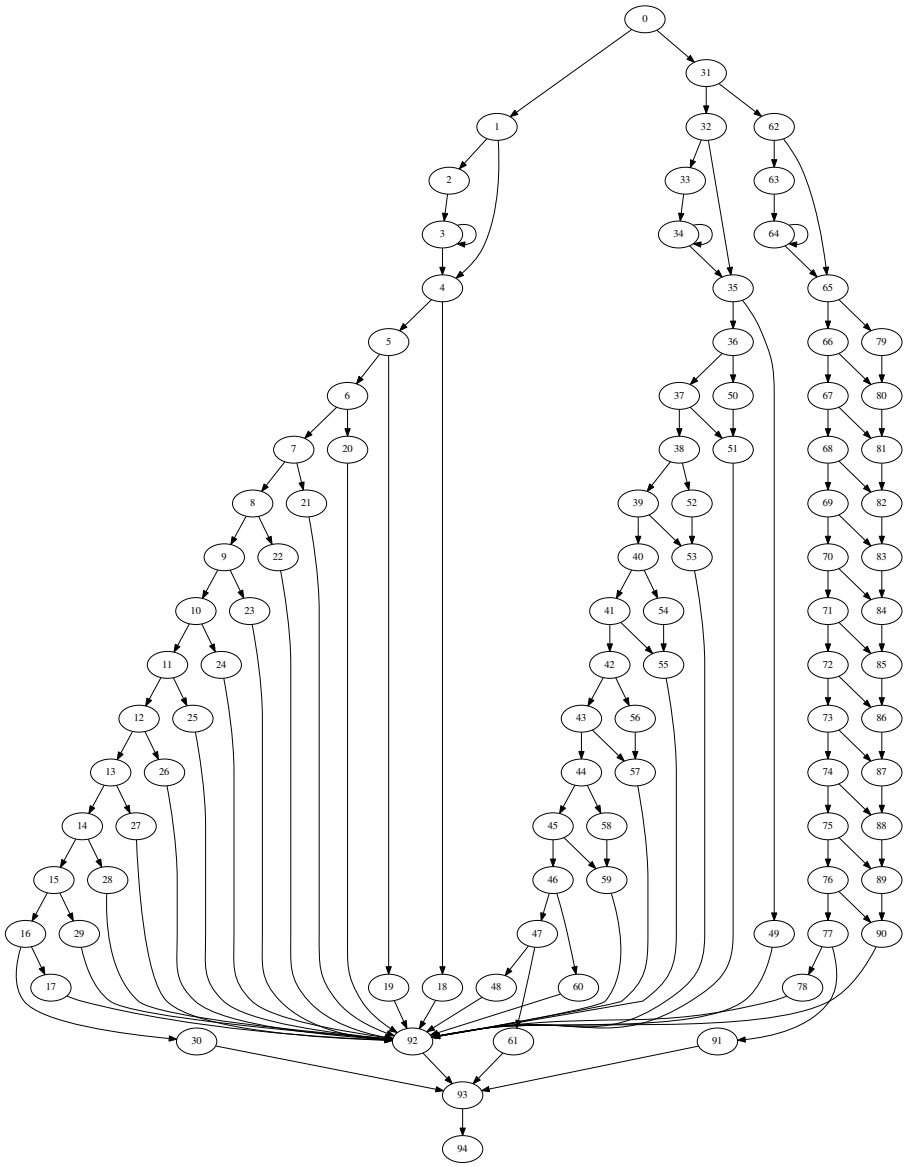


Figure 3.1: Control Flow Graph (CFG) from Memcached's Jenkins hash function.

Performing a deeper analysis into Memcached's code and runtime behaviour, we find that its control flow and memory access patterns are indeed dependent on the input data in the network requests. For example, Memcached's hash function, key comparison, and hash collision resolution technique may result in a different number of iterations or different operations between requests depending on the request data, potentially leading to branch divergence. Recall that SIMD efficiency refers to the fraction of scalar GPU threads that actually execute together in lockstep relative to the maximum number of threads capable of executing together (Section 2.1.8). Consider the control flow graph (CFG) corresponding to Memcached's key hash function (Jenkins Hash [86]), as shown in Figure 3.1. This hash function contains multiple fine-grained branches and loops, which are completely dependent on the format and length of the key to be hashed. If each of these branches is equally likely based on the input data, and each GPU thread is responsible for hashing a different key, the threads may take different paths through the CFG, resulting in low SIMD efficiency.

However, in most cases the probability of taking or not taking each branch is not equal. For example, branches may be responsible for handling errors, unaligned memory accesses, or handling special cases, which occur relatively infrequently. Without evaluating the runtime behaviour of the application, especially after tuning the code for the GPU's SIMD architecture, it is challenging to know the actual achievable SIMD efficiency. To this end, we perform an initial evaluation, which compares an estimation of the SIMD efficiency of Memcached to the actual achieved SIMD efficiency of Memcached on a GPU. The results are presented in Figure 3.2, where *Actual* is the actual SIMD efficiency and *Expected* is the estimated SIMD efficiency if all code paths through the CFG are equally likely. The actual SIMD efficiency was measured on the GPGPU-Sim GPU simulator after implementing Memcached in OpenCL (Section 3.4.2) and the estimated SIMD efficiency is measured using a custom control-flow simulator (Section 3.2). On average, Memcached's actual SIMD efficiency is approximately 2.7$\times$ higher than a naive assumption of equal branch probabilities in the code-path may suggest. These results are explained in greater detail in Section 3.4.

The rest of this chapter is organized as follows: Section 3.1 describes how Memcached was ported to the GPU, Section 3.2 presents the GPU control-flow

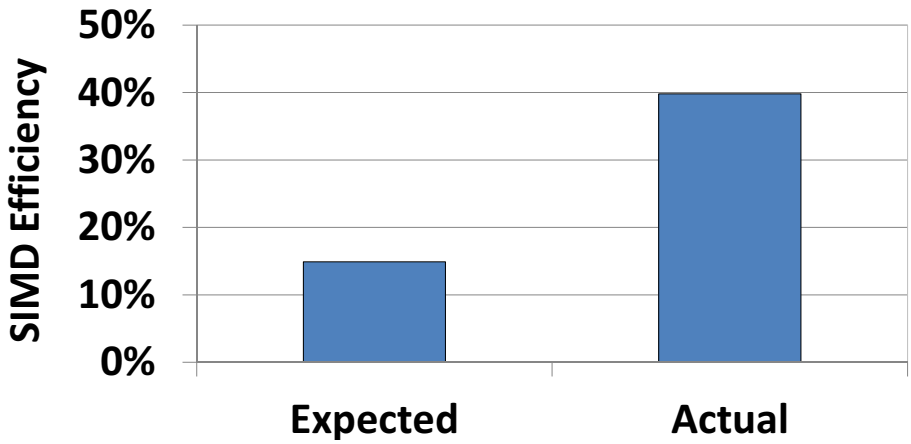


Figure 3.2: Memcached SIMD efficiency: Expected vs. Actual.

simulator, CFG-Sim, Section 3.3 describes the methodology and environment used to perform this study, Section 3.4 presents a characterization and evaluation of Memcached on hardware and GPGPU-Sim, and Section 3.5 summarizes this chapter.

3.1 Porting Memcached

This section describes the relevant implementation details of Memcached, design decisions, and modifications made to Memcached to offload the read (*GET*) request handler to the GPU. This section also describes the corresponding changes made to the host (CPU) code to efficiently interact and communicate with the GPU, as well as update entries in the hash table (*SET*s).

3.1.1 Offloading *GET* Requests

In our GPU implementation of Memcached, we focused on accelerating the read requests on the GPU while leaving the write requests to be handled by the CPU. Berezecki et al. [24] observe that read requests far outnumber write requests in real-world scenarios running Memcached in Facebook. The large composition of read requests in the total network traffic indicates high temporal locality, which is a key component in achieving the most out of the caching tier. Since Memcached

acts as a look-aside cache, write requests occur when a read request misses in the cache. Berezecki et al. also conducted experiments showing that write requests have negligible effects on read performance. From this, it is reasonable to assume that the read requests will have the most significant impact on total performance, and thus have the greatest benefit – in terms of overall system performance – from being accelerated on the GPU. Note that by partitioning the application between the CPU and GPU, special care must be taken to ensure that common data structures accessed by both CPU and GPU are kept coherent.

There are multiple levels of computation required for the end-to-end (receive-to-send) processing of a Memcached request. Before the user-level Memcached application receives the packet, the network request is first received by the network driver and processed by the operating system (OS). The OS processing consists of buffer management, network protocol handling (e.g., TCP or UDP), and the delivery of the packet data to the corresponding user-level application. The user-level Memcached application then decodes and performs the appropriate Memcached operation. Finally, a new network packet is constructed to send the response back to the initiator of the Memcached request through the OS and network driver. In this chapter, we focus only on the specific user-level application processing required to handle the Memcached request.

To take advantage of the massive amounts of available parallelism provided by the GPU, we exploit request-level parallelism in Memcached requests by assigning each GPU work item to process a single *GET* request. *GET* requests are batched into groups of requests on the host (CPU), transferred to the device on a kernel launch, and processed in parallel on the device (GPU). Batching requests results in a trade-off between throughput and latency. Requests are queued until a configurable number of requests has been batched together, which increases the latency per-request. However, the GPU's throughput-oriented architecture provides the potential to processes more requests in parallel within a given amount of time than the CPU can process sequentially, increasing the request throughput. However, this could have a significant impact on request latency under low traffic loads to the servers. A simple solution is to add a timeout, such that no request remains queued in a pending batch over a certain amount of time. This timeout value could be statically configured or dynamically determined based on the cur-

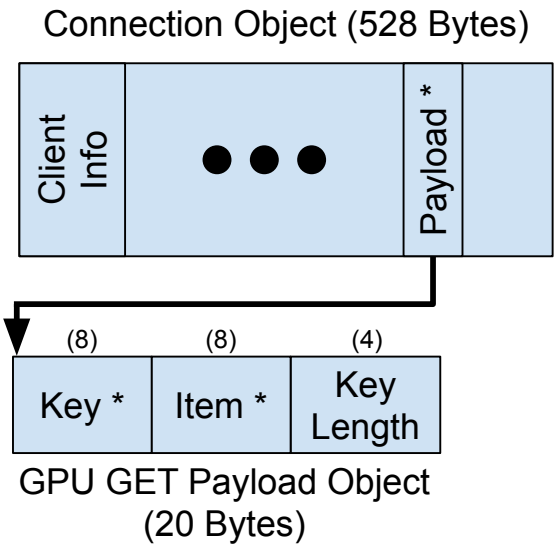


Figure 3.3: GPU *GET* Payload object. The original Memcached Connection object contains a large amount of information about the current Memcached connect, the requesting client network information, and the current state of the Memcached request. The GPU Payload object contains a much smaller subset of the relevant information required to process the *GET* request on the GPU.

rent traffic rates.

For each *GET* request, the work item performs common key-value look-up operations. These operations consist of computing the hash of the request's key, accessing the appropriate entry in the hash table using the resulting hash value, and comparing the request's key with the key – or multiple keys in the event of hash collisions – residing at that hash table location. If the keys match, the value corresponding to that key is returned to the requesting client. In this work, we assume that the requests have already been directed to the correct Memcached server, and thus the hash performed on the GPU corresponds to the second hash mentioned in Section 2.2.

When Memcached receives a request from a client via the network, it first creates a *connection* object that contains all information required to process any requests during the lifetime of this connection. The client is then able to send

requests through this connection to be handled by the Memcached server. These connections contain significant overhead when considering the amount of information required to process the actual Memcached request on the GPU, such as the network/Memcached protocol used and client information. To reduce the amount of data being sent to the GPU, we created a Memcached request *payload* data object that contains a subset of the connection information required to process a *GET* request, such as information about the search key and a pointer to the corresponding item if found. The Memcached *GET* payload object is shown in Figure 3.3. The original Memcached connection object is 528 Bytes, whereas the *GET* payload object is only 20 Bytes, which significantly reduces the amount of data to transfer on each *GET* request batch. On each *GET* request, we allocate and assign a payload object to the requesting connection and batch these payload objects to be transferred to the GPU.

3.1.2 Memory Management

To manage the data allocated and accessed in Memcached, we implemented a dynamic memory manager on the host CPU. This memory manager is used to store all of the data that needs to be visible to both the host and the device; it replaces the *malloc* and *free* system calls originally used in Memcached for any shared data structures with custom memory allocation calls. Depending on the system being used (discrete or integrated GPU), the allocated buffers reside in different memory regions on the host or device. On the discrete system, the buffers are allocated in the regular host memory space and transferred to the device when necessary. On the integrated AMD Fusion systems, however, these buffers are allocated in pinned memory to take advantage of the zero-copy memory regions, where data can be allocated on either the CPU or the GPU and accessed directly by both with varying bandwidth and latencies, as described below.

There are two types of zero-copy memory spaces available: the host-visible device memory and the device-visible host memory. As the names suggest, each memory space is targeted towards a specific device, but accessible from both. In either case, both memory spaces are allocated from pinned host memory, a subset of the host's memory space, at system boot time. Pinned memory locks pages

in physical memory, preventing the pages from being swapped out. The device-visible host memory is optimized for access by the host, whereas the host-visible device memory is optimized for the device [9]. The device-visible host memory is cacheable on the host, which enables the host to access the memory at its full bandwidth. However, this limits the device's memory bandwidth as all memory requests must go through the host's cache coherency protocol. The host-visible device memory, on the other hand, is a subset of the pinned host memory that is un-cacheable on the host. As this memory region is not bound by any coherency protocols, it can be directly accessed by the device at its full bandwidth. However, this limits the host's bandwidth to access the shared memory. As our main goal is to accelerate *GET* requests on the GPU, we used the host-visible device memory to minimize the data access time on the GPU.

3.1.3 Separate CPU-GPU Address Space

At the time of this study, while current AMD fusion hardware shared a physical memory region between the host and device, it did not share a common unified address space. The implication of this is that the virtual addresses returned by our custom *mem_alloc* function on the host, corresponding to the physical location in host-visible device memory, is not the same as the virtual address seen on the device, even though it corresponds to the same physical location. Thus, complex data structures consisting of many multiple-level pointers can not simply be de-referenced on the device.

What is common between the host and device when allocating memory, however, is the offset of each memory object from the start of the allocated memory region. For example, a memory object that is 32KB from the start of the memory region seen on the host is also 32KB from the start of the memory region seen on the device. Using this property, we pass the virtual address pointing to the start of the memory region seen by the host as an argument to the Memcached kernel and calculate the offset between this host memory buffer and the start of the memory region seen by the device. A macro is used to subtract this offset from every memory de-reference on the device: $translate(address, offset)$. The inverse operation is applied to all pointers set on the device, $inverse\_translate(gpu\_address, offset)$,

such as the return value for the corresponding item pointer on a *GET* request. This ensures that both the host and device access the same physical memory locations. Gelado et al. [60] implement a similar technique by either ensuring that the virtual pointers returned to the shared memory region are the same or by maintaining address mappings between the host and device. Using this offset-based address translation technique reduces the complexity for programmers by eliminating the need to traverse and reconstruct data structures that contain multiple nested pointers on the device, such as linked lists or tree structures.

3.1.4 Read-only Data

With the exception of the Memcached data structures written to with the results of the *GET* requests, such as the *GET* request payload objects, the majority of data structures between successive kernel launches are read-only when processing *GET* requests in Memcached. The AMD hardware evaluated in this chapter provides various hardware components, such as read-only caches, that can significantly decrease data access time. Where possible, we allocate data in read-only buffers to take advantage of the buffer's high-bandwidth, low-latency memory accesses.

3.1.5 Memory Layout

Using our custom dynamic memory manager, we can allocate data in specific layouts to take advantage of the GPU's memory-coalescing property discussed in Section 2.1.8. Two data structures that are guaranteed to be accessed by all work items with known access patterns on the GPU are the payloads and the keys corresponding to each payload. As introduced in Section 3.1.1, the payload contains a pointer to the request key, the length of the key, and a pointer to the item being requested. Each work item is assigned a single payload corresponding to a single *GET* request. Depending on the architecture, 32-bit or 64-bit, the size of each payload is only 12 bytes or 20 bytes respectively. With a wavefront size of 64 work items and a cache line size of 128 bytes, these 64 memory requests could be reduced to six or ten memory requests respectively to retrieve the same amount of data. Therefore, we ensure that the *GET* request payload objects are allocated contiguously in memory by allocating them in a separate dedicated buffer.

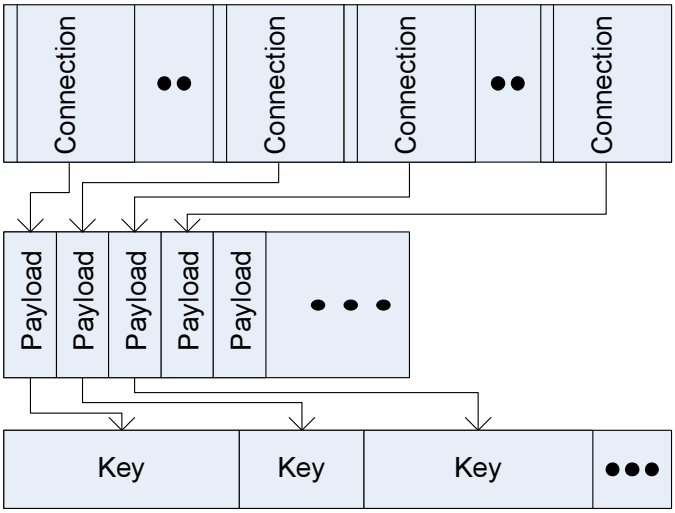


Figure 3.4: Contiguous memory layout.

To access the payloads, each work item requires only a pointer to the start of the payload buffer and uses its global work item ID to access the appropriate index in the payload array. This same technique is applied to the keys corresponding to the payloads, such that when each work item dereferences the pointer to the key, the keys will reside nearby in memory. Figure 3.4 shows how the connections, payloads, and keys are allocated and laid out in the different memory buffers.

3.1.6 *SET*s and *GET*s

As described in Section 3.1.1, this work focusses on accelerating *GET* requests on the GPU, while leaving *SET* requests to be processed on the CPU. This results in a requirement for synchronization between the CPU and GPU to ensure coherent accesses to the shared resources between the two devices. The purpose of this chapter is to understand the potential for the GPU to accelerate workloads that exhibit irregular behaviour. As such, evaluating the impacts of fine-grained synchronization between accesses and updates to the shared data structures is beyond the scope of this chapter and is considered in Chapter 4. Instead, we implement a coarse-grained synchronization in which *SET*s and *GET*s are batched separately and processed sequentially and independently. For example, as requests arrive at

the GPU Memcached server, *GET* requests are inserted into a pending batch and *SET* requests are processed immediately. As long as no *GET* request batch is currently being processed on the GPU, *SET* requests are able to proceed without requiring synchronization with the GPU. However, once a batch of *GET* requests is launched on the GPU, all *SET* requests are blocked until the *GET* request batch has completed processing on the GPU. This ensures that the data is not being updated while a *GET* batch is being concurrently processed on the GPU.

However, this coarse-grained synchronization has two limitations: (1) unnecessary blocking of *SET* requests that may be independent of all concurrent *GET* requests and (2) out-of-order execution of *SET* and *GET* requests. For (1), *SET* requests may be able to tolerate longer latencies than *GET* requests, since the requesting application does not depend on the result of the *SET* to continue making progress. For (2), Memcached has weak request ordering requirements. In the baseline Memcached, requests may be handled by multiple different CPU threads. Each thread acquires locks on shared resources, such as the hash table, prior to accessing or updating the resource. As such, it is possible for *SET* and *GET* requests to be reordered in the baseline Memcached between the time when the request was received at the server and when the request was processed, depending on which thread acquires the lock first. Consequently, the client application cannot rely on a strict ordering of Memcached requests. While our implementation does not break a requirement on request ordering, it does increase the chance of reordering requests.

3.2 Control-Flow Simulator (CFG-Sim)

Along with evaluating the behavior of Memcached on GPUs, we are also interested in understanding (1) how a programmer's intuition about how an application might perform on a GPU compares with how the application actually performs on a GPU, and (2) what are the effects of branch probabilities relative to correlated branch outcomes (described below) within GPU work items in a wavefront. To address these questions, we look to analyzing an application's control flow behavior.

We designed a stand-alone control-flow simulator, CFG-Sim, that simulates the behavior of a wavefront through an application's control-flow graph (CFG). The CFG can be generated either from a CPU version or an existing GPU version

of the application. Each branch in the application's CFG is annotated with an outcome probability. The branch probabilities can be set randomly, set based on intuition about the program's behavior, or set from actual branch probabilities extracted from profiling the application. At each branch, the simulated active work items generate a random number and compare it with the threshold outcome probability at that branch to determine if the branch will be taken or not. A SIMT stack (Section 2.1.8) handles tracking branch divergence, re-convergence, and deciding which work items to execute at a given time. The overall SIMD efficiency is measured for each iteration through the application's CFG and averaged across multiple iterations, as shown in Equation 3.1. In this equation, $E(SIMD_{Eff})$ is the expected SIMD efficiency returned by CFG-Sim; N is the total number of iterations to perform through the application's CFG; CF_j is the set of basic blocks that were traversed by CFG-Sim on the j^{th} iteration through the application's CFG; $\#BB_i$ is the number of instructions in basic block i; am_i is the number of work items active at basic block i calculated by CFG-Sim; and finally WF_{size} is the total number of work items in a wavefront.

$$E(SIMD_{Eff}) = \frac{1}{N}\sum_{j=1}^{N}\frac{\sum_{i\in CF_j}(\#BB_i \times \frac{am_i}{WF_{size}})}{\sum_{i\in CF_j}\#BB_i} \tag{3.1}$$

An estimate of the expected SIMD efficiency from CFG-Sim can be useful when performing an initial analysis to decide whether an application may benefit from GPU acceleration. Prior to writing any code for the GPU, a programmer can gain better insight into the average SIMD efficiency estimated to result on the hardware. CFG-Sim takes three input files: a *DOT* file [58] containing the information required to generate the control-flow graph, a file containing the number of instructions per basic block (specified in the DOT file format), and a file containing the estimated outcome probabilities for each branch in the application.

However, the branch probabilities themselves may not be enough to accurately simulate the SIMD efficiency of an application. Consider an application that is already ported to run on a GPU. After profiling the execution, it is possible to have

a SIMD efficiency of 100% with a branch outcome probability of 50% for every branch. This occurs if every work item in a wavefront takes the same path through the CFG, but alternates between different paths every execution. We refer to this property where work items in a wavefront move through the CFG together as *correlated branches*. Correlated branches may occur because the application was programmed for the GPU to maximize SIMD efficiency or because of different phases during the application's execution. This correlation among branches is currently not accounted for in CFG-Sim and is left to future work (Section 7.2.1). Despite this, by extracting Memcached's actual branch probabilities from a cycle accurate GPGPU Simulator, GPGPU-Sim (described in Section 3.3), and using them as input to CFG-Sim, we found that the estimated SIMD efficiency is within 1.3% of the actual SIMD efficiency of Memcached measured by running the application through GPGPU-Sim. One such application that requires modelling branch correlation to accurately estimate the SIMD efficiency is Ray Tracer, which is discussed in Section 3.4.2.

3.3 Experimental Methodology

This section describes the experimental methodology followed in this chapter.

3.3.1 Hardware and Simulation Frameworks

We performed experiments on three configurations of GPUs and accelerated processing units (APUs): a high-performance discrete graphics card (AMD Radeon HD 5870), a low-power AMD Fusion APU (Zacate E-350 [AMD Radeon HD 6310]), and a mid-to-high-end AMD Fusion APU (Llano A8-3850 [AMD Radeon HD 6550D]). The terms APU and integrated GPU are used interchangeably. The discrete GPU is connected to the host CPU via PCIe and has a physically separate graphics memory from the CPU's main memory. The integrated GPUs are integrated on the same die as the host CPU and share a common memory. The discrete GPU was chosen to show potential upper bounds on compute performance, while the low-power AMD Fusion APU provides insight into the performance capabilities of such integrated systems with shared memory. The mid-to-high-end AMD Fusion APU falls in the middle of these two systems, combining the higher

Table 3.1: GPU hardware specifications.

Name	**AMD Radeon HD 5870**	**Llano A8-3850 (AMD Radeon HD 6550D) AMD Fusion**	**Zacate E-350 (AMD Radeon HD 6310) AMD Fusion**
Engine Speed (MHz)	850	600	492
# Compute Units (CU)	20	5	2
# Stream Cores	320	80	16
# Processing Elements	1,600	400	80
Peak Gflops (single-precision)	2,720	480	78.72
# of Vector Registers/CU	16,384	16,384	16,384
LDS Size/CU (kB)	32	32	32
Constant Cache / GPU (kB)	48	16	4
L1 Cache / CU (kB)	8	8	8
L2 Cache / GPU (kB)	512	128	64
DRAM Bandwidth (GB/sec)	153.6	29.9	17.1
DDR3 Memory Speed (MHz)	1,000	933	533

Table 3.2: CPU hardware specifications.

Name	Llano A8-3850	Zacate E-350
# x86 Cores	4	2
CPU Clock	2.9 GHz	1.6 GHz
TDP	100W	18W
L2 $ / core	1MB	512 KB

compute performance with the benefits of the APU's shared memory space. The hardware specifications for these GPUs are outlined in Table 3.1. Table 3.2 provides additional information about the CPUs used in this study.

At the time of this study, the Linux drivers for the AMD Fusion system did not support zero-copy buffers. To access the Windows AMD SDK and the required Linux libraries for Memcached, we used Cygwin [28] to run Memcached on the AMD Fusion systems. One issue with Cygwin is its inability to access all provided GPU hardware counters. This significantly limited the amount and variety of data

we were able to collect from Memcached on the hardware. To gain additional information about Memcached's behavior on a GPU, we profiled Memcached on GPGPU-Sim [17], a cycle-accurate GPGPU Simulator.

Although the GPU architecture modeled by GPGPU-Sim differs from the physical hardware analyzed at the time this study was performed (Table 3.1) [1], such as in its use of a VLIW unit, we do not believe that this is an issue because the architecture resembles AMD's future GPU architecture [74]. Furthermore, many of the properties evaluated in this study, such as SIMD efficiency, memory divergence, and cache sensitivity, are relevant to any GPU architecture with a SIMD pipeline, data caches, and the ability to coalesce memory accesses. GPGPU-Sim simulates Parallel Thread Execution (PTX) code, a pseudo-assembly intermediate language used in NVIDIA GPUs. Table 3.3 presents the configurations used in GPGPU-Sim.

Control-Flow Simulator

In the current version of the control-flow simulator (CFG-Sim), we assume that the outcome of each work item's branches are independent of any other work items within the same wavefront. As such, we do not consider correlated branch outcomes between work items in a wavefront. Additionally, loops are treated as regular branches, which can impact the estimated SIMD efficiency. For example, a loop may be known to be taken 100 times before work items begin exiting the loop. However, setting the branch probability to $p(\frac{1}{100})$ does not result in the same branch outcome distribution as having both a fixed and probabilistic portion to loop branches. While CFG-Sim could be modified to include optimizations such as loop branches, it would require additional information about the behaviour of the application obtained through profiling or static analysis.

For a set of given branch probabilities, we ensure that the SIMD efficiency results converge by averaging 100K iterations through the application's control-flow graph.

[1] GPGPU-Sim models NVIDIA-like GPUs whereas this study evaluates AMD GPUs.

Table 3.3: GPGPU-Sim configuration.

Config Name	Config Value
# Streaming Multiprocessors	30
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	32KB, 64B line, 16-way assoc.
Number of Memory Channels	8
L1 Data Cache	32KB, 128B line, 8-way assoc.
L2 Unified Cache	512k, 128B line, 8-way assoc.
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
DRAM request queue capacity	32
Memory Controller	Out of Order (FR-FCFS)
Branch Divergence Method	PDOM [172]
Warp Scheduling Policy	Loose Round Robin
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8
Memory Channel BW	8 (Bytes/Cycle)

3.3.2 Assumptions and Known Limitations

Throughout this study, we assume requests are independent of each other. Thus, all *GET* (read) operations will view the most up-to-date data in Memcached. As discussed in Section 3.1.6, *SET* and *GET* requests may complete out-of-order with higher probability than the baseline Memcached, due to the batching of *GET* requests.

The size of memory accessible by the AMD GPUs and APUs evaluated in this study is limited. On the APU, each zero-copy buffer can be a maximum of 64 MB, with a system total of 128 MB [9]. This poses various problems for memory-intensive applications, such as Memcached, that require large amounts of memory

to be effective. This problem would be eliminated with a larger region of pinned memory available to the GPU and an appropriate interface to allocate and access the additional memory. Indeed, the industry has addressed the limited memory capacities available in previous graphics cards. For example, at the time this study was performed, AMD had already announced at the 2011 AMD Fusion Developer Summit (AFDS) that future AMD GPUs and APUs will support accessing CPU virtual memory [42]. The current HSA specification [55] requires that compliant HSA systems support access to shared system memory across devices through a unified virtual address space. Current NVIDIA GPUs [131] also support unified virtual memory, which enables transparent access for the same virtual address on both the CPU and GPU, as well as virtual memory paging from CPU memory to GPU memory.

Batching requests inherently increases the latency to process the requests. Offloading requests to the GPU in batches can help to reduce the total system-queuing latency if the CPU throughput becomes the bottleneck when experiencing high incoming request rates. Assuming the GPU can process requests with higher throughput than the CPU, the GPU would be able to drain the pending request queues faster than the CPU. While some applications may not be able to tolerate the increased latency impact of batching large numbers of requests (as is done in this study), we expect we could achieve many of the benefits while using smaller batch sizes. Chapter 4 evaluates batching fewer requests at a time to reduce the queuing latency, while launching multiple batches concurrently to maintain high throughput.

We initially profiled Memcached to locate sections of code that contribute to the majority of Memcached *GET* request processing and would benefit from running in parallel on the GPU. This revealed that a majority of execution time for *GET* requests is actually spent in I/O and network stack processing. The key-value lookup is the next-highest contributor to the overall execution time of Memcached. Thus, in this chapter we focused our efforts on porting the key-value lookup handler to the GPU. Chapter 4 evaluates offloading portions of the network stack to the GPU as well.

3.3.3 Validation and Metrics

To verify that the GPU version of Memcached returns the correct results, we first process a batch of *GET* requests on the CPU using the baseline version of Memcached, and then process the same batch of *GET* requests on the GPU. On completion of the GPU kernel, we compare the results from the CPU and GPU to ensure the correct items were found.

The execution times on the CPU were recorded using a fine-grained time stamp counter (TSC) that records the sequential look-up times for the batch of requests. When timing the CPU, all data was allocated in a cacheable memory region. For each GPU execution, we recorded the kernel execution times using the AMD APP Profiler tool (v. 2.3). We also verified that the comparison with the TSC timer was valid by timing the kernel execution time on the Llano A8-3850 system immediately before the kernel launch and immediately after the $clFinish$ synchronization function.

3.3.4 WikiData Workload

We simulated request traffic to our GPU Memcached server using a large input file consisting of HTTP read and write requests. Specifically, we used portions of Wikipedia workload traces [169], referred to as WikiData, to stimulate our application. These workload traces were recorded by Wikipedia's front-end proxy caches and, in total, contain billions of HTTP requests.

Memcached's front-end host code was modified to process the requests from the WikiData trace files instead of processing incoming requests from the network. The HTTP read and write requests are converted into Memcached *GET* and *SET* requests respectively. A configurable number of requests are processed on the CPU prior to offloading work to the GPU to set up the environment and to warm up Memcached's hash table and memory slabs. Once the setup phase completes, *SET* requests are handled immediately on the host and *GET* requests are placed into a buffer until the configurable number of *GET* requests have been received. Once a *GET* request batch has been launched on the GPU, *SET* requests are stalled until the *GET* request batch completes.

3.4 Experimental Results

This section presents our experimental results evaluating Memcached on both integrated and discrete GPUs, and a software GPU simulator. We first present the hardware results, then discuss the SIMD efficiency analysis using CFG-Sim, and finally present an evaluation of Memcached on GPGPU-Sim.

3.4.1 Hardware Evaluation

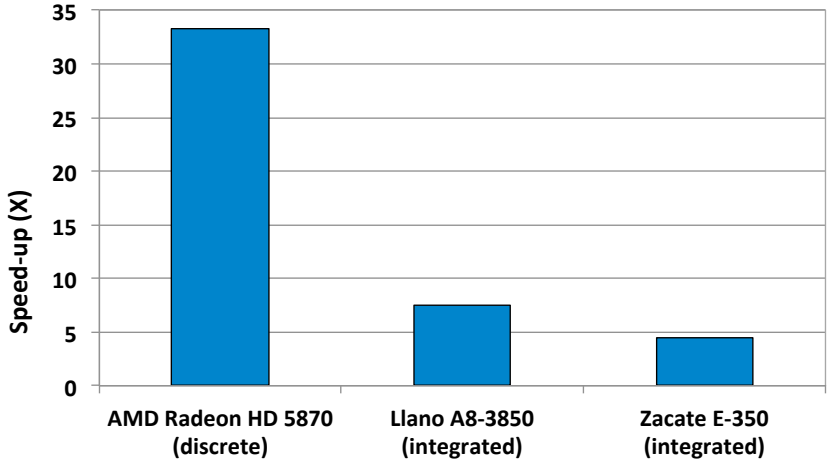
Memcached was run on the three GPU configurations introduced in the previous section, with the performance measured via hardware counters through AMD App Profiler. Both the AMD Radeon HD 5870 and the Llano A8-3850 (AMD Radeon 6550D) are compared against a single Llano x86 CPU core, while the Zacate E-350 system was compared against a single Zacate x86 CPU core.

We also evaluated Memcached's key-value look-up performance on a multi-threaded CPU implementation (using Linux pthreads). As shown in Table 3.2, the Llano and Zacate systems contain four and two cores respectively. In this experiment, we evenly partitioned the *GET* request batch between one, two, and four threads. Each thread performed the same key-value look-up operations on a different sub-batch of *GET* requests in parallel, and the time was measured for the last thread to complete its portion of the requests. From this experiment, we found that two threads had approximately the same throughput as a single thread, and four threads reduced the performance relative to a single thread, even on the 4-core Llano. As described in Section 3.3, this work was evaluated on Windows using Cygwin. Consequently, the lack of multi-threaded performance improvements may be attributed to Cygwin's pthread support. Additionally, Memcached's CPU implementation contains global locks, for example, to protect access to the hash table entries and the global LRU queue for managing item evictions, which serialize portions of the key-value look-up operation between *GET* requests, reducing the potential for linear speed-ups with additional cores. For these reasons, our results in this Chapter compare the GPU performance against a single CPU thread, which was the highest measured throughput on the CPU. An ideal multi-threaded CPU performance can be estimated by multiplying the measured throughputs by the number of cores on the corresponding system. However, as shown below, the

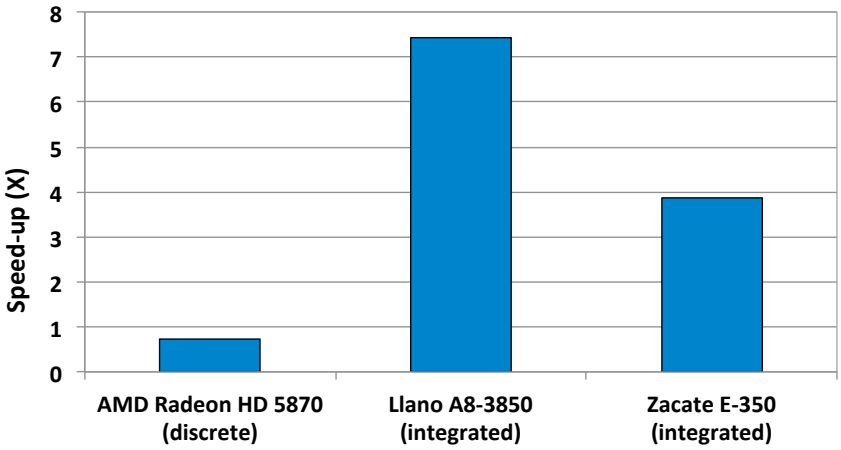
GPU is still able to outperform the estimated ideal CPU throughputs. Furthermore, the next chapter (Section 4.4.2) evaluates end-to-end multi-threaded Memcached CPU performance, which shows approximately a 2× increase in throughput when increasing from one to four threads and a steep drop-off in throughput after adding more threads than the number of cores. Hence, we believe that the GPU results presented in this section are representative of the GPU's ability to outperform the CPU-version of Memcached.

Figure 3.5a presents the average speed-up, in terms of key-value look-ups per second (LPS), for each GPU configuration normalized to the CPU's execution time. Because these results do not include any data transfer times, they explicitly highlight the computational performance benefits when performing Memcached's key-value look-up on the GPU relative to the CPU. Even with the potential for irregular control-flow and memory-access patterns present in Memcached, the AMD Radeon HD 5870 is able to perform the key-value look-up on a batch of 38, 400 *GET* requests approximately 33× faster, the Llano A8-3850 7.5× faster, and the Zacate E-350 4.5× faster than their CPU counterparts. We hypothesize that the performance improvements are largely a result of the increased computational and memory-level parallelism. As will be discussed further in Section 3.4.2, the SIMD efficiency of our Memcached implementation is approximately 40%, which indicates that a non-negligible number of the *GET* requests' key-value lookup instructions are executing in parallel. Furthermore, because Memcached is a highly memory-intensive application, the GPU's high memory-level parallelism enables multiple in-flight memory requests from different work items to be processed concurrently. While CPU threads must wait for a memory request to return from the cache or memory before continuing to operate on the requested data, the GPU can simply switch to another wavefront to perform any computations or initiate additional memory requests.

However, the results in Figure 3.5a ignore any data transfer times, which need to be considered when measuring the full end-to-end performance of a GPU application, especially for a streaming application such as Memcached. Including the data transfer times results in a large overall performance decrease on the discrete system, as can be seen in Figure 3.5b. The execution time is measured from immediately before the data transfer to the GPU is initiated to immediately after the



(a) Excluding data transfers.



(b) Including data transfers.

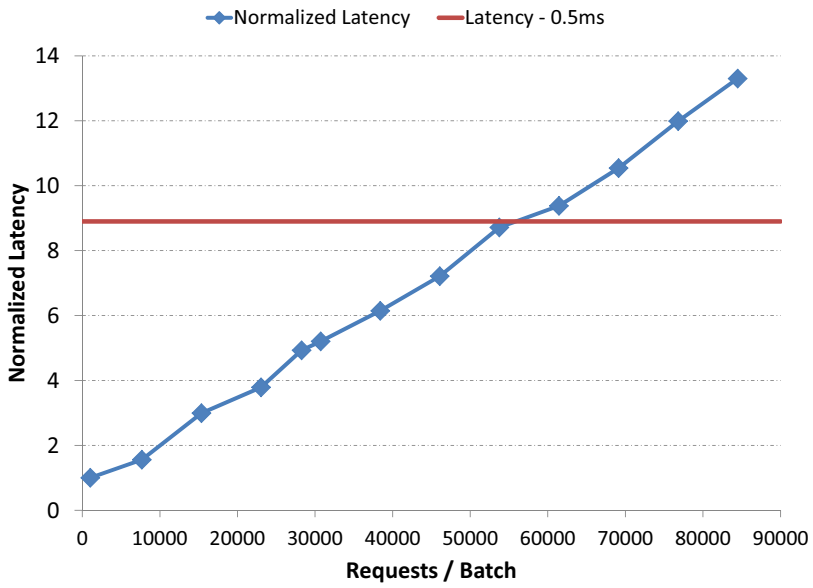
Figure 3.5: Memcached speed-up vs. a single core CPU on the discrete and integrated GPU architectures. Each batch of *GET* requests contained 38,400 requests/batch.

data transfer from the GPU is completed. Overlapping the data transfers with computation from another batch of *GET* requests can reduce the overall data transfer

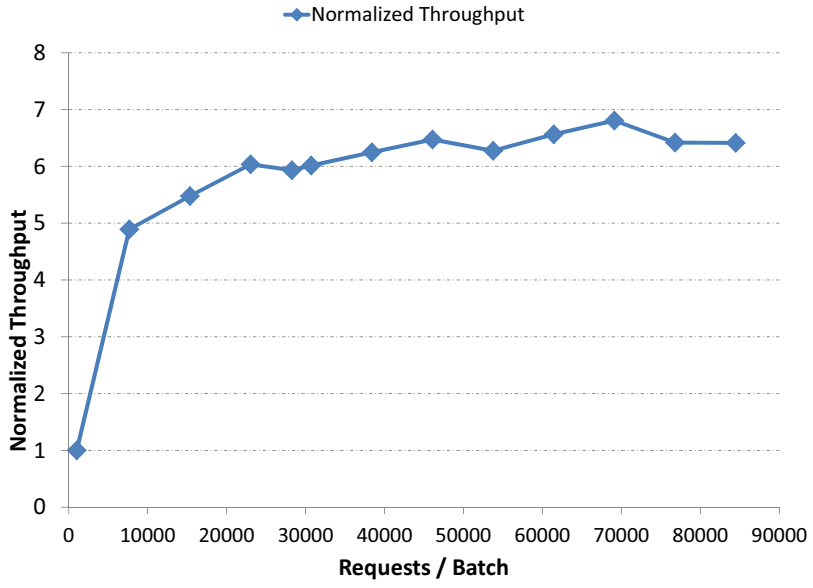
overheads, however, the latency for the request batch is still negatively impacted. Additionally, given the large request batch sizes (38,400 requests/batch), the incoming request rate would need to be high enough to have another batch of *GET* requests available once the previous request batches have completed transferring data and have started processing. The APUs have close to zero transfer time due to the shared memory space. These data transfer times are small, but non-zero, due to the mapping and un-mapping operations, which are required to ensure that the data the GPU is accessing is coherent with any CPU updates. Although the total compute power of the APUs is far less than the high-performance discrete AMD Radeon GPU, the ability to fully eliminate the transfer of data allows these devices to outperform the AMD Radeon HD 5870. Further optimizations to reduce the amount of data transferred to and from the GPU for each *GET* request are required to reclaim some of the computational performance benefits of the discrete GPUs. This is discussed further in Chapter 4.

Request Batching

Whenever considering batch processing, there is always a trade-off between throughput and latency. Specifically, as the number of queued requests increases, the time taken to process these requests also increases. The reason is two-fold. First, the requests sit in the batch queue for a longer period of time until processing begins. Second, the GPU takes longer to process a larger batch. With more requests to process concurrently, more active wavefronts are executing concurrently. This places higher contention on shared resources, such as the compute units and the memory system. While this increases the total system throughput, it also increases the per-work item processing latency. We measured the impact on request throughput and latency on the AMD Radeon HD 5870 by varying the batch request size and recording the average time taken to process that batch of requests. Figure 3.6 presents these results normalized to an initial batch size of 1,024 requests, excluding data transfer times. Also shown in Figure 3.6a is a 0.5ms latency reference line. Berezecki et al. [24] indicate that a 1ms delay for a complete Memcached request processing, including network transfer and network processing time, is a reasonable maximum tolerable latency. The data shown



(a) Normalized latency.



(b) Normalized throughput.

Figure 3.6: Throughput and latency while varying the request batch size on the AMD Radeon HD 5870 (normalized to 1,024 requests/batch).

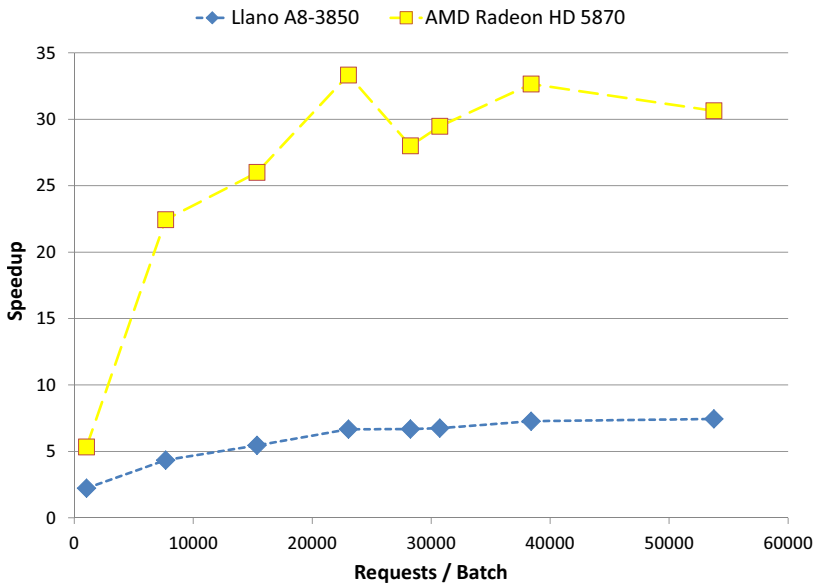


Figure 3.7: Speed-up of AMD Radeon HD 5870 and Llano A8-3850 vs. the Llano A8-3850 CPU at different request batch sizes.

in Figure 3.6 measures only the Memcached key-value lookup time. The large spike in throughput from 1-5× in Figure 3.6b is caused by the minimal increase in latency, approximately 1.3×, while increasing the number of requests/batch by 7.5× (1,024 → 7,680). This results in the behavior shown by the throughput, corresponding to the initial increase in latency, which begins to level off and fluctuates between 6× and 7× the throughput at 1,024 requests per batch.

This behaviour suggests that the GPU is underutilized when the request batch size is less than approximately 20,000. Assuming that a theoretical incoming request rate can be set to match any level of throughput, selecting a batch size of approximately 8,000 requests per batch qualitatively provides the maximum ratio of throughput to latency, whereas a batch size around 30,000 requests achieves most of the peak measured throughput. As will be discussed in Chapter 4, multiple smaller request batches can also be executed concurrently to improve the balance of throughput and latency.

Another property of batch processing to consider is how the performance between the GPU and CPU varies when the request batch size is increased. These

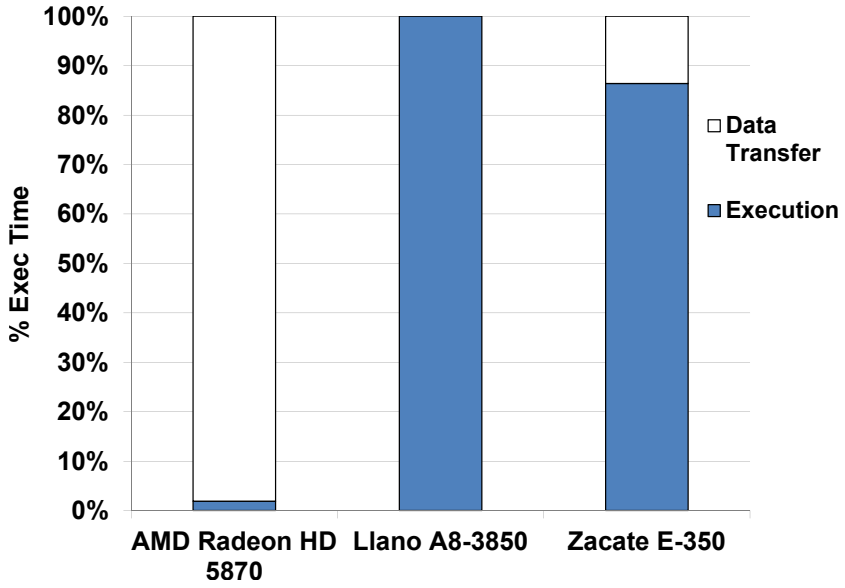


Figure 3.8: Memcached overall execution breakdown (23,040 requests/-batch).

results are presented in Figure 3.7 for both the AMD Radeon HD 5870 and the Llano A8-3850 when compared to a single CPU core on the Llano A8-3850 system. Both architectures show a large initial increase in performance when the batch size is increased from small values. Similar to the throughput behavior seen in Figure 3.6b, the speed-up compared to the CPU begins to level out on both architectures around 40,000 requests/batch. Again, these results are excluding data transfer times. Including the data transfers results in the integrated Llano GPU achieving higher overall performance relative to the CPU than the discrete Radeon GPU.

Data Transfer

In applications with large amounts of data needing to be transferred to and from the device, such as Memcached, transfer time can dominate the overall execution time of the kernel. Figure 3.8 shows the contribution of the execution time and data transfer times as a percentage of the overall execution time for each GPU when operating on a *GET* request batch size of 23,040 requests/batch. We optimisti-

cally selected the minimum amount of data that must be transferred to and from the device: the requests to be processed and the results of the requests respectively. Assuming cyclic [2] transfer of data, more than 98% of the overall execution time is spent transferring data for the discrete AMD Radeon HD 5870. These values were recorded assuming that none of the data could have been modified on the host between successive kernel launches, thus ensuring all data in the device memory is valid. Therefore, on kernel launch, the only data that must be transferred are the requests themselves; upon completion of the kernel, all of the results must be transferred back to the host. A more realistic assumption is that an unknown amount of data could have been modified between kernel launches, thus invalidating a portion of the data on the device and requiring explicit tracking and transfers of the modified data on every kernel launch. Tracking which data was modified could be avoided by pessimistically transferring all of the data on every kernel launch, however, this cyclic memory transfer model that transfers data regardless of whether it has been modified is sub-optimal.

Others [21, 104, 161, 168] have proposed solutions to this challenge (e.g., implementing frameworks to automatically and acyclically transfer modified data to the device or requiring programmer annotation of the code to specify memory regions to be explicitly managed) to reduce the impact data transfers have on performance. With the introduction of CPU-GPU architectures that share a physical memory space, such as the AMD Fusion systems, this transfer time can be virtually eliminated. As can be seen in Figure 3.8, the majority of the overall execution time for the Llano A8-3850 and Zacate E-350 systems is spent performing useful work, rather than waiting for the data transfer to complete. Being able to reduce the time required to transfer data, either by using an architecture with a unified memory space or using a method to reduce the transfer overhead, is crucial when porting an application requiring large data transfers to the GPU.

3.4.2 Simulation Evaluation

This section attempts to gain additional insight into the performance of Memcached on a GPU using simulation frameworks. Unless otherwise stated, the data pre-

[2] Cyclic refers to transferring data before and after successive kernel launches, whereas acyclic data transfer overlaps data transfer with the kernel execution [161]

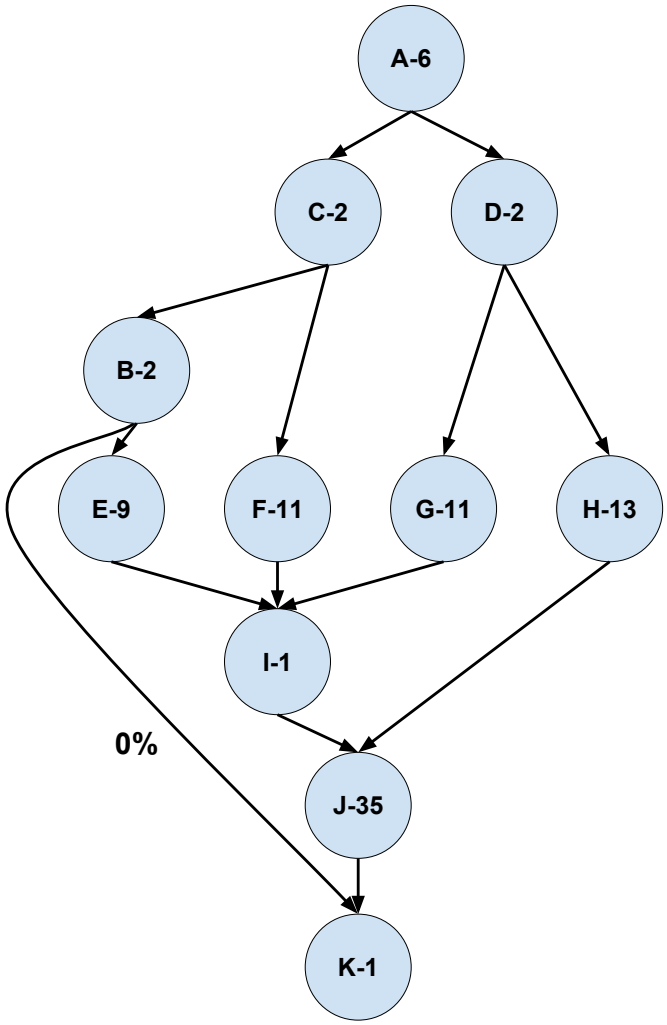


Figure 3.9: Example control-flow graph with an error handling branch from *B* to *K*. Each basic block contains the basic block identifier and the number of instructions in that basic block (*Basic Block ID-# Instructions*). All branches have a non-zero branch outcome probability except for *B* to *K*, which is never taken.

sented in this section is either collected from CFG-Sim discussed in Section 3.2 or the baseline GPGPU-Sim configuration presented in Section 3.3.

SIMD Efficiency of Memcached

As previously discussed, in single-instruction, multiple-data (SIMD) architectures, such as GPUs, groups of work items in a wavefront must execute instructions together in lock-step. If a work item branches away from the other work items in its wavefront, the GPU executes the two sub-groups separately, requiring more cycles than if they were executed together [9]. This reduces overall SIMD efficiency. Combining Memcached's complex control-flow graph, which contains multiple nested conditional branches, with the level of uncertainty in branch outcomes, one can reasonably expect Memcached to have poor SIMD efficiency, directly resulting in poor performance on the GPU. Although pessimistic, an initial view of the system might be that each branch outcome has an equal probability (50% not-taken and 50% taken). In many applications, this might be an unreasonable assumption; however, many of the branches in Memcached depend on input data, such as the length of the request key, that can vary greatly between requests. Without additional analysis, it is unclear to what extent these data dependent branches will impact branch divergence and, hence, SIMD efficiency. Assuming equal branch probability is marginally better than the worst case, where the thread groupings deterministically split in half at each branch.

After performing further analysis of the application, such as manually inspecting or profiling the application, certain branches may be reasoned to occur rarely or never (e.g., error handling or dead code). These branches can be removed from the analysis by forcing the work items to take a certain path (the path known to be always taken), since their inclusion would negatively bias the expected SIMD efficiency of the system. For example, consider the CFG in Figure 3.9 with a similar structure identified within Memcached. In this example, each ellipse contains the basic block identifier and the number of instructions in that basic block (*Basic Block-# Instructions*). Also note that the branch from *B* to *K* may be some error checking code, which should never, or rarely, occur during normal execution. There are two main issues here. First if we assume equal branch probabilities for all basic blocks, the estimated SIMD efficiency will be unnecessarily lower since work items will never actually diverge at *B*. Second, when work items diverge at *A*, their re-convergence point is set to the immediate post dominator (IPDOM) *K*,

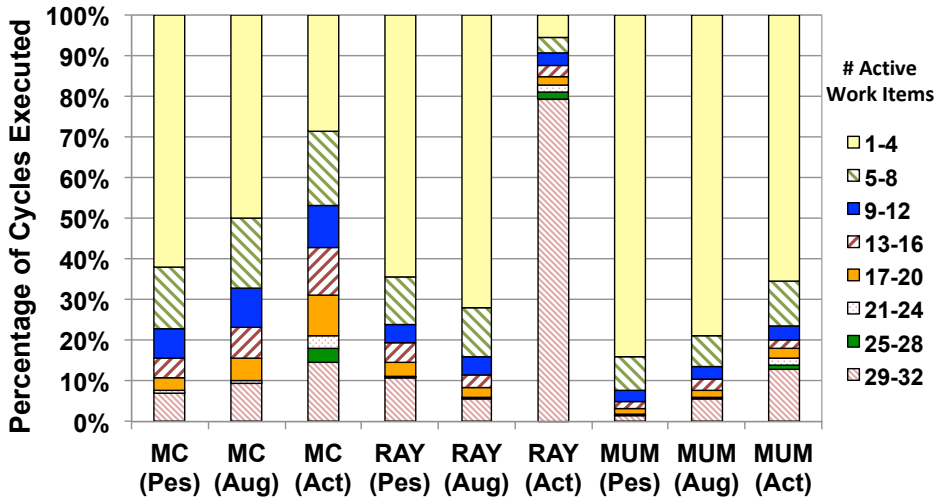


Figure 3.10: SIMD efficiency.

since this is the first basic block which all work items must execute. As a result, work items that diverge at *A* or *C* will all execute *J*, a large basic block, separately even though all work items will actually go through *J*, which further lowers the SIMD efficiency. The first issue can be solved by forcing all branches known to never be taken to have zero probability (see *Augmented* below), while the second issues requires either dynamically identifying early re-convergence points or removing the zero-branch from the CFG (left to future work).

Simulating the control-flow behavior of a single thread grouping with the control flow simulator (CFG-Sim) (Section 3.3), we can compare Memcached's SIMD efficiency with these initial views of how the application might behave on a GPU, resulting in the data in Figure 3.10. This figure compares the overall SIMD efficiency of Memcached's actual execution (*Act*) with the pessimistic view that all branches have equal outcome probabilities (*Pes*). In this experiment, there are 32 work items per wavefront. Each bin in the graph represents the fraction of total program execution in which the specified number of scalar threads were concurrently executing. For Memcached (MC), the SIMD efficiency with *Pes* is much lower than *Act*, where significantly more of the execution time contains only 1-4 work items in a wavefront.

We then improve on this pessimistic view by optimizing away all branch paths that are never taken during normal execution (Augmented - *Aug*) by setting the branch probability to 0%, and compare the recorded SIMD efficiency with the actual execution. All other branches remain at equal branch probabilities. Although there is an improvement, the SIMD efficiency of the actual execution of Memcached still outperforms the estimated behavior.

Using the actual branch probabilities measured from GPGPU-Sim for Memcached in CFG-Sim, instead of the equal branch probabilities, the estimated SIMD efficiency from CFG-Sim is within 1.3% of the actual SIMD efficiency, which highlights the potential for CFG-Sim to accurately estimate the SIMD efficiency.

We extend this analysis to applications known to perform well on the GPU, such as Mummer (MUM) and Raytracer (RAY), and measure how these results compare when similar assumptions are applied, which is also shown in Figure 3.10. Although MUM exhibits a relatively low SIMD efficiency, GPUs tend to have more memory bandwidth than CPUs, which can result in higher throughput on memory-limited applications even in the presence of significant control flow-divergence. This is a similar behaviour as measured in Memcached. However, the theoretical results in RAY perform significantly worse than the actual results. This is caused by high correlations between work items' branch outcomes within a wavefront. Although each branch outcome may have a relatively random probability, each work item is biased by the results of the other work item within the group.

The main takeaway from this experiment is that while an application may contain many data-dependent branches, which can lead to irregular control flow on a GPU, further analysis of the application is required to understand the actual attainable SIMD efficiency on a GPU, since branch outcomes can be far from random and branch outcomes may be correlated between work items.

Effect of Memcached on the Memory System

Memcached's key-value retrieval algorithm places a significant amount of stress on the memory system. Figure 3.11 shows the misses per 1,000 instructions (MPKI) for Memcached with a variety of L1 data cache configurations ranging from a small 8KB, 8-way set associative cache to a large 1MB full associative cache. This data

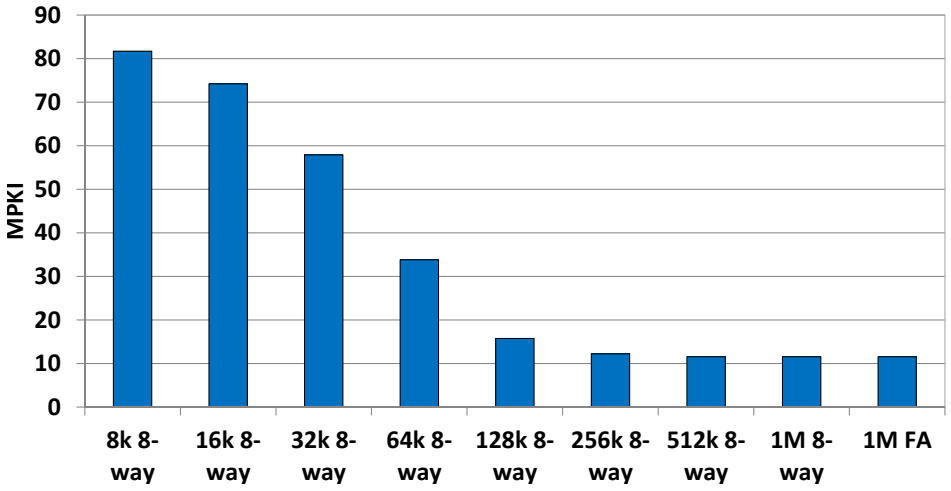


Figure 3.11: L1 data cache misses per 1,000 instructions at various configurations. FA = Fully Associative.

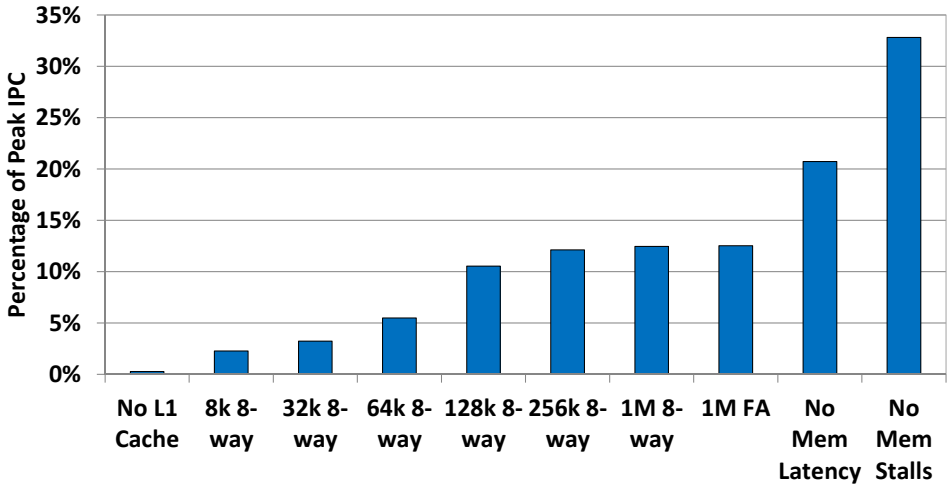


Figure 3.12: Performance as a percentage of peak IPC with various realistic L1 data cache configurations and two idealized memory systems.

shows that Memcached has some exploitable locality and that the working set of our simulated configuration fits in a 256KB cache. The remaining 10 MPKI are caused by cold start misses.

Figure 3.12 shows the performance of Memcached on GPGPU-Sim with a

number of L1 global data cache configurations and two variations of an idealized memory system. Performance is presented as a percentage of peak IPC (when every SIMD lane is active for every GPU cycle). Increasing the cache size results in a continuous performance improvement up to 256KB, beyond which it levels off. This result indicates that Memcached is a cache-sensitive workload. Further investigation of the source code reveals that two instructions receive a significant reduction in latency when cache size increases. These instructions are the loads performed inside the key comparison loop which compares the input key and a key found in the hashtable. This loop accesses memory sequentially, resulting in a high cache hit rate when the cache is large enough to capture the working set.

The 1MB fully associative (FA) configuration suffers only from cold-start misses. The No Memory Latency (No Mem Latency) data point models a system in which requests can be processed in a single cycle, but each compute core can send only one request per cycle to the global memory system. These two data points enable us to measure the amount of touched-once data loaded by the Memcached kernel. Since the No Mem Latency model places a very low penalty on loading data into the cache, one cycle, the difference between these two data points accounts for the speedup relative to a cache large enough to hold the entire working set after incurring the cold-start load penalty. Increasing from no cache to a cache that captures all the kernel's locality takes the IPC from less than 1% of the peak to 12%, and removing the cold-start misses achieves 21% of the peak IPC. This suggests that Memcached contains a high fraction of touched-once data. This was verified by measuring the number of accesses to each L1 cache line prior to eviction in the 1M cache configuration. While the No Mem Latency configuration removes the penalty for cache misses, a large number of memory requests may cause contention in the memory system, resulting in performance loss. The No Memory Stalls configuration sends memory requests though the pipeline as fast as they are generated without any blocking. No Memory Stalls results in an additional 12% increase in performance over the No Memory Latency system. This result tells us that Memcached spends a large fraction of its execution time with a backed-up queue of memory requests waiting to access the memory system. If wavefronts do not stall on memory, then the overall performance is largely limited by SIMD efficiency. The performance of the non-stalling memory

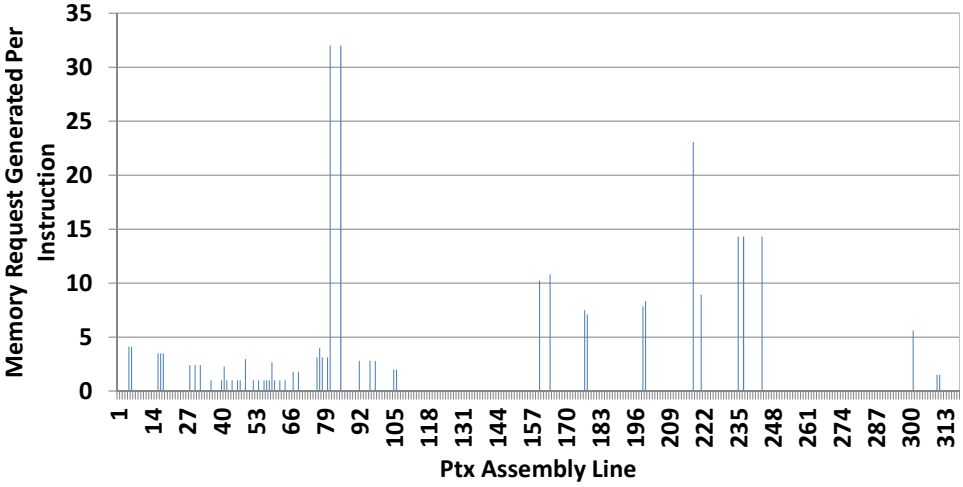


Figure 3.13: Memory requests generated per instruction for each static PTX instruction.

system is 33% of peak, while the measured SIMD efficiency of Memcached is ∼40%. This 7% discrepancy can be attributed to idle cycles when some cores take longer than others to complete the kernel.

Figure 3.13 illustrates the amount of memory divergence in Memcached (using AerialVision [13]). It presents the average number of global memory requests generated for each static PTX assembly instruction. In our GPGPU-Sim baseline configuration the maximum Y-value for each bar is 32 (all 32 lanes of the wavefront generate a request) and the minimum is two (because requests are coalesced per half-wavefront in GPGPU-Sim). A well-behaved GPU application will attempt to minimize this number and limit the stress on the memory system. From this graph we can see that many memory instructions do not fully coalesce their accesses into two requests. The bulk of the program's execution time is spent between PTX lines 157 and 253, where the instructions request between seven and 23 cache lines each on average. Further analysis of the Memcached code revealed that the main reason these instructions do not request closer to 32 lines is that Memcached's SIMD efficiency also drops during this phase, resulting in fewer active lanes, and hence fewer possible memory requests to be coalesced on each memory instruction. As a result, a relatively small amount of code repeatedly generates a large number of

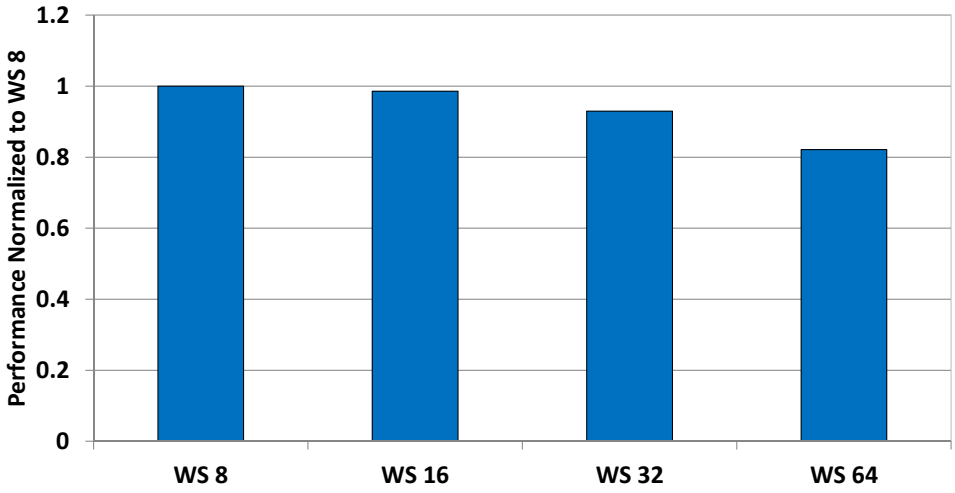


Figure 3.14: Performance of Memcached at various wavefront sizes (normalized to a warp size of 8).

memory accesses, which backs up the memory-request queue, leading to decreases in performance.

The preceding data indicates that the inclusion of an L1 data cache is critical to the performance of Memcached. Processing more than one memory request per cycle (e.g., through a multi-banked L1 data cache) would also improve performance because it allows the backed-up memory-request queue to empty sooner.

Effect of Wavefront Size on Performance

Figure 3.14 shows the performance of our modified Memcached on the baseline GPGPU-Sim simulator when varying wavefront lengths. The performance is normalized to a wavefront length of eight. A smaller wavefront length limits the amount of underutilized hardware resources when work items in the wavefront diverge, however, a larger wavefront length improves the maximum achievable performance as more instructions can execute per cycle and increases the number of memory requests potentially able to be coalesced. This data shows that there is an 18% decrease in performance between a wavefront size of eight and 64. This indicates that Memcached's SIMD efficiency is a limiting factor even in the presence of excessive memory stalls.

3.5 Summary

This chapter presents an initial characterization and evaluation of Memcached on both GPU hardware and GPU simulation frameworks. We identified many challenges with porting an application with irregular control-flow and memory access behaviour, as well as large data requirements, to a GPU system. We presented our solutions to address these challenges and mitigate their impact on performance, such as request batching, specific data structure layouts in memory to maximize memory coalescing, and reducing the CPU-GPU data transfer sizes. While this chapter focusses on Memcached, we believe that the presented methodology of batching user network requests for processing on a throughput-efficient device can be generalized such that other server-type applications with ample request-level parallelism could take advantage of this framework. We then presented an analysis using a GPU control-flow simulator, CFG-Sim, and a cycle-accurate GPGPU simulator, GPGPU-Sim, to gain additional insight into the behavior of Memcached on a GPU. From this analysis, we conclude that irregular applications, such as Memcached, should not be immediately disregarded when considering porting them to a GPU. Even though the SIMD efficiency may be lower than traditional GPU applications, we find that Memcached's SIMD efficiency is approximately 2.7$\times$ higher than a naive assumption of equal branch probabilities in the code-path may suggest. This, coupled with the GPU's high memory-level parallelism, enables Memcached to achieve a sizeable speedup over the baseline CPU implementation. We observed that the AMD Llano A8-3850 and Zacate E-350 integrated fusion GPUs outperformed their respective CPUs by factors of $\sim$7.5$\times$ and $\sim$4.5$\times$ respectively. We also showed that the discrete GPU system was able to significantly outperform the CPU when data transfers are ignored. However, when including data transfer times, results are hindered by the data transfer overheads of the large Memcached request batches.

In this chapter, we focussed solely on accelerating the key-value lookup portion of Memcached on a GPU. While this is the largest contributor to the Memcached user-level request processing, other portions of the full Memcached request processing, such as the network processing overheads and data movement, contribute to the majority of the end-to-end processing time. In the next chapter, we address

these issues by enabling the direct communication of Memcached data between the network interface and GPU, and offloading the network processing to the GPU.

Chapter 4

Memcached GPU

The previous chapter (Chapter 3) evaluated the potential for accelerating irregular datacenter-based applications, such as Memcached, on contemporary integrated and discrete GPUs. However, Chapter 3 focussed specifically on accelerating Memcached's key-value look-up on the GPU, with all of the other required networking and Memcached request processing remaining on the CPU. While the key-value look-up accounts for the core operations and the majority of user-space processing time for a Memcached request, when considering all of the operations required for the end-to-end network request processing, the key-value look-up results in a relatively small fraction of the overall processing. This is highlighted in Figure 4.1 and Figure 4.2, which present a fine-grained breakdown of the total Memcached processing time for a single *GET* request, including Memcached operations and Linux system calls, and a coarse-grained breakdown of Memcached versus Linux kernel processing time, including Linux network processing, respectively. The execution breakdown was measured for a single request using Intel VTune [80]. As a result, some of the Linux kernel overheads that may be amortized over multiple network requests are not captured in this analysis. Figure 4.1 shows the portions of the *GET* request processing that was accelerated in Chapter 3, indicated by the red boxes, which contribute to the majority of the user-space processing. However, as can be seen in Figure 4.2, the overall majority of processing time for a single *GET* request is spent in the Linux Kernel (e.g., Linux network stack) compared to the actual user-space Memcached processing ($\sim$10%).

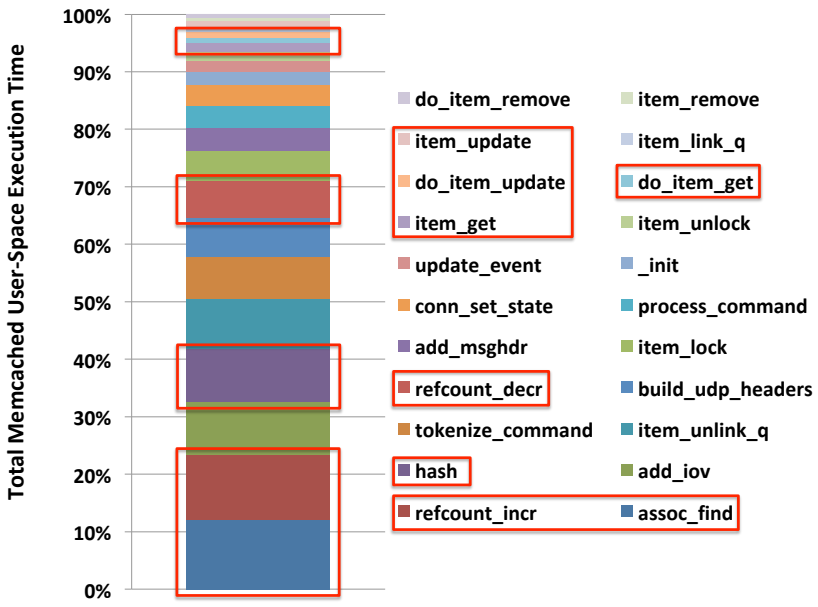


Figure 4.1: Breakdown of the baseline Memcached request processing time for a single *GET* request on the CPU.

Furthermore, a large portion of the Memcached user-space processing is also spent parsing the Memcached request and building the response packets. This chapter explores offloading the complete end-to-end processing for Memcached requests to the GPU, including the network packet processing.

To this end, we implement and evaluate an end-to-end version Memcached on commodity GPU and Ethernet hardware, MemcachedGPU. Memcached is a *scale-out* workload, typically partitioned across multiple server nodes. In this Chapter, we focus on using the GPU to *scale-up* the throughput of an individual server node. We exploit request-level parallelism through batching to process multiple concurrent requests on the massively parallel GPU architecture, and task-level parallelism within a single request to improve request latency. While the previous chapter and other works have evaluated batch processing of network requests on GPUs, such

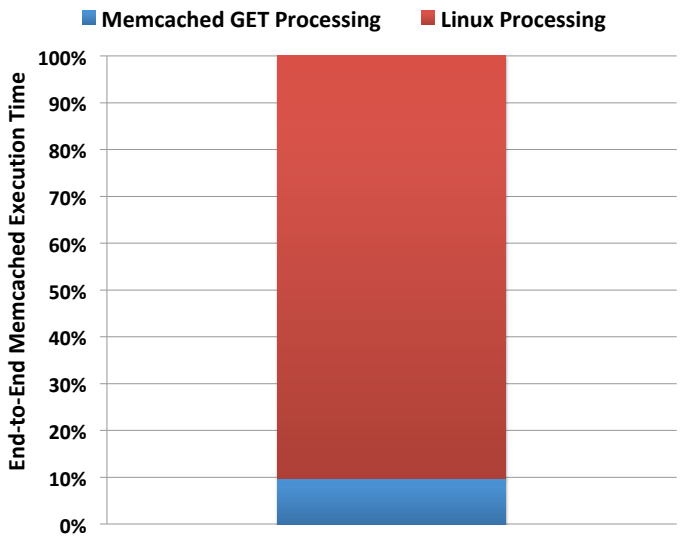


Figure 4.2: End-to-end breakdown of user-space and Linux Kernel processing for a single *GET* request on the CPU.

as HTTP [3], or database queries [18, 179], they focus solely on the application processing, which depending on the application, can be a small subset of the total end-to-end request processing. In contrast, this chapter describes the design and implementation of a complete software GPU network offload management framework, *GNoM* [1], that incorporates UDP network processing on the GPU in-line with the application processing (Figure 4.3). *GNoM* provides a software layer for efficient management of GPU tasks and network traffic communication directly between the network interface (NIC) and GPU.

This is the first work to perform all of the Memcached read request processing and UDP network processing on the GPU. We address many of the challenges as-

[1]Code for GNoM and MemcachedGPU is available at https://github.com/tayler-hetherington/MemcachedGPU.

sociated with a full system network service implementation on heterogeneous systems, such as efficient data partitioning, data communication, and synchronization. Many of the core Memcached data structures are modified to improve scalability and efficiency, and are partitioned between the CPU and GPU to maximize performance and data storage. This requires synchronization mechanisms to maintain a consistent view of the application's data. The techniques presented in this chapter can be generalized to other network services that require both CPU and GPU processing on shared data structures.

This chapter also tackles the challenges with achieving low-latency network processing on throughput-oriented accelerators. GPUs provide high throughput by running thousands of scalar threads in parallel on many small cores. *GNoM* achieves low latency by constructing fine-grained batches, such as 512 requests, and launching multiple batches concurrently on the GPU through multiple parallel hardware communication channels. Compared to the previous chapter, which evaluated batch sizes of tens of thousands of requests, the small batch sizes here minimize batching delay and processing latency. At 10 Gbps with the smallest Memcached request size (96 bytes), the smaller batches result in requests being launched on the GPU every $\sim$40μs, keeping the GPU resources occupied to improve throughput while reducing the average request batching delay to under 20μs.

This chapter is organized as follows: Section 4.1 presents *GNoM*, a software framework for UDP network and application processing on GPUs; Section 4.2 describes the design of MemcachedGPU, an accelerated end-to-end key-value store, which leverages *GNoM* to efficiently run on a GPU; Section 4.3 presents the experimental methodology in this chapter; Section 4.4 evaluates the feasibility of achieving low-latency, 10 Gbit line-rate processing at all request sizes on commodity Ethernet and throughput-oriented GPU hardware; Section 4.4.4 explores the potential for workload consolidation on GPUs running on servers in a datacenter during varying client demands, while maintaining a level of quality of service (QoS) for a GPU networking application; and finally Section 4.4.6 compares MemcachedGPU against prior Memcached implementations.

4.1 GPU Network Offload Manager (*GNoM*)

GNoM is a CPU-GPU software framework that enables high-throughput and low-latency processing of (UDP) network packets on the GPU. This section addresses some of the challenges with achieving this goal and presents the software architecture of *GNoM* that facilitated the design and implementation of MemcachedGPU. While we focus on accelerating Memcached using the *GNoM* framework, we believe that many parts of the design, such as the efficient packet data movement and GPU task management, can be generalized to support other applications in the domain of high-throughput, UDP request/response-type applications.

4.1.1 Request Batching

Request batching in *GNoM* is done per request type (*GET*, *SET*, *UPDATE*, *DELETE*) to reduce control flow divergence among GPU threads. While intra-request control flow divergence still exists (as discussed in Chapter 3), the high-level operations within each request type are the same (e.g., hashing the request key or accessing the hashtable). The previous chapter evaluated large batch sizes (e.g., 30K+ requests/batch) to maximize the GPU's utilization and request throughput. However, large batch sizes negatively impact request latency, since requests will be queued longer and per-request processing will be longer. Smaller batch sizes minimize batching latency, however, increase the GPU kernel launch overhead and lower GPU resource utilization, which reduces overall throughput. In this chapter, we evaluate the use of smaller batch sizes, but overlap multiple such smaller batches to reduce the impact on throughput. We empirically find that 512 requests per batch provides a good balance of throughput and latency (Section 4.4.3). As discussed in Section 2.1.2, NVIDIA GPUs support up to 32 concurrently running kernels via Hyper-Q. Assuming that the GPU has enough resources to support all 32 *GNoM* kernels, a batch size of 512 requests enables 16K requests to be processed concurrently.

While MemcachedGPU's main focus is in accelerating a single request type for batching, *GET* requests, workloads with many different request types could batch requests at finer granularities. For example, multiple smaller batches could be constructed at the warp-level of 32 requests and launched together on the GPU

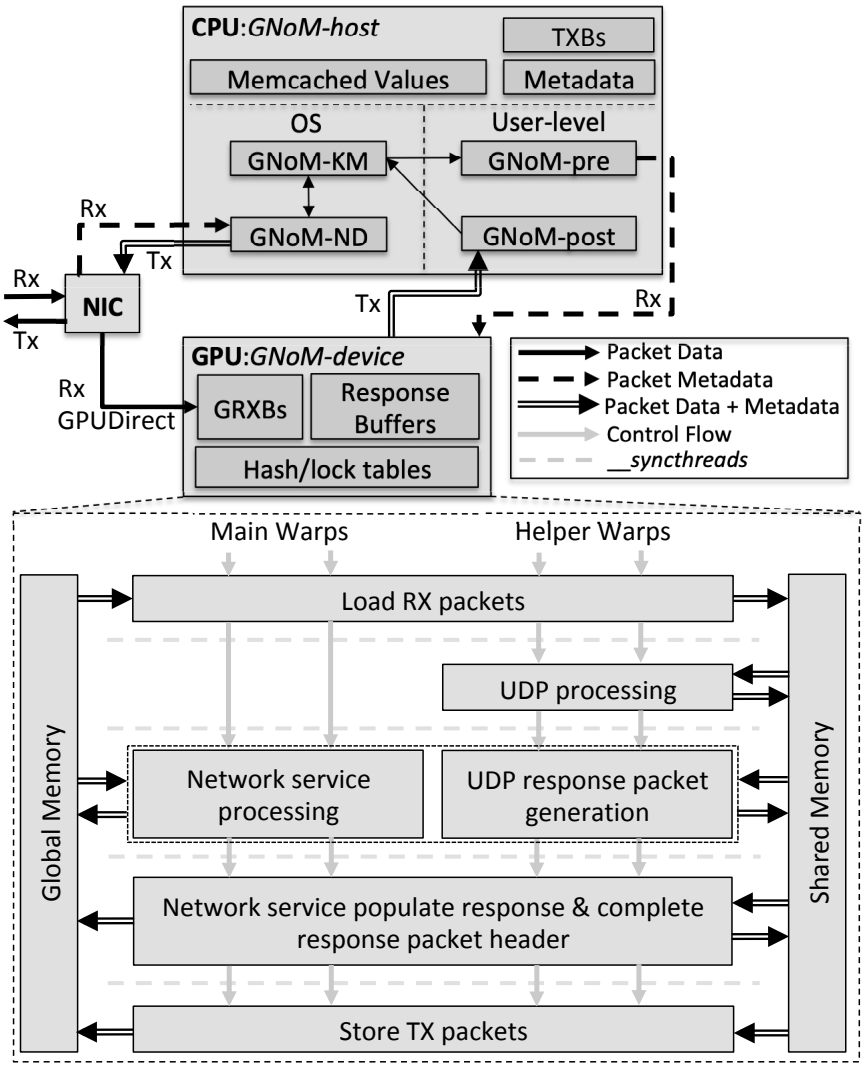


Figure 4.3: *GNoM* packet flow and main CUDA kernel. The figure contains the three main components, the NIC, CPU, and GPU, and the corresponding *GNoM* software frameworks that run on each device. The solid black arrows represent data flow, the dashed black errors represent metadata flow (e.g., interrupts, packet pointers), the double black arrows represent packet data and packet metadata, the solid grey arrows represent GPU thread control flow, and the dashed grey lines represent GPU synchronization instructions.

to perform different tasks [22] in parallel. MemcachedGPU does support other Memcached request types, such as *SET UPDATE*, and *DELETE*, however, the current version implements these requests in individual batches of a single warp. Batching and accelerating these requests types are left to future work.

4.1.2 Software Architecture

GNoM is composed of two main software frameworks that cooperate to balance throughput and latency for network services on GPUs: *GNoM-host* (CPU) and *GNoM-dev* (GPU). At a high-level, *GNoM-host* is responsible for interacting with the NIC, performing any pre/post-GPU data processing, and for GPU task management. *GNoM-dev* (GPU) is responsible for the main UDP packet and application-level processing. Figure 4.3 presents the *GNoM* software architecture, along with the interactions between the NIC, CPU, and GPU in *GNoM*, as well as the main CUDA kernel for *GNoM-dev*. RX (receive) packets are DMA-ed directly to the GPU's global memory using GPUDirect (Section 2.1.2). Only metadata describing the request batch is sent to the CPU. This metadata is described further below. We focus on accelerating the RX path in *GNoM*. While *GNoM* can implement a similar TX (transmit) path using GPUDirect, MemcachedGPU uses a third party CPU Linux network bypass service, *PF_RING* [125], to accelerate the TX path on the CPU. This decision was driven by the design of MemcachedGPU in which the main data structures used to populate the response packets are stored in CPU memory (Section 4.2). This design decision increases the total amount of storage available for Memcached, which is important for the effectiveness of a Memcached server (Section 4.2.2). However, this also requires CPU post processing, which decreases the total system energy-efficiency (Section 4.4).

The rest of this section describes *GNoM-host* and *GNoM-dev* in detail.

GNoM-host

GNoM-host provides task management and I/O support for the GPU. *GNoM-host* is required because current GPUs cannot directly receive control information or interrupts from other third party devices in a heterogeneous system through the standard CUDA interface. We evaluate this limitation further in Chapter 5. Other

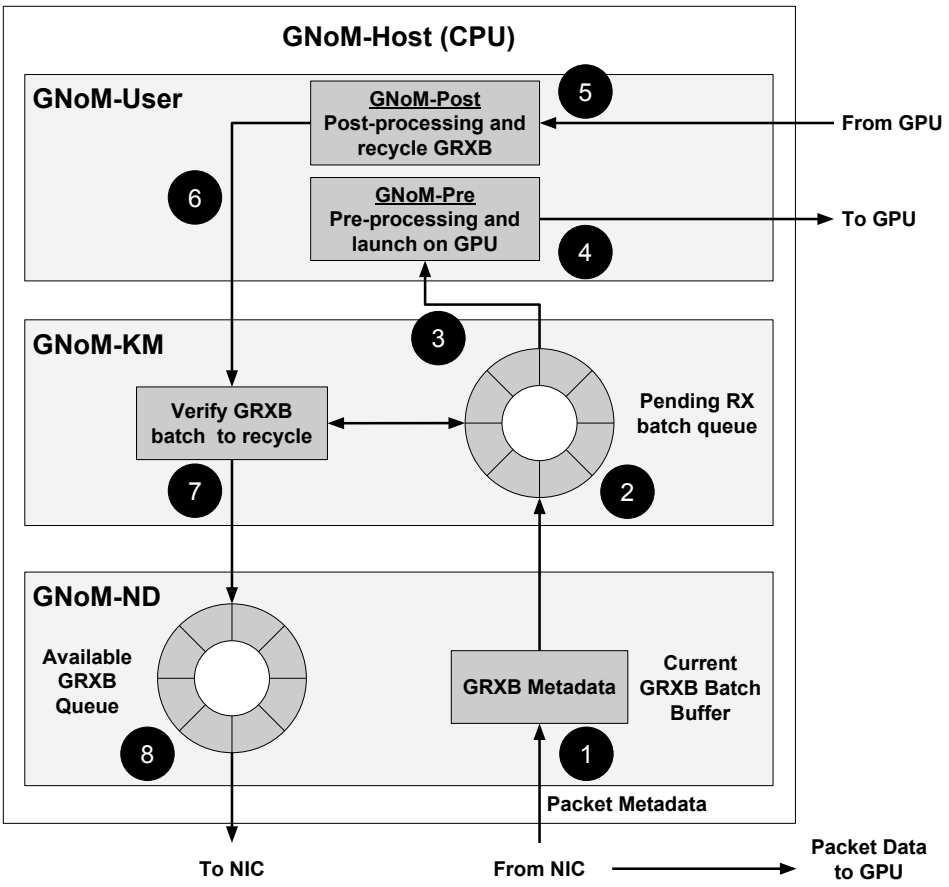


Figure 4.4: Software architecture for *GNoM-host* (CPU).

work has explored the reverse direction of GPUs communicating to third party devices, for example, by having the GPU write to the doorbell register on a NIC [183] to indicate when data is available to send. However, on current systems, third party devices must go through the GPU driver to initiate work on the GPU, which requires assistance from the CPU. A workaround for this is to use persistent GPU threads, which are described in Section 2.1.7. For *GNoM*, this limitation restricts the NIC from being able to directly initiate tasks on the GPU, even though the packet data is already in GPU memory. As a result, the CPU acts as a middleman between the NIC and GPU. *GNoM-host* is responsible for efficiently managing metadata movement between the NIC and GPU, managing tasks on the GPU, and

performing any post-processing tasks required prior to sending response packets. As described above, all packet data is directly copied to the GPU memory from the NIC. To accomplish these tasks, *GNoM-host* is composed of three software components: a modified Intel IXGBE network driver (v3.18.7), *GNoM-ND*, a custom Linux kernel module, *GNoM-KM*, and a user-level software framework, *GNoM-user*, which are described below. Figure 4.4 highlights the interaction between these three components. In the following sections, any reference to ❶ refers to Figure 4.4.

GNoM-KM and *GNoM-ND*

GNoM-KM is a custom Linux kernel module, which provides an interface between the NIC driver (*GNoM-ND*) and user-space application (*GNoM-user*). *GNoM-KM* includes hooks for *GNoM-user* to communicate indirectly with the NIC via *GNoM-ND*, such as configuring the NIC for use with *GNoM*, retrieving new request batches, and recycling completed batches of request buffers to the NIC. Multiple steps are required to initialize and configure *GNoM*.

GNoM-KM first allocates pinned, un-pageable GPU memory using GPUDirect to store incoming RX packets, referred to in this work as GPU RX Buffers (GRXB). A total of 220MB is allocated for the GRXBs, partitioned into 32 - 2KB buffers per 64KB GPU page. 220MB is the maximum amount of pinnable GPU memory one can allocate for GPUDirect on the NVIDIA Tesla K20c at the time this study was performed. However, future NVIDIA GPUs (starting from the NVIDIA K40) increased the amount of pinnable GPU memory that can be access across the PCIe to 16GB [183]. The limited pinnable memory on the GPUs evaluated in this study reduces the peak obtainable throughput, which is discussed further below. Evaluating *GNoM* on newer GPUs with increased support for RDMA-accessible memory is left to future work. *GNoM-KM* allocates the GRXBs and registers them with *GNoM-ND*. The GRXBs are maintained in a circular queue in *GNoM-ND* ❽ and can be in one of three states: *free*, *registered*, or *busy*. Free buffers are not allocated to either device (NIC or GPU) and are waiting to be registered with the NIC. Registered buffers are registered with the NIC, waiting for a new RX packet to fill the buffer. Busy buffers contain a valid RX packet and are either waiting or actively

being processed on the GPU. *GNoM-KM* also allocates a secondary circular buffer ❷ for storing batches of GRXBs which have been populated by *GNoM-ND*, but not yet consumed by *GNoM-user* (described in more detail below).

The NIC is then configured in *GNoM-ND*. The NIC evaluated in this study (Intel 82599 10GbE) contains multiple hardware RX queues and supports hardware packet filtering and receive side scaling (RSS) [67]. RSS enables RX hardware queues to be mapped to different CPU processors such that incoming packets can be distributed across processors to increase performance. The NIC can apply a hardware packet filter to steer packets to the different RX queues. We make use of this feature by assigning one of the RX hardware queues to be a GPU queue. In a multi-GPU system, multiple GPU packet filters could be installed to distribute packets across GPUs. *GNoM-ND* installs a hardware packet filter for a range of UDP ports to be processed by the GPU, such that any packet that matches the filter is directed to the GPU RX queue. *GNoM-ND* ensures that the GRXBs allocated by *GNoM-KM* are only registered with the GPU RX queue, which enables packet data to flow directly from the NIC to the GPU's memory. Another benefit of this organization is that all other network traffic not destined for the GPU is steered towards the non-GPU RX queues, which flows through the baseline Linux networking stack on the CPU.

Similar to the baseline Linux networking stack, *GNoM* uses Linux NAPI to mitigate the impact of high interrupt rates by scheduling a polling routine to service the received packets. As the packet rate increases, and consequently the rate of interrupts from the NIC increases, *GNoM-ND* disables any further interrupts for received packets and schedules a polling routine. At a later time, the polling routine is run, which queries the NIC for a specified number of packets to service. If there are fewer packets than this threshold, the polling routine is completed. Otherwise, the polling routine is rescheduled to service the remaining packets.

To reduce memory copies for packet data, the GRXBs hold the packets throughout their lifetime on the server, recycling them to the NIC for new packets only when the full processing is complete. This differs from the baseline Linux network driver flow, which recycles the RX buffers immediately to the NIC after copying the packets into Linux SKBs for further processing in the Linux kernel. In this case, the GRXBs acts as both the RX buffers and the Linux SKBs. As such,

GNoM requires significantly more pinned memory than the baseline network flow to ensure that a GRXB is available to store incoming packets. In contrast, the baseline Linux network driver flow can have small amount of pinned memory for the RX buffers and a large amount of pageable memory for the Linux SKBs.

A packet drop occurs in *GNoM* when GRXBs are not recycled quickly enough to accommodate newly received packets. The time to recycle packets is directly impacted by the time to process the packet on the CPU and GPU. As as result, along with increasing the request throughput in *GNoM*, we must also aim to minimize the request processing latency such that we always have a free GRXB for a new packet. If more GRXBs cannot be allocated, then the client's packet send rate must be reduced accordingly to avoid dropping packets on the GPU server. We evaluate the improvement in system throughput when the number of GRXBs can be arbitrarily increased through an offline study, where packets are read directly from system memory instead of from the network (Section 4.4.5).

GNoM-ND has two main responsibilities when receiving new packets from the network. First, *GNoM-ND* DMAs the incoming packets to the GRXBs directly in GPU memory, indicated by the solid black RX arrow from the NIC to the GPU in Figure 4.3 (labelled RX GPUDirect). Second, *GNoM-ND* constructs metadata describing the batch of GPU packets ❶ and passes the batch metadata to *GNoM-KM* once a batch is fully populated and ready for GPU processing ❷. The GRXB batch is stored in a circular queue in *GNoM-KM*. If a GRXB batch is not yet ready in *GNoM-KM* when a *GNoM-user* thread issues a read request for a new batch of GRXBs, the thread is blocked. When *GNoM-ND* transfers the GRXB batch to *GNoM-KM*, any waiting threads are notified, which resumes blocked threads to consume the new batch of packets. The GRXB batch metadata is then copied from kernel-space to user-space ❸ and a batch ID is stored in *GNoM-KM* to perform a sanity check when the corresponding batch is recycled.

We found that the NIC populates the GRXBs in the order that they were registered to the NIC. As such, the batch metadata only requires a pointer to the first packet (GRXB) and the total number of packets in the batch to identify all packets in the batch. Note that this requires all GRXBs to have the same size. With this optimization, the amount of data needing to be transferred to *GNoM-user* and across the PCIe to the GPU is reduced from 4KB (512 packets, 8 bytes per GRXB pointer)

to 12B (8 bytes for the first GRXB pointer and 4 bytes for the packet counter [2]).

Once the packets have completed processing on the GPU and CPU, the GRXBs must be recycled back to the NIC to be able to receive new packets. *GNoM-KM* provides an interface for the *GNoM-user* threads to recycle GRXBs to *GNoM-ND* ❻. *GNoM-KM* performs a sanity check ❼ to ensure that the GRXB batch is valid and then *GNoM-ND* marks the GRXBs as *free* prior to registering them back with the NIC ❽.

GNoM-User

GNoM-user consists of pre-processing (*GNoM-pre*) and post-processing (*GNoM-post*) user-level threads (See Figure 4.3 and Figure 4.4). *GNoM-pre* retrieves request batch metadata from *GNoM-KM* ❸, performs application specific memory copies to the GPU, launches CUDA kernels that perform the network service processing on the batch of requests, and constructs CUDA events to detect when the GPU processing completes ❹. For MemcachedGPU, *GNoM-pre* transfers the GRXB metadata describing the current batch (pointer to the first GRXB and the number of packets in this batch), a timestamp for this batch, and a pointer to the GPU memory to store the response packets. *GNoM* uses CUDA streams to overlap processing of multiple small batches to provide a better trade-off between packet latency and throughput.

GNoM-post ❺ polls CUDA events waiting for the GPU network service processing to complete, populates the response packets with application specific data, and transmits the response packets using *PF_RING*. For MemcachedGPU, this consists of copying the item's value, corresponding to the Memcached key in the original *GET* request, from the Memcached memory slabs in CPU memory to the TXBs (*PF_RING* transmit buffers) in CPU memory for each packet in the batch. The *PF_RING* buffers are then sent out the NIC. Finally, *GNoM-post* recycles the now free GRXBs back to *GNoM-ND* for future RX packets ❻. As will be discussed in Section 4.4.3, the performance of *GNoM* is highly dependent on the rate at which *GNoM-post* threads are able to complete the post-processing tasks and recycle the GRXBs. We empirically find that four *GNoM-post* threads are required to achieve

[2]While a 4 byte counter is unnecessary given that the batch size is 512 packets, there is little performance benefits when transferring small data sizes across the PCIe bus.

10 GbE line-rate throughput with the smallest Memcached packet sizes. However, the polling nature of *GNoM-post* threads negatively impacts energy-efficiency.

Non-GPUDirect (NGD)

GPUDirect is currently only supported on the high-performance NVIDIA Tesla and Quadro GPUs. As previously discussed, GPUDirect minimizes the amount of data which first needs to be copied to CPU memory prior to being copied to the GPU memory by directly transferring packet data to GPU memory from the NIC. To evaluate MemcachedGPU on lower power, lower cost GPUs, we also implemented a non-GPUDirect (NGD) framework. NGD uses *PF_RING* [125] to receive and batch Memcached packets in host memory before copying the request batches to the GPU. NGD uses the same *GNoM-user* and *GNoM-dev* framework; however, *GNoM-KM* and *GNoM-ND* are replaced by *PF_RING*. Section 4.4.3 evaluates NGD on the NVIDIA Tesla K20c and GTX 750Ti GPUs.

GNoM-dev

The lower portion of Figure 4.3 illustrates the *GNoM* CUDA kernel for UDP packet and network service processing (e.g., MemcachedGPU *GET* request processing). Once a network packet has been parsed (UDP processing stage), the network service can operate in parallel with the response packet generation since they are partially independent tasks. Similar to Singe [22], we make use of *warp specialization* via conditional operators on warp IDs to perform multiple different independent, but related tasks on different warps within the same thread block. In *GNoM*, the number of GPU threads launched per packet is configurable (MemcachedGPU uses two threads). *GNoM-dev* leverages additional *helper threads* to perform parallel tasks related to a single network service request, exploiting both packet level and task level parallelism to improve response latency and throughput.

GNoM-dev groups warps into main and helper warps. Main warps perform the network service processing (e.g., Memcached *GET* request processing) while helper warps perform the UDP processing and response packet header construction. The main and helper warps also cooperatively load RX data and store TX data (e.g., response packet headers and any application specific data, such as point-

ers to Memcached items in CPU memory) between shared and global memory efficiently through coalesced memory accesses. This requires CUDA synchronization barriers (*__syncthreads*) to ensure that the main and helper warps maintain a consistent view of the packet data in shared memory. The UDP processing stage verifies that the packet is for the network service and verifies the IP checksum. While most of the response packet header can be constructed in parallel with the network service processing, the packet lengths and IP checksum are updated after to include any application dependent values. For example, the length of the Memcached item value is not known until after the Memcached hash table has been accessed and the corresponding entry is retrieved (network service processing stage). After synchronizing, all warps cooperatively update the response packets with any application dependent data and then proceed to copy the response packet data and any application specific data (e.g., pointers to Memcached items in CPU memory) from shared memory to global memory to be processed by the *GNoM-post* threads on the CPU.

4.2 MemcachedGPU

This section presents the design of MemcachedGPU and discusses the modifications required to achieve low latency and high throughput processing on the GPU.

4.2.1 Memcached and Data Structures

As previously mentioned, in typical Memcached deployments [124], *GET* requests comprise a large fraction of traffic when hit-rates are high (e.g., 99.8% for Facebook's USR pool [16]). Hence in MemcachedGPU, we focus on accelerating Memcached *GET* requests and leave the majority of *SET* request processing on the CPU.

Memcached data structures accessed by both *GET* and *SET* requests include the hash table to store pointers to Memcached items, memory slabs to store the Memcached items and values, and a least-recently-used (LRU) queue for selecting key-value pairs to evict from the hash table when Memcached runs out of memory on a *SET*. Memcached keys can be an order of magnitude smaller than value sizes (e.g., 31B versus 270B for Facebook's APP pool [16]), placing larger storage

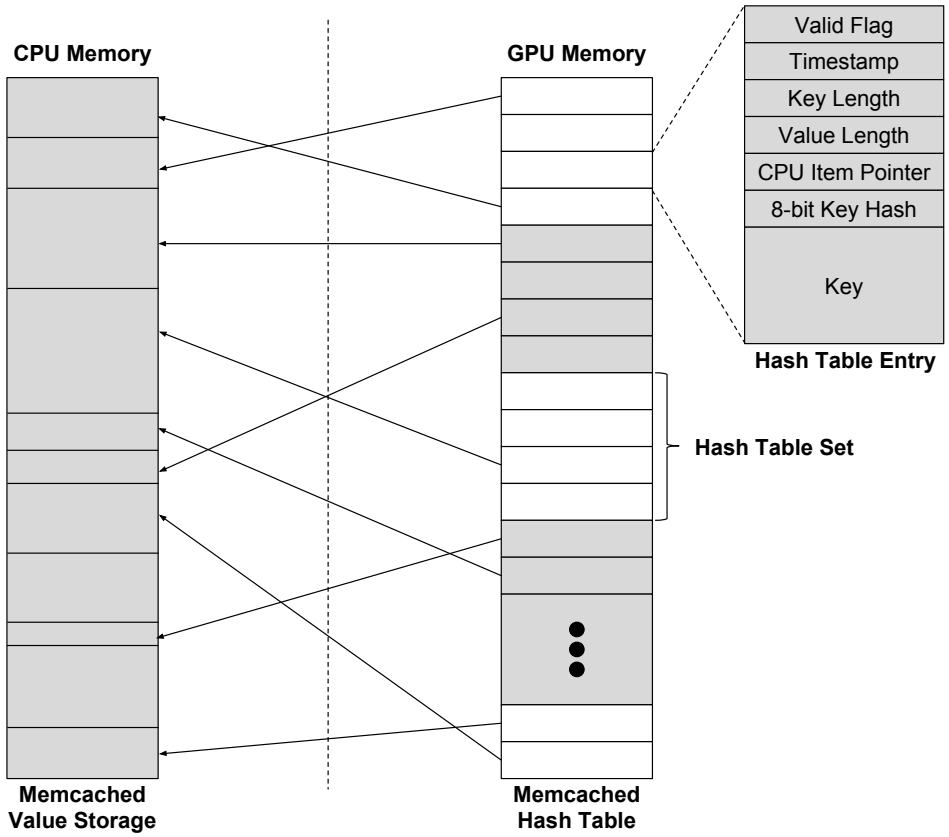


Figure 4.5: Partitioning the Memcached hash table and value storage between the CPU and GPU.

requirements on the Memcached item values.

These data structures need to be efficiently partitioned between the CPU and GPU due to smaller GPU DRAM capacity versus CPUs found on typical Memcached deployments and to ensure high performance and scalability. In MemcachedGPU, we place the hash table containing keys and item pointers in GPU memory, while the Memcached item values stored in the memory slabs remain in CPU memory. This partitioning ensures that the majority of data structures accessed for *GET* request processing are stored in GPU memory, which helps to minimize request processing latency. This partitioning is highlighted in Figure 4.5.

GET Requests

At a high level, *GET* requests perform a look-up in a hash table and return the corresponding Memcached item if found. MemcachedGPU performs the main *GET* request operations on the GPU. This includes parsing the *GET* request, extracting the key from the *GET* request packet, hashing the key, and accessing the hash table to retrieve the corresponding item pointer. Aside from the item values, all of the Memcached data structures accessed by *GET* requests are stored in GPU global memory. MemcachedGPU uses the same Bob Jenkin's lookup3 hash function [86] included in the baseline Memcached to hash the key. *GET* requests completely bypass the CPU and access the hash table on the GPU as described in Section 4.2.2. Each *GET* request is handled by a separate GPU thread, resulting in memory divergence on almost every hash table access. This is because each request should, based on the efficacy of the hashing function, be hashed to different parts of the hash table. However, the small number of active GPU threads and the GPU's high degree of memory-level parallelism mitigates the impact of memory divergence on performance. As such, multiple memory requests can be in-flight and processed concurrently. Additionally, when possible, *GNoM* stores packet data in GPU shared memory, which improves performance. After the GPU processing is complete, *GNoM-post* receives a list of Memcached response packet headers and corresponding item value pointers for each *GET* request (assuming the *GET* request hit in the hash table). Finally, *GNoM-post* copies the item value for each packet in the batch from CPU memory into a response packet (TXB) to be sent across the network.

SET Requests

While the main focus of MemcachedGPU is on accelerating *GET* requests, *SET* requests must also interact with the GPU to update the hash table. *SET* requests require special attention to ensure consistency between CPU and GPU Memcached data structures. In MemcachedGPU, *SET* requests follow the standard Memcached flow over TCP through the Linux network stack and Memcached code on the CPU. They update the hash table with the new or updated entry by launching a simple *SET* request handler on the GPU.

SET requests first allocate the item data in the Memcached memory slabs in CPU memory and then update the GPU hash table. This ordering ensures that subsequent *GET* requests are guaranteed to find valid CPU item pointers in the GPU hash table – a *GET* request will only encounter a entry in the hash table which holds a valid item stored in CPU memory. Another consequence of this ordering is that both *SET* and *UPDATE* requests are treated the same since the hash table has not been probed for a hit before allocating the new Memcached item. An *UPDATE* request simply acts as a *SET* that evicts and replaces the previous item with the new item. As such, the previous item is freed from CPU memory if the *SET* request returns that an existing entry corresponding to this *SET* request's key was found in the hash table.

SET requests update the GPU hash table entries, introducing a race condition between *GET* requests and other *SET* requests. Section 4.2.2 describes a GPU locking mechanism to ensure exclusive access for *SET* requests, while maintaining shared access for *GET* requests. As *GET* requests typically dominate request distributions, we have used a simple implementation in which each *SET* request is launched as a separate kernel and processed by a single warp. Aside from the usage of the TCP protocol, there is no fundamental reason why *SET* requests could not follow a similar flow as the *GET* requests in the *GNoM* framework. At a minimum, *SET* requests can be batched on the CPU (after flowing through the standard Linux protocol) and processed concurrently on the GPU. Accelerating *SET* requests through batch processing, potentially requiring a reliable network protocol, is left to future work.

Additionally, because *SET* requests and *GET* requests are handled by separate CPU threads, another race condition exists when a *GET* request attempts to access a Memcached item's value in CPU memory concurrently with a *SET* request. Section 4.2.3 addresses this race condition between dependent *SET* and *GET* requests.

4.2.2 Hash Table

This section presents the modifications to the baseline Memcached hash table to enable low-latency and high-throughput processing on the GPU while minimizing the impact on the hash table hit rate.

Hash Table Design

The baseline Memcached implements a dynamically sized hash table with hash chaining on collisions. A collision refers to the case where different values are hashed to the same entry and are caused by having a finite number of entries in the hash table. A hash table with hash chaining resolves collisions by dynamically allocating new entries and linking them into existing linked lists (chains) at conflicting hash table entries. This ensures that all items will be stored as long as the system has enough memory. However, depending on the rate of collisions, a long chain of entries may need to be traversed to find the correct entry.

This hash table design is a poor fit for GPUs for two main reasons. First, dynamic memory allocation on current GPUs can significantly degrade performance [75]. Second, hash chaining creates a non-deterministic number of elements to be searched between requests when collisions are high. This can degrade SIMD efficiency when chain lengths vary, since each GPU thread handles a separate request. In this case, some threads may find their request in the first entry in the chain but are blocked waiting for any other threads which have to traverse a long chain to find their entry.

The above observations drive the hash table design in MemcachedGPU, which implements a fixed-size *set-associative* hash table, similar to [26, 106]. We select a set size of 16-ways (see Section 4.4.1). We also evaluated a modified version of *hopscotch hashing* [68] that evicts an entry if the hopscotch bucket is full instead of rearranging entries. This improves the hit-rate over a set-associative hash table by 1-2%; however, the peak *GET* request throughput is lower due to additional synchronization overheads. Specifically, the hopscotch hash table requires locking on every entry since no hopscotch group is unique, whereas in MemcachedGPU, the set-associative hash table requires locking only on each set.

Each hash table entry contains a header and the physical key (Figure 4.5). The header contains a valid flag, a last accessed timestamp, the length of the key, the length of the corresponding item value, and a pointer to the item in CPU memory. MemcachedGPU also adopts an optimization from [52, 106] that includes a small 8-bit hash of the key in every header. When traversing the hash table set, the 8-bit hashes are first compared to identify potential matches. The full key is compared

only if the key hash matches, reducing both control flow and memory divergence.

Hash Table Collisions and Evictions

The baseline Memcached uses a global lock to protect access to a global LRU queue for managing item evictions. On the GPU, global locks would require serializing all GPU threads for every Memcached request, resulting in low SIMD efficiency and poor performance. Other works [124, 177] also addressed the bottlenecks associated with global locking in CPU implementations of Memcached.

Instead of a global LRU, we manage a local LRU per hash table set, such that *GET* and *SET* requests only need to update the timestamp of the hash table entry. The intuition is that the miss rate of a set-associative cache is similar to a fully associative cache for high enough associativity [144]. Whereas hash chaining allocates a new entry on a collision, collisions in MemcachedGPU are resolved by finding a free entry or evicting an existing entry within the hash table set. This introduces an additional eviction condition to Memcached, which previously only occurred when the maximum amount of item storage has been exceeded. While a set-associative hash table was also proposed in [26, 106], we expand on these works by evaluating the impact of the additional evictions on hit-rates compared to the baseline Memcached hash table with hash chaining in Section 4.4.1. We find that the additional evictions result in a decrease in hit-rate of approximately 0.01% to 3.6% for different key access distributions. This indicates that a global LRU replacement policy is not necessary to effectively capture the locality.

SET requests search a hash table set for a matching, invalid, or expired entry. If the *SET* misses and no free entries are available, the LRU entry in this hash table set is evicted. *GET* requests traverse the entire set until a match is found or the end of the set is reached. This places an upper bound, the set size, on the worst case number of entries each GPU thread traverses, which mitigates the impact on SIMD efficiency relative to hash chaining. If the key is found, the CPU value pointer is recorded to later populate the response packet.

Storage Limitations

As previously described, the hash table is partitioned from the value storage due to the relatively smaller GPU DRAM capacity versus CPUs. This static hash table places an upper bound on the maximum amount of key-value storage. Consider a high-end NVIDIA Tesla K40c with 12GB of GPU memory. *GNoM* and MemcachedGPU consume ∼240MB of GPU memory for data structures such as the GRXBs, response buffers, and *SET* request buffers. This leaves ∼11.75GB for the GPU hash table and the lock table (described in Section 4.2.2). The hash table entry headers are a constant size, however, the key storage can be varied depending on the maximum size. For example, the hash table storage increases from 45 million to 208.5 million entries when decreasing from a maximum key size of 250B to 32B. From [16], typical key sizes are much smaller than the maximum size, leading to fragmentation in the static hash table if each entry is allocated for the worst case maximum size.

If a typical key size distribution is known, however, multiple different hash tables with fixed-size keys can be allocated to reduce fragmentation. For example, [16] provides key and value size distributions for the Facebook ETC workload trace. If we create five hash tables with static key entries of 16, 32, 64, 128, and 250B with each size determined by the provided key distribution (0.14%, 44.17%, 52.88%, 2.79%, and 0.02% respectively), this enables a maximum of 157 million entries for a 10GB hash table. Using the average value size of 124B for ETC, this static partitioning on the GPU would enable indexing a maximum of 19.2GB of value storage in CPU memory compared to only 5.5GB when allocating for the worst case key size.

While there's a trend for growing GPU DRAM sizes, integrated GPUs may remove this limitation with access to far more DRAM than discrete GPUs. Our results on a low-power GPU (Section 4.4.3) and integrated GPUs (Chapter 3) suggest that integrated GPUs may be able to achieve high throughputs in MemcachedGPU and are an important alternative to explore for *GNoM* and MemcachedGPU.

GPU Concurrency Control

Aside from updating the timestamps, *GET* requests do not modify the hash table entries. Thus, multiple *GET* requests can access the same hash table entry concurrently as they are guaranteed to have similar timestamp values [3]. Furthermore, Memcached has a relaxed requirement on request ordering. As such, the race condition between concurrent updates to the timestamp from *GET* requests can be safely ignored. However, *SET* requests require exclusive access since they actually modify the hash table entries. To handle this, we employ a multiple reader (shared), single writer (exclusive) spin lock for each hash table set using CUDA atomic compare and exchange (CAS), increment, and decrement instructions. The GPU locks are also implemented as test-and-test-and-set to reduce the number of atomic instructions. The shared lock ensures that threads performing a *GET* request in a warp will never block each other, whereas the exclusive lock ensures exclusive access for *SET* requests to modify a hash table entry.

For *SET* requests, a single thread per warp acquires an exclusive lock for the hash table set. The warp holds on to the lock until the *SET* request hits in one of the hash table entries, locates an empty or expired entry, or evicts the LRU item for this set. The remaining threads in the warp are used to perform a coalesced store of the key into the hash table.

The hash table locks are maintained in a *lock table* residing in GPU global memory. The size of the lock table is dictated by the size of the hash table and the number of sets, with one lock entry per hash table set. Due to the high ratio of *GET* requests to *SET* requests, it would also be possible to have a smaller lock table, since *GET* requests obtain shared locks which do not block other *GET* requests.

4.2.3 Post GPU Race Conditions on Eviction

While the CPU item allocation order (Section 4.2.1) and GPU locking mechanism (Section 4.2.2) ensure correct access to valid items in CPU memory, a race condition still exists in *GNoM-post* for *SET* requests that evict items conflicting with concurrent *GET* requests. This race condition exists because separate CPU threads

[3] GPU request processing latency is on the order of hundreds of microseconds. Hence, requests being concurrently processed on the GPU either belong to the same request batch or to a request batch launched at a similar point in time.

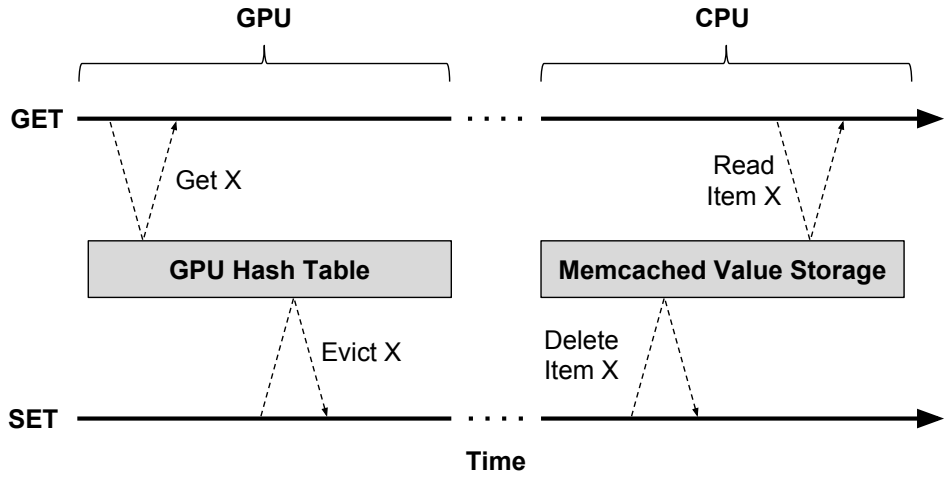


Figure 4.6: Race condition between dependent *SET* and *GET* requests in MemcachedGPU.

handle post GPU processing for *GET* and *SET* requests. Consider the example in Figure 4.6. Here, a *GET* request is accessing an item, X, which is concurrently being evicted by a *SET* request occurring later in time on the GPU. In this example, the *GET* request obtains the shared lock for the hash table set corresponding to item X and correctly retrieves the CPU item pointer from the GPU hash table. At a later time, the *SET* request obtains the exclusive lock for X and evicts X from the hash table. Any subsequent *GET* requests will correctly miss on X. However, the requests are then returned to the CPU for post processing on separate threads. As such, it is possible that the *SET* request thread responsible for evicting the item deletes the item value storage prior to when the *GET* request reads X's value to populate the response packet, as shown on the right side of Figure 4.6. This may result in the *GET* request accessing stale or garbage data.

Removing the race condition requires preserving the order seen by the GPU on the CPU. To accomplish this, each *GNoM-post* thread maintains a global completion timestamp (*GCT*), which records the timestamp of the most recent *GET* request batch to complete sending its response packets. This is the same timestamp used to update the hash table entry's last accessed time. If a *SET* request needs to evict an item, it records the last accessed timestamp of the to-be evicted

Table 4.1: Server and client configurations.

	Server	**Client**
Linux kernel	3.11.10	3.13.0
CPU	Intel Core i7-4770K	AMD A10-5800K
Memory	16 GB	16 GB
Network Interface	Intel X520-T2 10Gbps 82599 (modified driver v3.18.7)	Intel X520-T2 10Gbps 82599 (driver v3.18.20)

item from the GPU hashtable. After updating the GPU hash table, the *SET* request polls all *GNoM-post* GCT's on the CPU and stalls until they are all greater than its eviction timestamp before evicting the item. This ensures that all *GET* requests prior to the *SET* request have completed sending the response packets before the *SET* completes, preserving the order seen by the GPU. This stalling does not impact future *GET* requests since the *SET* allocates a new item prior to updating the hash table. Thus all *GET* requests occurring after the *SET* will correctly access the updated item. We have verified this mechanism by arbitrarily stalling *GET* request batches in *GNoM-post* while concurrently sending *SET* requests to update items conflicting with the *GET* requests, and ensuring the *SET* stalls until all conflicting *GET* requests complete.

An alternative method is to lazily delete items from the CPU Memcached value storage. Upon evicting an item from the GPU hash table, the *SET* request can add the evict operation to a queue on the CPU. At a later time, a separate CPU thread or *SET* request thread can clean up the items in this queue. The delay before garbage collecting the queue does not need to be long, since the only potential race condition candidates are those that accessed the GPU hash table prior to the corresponding *SET* request updated it, which should completely shortly following the *SET* request. As such, this delay could be set to a multiple of the worst case end-to-end *GET* request processing latency.

4.3 Experimental Methodology

This section presents our experimental methodology for evaluating *GNoM*, MemcachedGPU, and the modifications to the hash table design.

Table 4.2: Server NVIDIA GPUs.

GPU	Tesla K20c	Titan	GTX 750Ti
Architecture (28 nm)	Kepler	Kepler	Maxwell
TDP	225 W	250 W	60 W
Cost (2015)	$2700	$1000	$150
# CUDA cores	2496	2688	640
Memory size (GDDR5)	5 GB	6 GB	2 GB
Peak SP throughput	3.52 TFLOPS	4.5 TFLOPS	1.3 TFLOPS
Core frequency	706 MHz	837 MHz	1020 MHz
Memory bandwidth	208 GB/s	288 GB/s	86.4 GB/s

4.3.1 *GNoM* and MemcachedGPU

Unless stated otherwise, all of the hardware experiments in this chapter are run between a single Memcached server and client directly connected via two CAT6A Ethernet cables; one used for transmit and the other for receive. The main system configurations for the server and client systems are presented in Table 4.1. The GPUs evaluated in this chapter are shown in Table 4.2, all using CUDA 5.5. The high-performance NVIDIA Tesla K20c is evaluated using *GNoM*, which supports GPUDirect, and the low-power NVIDIA GTX 750Ti is evaluated using NGD. All three GPUs are evaluated in the offline limit study (Section 4.4.5). While Chapter 3 evaluated integrated AMD GPUs, this chapter focusses on discrete NVIDIA GPUs. Aside from the usage of GPUDirect on the Tesla K20c, there is no fundamental reason why *GNoM* would not also be beneficial on lower-power integrated GPUs. Additionally, our experiments in Section 5.6 highlight the potential for low-power discrete GPUs to achieve high performance with *GNoM*, and hence an important direction for future work is to evaluate integrated GPUs.

MemcachedGPU was implemented on top of Memcached v1.4.15. The hash table is configured as a 16-way set-associative hash table with 8.3 million entries assuming the maximum Memcached key-size. Note that this is only $\sim$46% of the maximum possible hash table size on the Tesla K20c given the 5GB global memory space and storage for the other *GNoM* and MemcachedGPU data structures. The hash table associativity was selected based on an offline hash table analysis

(Section 4.4.1) and an empirical performance analysis on the GPU.

The client system generates Memcached requests through the Memaslap microbenchmark included in libMemcached v1.0.18 [4]. Memaslap is used to stress MemcachedGPU with varying key-value sizes at different request rates. As described in more detail below (Section 4.3.2), we evaluate the effectiveness of the hash table modifications in MemcachedGPU on more realistic workload traces with different types of request distributions in Section 4.4.1 (zipfian, latest, and random distributions). The Memcached ASCII protocol is used with a minimum key size of 16 bytes (packet size of 96 bytes). The ASCII protocol places more stress on MemcachedGPU than the binary protocol, since this requires string processing and impacts the packet sizes. Larger Memcached value sizes impact both the CPU response packet fill rate and network response rate. However, we find that *GNoM* becomes network bound, not CPU bound, as value sizes increase. In our experiments, the larger value sizes result in larger packet sizes, which reduces the number of packets required to saturate the network bandwidth; hence lower processing requirements from MemcachedGPU. Thus, the smallest value size of 2 bytes is chosen to stress per packet overheads. Similarly, the smallest Memcached key size of 16 bytes is chosen in any experiments aimed at stressing *GNoM* and MemcachedGPU.

The single client system is unable to send, receive, and process the minimum sized Memcached packets at 10 Gbps with the Memaslap microbenchmark. As such, we created a custom client Memcached stressmark using *PF_RING* zero-copy [125] for sending and receiving network requests at 10 Gbps, and replay traces of *SET* and *GET* requests generated from Memaslap. With this method, the request ordering and key distributions are maintained with those generated by Memaslap, but are replayed at a much higher throughput. The MemcachedGPU server is initially warmed up with 1.3 million key-value pairs through TCP *SET* requests. Then, we send *GET* requests over UDP. However, processing all of the response packets on the client system still limits our send rate to $\sim$6 Gbps. To overcome this, we used a technique similar to [106], which forcefully drops response packets at the client using hardware packet filters at the NIC to sample a subset of packets. Using this technique, the client sends all of the packets to the server at the specified send rate and the server still performs all of the required per-packet op-

erations. The client NIC filter allows a subset of packets to flow back to the client for response processing, while the rest are dropped. The total number of received response packets are the sum of those received by the client and those dropped. Thus, our end-to-end latency experiments at high request rates measure a subset of the total response packets. However, because packets are processed in batches on the server, if we ensure that the subset of measured packets are distributed throughout a batch, these will be a good representation of all packets. If we instead create a static set of Memcached requests to repeatedly send to the server (e.g., 512 different requests), we are able to send and receive at the full rate at the client, since this frees up processing cycles on the client that were previously used to prepare each send packet. We use this static request technique to measure packet drop rates at the client more accurately. All other experiments use the Memaslap generated traces, as described above.

Power is measured using the Watts Up? Pro ES plug load meter [164] and measures the total system wall power for all configurations. An issue with PCIe BAR memory allocation between the BIOS and NVIDIA driver on the server system at the time this study was conducted restricted the NVIDIA Tesla K20c and NVIDIA GTX 750Ti GPUs from being installed in isolation. We measured the idle power of the Tesla K20c (18W) and GTX 750Ti (6.5W) using the wall power meter and the *nvidia-smi* tool. This inactive GPU idle power was subtracted from the total system power when running experiments on the other GPU. For example, when running an experiment on the Tesla K20c, the GTX 750TI's idle power of 6.5W was subtracted from the measured system power to calculate the total power. The GTX Titan did not have this issue and could be installed in isolation.

4.3.2 Hash-Sim (Hash Table Simulator)

To evaluate the impact of modifying the hash table structure, collision mechanism, and hash table eviction management in MemcachedGPU compared to the baseline Memcached hash table, we designed an offline hash table simulator, *hash-sim*. Hash-sim measures the hit rate for a trace of key-value *GET* and *SET* requests, which provides a platform for directly comparing different hash table and hash collision techniques. As in the baseline Memcached, a *GET* request miss triggers a

corresponding *SET* request for that item in hash-sim. Hash-sim uses the same Bob Jenkin's lookup3 hash function [86] included in the baseline Memcached.

We use a modified version of the Yahoo! Cloud Serving Benchmark (YCSB) [34] provided by MemC3 [52] to generate three Memcached request traces with different item access distributions: Zipfian, Latest, and Uniform. In the Zipfian (zipf) distribution, specific items are accessed much more frequently than other items. The Latest distribution is similar to the zipf distribution, except that the most recently inserted items are more popular. With Zipfian, the popularity of items are not affected by the insertion of new items. Finally, the Uniform distribution selects items at uniform random.

Hash-sim is single threaded, so no concurrency control is required. As such, the purpose of Hash-sim is not to measure the performance or scalability of the different hash table techniques in terms of lookups per unit time, but instead to measure the ability of the hash table to maximize the hit rate under different request distributions, hash table sizes, and eviction techniques. The hash table performance is considered independently from Hash-sim.

Lastly, MemcachedGPU implements a statically sized hash table to avoid dynamic memory allocation or resizing the hash table on the GPU. This differs from the baseline Memcached hash table with hash chaining, as well as other hashing techniques, which expand the hash table when hash chains become too long or a free hash table entry cannot be located when inserting an item. As a result, evictions occur in the hash table once the maximum number of items has been stored or a free entry is unavailable. As will be described further in Section 4.4.1, we evaluate global and local least recently used (LRU) techniques for managing item evictions. Global LRU considers all items in the hash table for eviction. Local LRU considers a subset of items in the hash table for eviction depending on the hash table structure and collision resolution technique.

4.4 Experimental Results

This section presents our evaluation of the Memcached hash table modifications, MemcachedGPU, *GNoM*, the non-GPUDirect (NGD) version of *GNoM*, and an offline limit study of *GNoM* and MemcachedGPU. Different experiments are per-

formed in simulation environments or on real hardware, as described in Section 4.3. If not explicitly mentioned, the experiment is evaluated on hardware.

4.4.1 Hash Table Evaluation

We first evaluate the impact of our modifications to the Memcached hash table on the *GET* request hit rate. The hash table modifications are required to improve the performance and scalability on GPU SIMD architectures. Every miss in the Memcached hash table results in an expensive access to the backing database and a potential *SET* request to update the hash table with the missing entry [4]. Thus, it is important to minimize the miss rate relative to the baseline Memcached hash table with hash chaining. Using Hash-sim (Section 4.3.2), we evaluate and compare the hit rate for multiple different hash table structures and techniques for handling collisions, which are more GPU-friendly than the baseline hash chaining technique. In addition to the hit-rate, each technique presents multiple trade-offs, for example, in the impact of the load factor on performance, requirement for dynamic memory allocation, worst-case number of accesses for *GET* requests, worst-case number of operations required to store an item, and concurrency control, which are important aspects to consider. Furthermore, some techniques may be better suited for the GPU's SIMD architecture than others.

Specifically, we evaluate four different hash table techniques and compare them with hash chaining:

- Hash chaining (HC): The default hash table technique used in the baseline Memcached. Keys are hashed and the item is inserted directly into the corresponding hash table entry. On a hash table collision, a new entry is dynamically allocated and linked into the hash table entry via a linked list. When searching for an item, the entire (potentially long) chain must be searched for the corresponding entry. A global least-recently used (LRU) queue is maintained to handle hash table evictions, such that all items in the hash table are considered for eviction based on their global ordering of usage. In our implementation, hash chaining is always able to store the maximum number of

[4] This is dependent on the application's implementation, as the application is responsible for managing when to store items in the hash table.

items in the hash table, which is equal to the size of the hash table, regardless of the number of collisions on a given entry.

- Set-associative (SA): The fixed-size hash table technique used in MemcachedGPU (similar to that proposed in [106] for CPUs and [26] for FPGAs). This technique is equivalent to a regular set-associative cache, which results in fast look-ups and insertions. The hash table is broken down into multiple hash table *sets*, which consist of a fixed number of hash table entries (the set size). Each hash value now corresponds to multiple hash table entries, which need to be searched for a matching or free entry. Only the fixed set size needs to be traversed when searching for an item. Items are inserted into an empty entry in the corresponding hash set. If no empty entries are available, an item must be evicted. A local LRU is maintained per hash table set, such that an eviction only selects from the items belonging to the hash table set corresponding to the hash value for a given key.

- Hopscotch (HS) [68]: Each entry in the hopscotch hash table has a corresponding hopscotch bucket, which consists of H consecutive entries. As such, each hash table entry/group overlaps with $H\text{-}1$ other hopscotch groups. When searching for an item, only the fixed hopscotch group size must be searched. When trying to insert an element into the hash table to entry X, similar to linear probing, consecutive entries are searched from X until a free entry is found at index Y. If the distance between the free entry at Y and X is less than $H\text{-}1$, the item is directly inserted into that entry in the hash table, otherwise the hash table must be reorganized to make room for the new value within X and $X+(H\text{-}1)$ (the hopscotch group size). This is achieved by performing a linear search to find the first free entry and then iteratively moving hash table entries to the free location in their hopscotch group in a reverse direction from $X+Y$ to X until we can insert X in its hopscotch group. If no free entries are found or the entries can not be reorganized, the hash table must be resized or an item must be evicted. In our implementation, we limit the search size to 512 entries such that for any given insertion, the search is limited between X and X+512 for a free entry. If no free entries are found, then the LRU item in the hopscotch group for item X (X to $X+H$) is evicted

and the item is directly inserted into this entry.

- Strided linear probing (SLP): Linear probing is similar to hash chaining, except that the chain consists of the consecutive hash table entries instead of dynamically allocated entries. In the strided implementation, each hash table entry contains N additional entries with some stride S. When inserting an item at entry X, we search from X to $X+(N\times S)$ to find a free entry. For example, if an item maps to the entry at X, the next entry is at $X+S$, and the last entry in the group is at $X+(N\times S)$. Similar to hopscotch hashing, each group overlaps with multiple different groups. Similar to the set-associative hash table, the hash table is not reorganized when no free entries are found. Instead, the LRU item belonging to the group entry X is evicted and the new item is inserted. When searching for an item, only the fixed-size group needs to be searched.

- Cuckoo hashing (CU) [141]: In Cuckoo hashing, each item can map to two different hash table entries. Two hash functions are used to select these entries. We follow the same methodology as MemC3 [52], which generates the first hash value using the baseline Memcached's Bob Jenkin's hash function, and performs a set of operations on the first hash index, using the hash constant from MurmerHash2, to generate the second hash index. When searching for a item, only the two entries need to be accessed, which limits the number of hash table accesses for *GET* requests. However, when inserting an item into the hash table, if the two entries are not empty, the hash table needs to be reorganized. This is achieved by recursively traversing each item's alternative hash table entry until a free entry is found, and then moving the items to the alternative hash entry in a reverse direction until one of the two entries for the current item is now free. Similar to our hopscotch hashing implementation, we limit this search to 512 entries. If no free entries are found within 512 searches, we evict the LRU item within this chain of items and reorganize the hash table accordingly. MemC3 [52] includes an additional optimization, which combines Cuckoo hashing with the set-associative hash table. Here, each hash table entry consists of four entries. We did not evaluate this optimization, but it should be noted that the

performance measured in this section will be lower than that achievable in MemC3's implementation.

For each hash table implementation, we fix the maximum number of key-value items to store, independent of the size of the corresponding values for each key. Hash chaining is always able to store the maximum number of elements as a new entry is dynamically allocated on a hash collision. Evictions only occur when the maximum number of items have been stored in the hash table. For all other hash table techniques, evictions may occur even with fewer elements than the maximum due to the conflict resolution techniques. This results in additional evictions compared to hash chaining, which needs to be minimized to reduce any extra expensive accesses to the back-end databases. MICA [106] also proposed an enhancement to avoid hash table evictions by including overflow bins, which are used to store items when the main sets overflow. An error is returned if no such overflow bins are available. MICA refers to hash tables with evictions as *lossy* and hash tables without evictions as *lossless*. In this work, we only consider *lossy* versions of hash tables due to the GPU's limited DRAM capacity and the expectation that the hash table can be sized large enough to capture the application's temporal locality.

For each workload distribution, we generate a runtime trace of 10 million key-value pairs with 95% *GET* requests and 5% *SET* requests. The hash tables are first warmed up using a *load* trace of all *SET* requests, and then the hit-rate is measured on a separate *transaction* trace. The hash tables are configured as follows: the hash chaining (HC) hash table can store any number of items in each chain, up to the maximum size of the hash table; the hopscotch (HS) hash table has a hopscotch group of 16 entries and a maximum search size of 512 entries for the linear probe; the strided linear probe (SLP) hash table has a group of 16 entries with a stride of 4 entries; The set associative (SA) hash table has a set size of 16 entries; finally, the cuckoo (CU) hash table has two entries per item (corresponding to the two hash functions) and a maximum depth of 512 for the recursive reorganization.

Figure 4.7a, Figure 4.7b, and Figure 4.7c measure the hit-rate for the five hash tables under the Zipfian (Zipf), Latest (Lat), and Uniform (Uni) distributions respectively. The x-axis shows different hash table capacities from 2 million entries to 16 million entries. With a request trace working size of 10 million items,

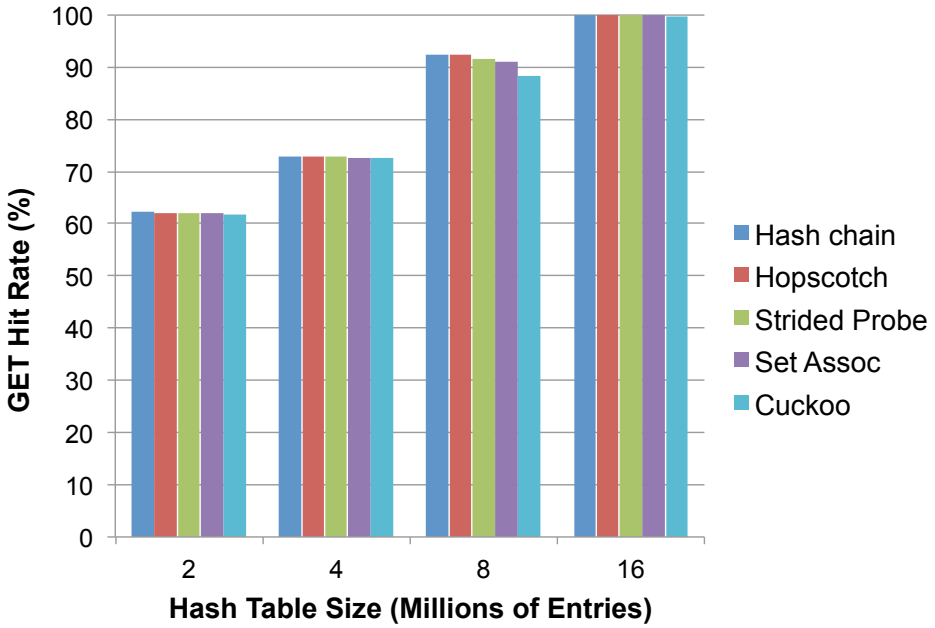


Figure 4.7a: Zipfian: Comparing the hit rate for different hash table techniques and sizes under the Zipfian request distribution. The request trace working size is 10 million entries.

HC is always achieve 100% hit-rate when the hash table is larger than 10 million entries, equivalent to a fully-associative cache. However, this may result in long hash chains based on the collision rate. There are three notable points from this experiment. First, the hash tables perform much better on the Zipf and Lat distributions than Uni. Since the hash tables implement a LRU eviction policy, the most most frequently accessed items are prevented from commonly becoming the LRU item. Similarly, Lat performs better than Zipf, as the most recently added item becomes the most frequently accessed item. Uni, however, suffers from low hit rates with smaller hash table sizes, as there is no temporal locality in the trace.

Second, all hash tables achieve similar hit rates to HC. For all hash table sizes less than the working set, HS, SLP, and SA achieve over 99% of the hit rate of HC on average, while CU averages over 97% of the HC hit rate. The difference between the techniques is most noticeable at a hash table size close to the working set size (e.g., 8 million entries compared to 10 million entries). As noted above,

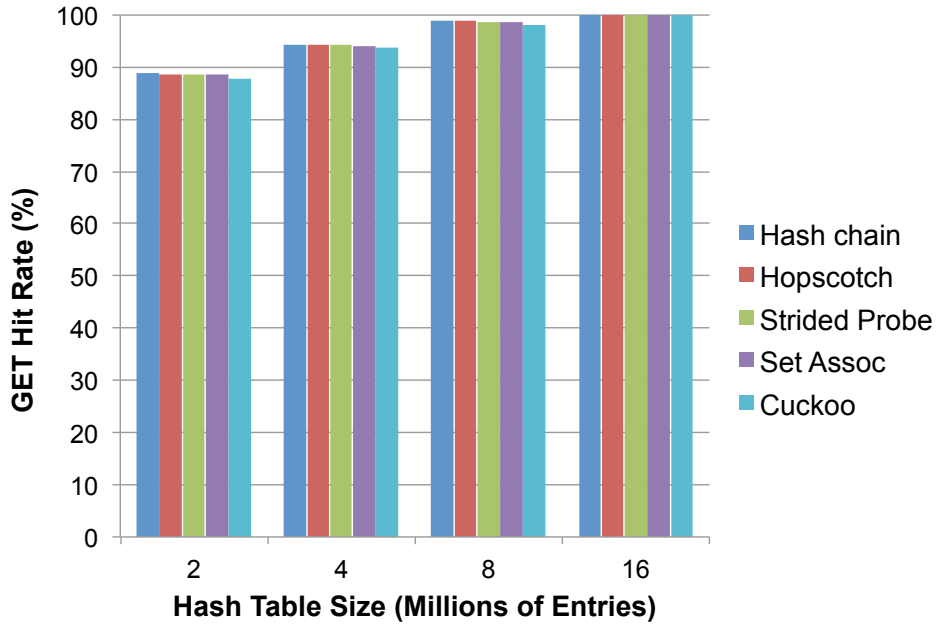


Figure 4.7b: Latest: Comparing the hit rate for different hash table techniques and sizes under the Latest request distribution. The request trace working size is 10 million entries.

CU can benefit from adding a small set (e.g., four entries) to each hash table entry; however, we did not evaluate this enhancement. In our experiments, HS was the best performing alternative. Similar to SLP and SA, HS contains multiple possible entries per hash value (16). However, unlike these techniques, HS is also able to reorganize the hash table when the local group becomes full, hence avoiding an eviction on the collision. CU is also able to reorganize the hash table, but we found that limiting the potential candidate entries for each item to two, instead of 16, was a limiting factor. While there is some loss in hit rate relative to HC, these results highlight that for the dataset evaluated, there is little benefit to maintain global LRU and global evictions over the local counterparts. As real workloads tend to follow a Zipfian-like distribution [34], where some items are accessed very frequently, while most others are accessed infrequently, we believe that this property will hold for other workloads.

Lastly, while SA is not the best performing alternative, it is able to achieve a

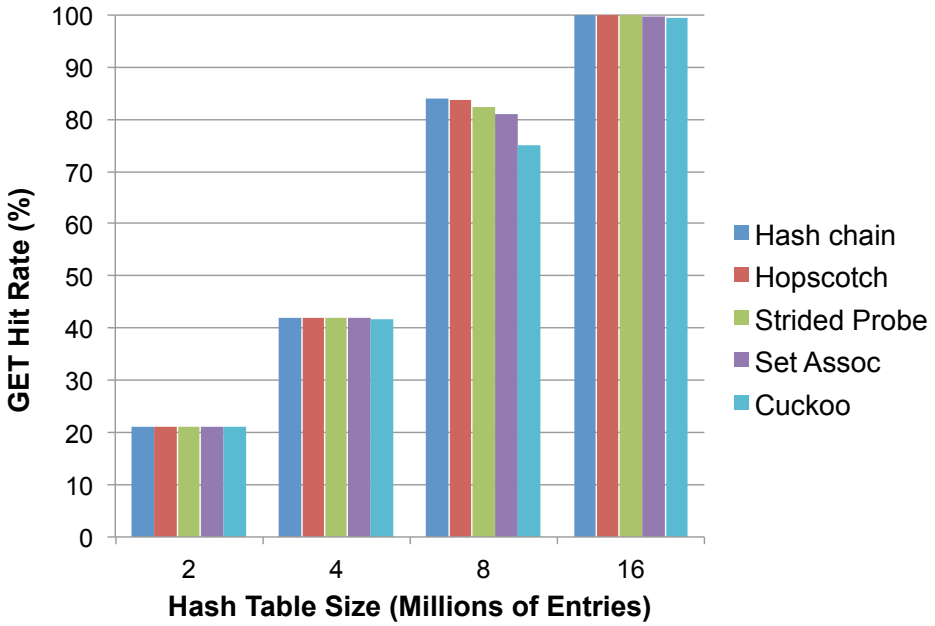


Figure 4.7c: Uniform Random: Comparing the hit rate for different hash table techniques and sizes under the Uniform Random request distribution. The request trace working size is 10 million entries.

hit rate within 96.6% - 99.99% of HC across the different distributions and hash table sizes. Furthermore, SA contains two main benefits over the alternatives. First, SA simplifies the locking mechanism required under concurrent accesses (Section 4.2.2) compared to the other hash table techniques. In SA, each set contains a single lock. As such, only a single lock needs to be acquired when accessing all elements in a set. This is possible since there is a one-to-many mapping between each hash table set and entry - each entry only belongs to a single set, while a single lock can lock all entries within that set. However, hash table entries in HS, SLP, and CU belong to multiple different overlapping groups, requiring acquiring multiple locks when accessing the group of entries corresponding to a given item. This increases the chance of threads in a warp blocking other threads, leading to reduced SIMD efficiency and performance. Additionally, the storage requirements for the locks are reduced in SA, since there are fewer sets than individual entries or overlapping groups. Second, SA achieves fast insertions even under high collision

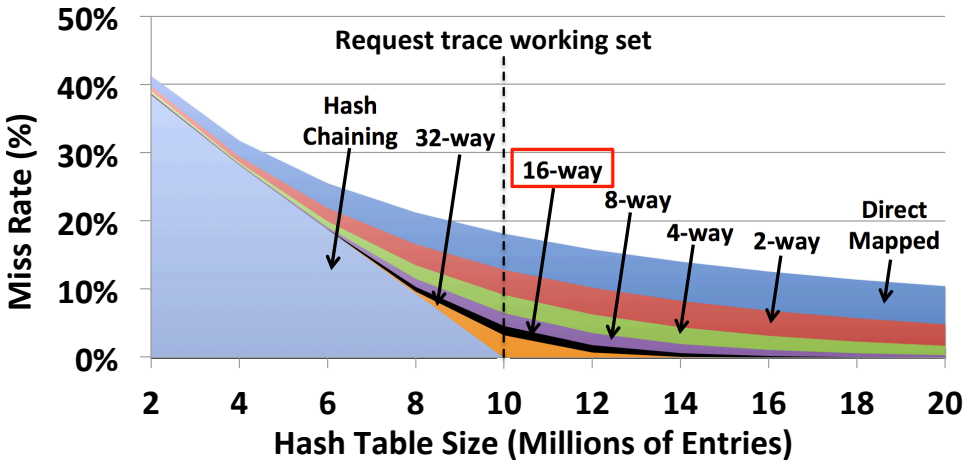


Figure 4.8: Miss-rate versus hash table associativity and size compared to hash chaining for a request trace with a working set of 10 million requests following the Zipf distribution.

rates. HS and CU require searching and reorganizing multiple entries in an attempt to avoid evicting an item from the hash table. While this can improve the storage efficiency under high collision rates, it comes at the cost of increased complexity and insertion times. From Figure 4.7a, Figure 4.7b, and Figure 4.7c, we see that the improvements in the hit rate of these techniques are small compared to SA.

As a result of the above experiments, we selected the set associative as the hash table for MemcachedGPU.

Next, we evaluate the miss rate of SA with different set sizes and hash table sizes under the Zipf distribution. The same request trace used in the previous experiment, with 10 million requests containing 95% *GET* and 5% *SET* requests, is used to evaluate the miss rate. The results are shown in Figure 4.8 and are compared to HC, which is equivalent to a fully associative hash table. For example, with a maximum hash table size of 8 million entries, SA has a 21.2% miss-rate with 1-way (direct mapped) and 10.4% miss-rate with 16-ways. HC achieves a 0% miss-rate when the hash table size is larger than the request trace (dotted line at 10 million entries) since it is able to allocate a new entry for all new key-value pairs. At smaller hash table sizes, none of the configurations are able to effectively capture the locality in the request trace, resulting in comparable miss-rates. As the

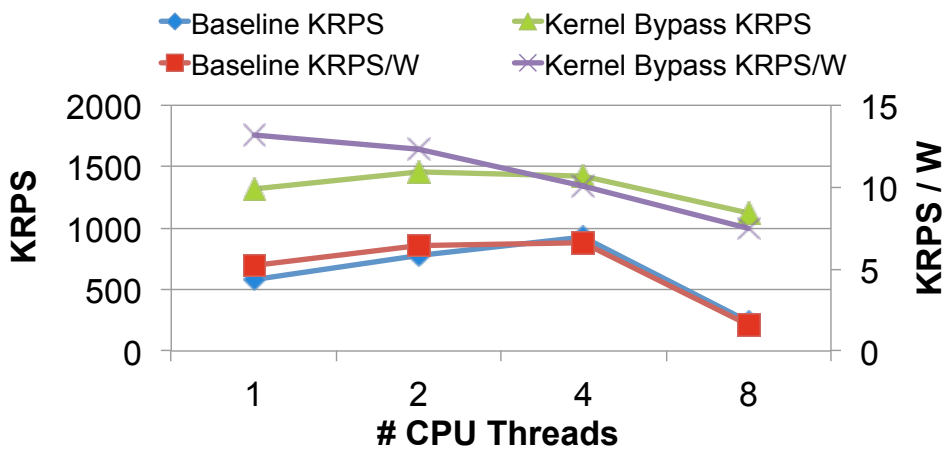


Figure 4.9: Impact of Linux Kernel bypass for *GET* requests vs. the baseline Memcached v1.5.20.

hash table size increases, increasing the associativity decreases the miss-rate. At 16-ways for all sizes, SA achieves a minimum of 95.4% of the HC hit-rate for the Zipf distribution. The other distributions follow similar trends with different absolute miss-rates. However, increasing the associativity also increase the worst-case number of entries to search on an access to the hash table. From experimentation, we empirically find that an associativity of 16-ways provides a good balance of storage efficiency and performance.

4.4.2 Impact of Linux Kernel Bypass

MemcachedGPU uses *GNoM* on receive and *PF_RING* on transmit to bypass both the Linux kernel and *libevent* (used by default for Memcached). To understand the impact of this optimization in isolation, we evaluate *PF_RING* for both receive and transmit in the baseline Memcached v1.5.20 running only on the CPU. Others have attained higher throughputs for Memcached on CPU-only systems using many other optimizations [52, 105, 106, 177]. We compare MemcachedGPU against published results in Section 4.4.6.

Figure 4.9 measures the improvement in throughput and energy-efficiency of bypassing the Linux kernel over the baseline Memcached using *PF_RING*. No other optimizations were applied. Removing the Linux network stack results in

Table 4.3: *GET* request throughput and drop rate at 10 GbE.

Key Size	16 B	32 B	64 B	128 B
Tesla drop rate server	0.002%	0.004%	0.003%	0.006%
Tesla drop rate client	0.428%	0.033%	0.043%	0.053%
Tesla MRPS/Gbps	12.92/9.92	11.14/9.98	8.66/9.98	6.01/10
Maxwell-NGD drop rate server	0.47%	0.13%	0.05%	0.02%
Maxwell-NGD MRPS/Gbps	12.86/9.87	11.06/9.91	8.68 10	6.01/10

a minimum improvement in both throughput and energy-efficiency over the baseline of 1.5X at 4 threads, and a maximum of 1.5 MRPS and 12.3 KRPS/W (1.9X) at 2 threads. Adding threads to the Kernel bypass has a larger increase in energy consumption relative to the increase in throughput, resulting in the continuously decreasing energy-efficiency. Increasing the number of threads beyond number of cores (4) results in a drop in throughput and energy-efficiency for both configurations. These results highlight that while bypassing the Linux kernel can provide a sizeable increase in performance and energy-efficiency, additional optimizations to the core Memcached implementation are required to continue improving efficiency and scalability.

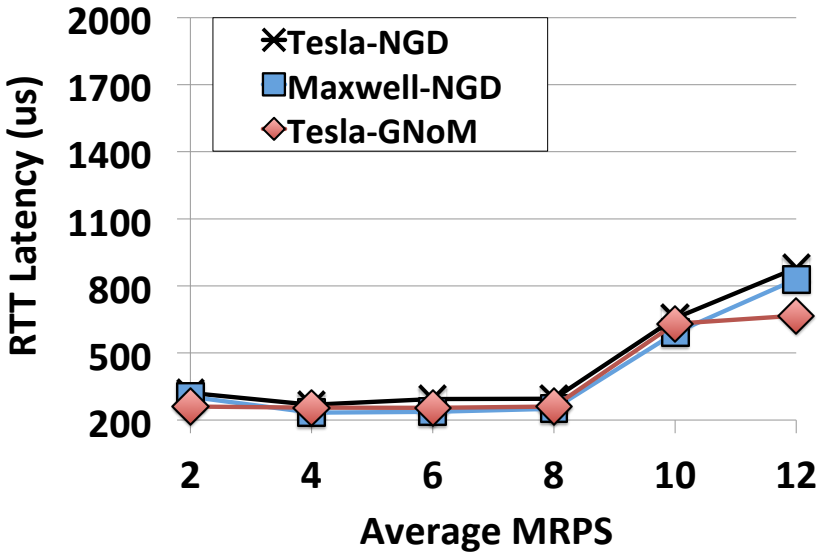
4.4.3 MemcachedGPU Evaluation

Next, we evaluate the full end-to-end MemcachedGPU on the high-performance NVIDIA Tesla K20c (Tesla) using *GNoM* with GPUDirect, and on the low-power NVIDIA GTX 750Ti (Maxwell) using the non-GPUDirect (NGD) framework (Section 4.1.2). Throughput is measured in millions of requests per second (MRPS) and energy-efficiency is measured in thousands of requests per second per Watt (KRPS/W). For latency experiments, the 8 byte Memcached header is modified to contain the client's send timestamp and measures the request's complete round trip time (RTT).

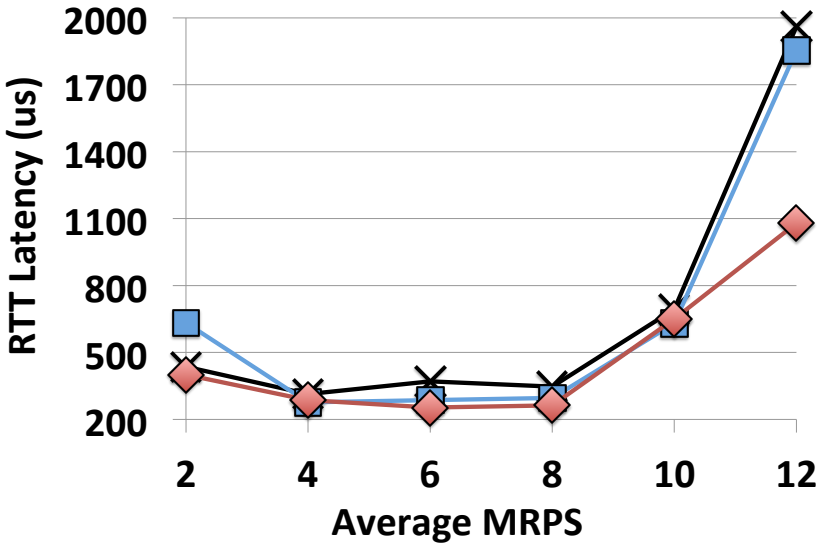
Throughput: Memcached typically uses UDP for *GET* requests and tolerates dropped packets by treating them as misses in the caching layer. However, excessive packet dropping mitigates the benefits of using Memcached. We measure the

packet drop rate at the server and client for packet traces of 500 million requests with equal length keys at peak throughputs, averaging over three runs. The impact of having different key lengths on SIMD efficiency is discussed later in this section. As shown in Table 4.3, MemcachedGPU is able to processes packets near 10 GbE line-rate for any Memcached request size with server packet drop rates $< 0.006\%$ (*GNoM*) and $< 0.5\%$ (NGD). The client drop rates are measured using the static trace described in Section 4.3 with no packet loss at the server. Increasing to the full 13 MRPS at 16B keys increases the server drop rate due to the latency to process the packets and recycle the limited number of GRXBs to the NIC for new RX packets. As a result, there are no available GRXBs when a new packet arrives and the NIC drops the packet accordingly. In Section 4.4.5 we evaluate the peak throughputs of MemcachedGPU assuming an unlimited number of GRXBs through an offline analysis.

RTT Latency: For many scale-out workloads, such as Memcached, the longest latency tail request dictates the total latency of the task [41]. While the GPU is a throughput-oriented accelerator, we find that it can provide reasonably low latencies under heavy throughput. Figure 4.10 measures the mean and 95-percentile ($p95$) client-visible RTT versus request throughput for 512 requests per batch on the Tesla using *GNoM* and NGD, and the Maxwell using NGD. Recall that NGD must copy the packet from the NIC to CPU memory and then from CPU memory to GPU memory. As expected, *GNoM* has lower latencies than NGD (55-94% at $p95$ on the Tesla) by reducing the number of memory copies on packet RX with GPUDirect. The latency increases as the throughput approaches the 10 GbE line-rate, with the $p95$ latencies approaching $1.1ms$ with *GNoM* and $1.8ms$ with NGD. We also evaluated a smaller batch size of 256 requests on *GNoM* and found that it provided mean latencies between 83-92% of 512 requests per batch when less than 10 MRPS, while limiting peak throughput and slightly increasing the mean latency by $\sim$2% at 12 MRPS. Although the smaller request batch sizes reduces the batching and potentially the kernel processing latency, it also increases the kernel launching and post processing overhead as these tasks are amortized over fewer packets. At lower throughputs (< 4 MRPS), we can see the effects of the batching delay on the $p95$ RTT (Figure 4.10b). For example, at 2 MRPS with a batch size of 512 requests, the average batching delay per request is already $128\mu s$, compared



(a) Mean RTT.



(b) 95-percentile RTT.

Figure 4.10: Mean and 95-percentile round trip time (RTT) latency versus throughput for Tesla GPU with *GNoM* and NGD, and Maxwell with NGD.

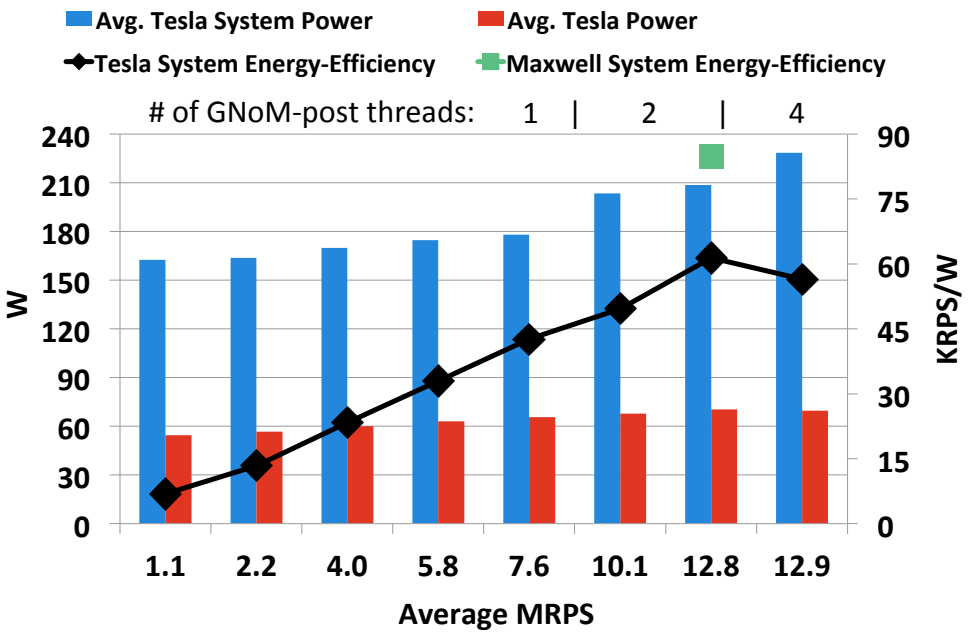


Figure 4.11: Total system and GPU power (left axis) and total system energy-efficiency (right axis) versus throughput for MemcachedGPU and *GNoM* on the NVIDIA Tesla K20c. The total system energy-efficiency for the Maxwell system is also shown at the peak throughput with two *GNoM-post* threads. The number of *GNoM-post* threads are shown above the graph, with 1 thread for 1.1 to 7.6 MRPS, 2 threads for 10.1 to 12.8 MRPS, and 4 threads for 12.9 MRPS.

to 32μs at 8 MRPS. While not shown here, a simple timeout mechanism can be used to reduce the impact of batching at low request rates by launching partially full batches of requests. The *GNoM* GPU kernel can either reduce the size of the kernel or use conditional operations to mask off unused threads accordingly.

Energy-Efficiency: Figure 4.11 plots the average power consumption, both the full system power and GPU power, and the full system energy-efficiency of MemcachedGPU and *GNoM* on the Tesla K20c at increasing request rates. The total system energy-efficiency for the Maxwell system is also shown at the peak throughput with two *GNoM-post* threads. The Tesla K20c power increases by less than 35% when increasing the throughput by $\sim$13X (1 to 13 MRPS), leading to the steady increase in energy-efficiency. The jumps in system power at 10.1 and 12.9

MRPS are caused by adding *GNoM-post* threads for post processing and recycling the GRXBs fast enough to maintain low packet drop rates at the higher throughputs. However, increasing the number of *GNoM-post* threads from two to four decreases energy-efficiency as the system power is increased by 9% while the throughput has a much smaller improvement. At peak throughputs, the low-power GTX 750Ti using NGD consumes 73% of the total system power consumed by the Tesla K20c using *GNoM* (151.4 W at 84.8 KRPS/W). These results highlight the importance of the CPU-side framework in contemporary systems for handling higher request rates, as well as the negative impact that including additional CPU resources has on energy-efficiency. Furthermore, placing increased load on the CPU reduces the potential for CPU workload consolidation and reduces the peak GPU performance, since the CPU may not be able to keep up with the requirements from multiple tasks (Chapter 5).

The Tesla K20c consumes roughly one third of the total system power. Note that the peak GPU power of 71W is less than 32% of the K20c's TDP (225W), suggesting low utilization of the total GPU resources. This also contributes to why the lower-power, lower-performance GTX 750Ti is able to handle the high request rates. The Tesla K20c system has an idle power of 84W without any GPUs. Thus, *GNoM-host* consumes roughly 15%, 25%, and 33% of the total system power when using one, two, or four *GNoM-post* threads respectively. Much of this is an artifact of GPUs being offload-accelerators, which rely on the CPU to communicate with the outside world. This leaves large opportunities to further improve the energy-efficiency of *GNoM* through additional hardware I/O and system software support for the GPU.

Branch Divergence: Next, we evaluate the impact of branch divergence on performance in MemcachedGPU, which stems from each GPU thread handling a separate *GET* request. For example, differences in key lengths, potential hits on different indices in the hash table set, or misses in the hash table can all cause GPU threads to perform a different number of iterations or execute different blocks of code at a given time. Each of these scenarios reduce the SIMD efficiency and consequently performance. Figure 4.12 plots the average peak throughput as a fraction of the theoretical peak throughput for a given key length distribution. We find that the throughput performs within 99% of the theoretical peak, regardless of the key

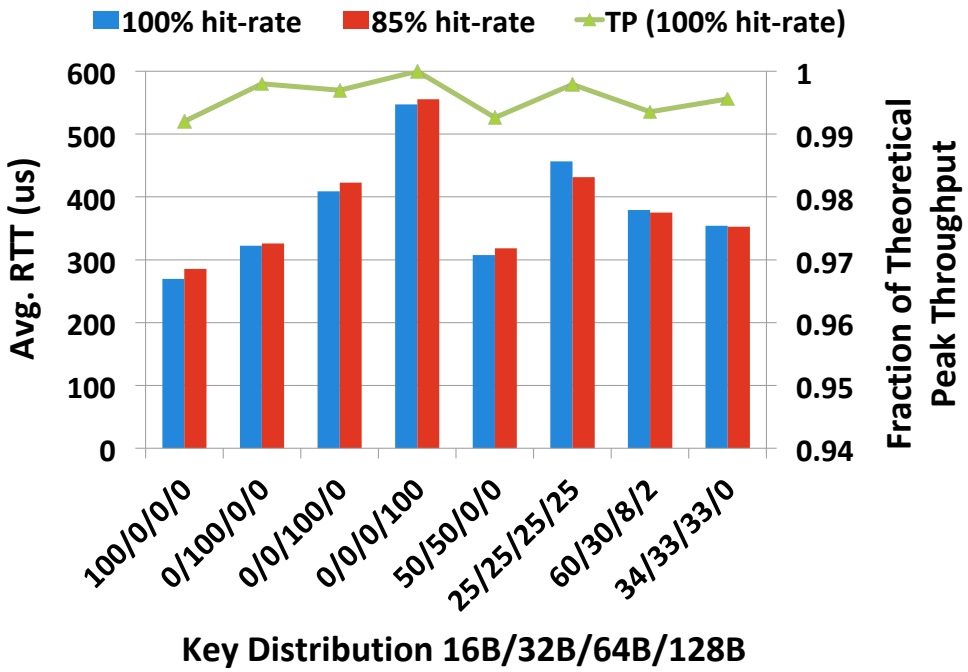


Figure 4.12: Impact of varying the key length mixture and hit-rate on round trip time (RTT) latency (left axis) and throughput (right axis) for MemcachedGPU and *GNoM* on the NVIDIA Tesla K20c. RTT latency is measured at 4 MRPS. Throughput is shown as the average fraction of peak throughput (at 10 Gbps) obtained for a given key distribution. The key distributions are broken down into four sizes (16B, 32B, 64B, and 128B) and the labels indicate the percentage of keys with the corresponding length.

distribution. That is, even when introducing branch divergence, MemcachedGPU becomes network bound before compute bound.

Figure 4.12 also plots the average RTT for *GET* requests at 4 MRPS under different distributions of key lengths and at 100% and 85% hit-rates. Note that this experiment still consists of 100% *GET* requests. The results match the intuition that because there is no sorting of key lengths to batches, the latency should fall somewhere between the largest and smallest key lengths in the distribution. For example, 50% 16 byte and 50% 32 byte keys have an average RTT between 100% 16 byte and 100% 32 byte key distributions. If there are large variations in key

Table 4.4: Concurrent *GET*s and *SET*s on the Tesla K20c.

GET MRPS (% peak)	7 (54)	8.8 (68)	9.7 (74)	10.6 (82)	11.7 (90)
SET KRPS (% peak)	21.1 (66)	18.3 (57)	18 (56)	16.7 (52)	15.7 (49)
SET:*GET* Ratio	0.3%	0.21%	0.19%	0.16%	0.13%
Server Drop Rate	0%	0.26%	3.1%	7.5%	8.8%

lengths and tight limits on RTT, the system may benefit from a pre-sorting phase by key length such that each request batch contains similar length keys. This could help reduce RTT for smaller requests, however, the maximum throughput is still limited by the network.

Typical Facebook Memcached deployments have hit-rates between 80-99% [16]. Figure 4.12 also measures the impact on RTT under 85% and 100% hit-rates. As can be seen, there is little variation between average RTT with different hit-rates. While reducing the hit-rate forces more threads to traverse the entire hash table set (16 entries), the traversal requires a similar amount of work compared to performing the key comparison on a potential hit.

GETs and SETs: While the main focus of MemcachedGPU is on accelerating *GET* requests, we also evaluate the throughput of the current naive *SET* request handler and its impact on concurrent *GET* requests. *SET* requests are sent over TCP for the same packet trace as the *GET*s to stress conflicting locks and update evictions. The maximum *SET* request throughput is currently limited to 32 KRPS in MemcachedGPU, $\sim$32% of the baseline. This is a result of the naive *SET* handler described in Section 4.2.1, which serializes *SET* requests. However, this is not a fundamental limitation of MemcachedGPU as, similar to *GET* requests, *SET* requests could also be batched together on the CPU prior to updating the GPU hash table. Unlike *GET* requests, however, each *SET* requests would need to be serialized or processed per GPU warp instead of per thread to avoid potential deadlocks on the exclusive locks (Section 4.2.2). Improving *SET* support is left to future work. Table 4.4 presents the *GET* and *SET* request throughputs, resulting *SET*:*GET* ratio, and server packet drop rate of MemcachedGPU on the Tesla K20c. As the *GET* request rate increases, the *SET* rate drops due to contention for GPU resources. The low peak *SET* request throughput limits the *SET*:*GET* ratio

to <0.5% for higher *GET* request rates. The average server packet drop rate approaches 10% when increasing *GET* throughput to 90% of the peak, which limits the effectiveness of MemcachedGPU. *GET* requests at ∼9 MRPS maintains comparable drop rates to the peak throughput, while also handling 0.26% *SET* requests.

4.4.4 Workload Consolidation on GPUs

Workload consolidation, running multiple workloads concurrently on the same hardware, improves datacenter utilization and efficiency [20]. While specialized hardware accelerators, such as ASICs or FPGAs, can provide high efficiency for single applications, they may reduce the flexibility gained by general-purpose accelerators, such as GPUs. For example, long reconfiguration times of reprogrammable hardware, milliseconds to seconds [142, 147], may mitigate the benefits gained by the accelerator when switching between applications. In this section, we evaluate the potential for workload consolidation on GPUs, which may provide advantages over other hardware accelerators in the datacenter.

However, at the time of this study, the evaluated GPUs do not support preemptive [163] or spatial [2] multitasking for GPU computing although they do support preemptive multitasking for graphics [132]. When multiple CUDA applications run concurrently, their individual CUDA kernel launches contend for access to the GPU and, depending on resource constraints, are granted access in a first-come, first-serve basis by the NVIDIA driver. Large CUDA kernels with many CTAs may consume all of the GPU resources, blocking other CUDA kernels from running until completed. However, we can potentially exploit this property through a simple approach to enable finer grained multitasking by splitting a single CUDA kernel into multiple kernels with fewer CTAs.

We study a hypothetical low-priority background task (BGT) that performs a simple vector multiplication in global memory requiring a total of 256 CTAs with 1024 threads each to complete. The low-priority BGT is divided into many smaller short running kernel launches, which can be interleaved with MemcachedGPU processing. This creates a two-level, software/hardware CTA scheduler. For example, if we reduce the background task to 16 CTAs per kernel, we require 16 separate kernel launches to complete all 256 CTAs in the task (16 CTAs / kernel launch ×

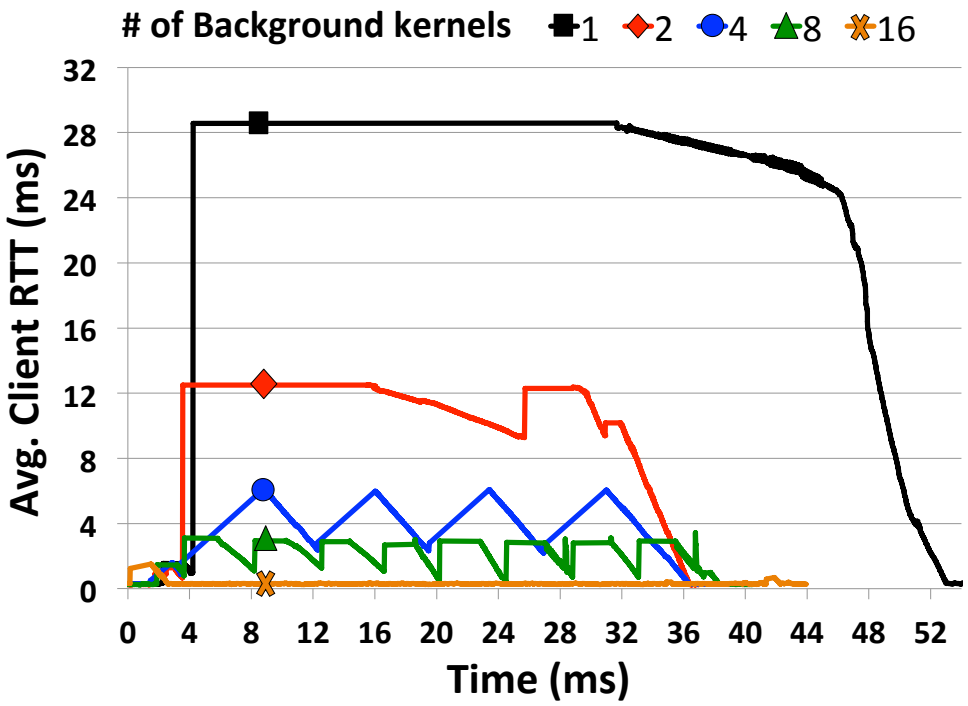


Figure 4.13: Client RTT (avg. 256 request window) during BGT execution for an increasing number of fine-grained kernel launches.

16 kernel launches = 256 CTAs).

We run MemcachedGPU using *GNoM* on the Tesla K20c at a lower *GET* request throughput of 4 MRPS using 16 byte keys. After some time, the BGT is launched concurrently on the same GPU, varying the number of CTAs per kernel launch. Figure 4.13 measures the average client RTT during the BGT execution. The average RTT is computed on a window of 256 *GET* request responses at the client. Without the BGT, MemcachedGPU has an average RTT $< 300\mu$s. With 256 CTAs (1 background task kernel launch), the BGT consumes the GPU resources causing a large disruption in the Memcached RTT. Even after the BGT completes with 256 CTAs (around 32*ms*), MemcachedGPU takes over 20*ms* to return back to the original average RTT. As the number of CTAs per kernel is reduced, the impact of the BGT on MemcachedGPU reduces significantly. For example, at 16 CTAs per kernel, the RTT experiences a short disruption for $\sim$2.4*ms* during the initial

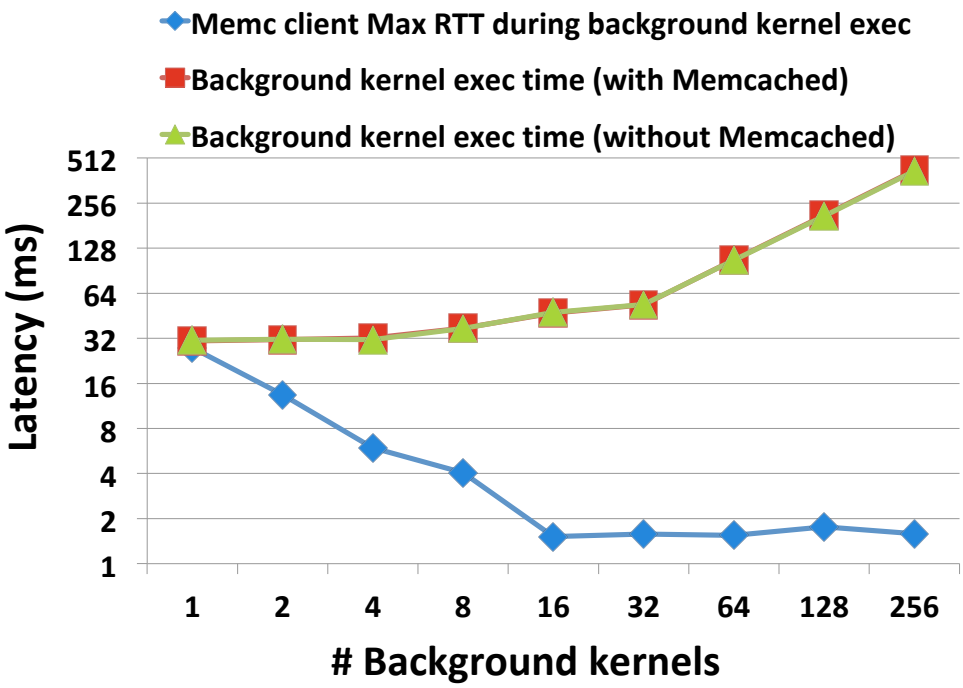


Figure 4.14: Impact on BGT execution time with an increasing number of kernel launches and max client RTT during BGT execution.

BGT kernel launch with a maximum average RTT of $\sim$1.5ms during this time, and then returns back to under 300μs while the BGT is executing. The other BGT configurations show multiple spikes in the RTT corresponding to the launches of the subsequent BGT kernels. It is interesting to note that 4 BGT launches (64 CTAs per kernel) results in a saw tooth RTT instead of the sharp spikes in RTT seen in the other configurations; however, we did not investigate the cause of this further.

While decreasing the size of BGT kernel launches reduces the impact on MemcachedGPU's RTT, increasing the number of BGT kernel launches also increases the BGT execution time. Figure 4.14 measures the BGT execution time with and without MemcachedGPU, as well as the maximum average RTT seen by MemcachedGPU during the BGT execution. At 4 MRPS, MemcachedGPU has very little impact on the BGT execution time due to its low resource utilization and small kernel launches. As the number of BGT kernels increases (more smaller BGT kernel launches), the execution time also increases due to the introduction of

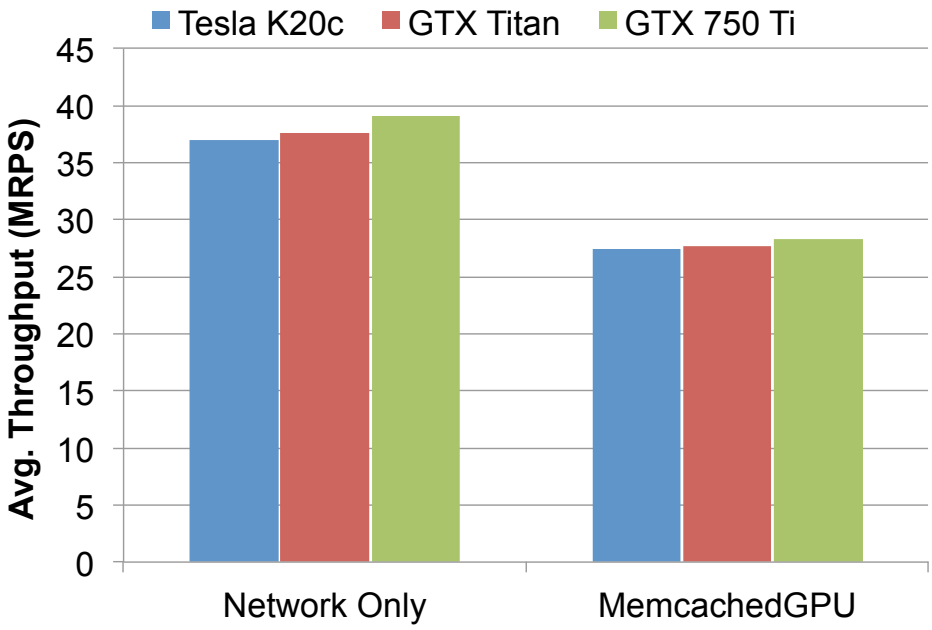


Figure 4.15: Offline *GNoM* throughput - 16B keys, 96B packets.

the software CTA scheduler and contention with competing kernel launches. However, the impact on MemcachedGPU RTT decreases much faster. At 16 CTAs per kernel, the BGT execution time is increased by ∼50% versus 256 CTAs, while the MemcachedGPU RTT is reduced by over 18×. Allowing for an increase in the lower-priority BGT completion time, *GNoM* is able to provide reasonable QoS to MemcachedGPU while running other applications on the same GPU.

4.4.5 MemcachedGPU Offline Limit Study

In this section, we evaluate an offline, in-memory framework that reads network request traces directly from CPU memory to evaluate the peak performance and efficiency of *GNoM* and MemcachedGPU, independent of the network. The same *GNoM* framework described in Section 4.1.2 is used to launch the GPU kernels (*GNoM-pre*), perform GPU UDP and *GET* request processing (*GNoM-dev*), and populate dummy response packets upon kernel completion (*GNoM-post*). However, unlike the actual *GNoM* framework, which uses GPUDirect to transfer pack-

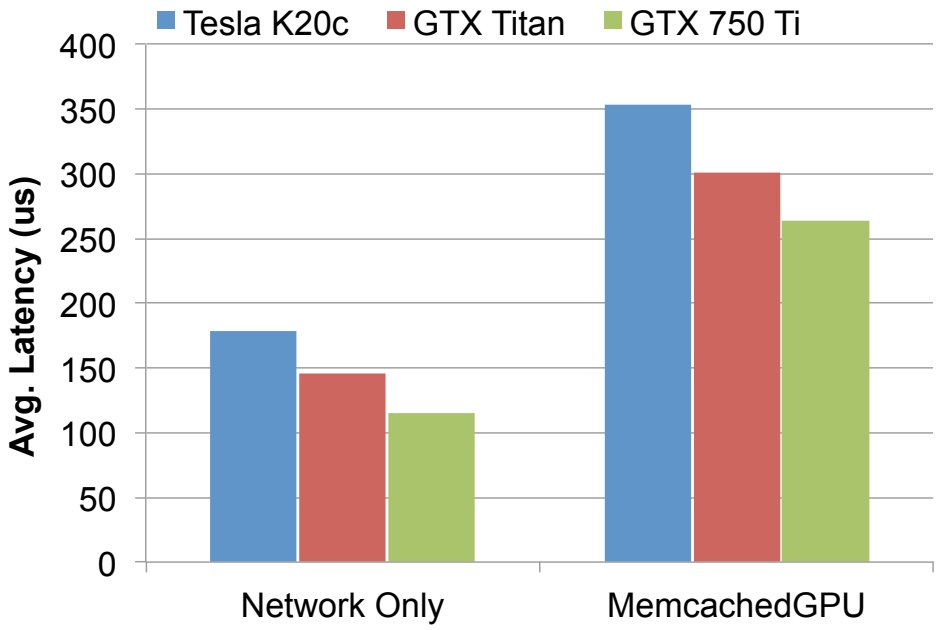


Figure 4.16: Offline *GNoM* processing latency - 16B keys, 96B packets.

ets from the NIC to the GPU one at a time as they arrive, the offline framework reads packets from CPU memory and bulk transfers request batches to the GPU across the PCIe bus. The same packet trace used in Section 4.4.3, with the minimum key size of 16 bytes to stress *GNoM*, is used in the offline evaluation. We also evaluate a simple *Network Only*, ping-like GPU kernel, which only performs the UDP network processing operations (no Memcached processing). The network only kernel also uses the same 96B packets. This experiment highlights the peak performance of *GNoM*.

Figure 4.15, Figure 4.16, and Figure 4.17 present the offline throughput, latency, and energy-efficiency of MemcachedGPU for the three GPUs in Table 4.2 respectively. As shown in Figure 4.15, Each GPU achieves over 27 MRPS ($\sim$21.5 Gbps), suggesting that the GPUs are capable of handling over 2$\times$ the request throughput measured in the online evaluations. The network only kernel is able to further increase this up to 39 MRPS on the GTX 750Ti. Assuming the PCIe bus is not a bottleneck, achieving this high throughput would require additional 10

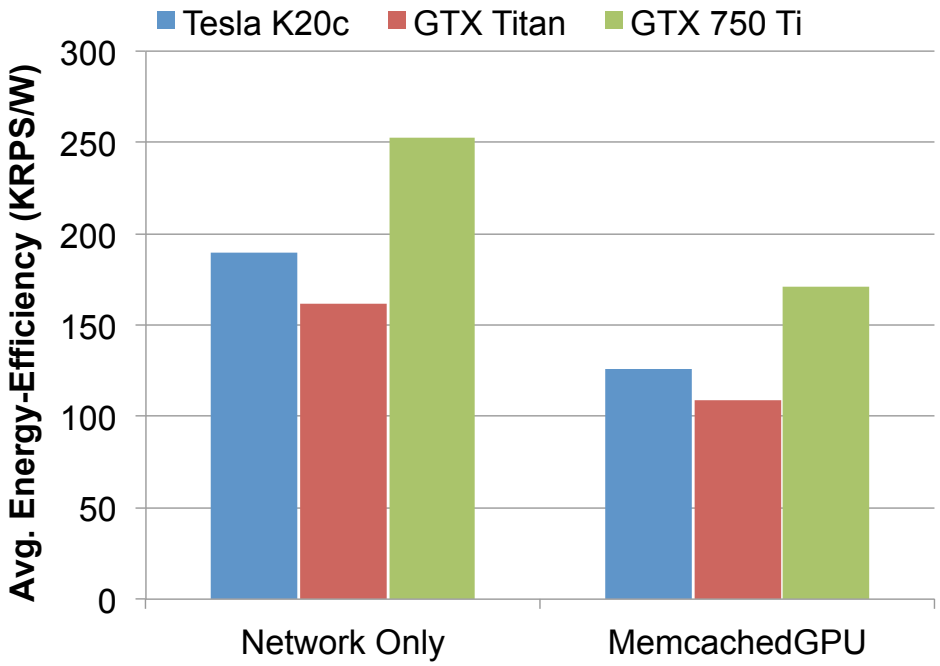


Figure 4.17: Offline *GNoM* energy-efficiency - 16B keys, 96B packets.

GbE NICs or higher throughput NICs, and removing the limitation on the amount of pin-able GPU memory to allocate more GRXBs for the increased request rate.

The latency in Figure 4.16 measures the time prior to copying the packets from the CPU to GPU and after populating the dummy response packets at peak throughputs. The network only measurements highlight the latency contributed by the *GNoM* framework and the network packet parsing and response packet generation code, which accounts for roughly 50% of the total processing latency. An interesting result of this study was that the low-power GTX 750Ti reduced the average MemcachedGPU batch latency compared to the Tesla K20c by ∼25%, while also slightly improving peak throughput. This improvement can be attributed to many of the different architectural optimizations in Maxwell over Kepler [136] and to the properties of the MemcachedGPU application in a GPU environment. MemcachedGPU is a memory bound application as there are many memory operations with little processing per packet. While the Tesla K20c has higher memory

bandwidth than the GTX 750Ti, the available GPU memory bandwidth is not optimally used since each GPU thread handles a separate packet, which reduces the opportunities for memory coalescing within a warp. Thus, providing little benefit to the Tesla's higher memory bandwidth. Although the Tesla K20c has 2.6× more CUDA cores capable of processing more request batches concurrently, the GTX 750Ti has a ∼61% higher core clock frequency, which can process warps faster. Additionally, the low processing requirements per packet further reduces the benefits of the higher computational throughput GPUs.

Finally, the energy-efficiency in Figure 4.17 measures the system wall power at the peak throughputs. As the packets are being read from memory instead of from the NIC, the wall power does not account for the power consumed by the NICs. As such, we also add the TDP for the additional NICs required to support the increased throughput. For MemcachedGPU, the GTX 750Ti is able to process over 27% and 43% more *GET* requests per watt than the Tesla K20c and GTX Titan respectively.

4.4.6 Comparison with Previous Work

This section compares MemcachedGPU against reported results in prior work. Table 4.5 highlights the main points of comparison for MemcachedGPU against multi-core CPUs [52, 106], an FPGA [26, 85], and an implementation solely using a GPU for the key hashing [44]. Results not provided in the published work or not applicable are indicated by "–". The cost-efficiency (KRPS/$) only considers the purchase cost of the CPU and the corresponding accelerator at the time of this study (2015), if applicable. All other costs are assumed to be the same between systems. The last column in Table 4.5 presents the year the processor was released and the process technology (*nm*).

Table 4.5 also presents our results for the vanilla CPU Memcached and vanilla CPU Memcached using *PF_RING* to bypass the Linux network stack. No other optimizations were applied to the baseline Memcached. These results highlight that while bypassing the Linux network stack can increase performance and energy-efficiency, additional optimizations to the core Memcached implementation are required to continue improving efficiency and scalability.

Table 4.5: Comparing MemcachedGPU with previous work.

Platform	**MRPS**	**Lat. (μs)**	**KRPS/W**	**KRPS/$**	**Year /** *nm*
MemcGPU Tesla (online)	12.9-13	m <800, $p95$<1100	62	4.3	'14/28
MemcGPU Tesla (online)	9	$p95 < 500$	45	3	'14/28
MemcGPU GTX 750Ti (NGD online)	12.85	m <830, $p95$ <1800	84.8	25.7	'14/28
MemcGPU GTX 750Ti (offline)	28.3	–	127.3	56.6	'14/28
Vanilla Memc - 4 threads	0.93	$p95$<677 - 0.5 MRPS	6.6	2.67	'13/22
Vanilla Memc + *PF_RING*- 2 threads	1.82	$p95$<607 - 1 MRPS	15.89	5.2	'13/22
Flying Memc [44]	1.67	$m < 600$	8.9	2.6	'13/28
MICA - 2x Intel Xeon E5-2680 (online,**4 NICs**) [106]	76.9	$p95 < 80$	–	22	'12/32
MICA - 2x Intel Xeon E5-2680 (offline) [106]	156 (avg. of uni. & skew.)	–	–	44.5	'12/32
MemC3 - 2x Intel Xeon L5640 [106]	4.4	–	–	12.9	'10/32
FPGA [26, 85]	13.02	3.5-4.5	106.7	1.75	'13/40

Aside from MICA [106], MemcachedGPU improves or matches the throughput compared to all other systems. However, an expected result of batch processing on a throughput-oriented accelerator is an increase in request latency. The CPU and FPGA process requests serially, requiring low latency per request to achieve high throughput. The GPU instead processes many requests in parallel to increase throughput. As such, applications with very low latency requirements may not be a good fit for the GPU. However, even near 10 GbE line-rate MemcachedGPU

achieves a 95-percentile RTT under 1.1*m*s and 1.8*m*s on the Tesla K20c (*GNoM*) and GTX 750Ti (NGD) respectively.

MemcachedGPU is able to closely match the throughput of an optimized FPGA implementation [26, 85] at all key and value sizes, while achieving 79% of the energy-efficiency on the GTX 750Ti. Additionally, the high cost of the Xilinix Virtex 6 SX475T FPGA (e.g., $7100+ on digikey.com) may enable MemcachedGPU to improve cost-efficiency by up to 14.7× on the GTX 750Ti ($150). While an equivalent offline study to Section 4.4.5 is not available for the FPGA, the work suggests that the memory bandwidth is tuned to match the 10 GbE line-rate, potentially limiting additional scaling on the current architecture. This provides promise for the low-power GTX 750Ti GPU in the offline analysis, which may be able to further increase throughput and energy-efficiency up to 2.2× and 1.2× respectively. Furthermore, the GPU can provide other benefits over the FPGA, such as ease of programming and a higher potential for workload consolidation (Section 4.4.4).

Flying Memcache [44] uses the GPU to perform the Memcached key hash computation, while all other network and Memcached processing remains on the CPU. *GNoM* and MemcachedGPU work to remove additional serial CPU processing bottlenecks in the *GET* request path, enabling 10 GbE line-rate processing at all key/value sizes. Flying Memcache provides peak results for a minimum value size of 250B. On the Tesla K20c with 250B values, MemcachedGPU improves throughput and energy-efficiency by 3× and 2.6× respectively, with the throughput scaling up to 7.8× when using 2B values.

The state-of-the-art CPU Memcached implementation, MICA [106], achieves the highest throughput of all systems on a dual 8-core Intel Xeon system with four dual-port 10 GbE NICs. Similar to MemcachedGPU, MICA makes heavy modifications to Memcached and bypasses the Linux network stack to improve performance, some of which were adopted in MemcachedGPU (Section 4.2). Additional modifications, such as the log based value storage, could also be implemented in MemcachedGPU. MICA's results include *GET*s and *SET*s (95:5 ratio) whereas the MemcachedGPU results consider 100% *GET* requests, however, MICA also modified *SET*s to run over UDP, which may limit the effectiveness in practice. Additionally, MICA requires modifications to the Memcached client to achieve peak

throughputs, reducing to ∼44% peak throughput without this optimization. In the online NGD framework, the GTX 750Ti may improve cost-efficiency over MICA by up to 17%. MICA presents an offline limit study of their data structures without any network transfers or network processing, reaching high throughputs over 150 MRPS. In contrast, all of the UDP packet data movement and processing is still included in the offline MemcachedGPU study (Section 4.4.5); however, UDP packets are read from CPU memory instead of over the network. In the offline analysis, the GTX 750Ti may improve cost-efficiency over MICA up to 27%. We were not able to compare the energy-efficiency of MemcachedGPU with MICA as no power results were presented.

A state-of-the-art GPU key-value store, Mega-KV [184], was published after the work in this study, which achieves very high performance on a multi-CPU, multi-GPU, multi-NIC server (e.g., 120+ MRPS). Mega-KV only performs the key-value look-up on the GPU, all other processing is on the CPU. Additionally, Mega-KV uses the AES SSE instruction for the hash function (instead of in software), smaller minimum sized keys, and a compact key-value protocol independent from Memcached's ASCII protocol. Mega-KV is discussed further in Section 6.1.1.

4.5 Summary

This chapter presented *GNoM*, a GPU-accelerated networking framework, which enables high-throughput, network-centric applications to exploit massively parallel GPUs to execute both network packet processing and application code. This framework allows a single GPU-equipped datacenter node to service network requests at ∼10 GbE line-rates, while maintaining acceptable latency even while processing lower-priority, background batch jobs. Using *GNoM*, this chapter described an implementation of Memcached, MemcachedGPU. MemcachedGPU is able to achieve ∼10 GbE line-rate processing at all request sizes, using only 16.1 μJ and 11.8 μJ of energy per request, while maintaining a client visible $p95$ RTT latency under 1.1 ms and 1.8 ms on a high-performance NVIDIA Tesla GPU and low-power NVIDIA Maxwell GPU respectively. We also performed an offline limit study and highlight that MemcachedGPU may be able to scale up to 2× the

throughput and 1.5× the energy-efficiency on the low-power NVIDIA Maxwell GPU. We believe that future GPU-enabled systems, which are more tightly integrated with the network interface and less reliant on the CPU for I/O, will enable higher performance and lower energy per request.

Overall, this chapter demonstrates the potential to exploit the efficient parallelism of contemporary GPUs for network-oriented datacenter services. However, a large portion of the *GNoM* framework presented here, specifically *GNoM-host* (*GNoM-KM*, *GNoM-ND*, and *GNoM-user*), is only required because current GPUs cannot directly receive control information or interrupts from other third party devices in a heterogeneous system through any standard interfaces. As a result, the CPU acts as a middleman responsible for handling the task management and control information between a third party device and the GPU. This increases the latency for launching and handling the completion of GPU tasks, decreases the energy-efficiency, increases complexity, and reduces the potential for the CPU to work on other useful tasks. In the next chapter (Chapter 5), we propose modifications to the existing GPU architecture and programming model to enable third party devices to launch tasks directly on the GPU.

Chapter 5

EDGE: Event-Driven GPU Execution

The previous chapter explored the potential for using GPUs as efficient accelerators for network applications in the datacenter. It highlighted that by offloading both the UDP network processing and Memcached processing to the GPU in micro batches, MemcachedGPU could obtain high throughput, low latency, and high energy efficiency on commodity Ethernet and GPU hardware. However, while the GPU is responsible for the main processing (UDP packet parsing, Memcached processing, and UDP response packet generation), and the network data is transferred directly to the GPU from the network interface (GPUDirect), a complex host framework, *GNoM-host*, is required to manage the interactions between the network interface (NIC) and the GPU. *GNoM-host* acts as the middleman between communicating devices, and is a result of a centralized CPU + Operating System (OS) design, where the CPU is typically responsible for handling IO and managing control between devices in a heterogeneous system.

This is not a unique property of *GNoM* or MemcachedGPU. Many other recent works utilize GPUs to accelerate tasks spanning multiple heterogeneous devices. For example, GPUNet [96], GPURdma [38], GASSP [170], and PacketShader [65], accelerate GPU network processing applications interacting with the CPU and NICs. Other works implement FPGA-GPU-CPU pipelined tasks for high-throughput processing tasks such as cardiac optical mapping [116] and pedes-

trian detection [23]. Each of these works require a CPU and/or GPU software runtime framework to orchestrate the communication of control between an external device (NIC/FPGA) and the GPU. This chapter [1] explores how to increase the independence and control of the GPU to enable third-party devices in a heterogeneous environment to manage the execution of GPU tasks without interacting with the CPU on the critical path or requiring CPU/GPU polling software frameworks.

5.1 Motivation for Increased GPU Independence

Figure 5.1 shows a comparison of three systems, in which the same streaming GPU task is launched repeatedly to process data provided by an external device. Examples of such applications are listed above. The CPU invokes the same GPU kernel with different parameters describing the new data to process; however, the CPU has little to no involvement in the application processing.

In the baseline CUDA programming model, Figure 5.1(a), the CPU is responsible for transferring data to the GPU, configuring the GPU task, and launching the task on the GPU through one or more CUDA streams (referred to as the Baseline in this chapter). Under the baseline, if an external device wants to initiate work on the GPU, it can directly transfer data to the GPU using GPUDirect; however, control is communicated with the CPU on both ends of the task. This is equivalent to the *GNoM* software framework described in Chapter 4. On the front end, the CPU must wait for work to arrive, either through interrupts or polling, configure the task for the GPU, and launch the task on the GPU. On the back end, the CPU must wait for the GPU task to complete, again either through interrupts or polling, handle the response, and communicate the response back to the device initiating the work on the GPU. The inclusion of such a CPU software framework for handling IO and managing control of the GPU when the CPU is not the initiator of the work has many drawbacks. First, the latency to launch tasks and handle the responses is increased. Second, including the CPU in the critical path increases the total system energy consumption. Third, the ability for the CPU to work on other useful tasks is reduced. This is especially important in the datacenter environment,

[1] A modified version of the material presented in this chapter was later published in the International Conference on Parallel Architectures and Compilation Techniques (PACT) 2019 [71].

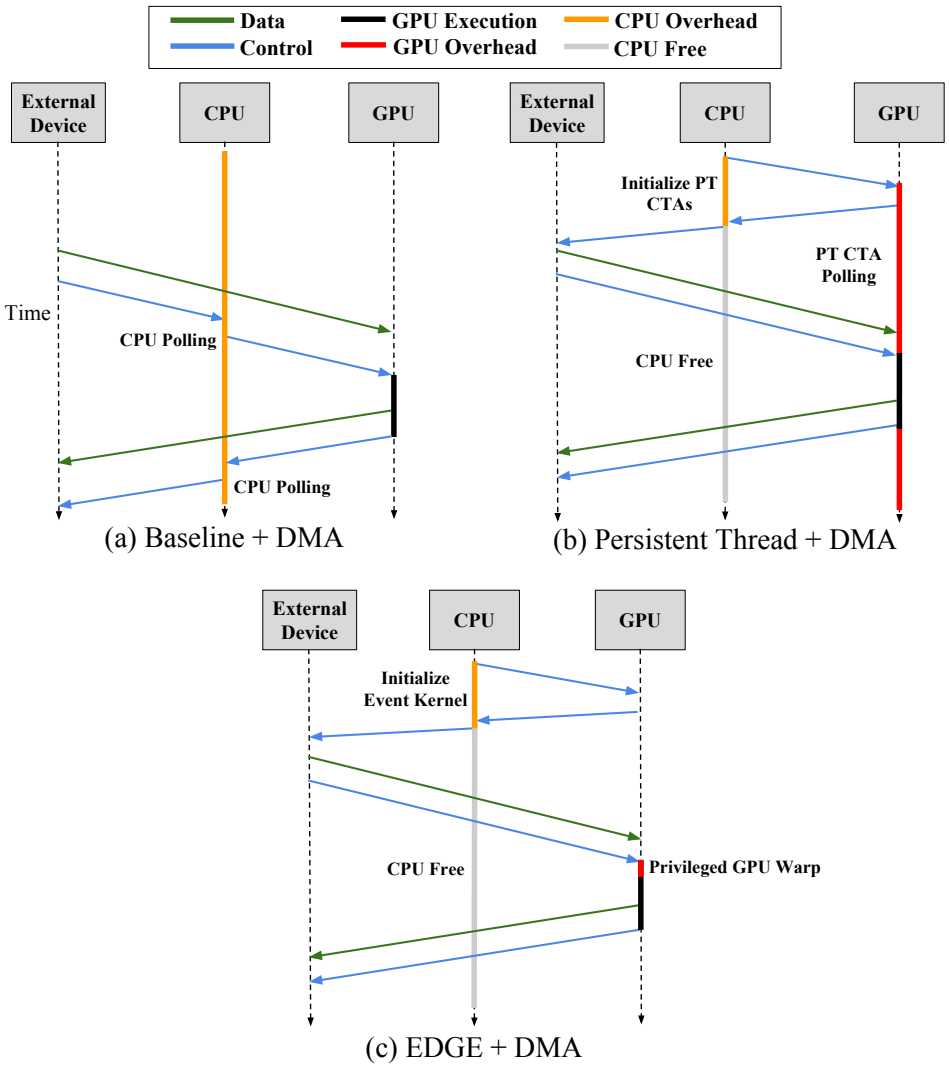


Figure 5.1: Example of the data and control flow when an external device launches tasks on the GPU for the baseline CUDA streams, Persistent Threads (*PT*), and *EDGE*.

where workload consolidation is used to increase utilization and efficiency, and to reduce total costs. Lastly, running any other work concurrently on the CPU can impact the end-to-end performance of the GPU task by increasing the task launch and completion latency.

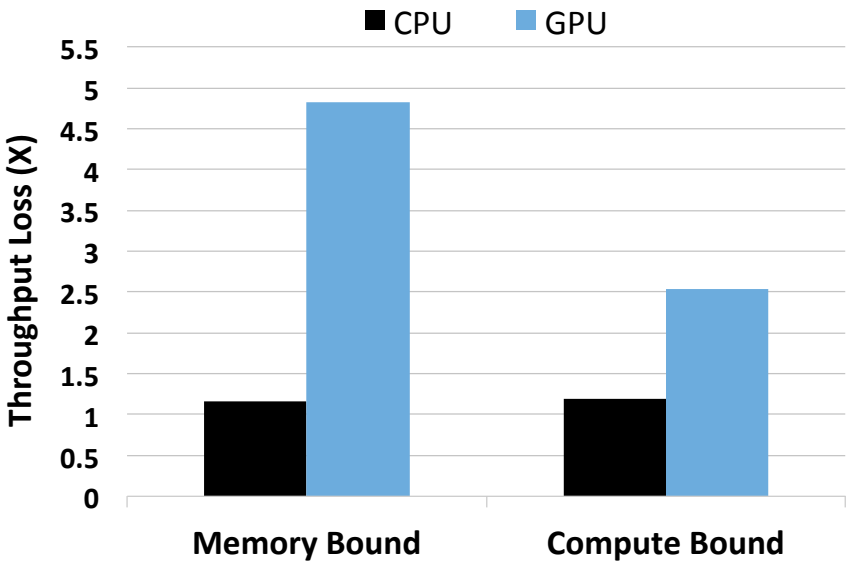


Figure 5.2: Evaluating the loss in throughput for CPU compute and memory bound applications (Spec2006) when running concurrently with and a GPU networking applications. The GPU's reliance on the CPU to launch kernels leads to inefficiencies for both devices.

Consider the experiment in Figure 5.2, which evaluates the performance for a set of memory and compute bound CPU tasks (Spec2006) concurrently running with a GPU UDP network ping (GPU-ping) benchmark under the baseline CUDA system. GPU-ping receives packets directly from an Ethernet NIC in GPU memory via GPU-Direct. It employs a CPU framework, *GNoM-host*, to handle NIC interrupts, manage the launch and completion of GPU kernels, and send the response packets to the NIC. Relative to running either application in isolation, the CPU memory and compute bound applications run 1.17× and 1.19× slower with GPU-ping, while the peak GPU-ping packet rate is reduced by 4.83× and 2.54×, respectively. This is a direct result of requiring the CPU to manage control of the GPU on the critical path of the streaming GPU application. As such, there is a need for enabling external devices to efficiently communicate both data and control directly with the GPU.

Persistent threads (*PT*) are an alternative technique for programming and

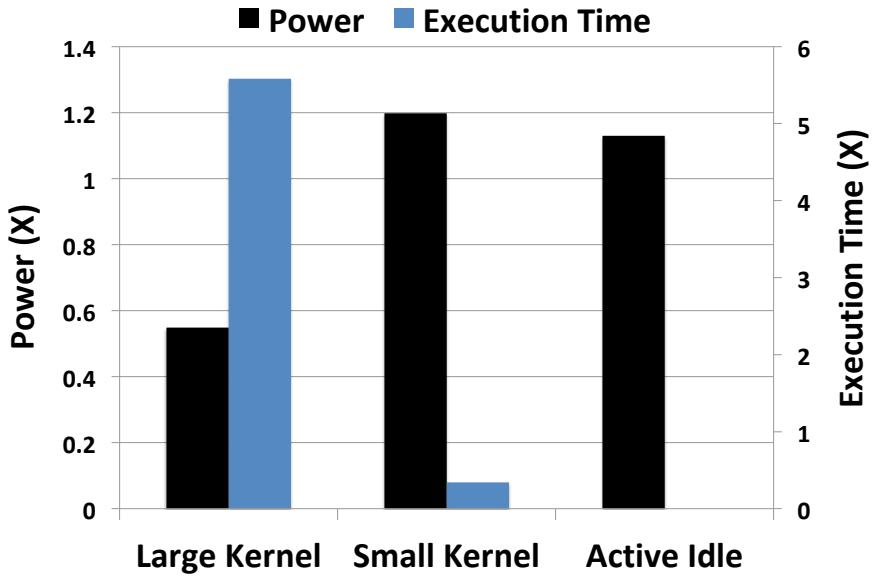


Figure 5.3: Measuring the performance and power consumption of persistent threads (*PT*) versus the baseline CUDA stream model for a continuous stream of large and small matrix multiplication kernels. Active Idle measures the power consumption of the polling *PT* threads when there are no pending tasks.

launching tasks on the GPU and are discussed in more detail in Section 2.1.7. With *PT* and RDMA, data and control is able to flow directly into the GPU's memory and software schedulers, relaxing the requirement for the CPU to be involved, as shown in Figure 5.1(b). However, the increased flexibility of *PT*, enabled by the software thread schedulers, can impact performance and energy efficiency depending on the size of kernels and the rate at which tasks are launched. *PT* also increases the code complexity, as the programmer is now responsible to implement the task launching, task scheduling, and synchronization logic. Additionally, by definition *PT* indefinitely consumes GPU resources for the persistent CTAs (pCTAs), which limits the potential for other kernels to concurrently use the GPU. As described in Section 2.1.5, recent proposals for GPU preemption [33, 92, 126, 131, 143, 157, 162, 176] can enable the pCTAs to be preempted. However, preempting and resuming the pCTAs reduces the benefits of having GPU threads continuously polling for work.

Furthermore, with *PT* the GPU's software scheduler threads are constantly polling for new work (red execution bars in Figure 5.1(b)), which can impact energy efficiency under varying task rates. Consider the example in Figure 5.3, which evaluates the power and performance of a CTA-level *PT* framework relative to the baseline GPU system performing a continuous stream of multiple matrix multiplications on two matrix sizes; small (2 CTAs / kernel) and large (120 CTAs / kernel). With the large kernel, *PT* is not able to take advantage of the GPU's hardware CTA schedulers, since this is now handled in software. As a result, less time is used for performing the matrix multiplication to perform the scheduling tasks, which lowers power by 45%, but increases execution time by 5.58×, consequently increasing energy. *PT* perform very well with small kernels, since the overheads for kernel launching and scheduling in the baseline system are higher relative to the amount of work to perform. This enables *PT* to spend more time performing the matrix multiplication, which increases power by 20%, but significantly decreases execution time by 2.9×, hence lowering energy. Finally, the polling nature of *PT* increases power consumption by 13% when no tasks are pending, relative to a GPU without *PT* in a high power state (p2), indicated by *Active Idle*.

Each technique described above poses a trade-off between the amount of control a GPU has and the efficiency it can provide. As such, there is a need for a technique that achieves the performance and complexity of the baseline CUDA model with the flexibility of the persistent thread model. This chapter proposes an event-driven GPU execution technique, *EDGE*, which is a form of GPU active messaging [46]. Event-driven programming is an alternative style of programming compared to threads, where programs are broken down into fine-grained pieces of code, callbacks, or event handlers, responsible for performing specific operations in response to IO events [37, 54, 140]. The event handlers can be triggered via interrupts or called in a continuously running event loop. The motivation behind *EDGE* is that by enabling GPU kernels to be triggered by external events, any device in a heterogeneous system can launch work on the GPU without requiring CPU interaction or GPU polling software frameworks. This can improve performance, efficiency, and server utilization, while reducing complexity.

The resulting control and data flow for *EDGE* is shown in Figure 5.1(c). Here, the CPU is required only to initialize the kernel once and then both the data and

control is transferred directly from the external device to the GPU, removing the CPU from the critical path and freeing it up to work on other tasks.

Additionally, this chapter proposes a new form of CTA barrier, the wait-release barrier, which enables running CTAs to halt execution indefinitely until another CTA releases the barrier in response to some external event. This is useful to help reduce the polling overheads of the persistent GPU thread style of programming, while retaining the benefits of having persistent CTAs ready to immediately begin processing the kernel.

The following section discusses the various design alternatives and requirements for supporting event-driven execution on a GPU.

5.2 Supporting Event-Driven GPU Execution

EDGE has three main requirements. First, the GPU needs to know which code to execute and which data to operate on for a given event. In the baseline CUDA model, a user-level CPU process passes the task and parameter information to the GPU driver via the CUDA API, which configures the GPU kernel and transfers it to the GPU to be executed. However, the GPU driver runs in the operating system (kernel) space, which introduces a dependence on the CPU for configuring and launching GPU tasks. Alternatively, in-memory user-level work submission queues provide a mechanism for configuring the GPU kernel without operating in the privileged kernel space or requiring the use of a device-specific API. With in-memory work queues, the initiator of the GPU task writes the task to run and the input parameters directly to pre-defined memory locations using regular store instructions. Once notified of the pending task (described below), the GPU can read this information, configure the task, and then execute the task. Additionally, user-level work queues support independently configuring and triggering GPU kernels concurrently from different processes or devices. There have been multiple different proposals for using in-memory, user-level work queues for communicating tasks with the GPU to improve multi-process support, reduce GPU kernel launch latency, or interact directly with GPU threads, such as NVIDIA's Multi-process Service (MPS) [129], the Heterogeneous Systems Architecture (HSA) [55], and the persistent GPU thread frameworks described above. In-memory work submis-

sion/completion queues are also used to enable a CPU to communicate with external devices, such as NVME [49]. As such, *EDGE* utilizes in-memory work queues to communicate tasks and data from a third-party device with the GPU, which is described in Section 5.4.1.

Second, there must be a method to notify the GPU of a pending task and schedule that task on the GPU. Assuming in-memory, user-level work queues are used to submit tasks to the GPU, there are multiple different techniques to achieve this. In MPS, a CPU daemon process identifies when tasks have been submitted to the work queues and launches the task on the GPU from the CPU. While this reduces the requirements of the user-level process for launching the task (i.e., the user-level process does not need to communicate with the GPU driver through the CUDA API), it still requires the CPU to communicate the task with the GPU via the daemon MPS process. In *PT*, persistent GPU threads poll the in-memory work queues to identify when new work is available. As discussed above, this polling can have negative impacts on performance, efficiency, and complexity . In HSA, a "Packet Processor" handles packets inserted into the work queues and initiates the task on the GPU. The HSA Programmer's Reference Manual [55] states that the Packet Processor is generally a hardware unit and may reside on either the device initiating the task (e.g., the CPU) or the device where the task will run (e.g., the GPU). The device initiating the task writes into a "doorbell register" associated with the work queue to notify the Packet Processor that a new task is available. In the context of a GPU, for example, the Packet Processor could be a dedicated hardware unit or a small scalar processor residing on the GPU. A dedicated hardware unit would provide a low-latency and high-efficiency path for identifying when a new task is available and for scheduling the task to be executed on the GPU. However, a dedicated hardware unit would also limit the types of operations that can be performed on an event. On the other hand, a small GPU-resident, scalar processor would provide the flexibility to implement any type of operation with additional hardware overheads. Depending on the GPU architecture, such scalar cores may already exist on the GPU [10].

In *EDGE*, we explore an alternative technique, which exploits the abundance of available computing resources on the GPU, the streaming multiprocessors (SMs), to act as the Packet Processor. Similar to *PT*, GPU threads are used to read the

in-memory work queues to configure and launch the GPU task, referred to as privileged GPU warps (*PGW*). However, unlike *PT*, *PGW*s are launched in response to an external event instead of continuously polling the in-memory work queues. To support the scheduling of *PGW*s, *EDGE* exposes a light-weight, warp-level preemption mechanism on the GPU that can be triggered by any device in the system. *PGW*s can implement a set of OS-like abstractions to increase the independence of the GPU, such as launching internal GPU kernels or ensuring fairness between competing tasks, and can be initiated via interrupts or writes to doorbell registers. The *PGW*s can also be used to release any persistent CTAs blocked on the proposed wait-release barriers to reduce the overheads of *PT* polling. Using the software *PGW*s to launch event kernels, which may require preemption to begin executing, trades off generality for performance (e.g., when compared to a dedicated hardware unit). In *EDGE*, we also consider the possibility of using a dedicated hardware unit to trigger the execution of event kernels, which would significantly reduce the event kernel scheduling latency. *PGW*s and the warp-level preemption mechanism are described in Section 5.3.

Finally, the latency to trigger events on the GPU should be minimized. In *EDGE*, we make the observation that GPU kernels in a streaming application typically have similar, if not identical, configurations. For example, the MemcachedGPU kernel performs the same task (key-value lookup), on the same number of packets, with the same set of parameters pointing to different buffers in memory populated by the NIC. In a cardiac optical mapping application [116], the FPGA triggers the same GPU image processing kernel on different images with the same dimensions. However, all of the kernel configuration parameters must be specified for every GPU kernel launch. *EDGE* exploits this opportunity by providing a platform to register pre-configured *event kernels* with the GPU. Event kernels are associated with an ID, which can be specified by the device initiating the task through an interrupt or doorbell register. If the format of the next kernel and the location of the next kernel's parameters are known a priori, event kernels can be efficiently scheduled from the *PGW*s with little to no additional information aside from the event kernel ID. Event kernels are discussed in Section 5.4.

Previous work, XTQ [101], has also evaluated active messaging for GPUs to reduce GPU kernel launch overheads by modifying Infiniband NICs to sup-

port the HSA user-level in-memory work queues via remote direct memory access (RDMA). In XTQ, the complexity is pushed to the Infiniband NIC, such that any remote agent can directly write to the HSA user-level work queues via the Infiniband NIC to launch a task on an HSA-enabled GPU without requiring interaction with the CPU. In contrast, *EDGE* pushes this complexity to the GPU, such that *any* external device capable of interacting with the GPU can launch a task directly on the GPU. *EDGE* also explores using *PGW*s as the Packet Processor for managing the launching of tasks submitted to user-level in-memory work queues. XTQ is described further in Chapter 6.

The next sections describe the GPU interrupt mechanism, GPU privileged warps, fine-grained warp-level preemption, event kernels, and the wait-release barriers in more detail.

5.3 GPU Interrupts and Privileged GPU Warps

As described in the previous section, *EDGE* explores utilizing privileged GPU warps (*PGW*) to perform higher level operations, such as internally scheduling tasks on the GPU or releasing CTA barriers, corresponding to an external event. However, a mechanism is required to initiate the execution of the *PGW*s to avoid continuous polling, as is done in *PT*. In this section, we present the design of a GPU interrupt architecture and propose modifications to the current GPU architecture to efficiently support fine-grained, warp-level GPU preemption to trigger the *PGW*s.

Interrupts provide a simple path for any device in a heterogeneous system to signal an event to the GPU. While traditional interrupts require dedicated interrupt lines, message signaled interrupts (MSI) are an alternative in-band method of sending interrupts over the same bus used to communicate with a device, supporting significantly more interrupt handlers. For example, PCIe, the communication bus typically used with discrete GPUs, supports up to 2048 different interrupts through MSI-X [2]. As such, any device that can send a message to the GPU over PCIe can also signal a variety of interrupts on the GPU.

While interrupts provide a mechanism for signalling devices of an event, interrupts can negatively impact tail latencies. For example, consider a distributed

[2]MSI-X is supported in PCIe 3.0 and higher.

networking or map-reduce task, which spawns off multiple different operations in parallel to solve a given problem. A delay in one of the parallel operations will delay the entire task [40]. As such, it is desirable to maintain low and predictable delays. A single interrupt on a traditional CPU running an OS results in multiple operations: an interrupt is received by an interrupt controller (IC), a CPU core is selected and notified of the interrupt, any application running on the interrupt core must be temporarily suspended, the core jumps to an interrupt descriptor table in the OS, which queries the IC to perform the corresponding task for the interrupt, and finally the interrupt is either processed in place or scheduled to be processed at a later time. Each of these operations introduces uncertainty into the latency for a task dependent on an interrupt. As a result, applications with low latency requirements typically resort to some form of polling, which can negatively impact energy under low task rates. However, there are fundamental differences in the GPU's hardware and software compared to a CPU that can mitigate these challenges. First, GPUs contains hundreds of cores and thousands of thread contexts, each potentially capable of handling an interrupt. At any given time, there may be a free thread context available to process an interrupt without requiring context switching (necessary on a CPU to handle an interrupt when currently processing another task), even if another application is currently running on the GPU. Second, current GPUs are offload accelerators that rely on hardware task schedulers, instead of an OS, to improve performance. As such, the latency from receiving an interrupt to running the interrupt service routine (ISR) may be reduced. Finally, GPUs are throughput oriented architectures, which are designed to tolerate long latency instructions by running multiple warps concurrently. Assuming that a warp selected to handle an interrupt is not blocking any other warps (e.g., at a synchronization barrier), the GPU application can still make progress through other concurrently running warps, whereas on a CPU, the interrupted task may be blocked from making progress during the interrupt processing.

In *EDGE*, we propose a fine-grained warp-level preemption mechanism, initiated by a GPU interrupt (or write to a doorbell register), that reduces the impact on concurrently running tasks and enables any warp to be a candidate for handling an external event. To use this mechanism, *EDGE* reserves certain interrupt vectors for user-space processing, such that any device can directly signal the GPU

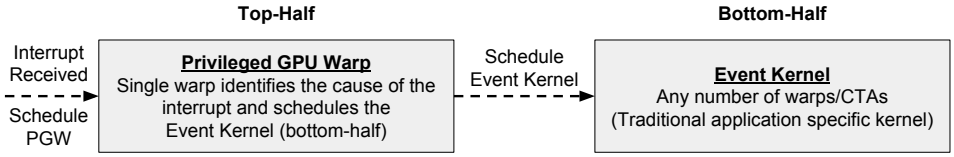


Figure 5.4: *EDGE* interrupt partitioning.

to begin work corresponding to a specific interrupt vector. *EDGE* also supports timer interrupts, which are useful for scheduling tasks that need to run periodically. Timer interrupts could be used to, for example, support time multiplexing of GPU resources as a method to coalesce interrupts, or to reduce the external device complexity by removing the need for the device to send an interrupt to initiate work by scheduling regular queries to the in-memory work queues.

5.3.1 Interrupt Partitioning and Granularity

Similar to the Linux interrupt handling mechanism, *EDGE* is designed with a notion of a *top-half* and *bottom-half* [35], as shown in Figure 5.4. The top-half is a privileged piece of code capable of performing a programmable set of operations. It is responsible for determining the cause of the interrupt and for configuring and scheduling the user-defined operation, the bottom-half, accordingly by triggering the launch of a GPU kernel, referred to as an *event kernel* (Section 5.4.1). This partitioning keeps latency low for the immediate processing stages of the interrupt, while enabling the GPU to defer the scheduling of the interrupt handler's bottom-half, which may require considerably more processing with a configurable priority. Additionally, once scheduled, the bottom-half event kernel can utilize the efficient hardware kernel and thread schedulers like any other kernel, which minimizes the modifications required to the GPU architecture to support *EDGE*.

EDGE reduces the impact of processing control messages within the GPU through fine-grained, warp-level interrupt handling. GPUs contain multiple SMXs and warp contexts, which can all be candidates to handle the interrupt. For example, the NVIDIA GeForce 1080 Ti supports up to 64 warps per SMX, with 28 SMXs, for a total of 1792 warp contexts. If all warps are occupied, a single warp must be preempted; however, if a free warp context is available, no preemption

is required. The ability for the interrupt to be processed by an idle or underutilized SMX can reduce the impact on concurrently running GPU kernels, as well as reduce the latency to schedule and process the interrupt (Section 5.6.1).

5.3.2 Privileged GPU Warp Selection

The *PGW* is similar to any other warp in the GPU, with the main difference being that it does not belong to a user-level kernel or CTA – it is a system-level warp. The anticipated operations to be performed by the *PGW* (the ISR) are inherently sequential, such as launching an event kernel or releasing CTA barriers. Consequently, the SIMD aspects of the warp may go underutilized, indicating that a dedicated hardware unit or scalar processor may be more appropriate. However, using an existing GPU warp to handle these operations removes the need for additional hardware and enables future operations that may benefit from the SIMD architecture. Additionally, the current proposed ISR operations for the *PGW* require minimal processing, which mitigate the inefficiencies of using a warp for sequential processing.

When an external or internal event is triggered, *EDGE* must select a *PGW* warp to handle the event. We consider four possibilities for selecting a *PGW*: (1) utilizing dedicated hardware for the interrupt warp context (*Dedicated*); (2) reserving existing warp contexts (*Reserved*); (3) selecting a free warp context ((*Free*)); and (4) preempting a running warp context with some selection policy (e.g., *Oldest* or *Newest* warp).

Qualitatively, the trade-offs between these approaches are as follows: (1) Adding dedicated hardware for *PGW*s or adding small scalar cores on the GPU guarantees that the interrupt can run immediately (i.e., no preemption is required as dedicated computing resources are always available to process an interrupt). However, this requires additional hardware resources for managing the *PGW* and CTA contexts (e.g., program counter, special registers, SIMT stack, local memory) or for any specialized processing units. For example, AMD's Graphics Cores Next (CGN) Architecture describes scalar cores integrated within the Compute Units, which could be used to process an event. Furthermore, as described above, the sequential nature of the *PGW* ISR operations could limit the additional

hardware required, while still reusing the existing SIMD execution pipeline and warp schedulers. For example, the *PGW* may not require a SIMT stack to track divergent warp threads, wide register-file to support warp-wide register accesses, CTA barrier management, or special thread / CTA dimension registers. While the hardware overheads may not be high, this chapter focuses on exploring the potential for reusing the existing GPU computing resources.

(2) Reserving specific warp contexts from the existing set of warps for *PGW*s achieves the performance benefits of additional dedicated *PGW* resources without any of the hardware overheads. However, reserving *PGW* resources reduces the amount of available parallelism that an application can exploit by the number of warps reserved for the *PGW*s. In such a case, applications that fully utilize the GPU resources will be negatively affected, even when the *PGW* is not executing.

(3) If there are unused warp contexts available, for example, due to a GPU application underutilizing the GPU resources, selecting a free warp does not require additional hardware for the *PGW* context, does not permanently reduce the parallelism available to an application, nor does it delay the time to schedule the *PGW*. However, a free warp context may not be available if one or more active GPU applications fully utilize the GPU's resources, which would require blocking the *PGW* until a free warp becomes available.

(4) Lastly, preempting a running warp enables the interrupt handler to run once the running warp has been properly halted and preempted, does not require additional hardware for the *PGW* context, and only temporarily reduces the available thread contexts available to an application. However, preempting a running warp increases the latency to schedule the *PGW* and requires additional hardware to support the warp-level preemption. The selection policy of a *PGW* to preempt, referred to as the *victim warp*, is important to minimize both the latency to preempt to the victim warp and the impact on the runtime of the victim warp's CTA. In this dissertation, we evaluate selecting the oldest and newest warp within a CTA as the victim warp. The evaluation of more complex victim warp selection policies, which also increase the hardware complexity of *EDGE*, is left to future work. Section 5.6.1 evaluates and compares the different *PGW* selection policies and highlights that *EDGE* can achieve similar performance by preempting running warps instead of adding dedicated *PGW* hardware or reserving existing warps for the *PGW*s.

The main challenge with interrupting a running warp is minimizing the latency required to preempt the warp, which is discussed further in the following section.

5.3.3 Privileged GPU Warp Preemption

EDGE preempts warps at an instruction granularity, meaning that it does not wait for a warp in a CTA to complete all of its instructions before preempting. Instead, the warp is preempted as soon as possible. This minimizes the latency to preempt a running warp at the cost of increased preemption complexity. To correctly save a warp's context and minimize complexity, any pending (un-executed) instructions are first flushed from the pipeline prior to starting the ISR. Depending on the state of the victim warp to preempt for a *PGW*, or the instructions currently in the pipeline, the victim warp preemption latency can be quite large. The terms "victim warp preemption latency" and "ISR scheduling latency" are used interchangeably in this dissertation.

We identify four main causes for large victim warp preemption latencies: (1) low scheduling priority for the victim warp, (2) pending instructions in the instruction buffer (i-buffer), (3) victim warps waiting at barriers, and (4) in-flight loads. (1) To ensure that the victim warp completes its current instructions promptly, the victim warp's priority is temporarily increased. This overrides the GPU's current warp scheduling policy, such as greedy-than-oldest (GTO) or round robin, and schedules the victim warp as soon as it is ready until all in-flight instructions have completed.

(2) Flushing any non-issued instructions from the i-buffer limits how many instructions the *PGW* needs to wait for before being able to preempt the victim warp. This can further increase the execution time of the victim warp, if the flushed instructions are evicted from the instruction cache before being restored. However, assuming the victim warp is not the final warp in the CTA, the GPU's FGMT mitigates the impact by hiding the instruction fetch latency through executing other warps.

(3) A victim warp waiting at a barrier is a perfect candidate for interrupting, since the warp is currently sitting idle waiting for other warps in the CTA to hit the barrier. As such, preempting this warp may not impact the CTA's progress. How-

ever, special care needs to be taken to ensure that the victim warp is conditionally re-inserted into the barrier when the ISR completes depending on if the barrier has been released or not.

(4) Finally, in-flight load instructions can be dropped and replayed for victim warps, which significantly decreases the variability in preempting a running warp, since load instructions potentially have a much higher latency than ALU operations. Dropping loads involves releasing the miss-status holding register (MSHR) entry for the pending load, releasing the registers reserved in the scoreboard, and rolling back the program counter to the dropped load instruction such that the victim warp re-executes the load instruction once being rescheduled after the ISR. Additionally, a register is required to store the address of the dropped load to identify and discard the load when it has returned from the memory subsystem. The load can still be safely inserted into the cache, which decreases the latency to replay the load.

Section 5.6.1 evaluates the impact of applying the above victim warp flushing techniques on the ISR scheduling latency.

Preemption requires saving the victim warp's state prior to switching to the *PGW* and requires restoring the victim warp's state after the *PGW* completes. When preempting a running victim warp context, the *PGW* can make use of the victim warp's SIMT stack to save the program counter and warp divergence state, since the SIMT stack already contains the full history of the victim warp's execution state. Then, a new entry for the *PGW*'s ISR can be pushed onto the victim warp's SIMT stack to configure the *PGW*'s execution state. When the ISR completes, an interrupt return function can pop the *PGW*'s entry off of the SIMT stack to resume execution of the interrupted victim warp. Additionally, any registers used by the *PGW* must be saved and restored. We allocate a small region of global memory per SM to save the victim warp's registers. This could also be implemented as a small on-chip buffer to reduce the *PGW* preemption latency.

5.3.4 Privileged GPU Warp Priority

Along with minimizing the time to schedule a *PGW*, the ISR runtime should be minimized. This reduces the latency to begin performing the actual task for the

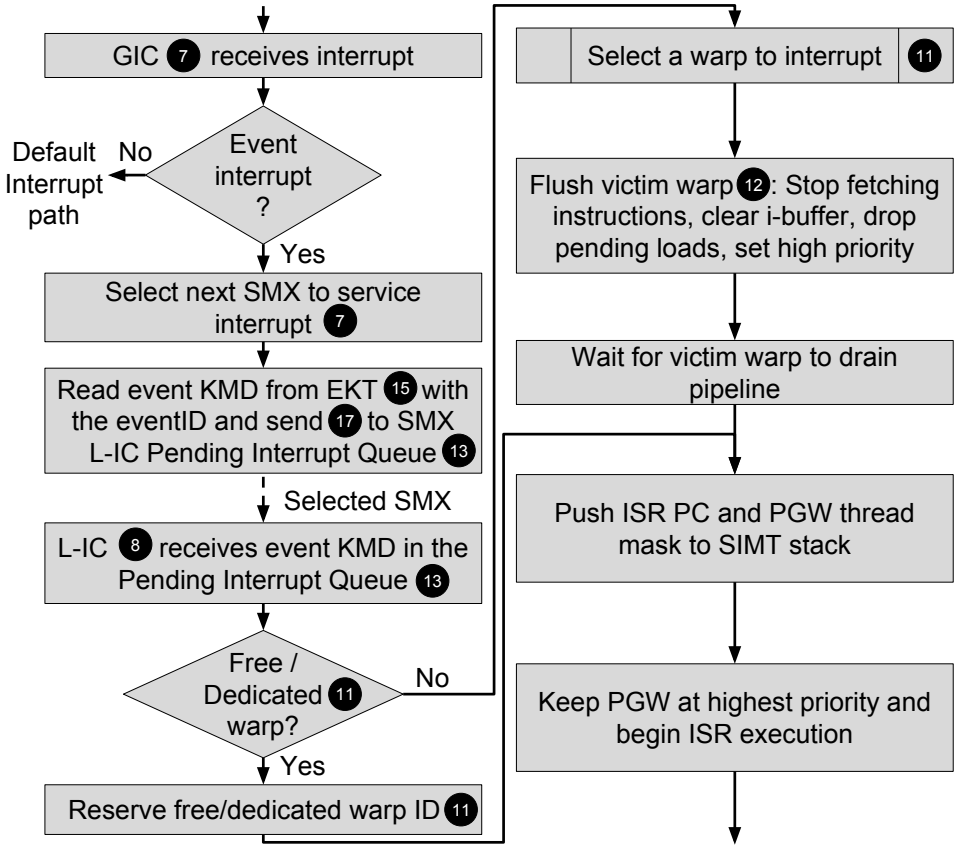


Figure 5.5: Interrupt controller logic (reference Figure 5.7).

corresponding event and reduces the amount of time the victim warp is blocked from performing its original task. In Section 5.6.1, we evaluate two techniques for minimizing the ISR runtime: prioritizing the *PGW*'s instruction fetch and scheduling, and reserving its entries in the instruction cache. The ISR code can be very small, requiring only a few entries in the instruction cache. For example, the current ISR implementation in *EDGE* occupies only three cache lines. We find that the ISR execution time is significantly reduced when the *PGW* never misses in the instruction cache.

5.3.5 Interrupt Flow

Figure 5.5 presents the high-level *EDGE* interrupt controller logic, which is implemented in hardware. The numbers in this figure indicate which parts of the GPU hardware shown in Figure 5.7 are responsible for performing the operation. The interrupt controller is split into a global interrupt controller (G-IC) and per-SMX local (L-IC) interrupt controllers. When an interrupt is received by the G-IC, it first checks if this is an *EDGE* interrupt or a GPU-wide interrupt. If an *EDGE* interrupt, an SMX is selected to service the interrupt based on a selection policy, such as simple round-robin or a more complex policy that selects the SMX with the most free resources. The GIC then pushes the interrupt metadata, such as the event KMD described in Section 5.4.2, into a queue in the SMX's L-IC. At this point the SMX and L-IC are responsible for processing the interrupt and the G-IC can safely clear the interrupt. Additionally, because each SMX has its own L-IC, there is no contention between concurrently running interrupts on different SMXs. The selected SMX then selects a victim warp to interrupt based on the available hardware and *PGW* selection policy (Section 5.3.2). If the victim warp is currently active, it is flushed through the pipeline using the techniques described in Section 5.3.3. Next, a *PGW* entry is pushed onto the victim warp's SIMT stack with the interrupt vector PC. Finally, the *PGW* priority for instruction fetch and scheduling is increased to minimize the total ISR runtime.

Figure 5.6 presents the software ISR flow performed by the *PGW*. The ISR first saves any registers that will be overwritten, if necessary. Depending on the number of registers required for the *PGW*, additional hardware for dedicated *PGW* registers could also be added to avoid saving the victim warp's registers. The *PGW* then communicates with the L-IC to determine the cause of the interrupt and proceed accordingly. Section 5.4 discusses the operations that the ISR may perform to orchestrate event-driven GPU execution.

5.3.6 Interrupt Architecture

Figure 5.7 presents the modifications to the current GPU architecture required to support fine-grained, warp-level preemption initiated by interrupts. The G-IC ❼, which is part of the GPU front-end ❸, requires logic to identify the *EDGE*

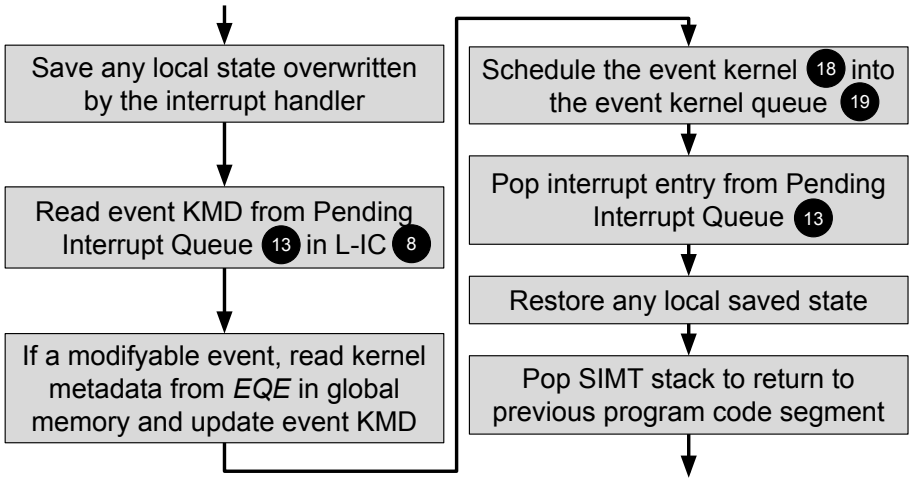


Figure 5.6: Interrupt service routine (reference Figure 5.7).

interrupt and select an SMX L-IC ❽ to process the interrupt. The SMX selection policy could take many factors into account, such as identifying the SMX with the lightest load or with free warps available to avoid warp preemption, or use a more simple policy, such as round-robin. The interrupt is then forwarded to an SMX L-IC. Enabling this division between G-IC and L-IC requires new connections ❾, or can make use of the existing connections between the SMX scheduler and SMXs.

The L-IC requires control registers to store the PC corresponding to the GPU's interrupt vector ❿, which may be able to use the existing IC metadata registers. The L-IC also maintains pending interrupt queue ⓭ for interrupts assigned to it, which in *EDGE*, contains metadata describing the event kernel to launch (Section 5.4.2). In the steady state, with 28 SMXs and a single pending interrupt queue entry, the ISRs can at most take 28× as long as the interrupt period. The correct depth of the pending interrupt queue depends on the distribution of the arrival rate of interrupts. The L-IC requires logic for identifying and selecting a victim warp for the *PGW*⓫ based on some policy (free, newest, oldest, etc), which can potentially make use of the existing warp scheduler logic. Once selected, the L-IC needs to flush the victim warp ⓬. The logic for the victim warp flushing optimizations (Section 5.3.3) includes modifications to the warp scheduler to prioritize individual warp scheduling, manipulating valid bits to flush entries from the instruction

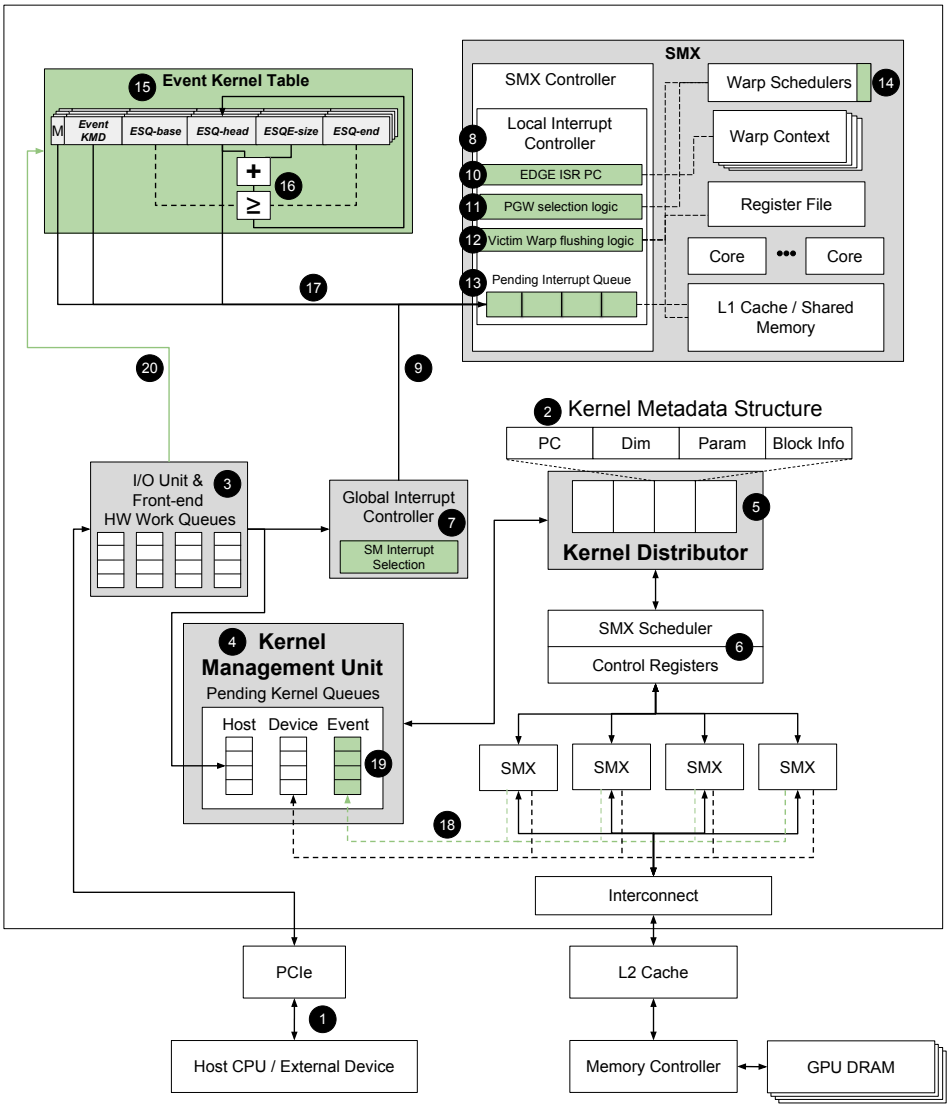


Figure 5.7: *EDGE* GPU Microarchitecture. Baseline GPU diagram inspired by [87, 111, 155, 175]. *EDGE* components are shown in green.

buffer, invalidating MSHR entries (or equivalent structure for in-flight loads), a victim warp ID register to identify and drop in-flight loads, and a register to indicate if a victim warp should return to a warp barrier after completing the ISR. *EDGE* also requires logic to lock instruction cache entries for the ISR. Finally, the

pending interrupt queue in each SMX requires 168B to store four pending events.

The *PGW* communicates information about the interrupt with the L-IC as a memory-mapped component via load and store instructions to special addresses. In our implementation, *EDGE* reserves certain memory addresses to access the components described above, such as the pending interrupt queue. This API could also be implemented using specialized instructions.

5.4 Event-Driven GPU Execution

GPU interrupts and the *PGW* warp-level preemption mechanism provide a path for initiating and managing tasks on the GPU without requiring CPU intervention. However, supporting this form of GPU task execution has several requirements. Specifically, external devices need to be able to specify which work to perform, what data to perform the work on, and determine when work has completed execution on the GPU. To address these requirements, this section proposes a new type of GPU kernel, the *event kernel*, a host API for pre-configuring event kernels and corresponding data, GPU hardware structures for storing the event kernel meta-data, and user-level work queues to communicate task initiation and completion. This section also proposes a new form of CTA barrier instruction, the *wait-release barrier*, which enables CTAs to halt execution until an external event releases the barrier through a *PGW*. Wait-release barriers are useful for persistent-thread style programming models to avoid continuously polling external memory for new tasks.

5.4.1 Event Kernels

An event kernel is a user-defined GPU kernel, which is launched internally by a *PGW* running on the GPU in response to an internal or external event. Once launched, an event kernel is identical to any normal GPU kernel. In a traditional GPU environment, the CPU is required to configure and launch every GPU kernel. *EDGE* relaxes this requirement if the same kernel is repeatedly launched on different input data (e.g., a network processing kernel when a group of packets arrive). In such cases, the kernel needs to be configured only once and an external device can directly trigger the execution of this kernel without unnecessarily involving the CPU. The following section describes how *EDGE* enables an event kernel to

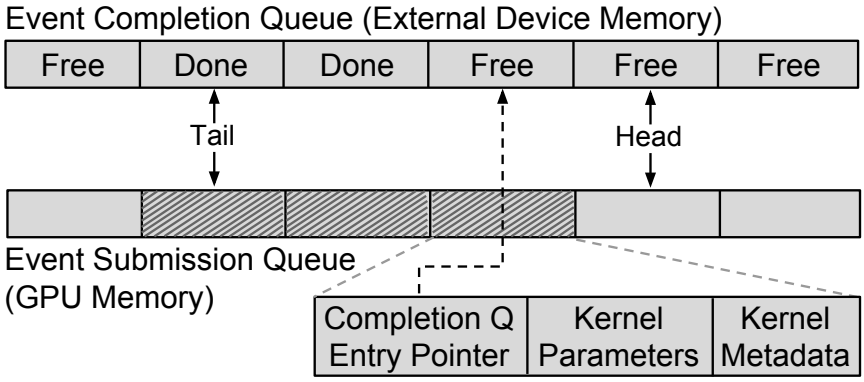


Figure 5.8: Event submission and completion queues.

concurrently support multiple kernel parameter memories to communicate with an external device.

Event Submission and Completion Queues

In current deployed systems, the host communicates data to the GPU through global and kernel parameter memory (which may be separate or part of the same physical memory). Large input and output buffers are stored in global memory, while the kernel parameter memory stores pointers to these buffers. The host CPU configures and communicates these buffers with the GPU through the GPU driver when launching a kernel. *EDGE* exploits the fact that the same event kernel will have the same parameter memory structure (i.e., the same type, number, and order of parameters), but works on different data. As such, the parameter memory could be pre-configured, removing the need to repeatedly configure it on the CPU from the critical path.

To communicate data and task status with the GPU, *EDGE* implements a pair of in-memory circular queues, called the *event submission queue* (*ESQ*) and *event completion queue* (*ECQ*), shown in Figure 5.8. These queues are similar to those used in persistent threads, NVIDIA MPS [129], and HSA [55] as a mechanism for launching tasks and managing task completion. However, unlike NVIDIA MPS and HSA, the *ESQ* and *ECQ* are tailored to a specific event kernel structure, as described below. The *ESQ* is stored in GPU memory and the *ECQ* is stored in

external memory (e.g, CPU memory or a third-party device's memory). Each *ESQ* entry (*ESQe*) contains a list of kernel parameters corresponding to a specific event kernel, optional event kernel metadata, and a pointer to the corresponding *ECQ* entry (*ECQe*) to signify the completion of an event. When launching an event kernel, the *PGW* simply sets the event kernel parameter memory to the current *ESQe* in the ISR (Figure 5.6). Since the event kernel parameter memory is pre-initialized, subsequent event kernels only need to update the kernel parameter memory to the next *ESQe*. Other required information, such as the event kernel dimensions, are already known from the event kernel pre-registration process. The *ESQe* can also be configured to store optional event kernel metadata (e.g., kernel dimensions), such that each event kernel structure can be dynamic between invocations. We refer to this as a modifyable event kernel. The *PGW* is responsible for updating the event kernel metadata (KMD) structure with the optional metadata specified in the *ESQe*. However, dynamically configuring the event kernels increases the ISR runtime, as the *ESQe* are stored in global GPU memory. There is a one-to-one mapping between an *ESQe* and *ECQe*, which the GPU uses to signal event kernel completion. The size of an *ESQe* is dictated by the number and size of kernel parameters. The size of an *ECQe* is 4 bytes. The number of entries in the circular queues dictates the maximum number of in-flight events.

EDGE provides an API for registering an event kernel and allocating the *ESQ* (Table 5.1), which is described in more detail in Section 5.4.1. *EDGE* is responsible for allocating the *ESQ* in GPU memory based on the structure of event kernel parameters and maximum number of in-flight events, which are specified by the CPU during event kernel registration. A pointer to the *ESQ* is returned to the CPU. The CPU is then responsible for allocating the *ECQ* in external device memory, pre-allocating the GPU input/output buffers, assigning the corresponding *ECQe* pointers, and setting the kernel parameters (GPU buffer pointers and any constant parameters) in each *ESQe*. Note that this requires pre-allocating kernel parameters for all possible in-flight event kernels (size of the *ESQ*). The CPU then communicates this information (*ESQ* and *ECQ*) with the external device that will launch the tasks on the GPU.

The external device maintains a single *head* and *tail* pointer for both the *ESQ* and *ECQ* to manage event kernel launching and completion (Figure 5.8). To

launch an event kernel, the external device checks the status of the *head* in the *ECQ*. If *free*, the kernel parameters in the *ESQe* at *head* can be filled with the data for the next event kernel. The external device then sends an interrupt to the GPU, which triggers the execution of the next event kernel on the GPU, and increments *head*. The queue is full if incrementing *head* reaches the *tail*. The external device uses the *tail* to identify when an event kernel has completed and the output buffers contain valid data. This is indicated by *Done* in an *ECQe*, which is updated by the GPU through an *ECQe* pointer specified in the *ESQe*. The external device is responsible for consuming the output buffers, setting the *ECQe* to *free*, and incrementing the *tail* to the next in-flight event.

As described, the proposed structure of the *ESQ* and *ECQ* results in in-order task submission and completion; the external device can not push a new request into the *head* of the *ESQ* until the corresponding *tail* in the *ECQ* is free. This limitation could be removed by adding complexity to the external device and GPU, for example, by having the GPU send interrupts to the external device to signal event completion. Note that the GPU is invoked only once a valid event has been configured at the *head* and an interrupt has been issued. As such, the GPU requires only a *head* pointer to keep track of the next *ESQe* to process, while the *ECQe* to signal completion is specified in the *ECQ*.

Event Kernel Priority and Preemption

As described in the previous section, the *PGW*s begin execution via warp-level preemption when no free warp contexts are available to process the interrupt. However, event kernels launched by the *PGW*s are regular GPU kernels, which are usually much larger than a single warp and make use of the baseline GPU hardware task/thread schedulers. Consequently, if other kernels are currently occupying the GPU, the event kernels will be blocked until sufficient resources are available. Similar to existing host and GPU-launched kernels, the priority of an event kernel is configurable. This can enable high-priority tasks to maintain a level of QoS in an environment where other GPU tasks are run concurrently. Furthermore, *EDGE* may use many recently proposed preemption and context switching mechanisms, either existing [126] or research-based [33, 92, 143, 157, 162, 176], to enable a

edgeEventId = **edgeRegisterEvent**<<<paramType0,... paramTypeN>>>(kernelPointer, &eventQueuePointer, gridDimensions, blockDimensions, sharedMemorySize, maxNumOfEvents, priority, isModifyable)
edgeUnregisterEvent(eventId)
edgeScheduleEvent(eventId)
EDGE GPU MSI-X Interrupt (eventId)
edgeReleaseBarrier()

Table 5.1: *EDGE* API extensions.

higher priority event kernel to preempt a lower priority host launched GPU kernel or other lower priority event kernel.

We evaluate two variations of CTA draining preemption for event kernels, *P1* and *P2*, which are modified versions of the mechanism proposed in [162]. These preemption mechanisms partially preempt a running kernel by blocking any non-event kernel CTA launches until enough resources are available to begin executing the event kernel. In P1, once the event kernel has scheduled all of its CTAs, any non-event kernel CTAs may be scheduled when enough resources are available. However, in P2, scheduling any non-event kernel CTAs is completely blocked until the event kernel completes. P2 places a higher priority on the event kernel than P1.

Event Kernel API

Table 5.1 presents extensions to the GPGPU API (e.g., CUDA or OpenCL) required to support *EDGE*. These functions enable the CPU to register an event kernel with the GPU, configure the parameter memory for the *ESQ*, and trigger or schedule event kernel launches.

Registering an event kernel is similar to launching a kernel through the baseline CUDA API. The main differences are that the kernel launch is delayed by storing the kernel metadata structure on the GPU, which consists of the *kernelPointer* to specify the GPU kernel function to execute for this event, the kernel dimensions (*gridDimensions* and *blockDimensions*) to specify the shape of the event kernel, and the *sharedMemorySize* to specify the amount of shared memory required by

the event kernel. Additionally, the parameter memory structure is provided instead of passing actual parameters, indicated by *paramType#* in the angle brackets. This enables *EDGE* to allocate the correct size for each *ESQe*. The priority of the event kernel is set through *priority*. The event kernel can also be marked as modifyable via the *isModifyable* flag, as described previously. The event registration allocates the *ESQ* as described in Section 5.4.1, with the total size of the queue being specified by the maximum number of in-flight event kernels (*maxNumOfEvents*) and the size of the event kernel parameters, and is returned in *eventQueuePointer*. The CPU is required to configure each *ESQe* accordingly, which consists of allocating the necessary CUDA buffers in GPU-accessible memory and updating the buffer pointers in the *ESQ*. All interactions with the *ESQ* are performed through generic memory accesses. Finally, the corresponding *edgeEventId* is returned, which is required to specify which event kernel to trigger on an interrupt, as described below. An event kernel and the corresponding *ESQ* can be freed through *edgeUnregisterEvent(edgeEventId)*.

Once registered and configured, any device capable of sending MSI-X interrupts can trigger the event kernel by sending an interrupt message with the *edgeEventId* as the interrupt identifier. *EDGE* also provides an alternative path to trigger an event kernel directly from the CPU through *edgeSchedulEvent(edgeEventId)*, which launches the event kernel corresponding to the edgeEventId using the *ESQ* interface.

The function, *edgeReleaseBarrier()* is described in Section 5.4.3.

5.4.2 EDGE Architecture

Returning to Figure 5.7, we present the modifications to the baseline GPU architecture required to support *EDGE*, indicated by the components in green. The main additional hardware components for supporting GPU launched event kernels are the storage buffers for the Pending Interrupt Queue ⓭ in the L-IC, the Event Kernel Table (EKT) ⓯, and the Event Kernel Queue (EKQ) ⓳ in the KMU ❹. The *ESQ* and *ECQ* (not shown here) are stored in global GPU memory and external device memory accordingly, as previously described.

Each entry in the EKT ⓯ stores a pre-allocated event KMD ❷, a single bit,

M, to indicate if the event kernel is modifyable, and metadata to describe the *ESQ*. The metadata consists of the *ESQ*-base pointer, *ESQ*-head pointer, *ESQ*-size, and the last *ESQ* entry, *ESQ*-end. *ESQ*-head is initialized to *ESQ*-base. When the G-IC transfers an interrupt to an L-IC ❾, the corresponding event KMD is read from the EKT ⓯ and transferred ⓱ into the Pending Interrupt Queue in the L-IC ⓭. The *ESQ*-head is then automatically incremented by *ESQ*-size via a hardware adder ⓰ and is stored back into *ESQ*-head in the EKT entry. *ESQ*-head is reset to *ESQ*-base when exceeding the allocated region, *ESQ*-end, which implements the logic for the circular *ESQ* on the GPU in hardware. The EKT simplifies the process of launching a kernel from the GPU, since the next event kernel is already configured at the next *ESQ*-head. Storing the event's KMD locally in the Pending Interrupt Queue ⓭ enables multiple SMXs to process different instances of the same event kernel concurrently, since each SMX is working on an event with unique parameter memory. Additionally, this enables the SMX to modify a local copy of the event KMD, if the event kernel is marked as modifyable, without requiring synchronization.

The EKQ ⓳ is a hardware queue in the KMU ❹ responsible for queuing pending event KMDs until there is a free entry in the KD ❺ to begin executing the event kernels. This queue is similar to the existing host and device-launched queues [87]. There can be any number of EKQs, each with different priorities relative to other event, host, or device-launched queues.

In our proposed implementation, the EKT ❼ is modeled as a small on chip buffer with a single read/write port. Each row contains five 8-byte values for the kernel pointer and parameter buffer pointers, six 4-byte values for the kernel dimension, a 2-byte value for the shared memory, and a single modifyable bit, for a total width of 529 bits. Assuming a size of 32 entries (enabling a max of 32 different event kernels), the total storage overhead of the single GPU EKT is 2.1KB, less than 1% of a single SMX's register file. The EKT also requires a single 8B adder and comparator to implement the logic for the circular buffers in hardware.

5.4.3 Wait-Release Barrier

Aside from indefinitely consuming the majority of GPU resources, an inefficiency with the persistent GPU threads (*PT*) style of programming is that GPU threads

continuously poll global memory to query for new tasks to perform or responses to remote procedure calls (RPC). If the incoming task rate is high or the RPC processing latency is low, polling can have little effect on efficiency. However, long waiting times for persistent CTAs (pCTA) result in repeated unsuccessful reads from global memory, which can lower efficiency over the traditional kernel launching. An attractive, and obvious, alternative to polling is to instead have the GPU threads block until work is available. This maintains the benefit of not requiring a full new GPU kernel to be launched when work is available, while removing the unnecessary reads to global memory, at the cost of increased latencies for identifying when work is available. Current GPUs support warp-level barriers, which block the execution of warps until all warps in a CTA have reached the barrier. However, there are currently no methods for blocking all warps in a CTA or blocking GPU threads until some external condition is met.

To address this, we propose the *wait-release barrier*, a special set of CTA barrier instructions that block all warps in a CTA at the wait barrier instruction (*__wait_threads()*) until a subsequent release barrier instruction (*__release_threads()*) is performed. While the wait-release barrier itself could be easily implemented in current GPUs, there is still the challenge of how to notify the CTA that it should be released if all warps are currently waiting at the barrier. This can be solved with the *PGW*s described in Section 5.3. A special interrupt vector is reserved for the wait-release barrier. Instead of continuously polling global work queues for new tasks, each persistent CTA (pCTA) can check the global work queue once and, if no tasks are available, block at a wait-release barrier. Pseudo examples of a baseline persistent GPU thread implementation and a persistent GPU thread implementation using the wait-release barrier are shown in Example 1 and Example 2, respectively. With the wait-release barrier, the continuous polling of the in-memory work queue is guarded with the __wait_threads() barrier instruction. At some later time when work is available, the external device can configure the task in the global work queue and trigger a GPU interrupt to release any pending wait-release barriers to check for the new work. While not shown here, a *PGW* executes the corresponding __release_threads() instruction in response to the external event. The CPU can also release the wait-release barriers through the *edgeReleaseBarrier()* API shown in Table 5.1.

Example 1 Pseudo example of a baseline persistent GPU thread implementation.

```
1:  while (true) {
2:      if (tid == CTA_SCHEDULER_TID) {
3:          // Spin for work to become available
4:          while(no_work_available()) { }
5:          // Configure task for the pCTA
6:          ...
7:      }
8:      __syncthreads();
9:
10:     // Specific kernel processing
11:     ...
12:
13:     if (tid == CTA_SCHEDULER_TID) {
14:         signal_task_complete();
15:     }
16:     __syncthreads();
17: }
```

Example 2 Pseudo example of a persistent GPU thread implementation using the wait-release barrier.

```
1:  __shared__ bool work_available = false;
2:  while (true) {
3:      // Spin for work to become available
4:      while (!work_available) {
5:          __wait_threads(); // Wait-release barrier
6:          if (tid == CTA_SCHEDULER_TID &&
7:              !no_work_available()) {
8:                work_avilable = true;
9:                // Configure task for the pCTA
10:               ...
11:         }
12:         __syncthreads();
13:     }
14:
15:     // Specific kernel processing
16:     ...
17:
18:     if (tid == CTA_SCHEDULER_TID) {
19:         signal_task_complete();
20:         work_available = false;
21:     }
22:     __syncthreads();
23: }
```

Depending on the application, there may be multiple pCTAs waiting at different wait-release barriers at any given time. While there are opportunities to optimize the selection of which barriers are released and how many barriers to release per interrupt, we opt for simplicity by releasing all pending wait-release barriers (even across concurrent kernels). The application logic does not need to change beyond adding the *__wait_threads()* prior to querying the global work queue and minor restructuring to avoid SIMD deadlocks. When an interrupt releases all waiting pCTAs from a wait-release barrier, each pCTA must recheck the global work queue to see if a new task is available. Additionally, the external device logic does not need to increase complexity by specifying a specific pCTA to release. However, a drawback of releasing all waiting pCTAs is additional accesses to the global work queue, since only a subset of the pCTAs will get to process the new task.

To avoid potential race conditions, in which an interrupt releases a wait-release barrier before a pCTA has hit the barrier, we implement the wait-release barrier as a level-sensitive barrier – the wait-release barrier is only relocked after all warps in a pCTA have passed through it. Thus, if a release operation occurs and no pCTA is waiting at a wait-release barrier, the barrier is unlocked until a full pCTA passes through it, meaning that the pCTA will then check the global memory queue to find if a pending task is available.

The wait-release barrier could be extended to support multiple different barriers. For example, similar to event kernels, wait-release barriers could be registered with the GPU and an ID could be returned. One or more barrier IDs could be passed to the GPU kernel such that within or between kernels, different CTAs could block on different wait-release barriers. The interrupt or *edgeReleaseBarrier()* API could be modified to include the wait-release barrier ID accordingly. However, this would require additional hardware to maintain the mappings between a given wait-release barrier ID and the CTAs that are waiting at that barrier, and to notify a subset of wait-release barriers. Improving the granularity of the wait-release barrier is left to future work.

Component	Configuration
GPU SMX frequency	700MHz
SMXs	16
Max threads per SMX	2048
Max CTAs per SMX	64
Register File size per SMX	65536
Shared memory size per SMX	48KB
L1 $ size per SMX	64KB
L2 $ size	1MB
Memory Configuration	Gem5 fused
GPU Warp Scheduler Base	Greedy-then-Oldest
Gem5 CPU model	O3CPU

Table 5.2: Gem5-GPU configuration.

5.5 Experimental Methodology

EDGE is implemented in Gem5-GPU v2.0 [146] with GPGPU-Sim v3.2.2 [17]. Gem5-GPU was modified to include support for CUDA streams, concurrent kernel execution, concurrent CTA execution from different kernels per SM from the CUDA Dynamic Parallelism (CDP) changes in GPGPU-Sim [174], and kernel argument memory using Gem5's Ruby memory system. The Gem5-GPU configuration used in this work is listed in Table 5.2.

We modified the baseline architecture to include a timing model for the G-IC and L-IC, *PGW* selection, victim warp flushing, interrupt vector logic, and interrupt service routine (ISR). The event kernel table is modeled as an on-chip buffer, which is read by the G-IC using the event kernel ID as the address to select an event KMD to send to an L-IC. Similar to CDP, event kernels are launched from the GPU into a separate event kernel hardware queue, which can specify different priorities relative to the host-launched kernel queues. However, unlike CDP, there is no need for the GPU to dynamically allocate argument buffers or configure the kernel to launch, since the event kernels are pre-configured by the host CPU and stored in the event kernel table. Additionally, there is no requirement to set up parent/child mappings for the event kernels as in CDP, which reduces the overheads of launching event kernels relative to child kernels in CDP. Gem5-GPU's CUDA

runtime library was extended with the EDGE API in Table 5.1 to enable configuration and management of events on the GPU. The *PGW* communicates with the L-IC via load/store instructions to special reserved addresses.

The proposed wait-release barrier instructions are implemented as in-line PTX assembly using the previously unsupported *bar.sync #* in GPGPU-Sim to model the *wait* barrier (*bar.sync 1*) and the *release* barrier (*bar.sync 2*). In the full system, we expect that these instructions would instead be implemented by modifying the CUDA compiler to support the proposed wait and release barrier instructions.

The benchmarks evaluated in this chapter are taken from Rodinia [31], two convolution kernels from Cuda-convnet [99] using a layer configuration similar to LeNet [102], and a GPU networking application, MemcachedGPU, as described in Chapter 4. The Rodinia benchmarks are Back Propagation (BACKP), Breadth First Search (BFS), Heart Wall (HRTWL), Hot Spot (HOTSP), K-Means (KMN), LU Decomposition (LUD), Speckle Reducing Anisotropic Diffusion (SRAD), and Streamcluster (SC). The size of the input data used in these benchmarks was selected to ensure that the GPU was fully utilized given the benchmark's implementation, such that an interrupt warp is not optimistically biased to having free resources available due to undersized inputs. The convolution kernels from Cuda-convnet are filterActs_YxX_color and filterActs_YxX_sparse. Finally, the MemcachedGPU kernel evaluated is the *GET* kernel using 16B keys, 2B values, and 512 requests per batch. The hash table size is set to 16k entries, and is warmed up with 8k *SET*s.

Any hardware measurements presented in this work are run on an Intel Core i7-2600K CPU with an NVIDIA GeForce GTX 1080 Ti, using CUDA 8.0 and driver v375.66. CPU timing measurements were recorded using the TSC registers and GPU power was measured using the NVIDIA Management Library (NVML) [130].

5.6 Experimental Results

This section first evaluates the interrupt warp architecture and then evaluates how the *PGW*s can be used to initiate the execution of event kernels and support the wait-release barriers.

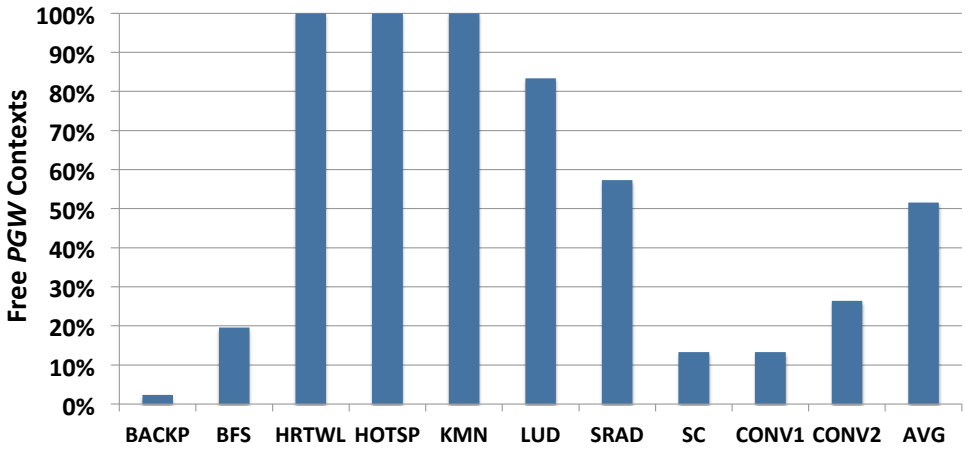


Figure 5.9: Percentage of cycles a free warp context is available for a *PGW*. This limits the amount of time a warp must be preempted to schedule the *PGW*.

5.6.1 GPU Interrupt Support

PGW Resource Requirements

Like any other warp, a *PGW* requires a hardware context to execute, such as registers and a program counter. If the GPU is underutilized by any currently running kernels, there may be enough free hardware resources (warp contexts) available to execute the *PGW* without requiring a running warp be preempted. GPUs have a range of resources (registers, thread contexts, shared memory) that are used in varying amounts by warps and CTAs. The amount of parallelism achievable on the GPU depends on which resources have the highest demand relative to the amount available. Previous work has shown that applications may underutilize GPU resources, which can benefit GPU multiprogramming through resource partitioning/sharing [176, 181]. We also measure the same behavior from many GPU applications. As a result, a GPU application may not be able to schedule an additional CTA due to insufficient resources, however, the *PGW*'s limited resource requirements may still be able to make use of the remaining fragmented resources. For example, assume we have a GPU kernel *X* that uses a large amount of shared memory, but does not require many threads. The GPU CTA schedulers dispatch

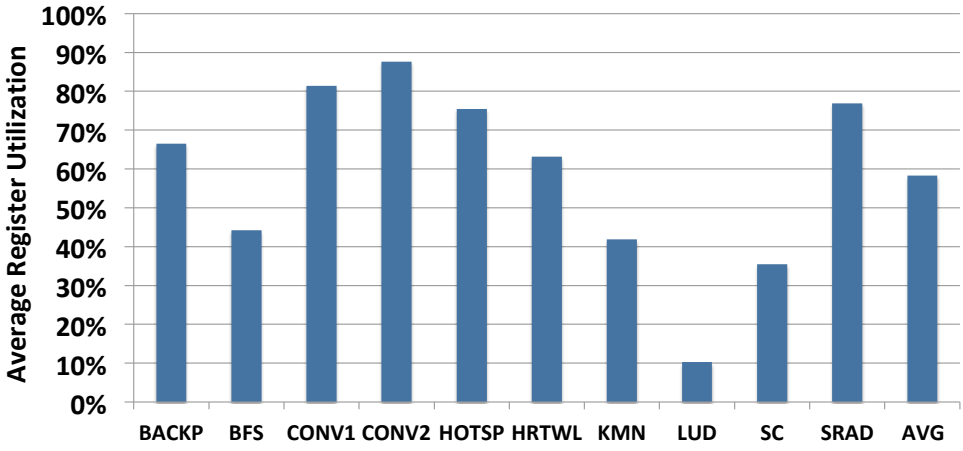


Figure 5.10: Average register utilization of Rodinia benchmarks on Gem5-GPU. Register utilization is measured for each cycle and averaged across all cycles of the benchmark's execution.

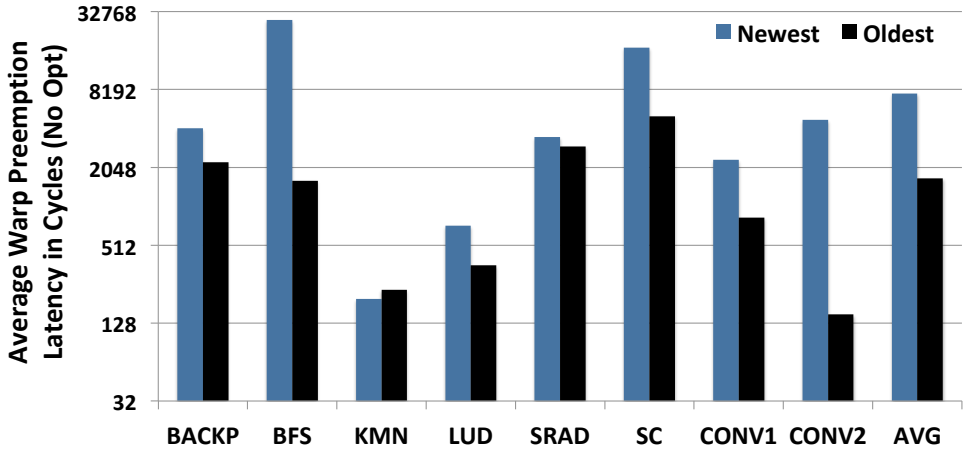
CTAs to SMXs until launching an additional CTA from *X* will exceed the shared memory size. The remaining CTAs from *X* must block until previous CTAs complete. Even though *X* is blocked, there are still unused warp contexts and registers available, which can be used by other GPU applications with low shared memory requirements. To measure this opportunity, we first profiled the Rodinia and Convolution benchmarks to identify the fraction of cycles where there are enough resources to support a *PGW* (free *PGW* context that does not require preempting a running warp) in Figure 5.9. As shown, the fraction of cycles a free *PGW* is available varies between applications, with the average around 50%. While not shown here, a cycle-by-cycle analysis highlights that a kernel goes through multiple phases with varying resource utilization, for example, as warps from a CTA begin to complete and a subsequent CTA has not yet been launched. These results highlight that there are sufficient opportunities to exploit underutilized resources for executing *PGW*s instead of preempting a running warp or requiring additional dedicated hardware to service the interrupt.

Furthermore, while an application may use all of the available thread contexts, it may still underutilize the large per-SMX register files, which limits the amount of resources needed to be saved on preemption. Figure 5.10 highlights the aver-

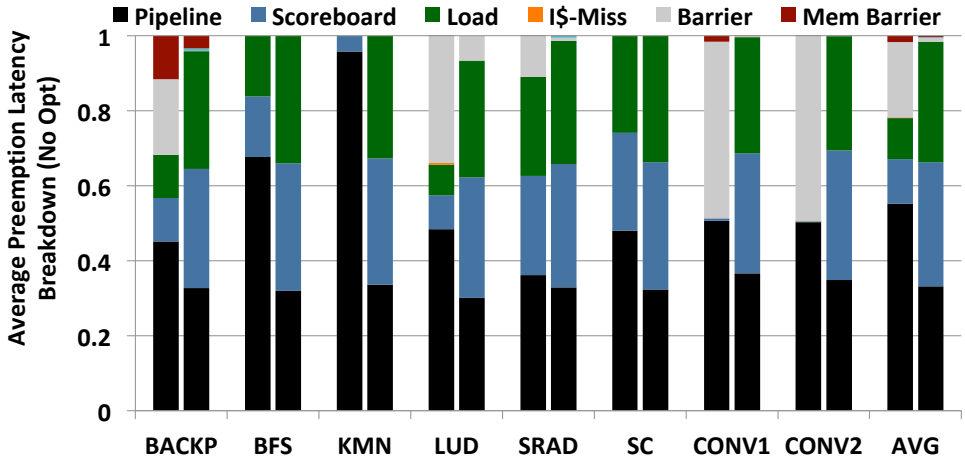
age register utilization (averaged over all cycles of the benchmark's execution) for the Rodinia and Convolution benchmarks on the Gem5-GPU simulator. We find that the evaluated benchmarks use ∼60% of the available registers. A *PGW* only requires 0.4% of an SMX register file (8 registers * 32 threads/warp = 256 / 65536 registers, assuming a full warp's worth of registers need to be reserved for the single thread in the *PGW*), which provides an opportunity to utilize the free registers instead of requiring register saving/restoring in the software ISR.

PGW Preemption Latency and Runtime

A key design goal for supporting interrupts on any system is low and predictable latency. However, this can be challenging to achieve when preemption is required and the to-be-preempted victim warp can be in any given state of execution in the pipeline. As such, the latency to preempt a running victim warp must be minimized. For any of the dedicated interrupt warp policies (Section 5.3.2), the preemption latency is zero since we always have a free interrupt warp context or hardware unit available to begin processing the interrupt immediately. Similarly, if *EDGE* includes a small hardware unit to launch the event kernel in response to an external event/interrupt, instead of using a *PGW*, the event kernel can be immediately transferred to the Event Kernel Table (15 in Figure 5.7). However, preempting a running warp requires flushing the victim warp's instructions from the pipeline. For ALU operations, as modeled in GPGPU-Sim, this can range from 4 to 330 cycles depending on the operation being performed and the precision. Long-latency memory operations, for example, load instructions that miss in the cache and must access DRAM, can take on the order of 1000's of cycles. Naively waiting for the victim warp to flush the pipeline results in unacceptably high scheduling latencies. Figure 5.11a measures the average victim warp preemption latency for interrupts that require preemption. We evaluate two victim warp selection policies, *Newest* and *Oldest*, where the most or least recently launched warps are selected to preempt, respectively. As our baseline GPU warp scheduler uses a greedy-then-oldest (GTO) policy, the oldest warp selection techniques tend to have a much lower preemption latency (∼4×) because the oldest warps are prioritized over the newest warps and are the first to complete. As such, the *PGW* selection technique



(a) Average preemption stall cycles without the victim warp flushing optimizations.



(b) Breakdown of preemption stalls for two victim warp selection policies (Left: Newest, Right: Oldest).

Figure 5.11: Interrupt warp preemption stall cycles.

alone can have a large impact on scheduling latency, which ranges from 2k-8k cycles on average (or 2.9-11.4 μs at 700 MHz) for the two warp selection policies. These are relatively high latencies before the *PGW* is even be able to begin executing the ISR.

Figure 5.11b measures the breakdown of causes for scheduling stalls. *Pipeline* means that a warp instruction is in the pipeline, *Scoreboard* means that the instruction has registers reserved in the scoreboard, *Load* indicates the warp is waiting for

in-flight loads, I$-Miss an instruction cache miss, and *Barrier* and *Mem Barrier* indicate that warps are waiting at thread or memory barriers respectively. These scheduling stall conditions are not mutually exclusive. For example, an in-flight load instruction also has a register reserved in the scoreboard. The trend for Oldest policy is very similar - the selected victim warp to preempt is waiting for load instructions to complete. The Newest policy leads to more diverse outcomes, such as waiting for thread barriers. This is a result of some evaluated benchmarks having thread barriers near the start of the kernel. For example, the initial portion of the CONV1/2 benchmarks load data from global memory to shared memory and then wait at a barrier. Selecting the newest warp tends to fall into this phase of the kernel, whereas the oldest warps are performing the convolution operation on the data in shared memory. Evaluating methods to optimize the warp scheduling to reduce preemption latency is an area for future work.

Figure 5.12 measures the reduction in preemption latency when applying each of the optimization techniques for victim warp flushing discussed in Section 5.3.3. Note the log scale y-axis. The average preemption latency is reduced by 35.9× and 33.7× for the Oldest and Newest policies respectively. At 700MHz, the preemption latency is $\sim$70*ns* (50 cycles) and $\sim$314*ns* (220 cycles) respectively. For Oldest, the largest benefit comes from dropping and replaying load instructions, which is explained by the large fraction of stalls on pending loads in Figure 5.11b. For Newest, many benchmarks are also stalled on barrier instructions. Only applying the barrier skipping, however, does not result in a large reduction in Figure 5.12 because BFS and SC, which are not blocked on thread barriers and have a large preemption latencies, dominate the average calculation. After reducing the preemption latency with the other optimizations, such as replaying loads, thread barriers result in a larger contribution to the average preemption latency, which is highlighted by the large improvement with all optimizations applied.

Along with the *PGW* scheduling latency, the ISR runtime should be minimized to quickly process the interrupt and reduce the impact on performance of any concurrently running tasks. We find that by prioritizing the *PGW*'s instruction fetch and warp scheduling over other concurrently running kernels, the ISR runtime can be reduced by 55% on average. However, depending on the rate and duration of interrupts, prioritizing a *PGW* can negatively impact other concurrently running

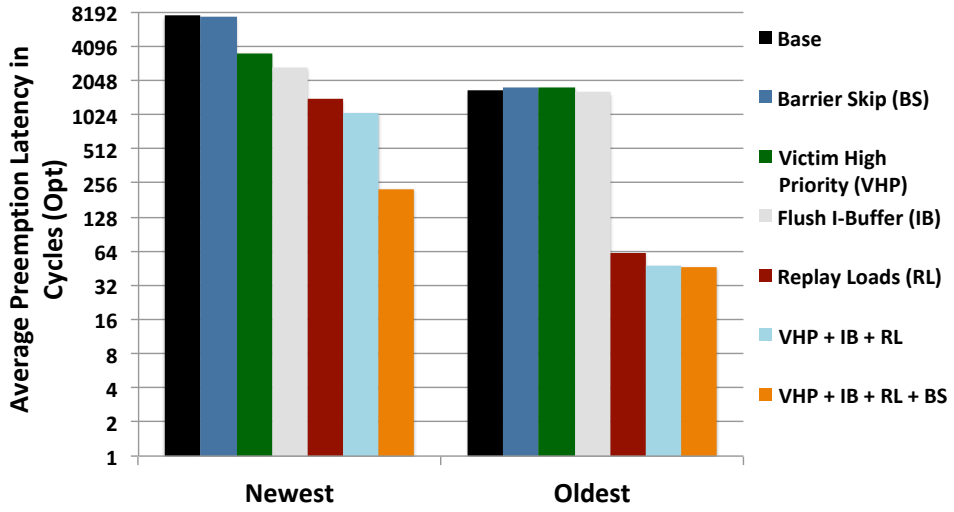


Figure 5.12: Preemption stall cycles with the victim warp flushing optimizations applied averaged across the Rodinia and Convolution Benchmarks. Base is the baseline preemption latency without any optimizations applied (Figure 5.11a). Barrier Skip immediately removes a victim warp if waiting at a barrier. Victim High Priority sets the victim warp's instruction fetch and scheduling priority to the highest. Flush I-Buffer flushes any pending instructions from the victim warp's instruction buffer. Replay Loads drops any in-flight loads from the victim warp and replays them when the victim warp is rescheduled after the ISR completes.

kernels' performance.

Impact of *PGW* on Concurrently Running Kernels

Figure 5.13 measures the impact of running a *PGW* on the IPC of concurrently executing kernels under the different *PGW* selection policies (Section 5.3.2), different interrupt rates (Slow/Fast), and different ISR durations (Short/Long). Slow and Fast interrupts are generated every 10k and 1.5k GPU cycles respectively. In these experiments, the ISR runtime is varied by iterating a delay loop. Additionally, the ISR does not actually launch an event kernel. With the delay loop, the average measured runtimes for Short and Long are $\sim$15$\times$ and $\sim$212$\times$ longer than the

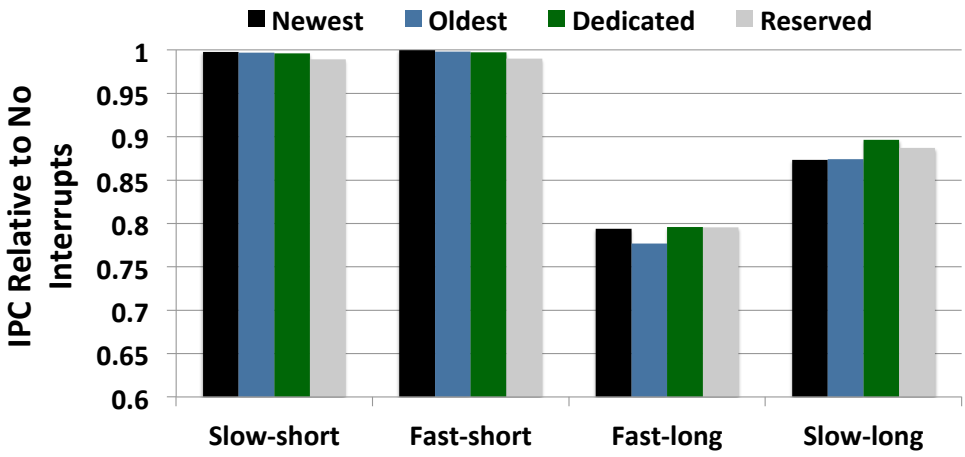


Figure 5.13: Average impact of the *PGW* selection, interrupt rate, and ISR duration on concurrent tasks' IPC (x-axis labels are <*interrupt rate*>-<*interrupt duration*>).

actual average ISR runtime measured in Section 5.6.2, respectively. All preemption and runtime optimizations described earlier are applied to the victim warp and *PGW*. When the ISR is short, the *PGW* has very little impact on the background task, regardless of the selection policy. However, reserving a warp for the *PGW* in the Reserved policy, hence reducing the amount of available parallelism to other tasks, does have a relatively higher impact on running tasks. When the ISR runs for a much longer time, the impact on the background task becomes much larger. The Dedicated and Reserved policies have slightly less of an impact on the background task than the preemption policies (Oldest/Newest), since running warps are not preempted indefinitely. Even though the Reserved policy continually reserves a warp context, all of the background task warps are able to make progress while the *PGW* is running. These results highlight that the impact on a background task is more due to sharing of execution cycles with the prioritized *PGW* than stalling the execution of a given warp. Furthermore, since the ISR runtime for Short is 15× longer than the measured actual ISR, the *PGW* and ISR have little impact on the performance of concurrently running kernels. As such, from the perspective of concurrently running applications, there is little benefit to having dedicated resources for the *PGW*.

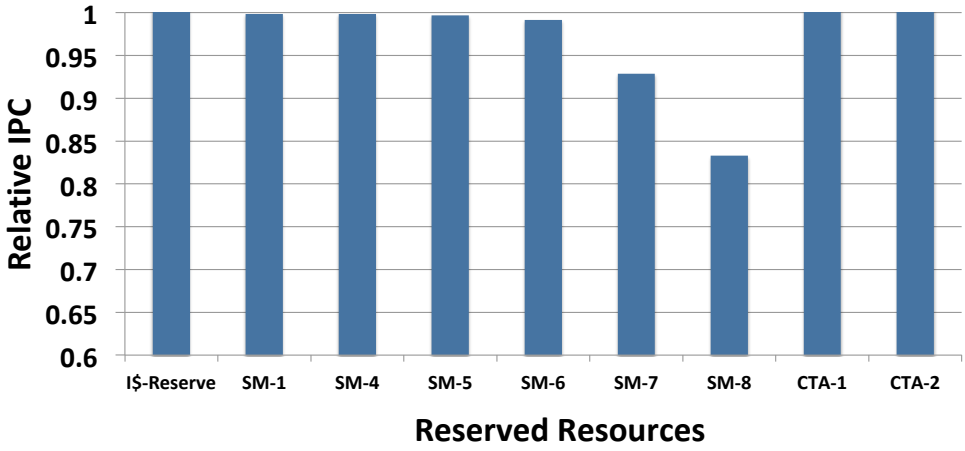


Figure 5.14: Impact on the Convolution kernel's IPC when reserving resources for the *PGW* and MemcachedGPU GET kernel.

5.6.2 Event Kernels

In this section, we evaluate event-driven kernels in *EDGE* by highlighting a multiprogrammed use-case for *EDGE* where the GPU is executing long running kernels for training a deep neural network (DNN), while also using the GPU to service a higher-priority network kernel, such as MemcachedGPU. Specifically, we evaluate launching multiple Memcached kernels, MEMC, through the *EDGE* flow with varying request rates, while performing a single Convolution kernel, CONV1, through the standard CUDA flow. We evaluate different prioritization techniques, P1 and P2 (Section 5.4.1), for sharing the GPU between the low and high priority kernels.

In addition to the different event kernel priorities, P1 and P2, we also evaluate different hardware reservation techniques for the event kernels to spatially share the GPU resources between concurrent kernels, such as reserving CTAs or entire SMXs. We first measure the impact on IPC of the CONV1 kernel when reserving hardware resources only for event kernels, but without running any event kernels. *SM-N* reserves *N* SMXs for an event kernel, which can not be used for the CONV1 kernel. *CTA-N* reserves *N* CTAs per SMX for the event kernel. Finally, *I$-Reserve* reserves entries for the ISR in the instruction cache (Section 5.3.4). Figure 5.14 highlights that there is negligible impact on performance when reserving the ISR

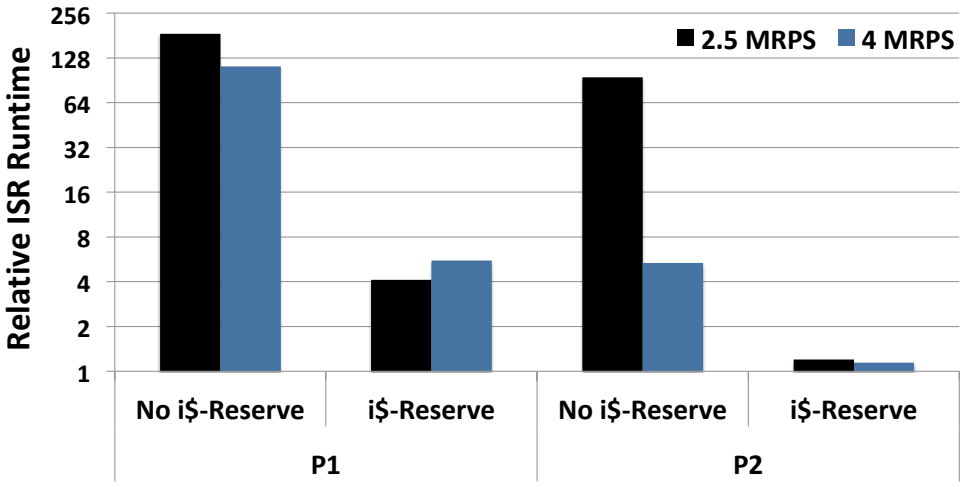


Figure 5.15: Average ISR runtime with CONV1 and MEMC kernels at varying MEMC launch rate relative to a standalone ISR. The runtime is measured with and without reserving the instruction cache entries for the ISR (i$-Reserve) and for both the P1 and P2 event kernel priorities. The ISR requires three entries in the instruction cache.

i$ lines, reserving 1-2 CTAs for the event kernel, or even reserving up to 6 SMXs. Rogers et al. [149] have identified similar behavior where performance can actually increase when reducing the kernel's parallelism (by limiting the number of active warps per SMX) due to improved cache access patterns. However, as we continue to decrease the number of SMXs available for the CONV1 kernel, the performance starts to drop. With 50% of the SMXs, CONV1's performance drops to 85%. Note that the performance for CONV1 does not drop linearly with the amount of compute resources removed, which suggests that the performance of CONV1 is memory bound.

In the following experiments, we launch multiple MEMC kernels at different packet rates, 2.5 MRPS and 4 MRPS, while running a single background Convolution kernel, CONV1. Figure 5.15 measures the ISR runtime slowdown for the different launch rates, priority levels, and with the ISR instruction cache entries reserved or not (three lines), relative to the ISR running in isolation. Without reserving the instruction cache entires for the ISR, the *PGW* ISR runtime is significantly increased ($\sim$100-200$\times$) relative to running in isolation. A large contributor to this

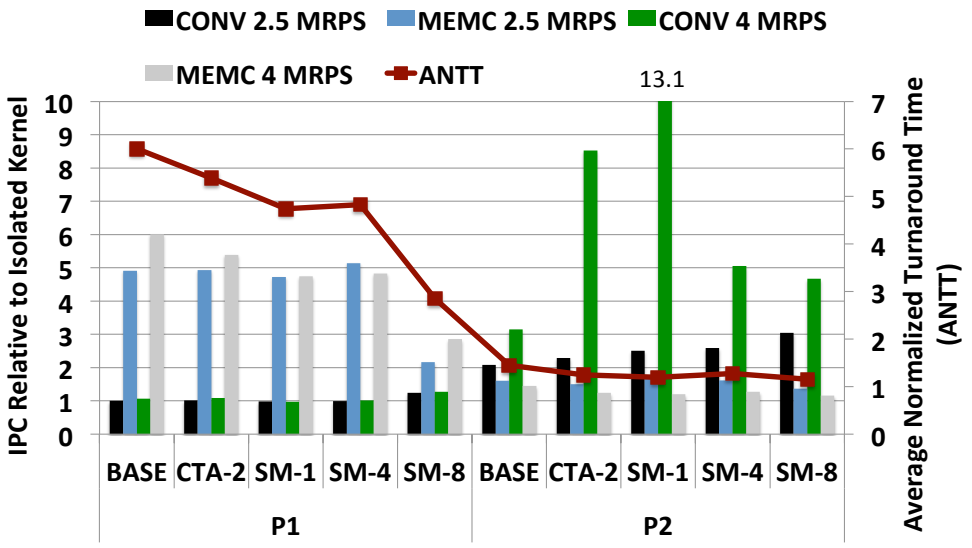


Figure 5.16: Runtime of the Convolution and Memcached kernels at different request rates and resource reservation techniques, relative to the isolated kernel runtimes, and average normalized turnaround time.

is that the multiple concurrently running kernels occupy the instruction cache, continuously evicting the ISR's cache lines. However, when reserving entries in the instruction cache for the ISR, the ISR runtime for P1 is reduced by 20-45× depending on the packet rate. With P2 at 4 MRPS, the higher packet rate causes MEMC kernels to be launched quicker (and overlapping), which limits the amount of time the CONV1 kernel can execute, hence reducing contention on the instruction cache. However, at 2.5 MRPS, CONV1 is able to resume execution, which results in a 80× reduction in ISR runtime when reserving instruction cache entries. This data highlights that reserving instruction cache entries for the ISR is necessary to improve ISR performance when there is contention with shared resources between concurrent tasks, such as the memory system. Since the ISR does not have a large code footprint, a small on-chip buffer could also be included to store the ISR's code to avoid reducing the instruction cache size for other GPU kernels.

Figure 5.16 measures the performance of MEMC and CONV1 when launching multiple MEMC tasks under the different launch rates (2.5 and 4 MRPS), priority levels (P1 and P2), and event kernel resource reservation techniques, relative to a

single MEMC and single CONV kernel run in isolation. For example, CONV 2.5 MRPS measures the performance of the CONV1 kernel (relative to CONV1 in isolation) while MEMC is running at 2.5 MRPS (relative to MEMC in isolation). We also measure the average normalized turnaround time (ANTT) [50] of the CONV1 and MEMC kernels. There is a large variation in the throughput of MEMC kernels per convolution kernel ranging from 2-10 for 2.5 MRPS and 2-70 for 4 MRPS for the different priority policies. The performance of CONV1 also varies widely. For example, SM-1 at P2 runs 70 MEMC kernels before CONV1 completes, but at a 13.1X slowdown in CONV1. The large slowdowns for CONV1 with P2 are caused by the higher priority placed on the MEMC kernel. Whenever a MEMC kernel is pending or running, the CONV1 kernel is blocked. At 4 MRPS, the MEMC kernels arrive at the GPU more frequently, which significantly reduces the CONV1 kernel's ability to run.

The ANTT takes into account the number of kernels that execute on the GPU concurrently and the slowdown of each kernel relative to being run in isolation. An ANTT of 1 is ideal. With P1, the CONV1 kernel is able to continue scheduling CTAs alongside MEMC, which impacts the MEMC throughput drastically, resulting in a high ANTT between 2.9 and 6. However, as the reserved resources are increased and higher priority is given to MEMC (P2), we measure a large reduction in ANTT to 1.44 at BASE P2 and 1.15 at SM-8 P2. The significant improvements in ANTT between the P1 and P2 priorities indicate that spatial multitasking alone may not be enough to maximize the overall system throughput due to contention on other shared resources, such as the memory system. This is highlighted in Figure 5.16, where both CONV1 and MEMC are memory bound applications. For example, even with SM-8 in P1, the ANTT is $\sim 2\times$ higher than BASE in P2. While SM-8 reserves half of the GPU's SMXs for MEMC, the P1 priority allows the CONV1 CTAs to continue running on the remaining 8 CTAs while MEMC is running, which slows down both the CONV1 and MEMC kernels. In P2, however, all of the CONV1 CTAs are blocked until the MEMC kernel completes, which, even with no resources reserved for the MEMC kernel, enables the MEMC kernel to complete faster.

These results highlight the potential to maintain a high quality of service (QoS) for high-priority event kernels, while concurrently processing a lower-priority

background application to improve GPU utilization. Additionally, with *EDGE*, the CPU is not required to manage the launching or completion of the event kernels, freeing it up to work on other tasks or enter a lower power state.

5.6.3 Wait-Release Barrier

Finally, we evaluate the second use-case of the *PGW*s for event-driven GPU execution: the wait-release barrier. Specifically, we evaluate the benefits provided by the wait-release barrier compared to a persistent thread environment (*PT*) to control GPU execution. For these experiments, we set up a *PT* framework with local CTA-level work queues, where each persistent CTA (pCTA) has its own entries in the work queue to avoid any contention with other pCTAs. There are 16 pCTAs in total, one per SMX. Each pCTA is responsible for checking the work queue for available work and then performing the corresponding kernel. For this experiment, we use an empty kernel with no actual processing, such that only the *PT* and *PGW* overheads are measured. For *PT*, each pCTA continuously polls the work queue in global memory for new work. We then modified the *PT* framework to include a wait-release barrier prior to checking the memory queue for work in the polling loop. Each pCTA blocks on a wait-release barrier until an interrupt is received, which releases all blocked pCTAs. Once released, each pCTA checks the work queue for work and continues to process the empty kernel if available, or returns to the wait-release barrier if not. Note that this naive implementation releases all pending wait-release barriers (16 pCTAs), which results in a larger number of queries to the work queue than necessary.

Due to the polling nature of *PT*, when no work is available, we expect an increase in the total global load memory traffic and number of dynamic warp instructions to query the work queue. Figure 5.17 measures the reduction in global load traffic when guarding the work queue with a wait-release barrier for three different interrupt coalescing sizes, where an interrupt is generated for every 1 (WRB-1), 8 (WRB-8), and 32 (WRB-32) pCTA tasks inserted into the global work queue. We also vary the launch frequency of each pCTA task from as fast as possible (Launch delay == 0) to a larger delay (Launch delay == 100). Here, the delay corresponds to the number of iterations of an empty delay loop performed on the CPU before

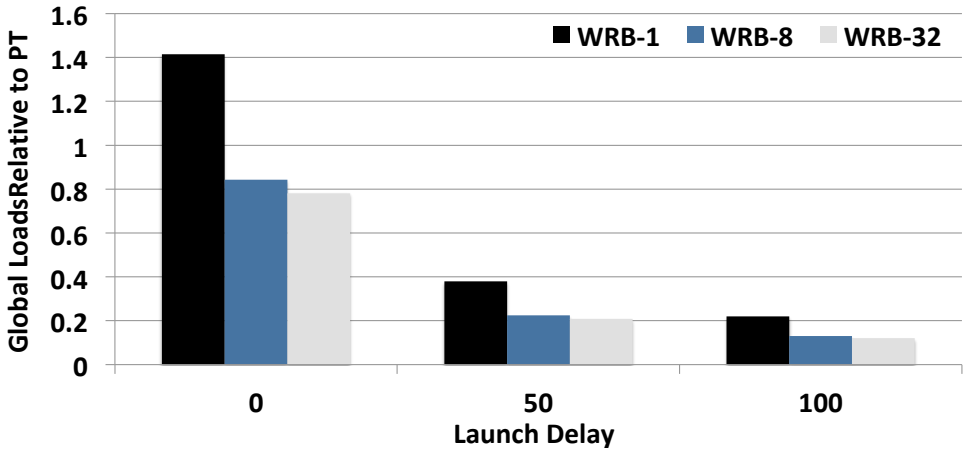


Figure 5.17: GPU global load instructions issued relative to Persistent Threads (*PT*).

launching the next GPU task. When the launch frequency is high, triggering an interrupt to release the wait-release barrier for every pCTA task results in additional global loads due to the *PGW* running on all 16 SMXs to release the blocked pCTAs, and only a single task being available to process. When the launch frequency is decreased or when the wait-release barrier request coalescing is increased, wait-release barrier results in a significant decrease in global load traffic. For example, even when the CPU is pushing tasks to the GPU as quickly as possible in a loop, triggering an interrupt every eight tasks results in a 20% reduction in global load traffic. Furthermore, when a delay is introduced in between task launches, having an interrupt per task results in a large reduction in global load traffic. As the launch delay is increased, the impact of coalescing interrupts on the total task latency is reduced further. The main takeaway from this experiment is not the absolute value of the relative improvements, but instead the ability for the event-driven programming model and wait-release barriers to reduce the GPU global load traffic under variable task rates.

In addition to global load traffic, blocking the pCTAs from continuously polling will impact the number of dynamically executed instructions. Figure 5.18 measures the effect on dynamic issued warp instructions on the GPU for the same experiment as above. In the baseline *PT*, each pCTA has a single warp spin-looping

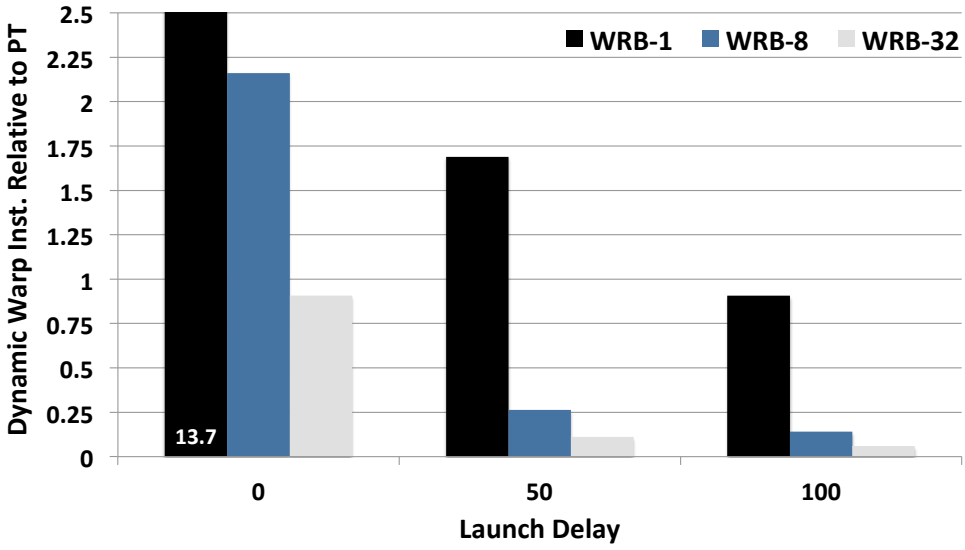


Figure 5.18: Dynamic warp instructions issued relative to Persistent Threads (*PT*).

on the work queue. However, wait-release barrier stalls all warps in the pCTA when waiting at the barrier, which can reduce the amount of unnecessary warp instructions used for polling. This can be seen by the large reductions in the number of dynamic instructions when the launching delay and interrupt coalescing are increased. Similar to the number of global loads, there can be an increase in the number of dynamic warp instructions with wait-release barrier when the launch frequency is high or interrupt coalescing is low due to the increased number of *PGW*s running the ISR on the GPU than tasks being launched. This is very apparent with WRB-1 at 0 launch delay, which increases the total number of dynamic warp instructions by 13.7× compare to *PT*, since there is no kernel processing and 16× the number of required *PGW*s are launched for every new task.

An obvious enhancement to the current wait-release barrier implementation would be to selectively release only the number of pCTAs required to service the new tasks, or release the specific pCTA waiting for the result of an RPC. This would require modifying the *PT* framework to have a shared global work queue with synchronization locks, as well as a global structure for recording which pCTAs are currently blocked at a wait-release barrier as candidates to release on an interrupt.

Evaluating these enhancements is left to future work.

Lastly, the use of the *PGW* to release the wait-release barriers increases the latency for the pCTA to identify when a task is available to process. This increase in latency is equal to the *PGW* ISR runtime, which as shown in Section 5.6.2, is dependent on the other kernels being concurrently run with the ISR.

5.7 Summary

This chapter proposes *EDGE*, a novel event-driven programming model consisting of a set of hardware mechanisms and software APIs that can increase the independence of specialized processors, such as GPUs, from their reliance on CPUs. *EDGE* makes use of a fine-grained warp-level preemption mechanism, which initiates the execution of privileged GPU warps (*PGW*s) that can launch work internally on the GPU from any device in a heterogeneous system without involving the CPU. This can help to reduce task launching latency from an external device and free up the CPU to either work on other tasks or enter a lower power state to improve efficiency. The *PGW*s can also be used to support a new form of CTA barrier, the wait-release barrier, which enables running CTAs to block indefinitely until a *PGW* releases the barrier in response to some external event. This is useful, for example, to reduce the polling overheads of the persistent GPU thread style of programming.

The *PGW* requires GPU resources to operate. We evaluated several techniques for efficiently selecting, flushing, and preempting a warp from concurrently running applications, such that the time to schedule and process an interrupt with a *PGW* is minimized. On a set of GPU benchmarks, we found that there is a free warp context available for a *PGW* 50% of the time, on average. When no free warp contexts are available, we showed that the evaluated flushing optimizations can reduce the average *PGW* preemption latency up to $\sim$36$\times$ (to $<$50 cycles) on the Gem5-GPU simulator.

While traditional GPU kernels require a host CPU to configure and launch each kernel, with *EDGE*, the host CPU pre-allocates and configures GPU event kernels a single time and utilizes the *PGW*s as a mechanism to internally launch any subsequent GPU tasks via standard memory operations and interrupts. By

removing interference between the CPU and GPU, we estimate that *EDGE* can increase CPU throughput by 1.17× and GPU networking throughput by 3.7× when running SPEC2006 on the CPU in parallel with a networking application, MemcachedGPU, on the GPU.

Chapter 6

Related Work

This chapter discusses and contrasts the key related works with the research presented in this dissertation. This chapter is partitioned into two high-level sections, which describe related work on using the GPU to accelerate server-based applications and network processing in Section 6.1, and related work on event-driven GPU execution and improving GPU system support in Section 6.2.

6.1 Related Work for Accelerating Server Applications and Network Processing on GPUs

The work performed in Chapter 3 and Chapter 4 focuses on improving the performance and efficiency of Memcached, a server-based application, using GPUs, and on using GPUs to accelerate network processing. This section is further partitioned between these two categories.

6.1.1 Memcached and Server-based Applications

Many prior works have looked at improving the throughput and scalability of Memcached or other general key-value store applications, through software or hardware modifications. While much of the focus in this dissertation is on Memcached, one of the long term goals of our research is to provide a general framework for accelerating high-throughput network services on GPUs. Many of these works propose optimizations that are complementary to our work and can be evaluated to pro-

vide additional improvements to the performance and efficiency of Memcached on GPUs using *GNoM*. This section summarizes and contrasts a set of key related works with this dissertation.

Concurrent with our work in Chapter 3, Berezecki et al. [24] presents a many-core architecture, the Tilera TILEPro64 64-core CPU, used to accelerate Memcached. In their work, different parts of Memcached are modified to run on individual processors, such as network workers, hash-table processes, TCP and UDP cores, and the operating system itself. Although the focus of their work and ours is similar (i.e., accelerating Memcached on a many-core architecture), our work differs in the method of achieving this goal, focusing on the feasibility of running such an application on a GPU and providing a detailed characterization of Memcached on hardware and on a simulator.

Andersen et al. [11] propose a log-structured data store system that utilizes lower power/performance (*wimpy*) CPUs and flash memory to maintain performance and reduce power consumption for key-value store applications. This is effective for key-value store applications, such as Memcached, in which large amounts of computation are replaced with long I/O operations and network latencies that are not significantly affected by low clock frequencies. The use of multiple wimpy cores with an optimized memory system to improve the efficiency of memory-bound applications is also similar to the use of GPUs in our work, which contain large numbers of small in-order cores and high-bandwidth memory. However, the GPU architecture is able to provide high levels of computational throughput for structured compute-bound applications, whereas the wimpy cores may not be able to efficiently scale-up performance for other types of compute-intensive applications.

Wiggins et al. [177] improve the scalability of Memcached on CPUs through a set of software enhancements. This work removes the global cache lock overheads in Memcached. Specifically, this work evaluates striped locking on the baseline hash-chaining hash table and a bagging LRU mechanism to replace the global lock with atomic CAS instructions. Our work also proposes removing the global locks on the hash table and LRU management; however, our work evaluates an alternative set-associative hash table and per-set shared/exclusive locking mechanism, which reduces complexity and significantly improves performance and energy-efficiency

on the GPU architecture.

Fan et al. [52] evaluate a set of software enhancements to improve the performance and memory efficiency of Memcached on a CPU. Memc3 proposes a novel optimistic cuckoo hashing mechanism with set associative cuckoo entries, which improves memory efficiency over the baseline Memcached and efficiently supports concurrent accesses to the hash table. This is achieved by separating the cuckoo path discovery phase from the update phase and only updating a single key move at a time, removing the opportunity for false misses when reorganizing the hash table. In Memc3, writes to the hash table are serialized, but reads can operate concurrently with writes. This enables a set of optimizations to further improve the performance of Memcached, such as replacing the global LRU queue with an approximate LRU based on the CLOCK replacement algorithm, and the use of optimistic locking to avoid expensive synchronization overheads on shared data structures. In our work, we also evaluate modifications to the hash table, proposing a more GPU-friendly set-associative hash table, which maintains similar read performance but improves write performance. Furthermore, our set-associative cache uses per-set exclusive locks on writes, which supports concurrent write requests. Our work also evaluates approximate LRU policies, which consider local LRU per hash table set instead of per memory slab, reducing the complexity compared to Memc3. MemcachedGPU borrows an optimization from Memc3, which compares small hashes of the key to avoid expensive key comparisons to reduce branch divergence among GPU threads. Overall, MemcachedGPU improves the read request throughput over Memc3.

Lim et al. [106] propose a state of the art in-memory key-value store application, MICA, on CPUs. MICA rethinks the design of a key-value store application, such as Memcached, to improve scalability under parallel access. MICA achieves very high throughputs on the order of 75 MRPS for 8B keys (half the minimum size evaluated in Chapter 4), using two 8-core CPUs with four 10 Gbps NICs (eight ports). MICA partitions keys across cores to improve locality and reduce interference between parallel cores. MICA can operate in exclusive-read, exclusive-write mode, which removes the need for synchronization as only a single core operates on a partition, or concurrent-read, exclusive write mode, which uses optimistic locking to reduce the synchronization overheads. The value storage is completely

redesigned from Memcached, which includes an efficient append only log structure, similar to log structures used in flash memory (e.g., FAWN [11]), for their lossy key-value store design and efficient memory allocators with segregated fits for their lossless key-value store design. Similar to MemcachedGPU, the hash table is implemented as a set-associative cache. Also similar to MemcachedGPU, MICA reduces the UDP networking overheads through direct NIC access and user-level network processing (using Intel DPDK [79]). Furthermore, to take advantage of the sharded key-space across cores, MICA uses RSS support in modern NICs to direct requests to the specific core responsible for that key. However, this requires the client application to be modified to include knowledge of the server-side key-space partitioning; dropping throughput by 55-60% without this optimization. MICA represents the high-end for CPU performance on key-value store applications, taking a holistic approach to redesign all aspects of the application to take advantage of multiple different hardware and software features available on contemporary CPUs and NICs. In contrast, our work explores how the GPU, a stranger to this type of streaming network application, can be used to achieve high levels of performance and efficiency, while minimizing the modifications to Memcached (e.g., only the hash table and eviction mechanisms are modified – the value storage, hashing mechanism, Memcached protocol, and client application remain the same). We believe that by taking a similar approach of fully redesigning a key-value store specific to a GPU, as well as including additional NIC and GPU hardware, the performance and efficiency of MemcachedGPU could be significantly improved. This is left to future work.

Blott et al. [26] evaluate the potential for accelerating Memcached on FPGAs. Other works by Lim et al. [108] and Chalamalasetti et al. [30] also evaluate using FPGAs for Memcached; however, Blott achieves the highest performance. In [26], the FPGA is able to achieve 10Gbps processing for all Memcached key sizes, extremely low latencies under 4.5 uS per request, and high efficiencies of 106.7 KRPS/W. This is achieved by pipelining requests through a dedicated hardware path on the FPGA architecture. Given the low request latency and high efficiency, the FPGA is a strong candidate for accelerating such workloads. Similar to MemcachedGPU, this work proposes a set-associative based hash table with local LRU management on the FPGA. Istvan et al. [85] expands on the description of the

FPGA hash table. However, in Chapter 4 we show that we can achieve similar throughputs at significantly lower costs on a lower power GPU. Additionally, while the FPGA architecture enables high performance and energy-efficiency, the flexibility of the general-purpose GPU architecture (e.g., ease of programming, multitasking) may outweigh some of the efficiency gains in a datacenter environment, where workload consolidation is necessary to improve server utilization and reduce costs.

Nishtala et al. [124] describe enhancements to improve Memcached at Facebook. As presented, their Memcached deployment is the largest in the world, responsible for handling billions of requests per second. They propose multiple enhancements to improve both the scalability of Memcached across servers, as well as improve the performance of an individual server. Many of the scalability improvements are complementary to our work, such as reducing the request latency through request batching (multi-GET), request parallelization (multiple Memcached servers), client sliding request windows (similar to TCP), replication across pools of Memcached servers (within and across regions) to reduce latency and improve efficiency and consistency, or adaptive slab allocators for item value storage. Other enhancements, such as using UDP over TCP for *GET* requests and using fine-grained locking mechanisms, are explored in our work.

Other works, such as Jose et al. [88, 89] and Dragojevic et al. [45], identify limitations with the networking overheads in Memcached and evaluate the use of RDMA over Infiniband and Ethernet network hardware, respectively, to improve Memcached performance. Jose et al. [88] propose an efficient unified communication runtime (UCR), containing enhancements to the RDMA connection establishment and support for active messaging over RDMA, to enable middleware applications to efficiently utilize RDMA capable interconnects. They then propose multiple modifications to Memcached to support UCR on Infiniband NICs. Jose et al. [89] propose a hybrid reliable connection and unreliable datagram protocol on top of UCR, and a set of enhancements to support the hybrid communication mechanism. Similar to the use of UDP over TCP, they find that the performance and scalability of Memcached with Infiniband can be improved with such hybrid communication mechanisms. FaRM [45] propose multiple modifications to Memcached to support RDMA primitives and distributed memory management across

a cluster of RDMA enabled Memcached servers on Ethernet hardware. FaRM also addresses locking issues in Memcached. Similar to our work, FaRM evaluates a new type of hash table (associative hopscotch hashing) to better support the underlying hardware changes. In contrast to our work, which reduces the network overheads by accelerating UDP network processing on the GPU, these works orthogonally reduce the network overheads through the use of different networking mechanisms/hardware.

Following our initial analysis of Memcached on GPUs in Chapter 3, other works have evaluated using GPUs to accelerate Memcached. For example, Dimitris et al. [44] propose Flying Memcached, which uses the GPU for performing the key hash, while all other processing remains on the CPU. Flying Memcache also addresses the networking bottlenecks in Memcached, achieving sizeable improvements in throughput by performing the UDP network processing in user-space on the CPU. In contrast, our work performs all Memcached *GET* request and UDP network processing on the GPU, addressing many of the challenges associated with a full system implementation of a GPU networking application. Furthermore, our work improves throughput and efficiency over Flying Memcache.

Concurrent with the work in Chapter 4, Zhang et al. [184] also propose using GPUs to accelerate in-memory key-value stores. They use two 8-core CPUs, two dual-port 10 GbE NICs (max. 40 Gbps), and two NVIDIA GTX 780 GPUs to achieve throughputs over 120 MRPS. In contrast to MemcachedGPU, these results use a smaller minimum key size (8B vs. 16B), use a compact key-value protocol independent from the Memcached ASCII protocol, perform the hashing function using AES SSE instructions, and batch multiple requests and responses into single network requests through multi-GETs to reduce per-packet network overheads. However, for *GET* requests, the GPU is only used to perform a parallel lookup for the corresponding CPU key-value pointer in a GPU Cuckoo hash table. All other processing, including UDP network processing, request parsing, key hashing, and key comparisons, are done on the CPU. In contrast, the goal of our work was to achieve 10 GbE line-rate performance for all key-value sizes while performing all of the UDP network processing and *GET* request processing on the GPU. We believe that MemcachedGPU could achieve sizable improvements in performance and efficiency when modifying the Memcached packet format and hashing

mechanism to be more GPU-friendly. We have not yet evaluated the potential for additional scaling of *GNoM* and MemcachedGPU using a more powerful CPU with multiple NICs and GPUs.

Prior and concurrent work have evaluated accelerating other types of traditional server workloads on GPUs, such as HTTP workloads by agrawal et al. [3], or databased queries by Bakkum et al. [18] and Wu et al. [179]. Similar to our work, these works highlight the benefits of exploiting request level parallelism on the GPU through batching requests with similar behavior. However, these works focus solely on the workload specific processing, which can limit the potential end-to-end performance improvements gained by the GPU. In contrast, our work considers a complete end-to-end system implementation, which performs both the network and application processing on the GPU. Evaluating how *GNoM* could be generalized to support accelerating the network path for these GPU server applications is an interesting direction for future work.

A core operation when processing a *GET* request is the hash. Because our focus was parallelizing independent requests, the hash algorithm is computed by each GPU thread (work item) individually. Massively parallel hashing algorithms, such as the one implemented by Al-Kiswany et al. [5] in StoreGPU, provide significant performance increases when the data being hashed is large. However, the keys hashed in this study were all less than 128 bytes and would not benefit from the efficient divide-and-conquer techniques proposed in their work.

6.1.2 GPU Networking

This section discusses some of the key prior works related to networking and packet-based processing on GPUs.

Related work by Kim et al. [96] present GPUnet, a networking layer and socket level API for GPU applications. Similar to *GNoM*, GPUnet improves the support for applications requiring networking I/O on GPU accelerators. GPUnet provides a mechanism for GPU threads (specifically CTAs) to send and receive network requests directly from the GPU, which is achieved through persistently running CPU and GPU software frameworks. The GPU initiates network requests (send or receive) by sending RPCs to the CPU threads, which carry out the requested oper-

ation and return the results to the request CTAs. In contrast, our work maintains the GPU offload accelerator abstraction by launching GPU network kernels in response to network events, removing the need for persistent GPU kernels. Furthermore, GPUnet is designed for Infiniband hardware and is directed towards more traditional GPU applications, such as image processing and matrix multiplication. The current cost of Infiniband hardware is a large contributor to the restricted usage outside of HPC. Our work instead targets commodity Ethernet hardware typically used in datacenters, which requires processing network packets in software, and evaluates a non-traditional GPU application, Memcached.

Following the work in Chapter 4, Daoud et al. [38] (GPUrdma) expand on GPUnet to completely bypass the CPU for sending and receiving network packets to/from the Infiniband NIC. This is achieved through the use of persistently running GPU kernels that directly write to the Infiniband NIC's doorbell register for sending packets and directly receive notifications of incoming packets through memory writes by the Infiniband NIC. GPUrdma is evaluated on network ping-pong and multi-matrix-vector product applications. In contrast, our work focuses on accelerating the end-to-end processing for a more complex server-based application using Ethernet NICs, requiring UDP packet processing on the GPU, and does not rely on the use of persistent kernels, which enables more opportunities for GPU workload consolidation during varying traffic demands. Furthermore, *EDGE* can improve efficiency by removing the CPU from any GPU task management and removing the requirement for persistent GPU kernels.

Other works have also evaluated packet acceleration on GPUs such as PacketShader by Han et al. [65], GASPP by Basiliadis et al. [170], and Network Balancing Act (NBA) by Kim et al. [95]. Similar to *GNoM*, these works exploit packet-level parallelism through batching on the massively parallel GPU architecture, requiring a host framework to efficiently manage tasks and data movement with the GPU. PacketShader [65] implements a high-performance software router framework on GPUs to accelerate the lower-level network layer, such as packet forwarding and encryption, whereas GNoM and MemcachedGPU presented in this work focus on transport and application levels.

GASSP [170] is the first work to evaluate stateful packet processing (TCP) on GPUs and highlights the ability for the GPU to achieve significant performance im-

provements over a GPU-based implementation where the packet processing occurs on the CPU. In contrast to our work, GASSP requires larger batch sizes (increases packet latency), uses main CPU memory for storing packets (does not make use of GPUDirect), and evaluates GASSP on packet processing based applications, such as firewalls, intrusion detection, and encryption. Using the insights from GASPP to support stateful packet processing in *GNoM* is an interesting direction for future work (Section 7.2).

NBA [95] proposes a software-based packet processing framework that employs a modular approach, following the Click [97] programming model. NBA abstracts the low-level architecture-specific optimizations in modern NICs, CPUs, and GPUs, and load balances network traffic between heterogeneous devices to reduce programming complexity and maximize performance without manual optimizations. Similar to our work, NBA bypasses the Linux kernel network processing; however, NBA transfers packets to CPU memory using Intel's DPDK instead of directly transferring to the GPU (as is done in *GNoM* for direct GPU processing). NBA proposes abstractions to offloadable accelerators (e.g., GPU), which specify a CPU-side function, GPU-side function, and the required input/output data. The NBA framework handles data transfers and kernel invocations internally (if offloading to the accelerator is expected to improve performance), and provides optimizations to support data reuse between offloadable elements. Similar to GASPP, NBA is a packet processing framework and is evaluated on individual packet processing applications, such as packet filtering, encryption, and pattern matching. In contrast, our work takes a holistic approach starting from the application layer and optimizing the end-to-end network flow for such applications on the GPU - performing both the network processing and application processing concurrently on the GPU. Furthermore, our work proposes enhancements to the GPU hardware and programming model to improve support for future networking applications. Combining insights from our work and NBA could be useful to improve the GPU offload performance in a modular packet processing framework on heterogeneous systems when the choice of offload accelerator is known (or can be predicted) in advance.

6.2 Related Work on Event-Driven GPU Execution and Improving GPU System Support

This section presents the related work with Chapter 4 and Chapter 5. Some of the works discussed in the previous section are included in the context of this section as well.

The mechanisms employed by *EDGE* to communicate tasks between a device and the GPU to support event-driven execution are similar to in-memory submission/completion queues that enable a CPU to communicate with external devices, such as NVME [49]. NVIDIA's Multi-process service [129], HSA [55], and persistent GPU threads [63] also propose using user-level, in-memory work queues for communicating tasks with the GPU to improve multi-process support, reduce kernel launch latency, or improve GPU independence from the CPU and GPU hardware task schedulers. MPS and HSA require a separate runtime environment (e.g., CPU daemon process, hardware unit, or scalar processor) to recognize when work has been submitted into the work queues, for example, via writes to global memory or doorbell registers. *EDGE* proposes using privileged GPU warps (*PGW*) on the existing GPU vector compute resources for processing new tasks inserted into the in-memory work queues. The *PGW*s can be initiated by interrupts or writes to doorbell registers. Additionally, *EDGE* avoids the re-configuration of kernels through event kernels.

LeBeane et al. [101] present the closest related work to *EDGE*, Extended Task Queuing (XTQ), which proposes active messages for heterogeneous systems over Infiniband networks using remote direct memory access (RDMA). This work has the same goal of removing the CPU from the critical path for launching tasks on the GPU from remote machines. XTQ focuses on integrated CPU/GPU systems, such as AMD's Accelerated Processing Units (APU), which utilize HSA and HSA's user-level command queues to launch GPU tasks from the user space, instead of interacting with the GPU driver and OS on the CPU. As described in Section 5.4.1, the *EDGE* event submission/completion queues are similar to those used in HSA, and hence similar to XTQ. XTQ proposes enhancements to the Infiniband NIC to construct XTQ active messages that can trigger GPU tasks via the HSA work queues. XTQ registers GPU kernels with the Infiniband NIC, such that the RDMA

write operations only need to specify a kernel ID to launch the corresponding task on the GPU. This is similar to *EDGE*, which registers event kernels with the GPU and uses a kernel ID to specify which event kernel to launch on an event. The main difference between XTQ and *EDGE* is that XTQ pushes the extensions to the NIC and requires HSA-enabled GPUs, whereas *EDGE* pushes the extensions directly to the GPU. As such, *EDGE* requires only that the GPU and external device are able to communicate with each other via shared memory and that the external device can be configured to send specific interrupt IDs to the GPU or write to specific addresses in GPU memory (e.g., doorbell registers). This enables devices other than a NIC, such as an FPGA, to utilize the same *EDGE* API to launch event kernels directly on the GPU instead of requiring a similar XTQ framework per device on top of HSA. Lastly, the HSA user-level work queues, as described in XTQ, use an HSA Command Processor (CP) to recognize when the doorbell registers have been written to to begin processing the new task in the user-level work queue. However, it is not clear exactly what the CP is in their evaluation, for example, a hardware unit, a small scalar processor residing on the GPU, a CPU daemon software framework as in NVIDIA's MPS, or some other mechanism. In *EDGE*, we explore a specific implementation of the CP, the *PGW*, which utilizes existing GPU vector compute resources and fine-grained warp-level preemption mechanisms to efficiently process tasks in the user-level work queues.

Recent work by Suzuki et al. [160] also propose an event-driven GPU programming model, GLoop, consisting of GPU *callback* functions invoked on events, such as a file read completion. Similar to GPUnet [96] and GPUfs [158], GLoop employs persistent CPU and GPU runtime event loops, which continuously poll for RPC events from either device, or require re-launching kernels from the host upon RPC completion. Unlike these works, GLoop enables multiple different GPU applications to concurrently execute on the same persistent thread-like framework by efficiently scheduling callbacks from different applications. While event kernels can be thought of as callback functions, *EDGE*'s privileged GPU warp architecture and fine-grained warp-level preemption mechanism can remove the requirement for continuously running GPU event loops, while minimizing the impact on performance required to re-launch full kernels. For example, the GPU-side event loops described in these works could be implemented directly by a *PGW*, or as an event

kernel launched by a *PGW*.

NVIDIA announced enhancements to GPUDirect RDMA with GPUDirect Async [153], which expose parts of the networking flow to CUDA streams to remove the CPU from the critical path when scheduling kernels in response to network events on Infiniband hardware. With GPUDirect Async, the CPU pre-launches a GPU networking kernel in a CUDA stream and configures the corresponding network receive buffers in GPU memory. The GPU kernel is asynchronously executed when receiving the network packets directly from the Infiniband NIC. This removes the GPU kernel launch from the CPU on the network critical path and enables the CPU to enter an idle state earlier while waiting for the packet receive, not just during the GPU kernel processing. GPUDirect Async has a similar goal as *EDGE* for removing the CPU from the critical path by pre-registering the GPU kernel; however, GPUDirect Async requires configuring every kernel launch, whereas *EDGE* reuses the same event kernel registration for subsequent network event-driven kernels. Additionally, the GPU kernel in GPUDirect Async returns control back to a waiting CPU thread on completion, whereas *EDGE* enables completely removing the CPU from any pre or post kernel processing by supporting direct communication to and from external devices in the system.

Multiple works have evaluated preemption/context switching, multiprogramming, and priority mechanisms on GPUs [33, 92, 117, 143, 157, 162, 176, 178, 183]. Zeno et al. [183] propose an I/O-Driven preemption mechanism for GPUs, GPUpIO, which removes the spin locking on persistent CTAs waiting for long I/O operations. Instead, persistent CTAs initiate self-preemption after executing expensive I/O or RPC operations (e.g., network operations in GPUnet [96] or file operations in GPUfs [158]) through a set of software checkpoint-restore operations. When the I/O or RPC operation completes, the CTA context is restored in software to resume execution following the operation that initiated preemption. GPUpIO also makes the case for adding hardware support to yield running CTAs, similar to existing CPU mechanisms. Tanasic et al. [162] propose two kernel preemption mechanisms, explicit context switching (requires saving and restoring thread contexts) and SM draining (blocks new CTAs and waits for currently executing CTAs to complete), support for multiple kernel execution, and corresponding architec-

tural enhancements. Park et al. [143] propose a collaborative and configurable approach to minimize GPU kernel preemption overheads depending on the kernel and active CTAs' states; consisting of the context switching and SM draining techniques proposed in [162], and an additional SM flushing technique, which drops and replays CTAs from idempotent kernels. Kato et al. [92] propose a CPU runtime engine for managing kernel priorities by breaking up large memory transfers and adding a priority-aware software kernel scheduling layer prior to launching kernels on the GPU. Wu et al. [178] propose FLEP, a compilation and runtime software framework for transforming GPU kernels into preemptable kernels to enable temporal and spatial multitasking. This is achieved by breaking the kernel down into CTA-level tasks (similar to persistent threads) and having the CPU runtime schedule CTAs from different kernels depending on given priorities and policies. Chen et al. [33] propose Effisha, a similar framework to FLEP, which applies compiler transformations to the CPU and GPU application and includes CPU/CPU runtimes to work on CTA-level tasks in a persistent thread style fashion. Wang et al. [176] propose simultaneous multikernel (SMK), a fine-grained dynamic kernel sharing mechanism on GPUs. SMK includes a partial preemption mechanism, which preempts CTAs one at a time to enable fine-grained sharing of GPU resources while preempting a running kernel. SMK also exploits heterogeneity between kernel resource requirements to enable fair resource allocation and sharing of SM resources for concurrent kernel execution, and proposes a fair warp scheduler mechanism to allocate a fair number of cycles between competing concurrent kernels. Shieh et al. [157] propose a kernel preemption mechanism, Dual-Kernel, which co-executes a preempting kernel's CTAs with the preempted kernel's CTAs by selectively preempting CTAs one at a time until enough resources are available to run a CTA from the new kernel. Dual-Kernel aims to minimize preemption latency, throughput overhead, and resource fragmentation while preemption is occurring. Lastly, Menon et al. [117] propose iGPU, a set of compiler, ISA, and hardware extensions that enable preemption support and speculative execution on GPUs. iGPU identifies and exploits sparse idempotent regions to minimize the impact on re-executing GPU threads and minimize the amount of state required to be saved/restored on a context switch.

Throughout this dissertation, we propose and evaluate different forms of

preemption and GPU resource sharing techniques related to the works described above. In Chapter 4, we evaluated a technique to enable fine-grained GPU multi-tasking between a high-priority Memcached kernel and lower-priority background task by splitting the background task into groups of CTA-level sub-tasks and adding a software scheduler to manage the execution of these sub-tasks. This reduces the ability for larger low-priority tasks to monopolize the GPU resources, enabling other, potentially higher-priority, tasks to interleave their execution of CTAs on the GPU and reduce the latency to wait for available GPU resources. FLEP [178] and Effisha [33] expand on this technique with a general CPU and GPU software framework for transforming GPU applications to support this CTA-level programming paradigm across multiple concurrent applications, which is used to enable low overhead preemption. In Chapter 5, we evaluate two forms of preemption to support the *PGW* and high-priority event kernel execution. Unlike previous work, which focus on kernel and CTA-level preemption, we propose fine-grained warp-level preemption to support executing the *PGW*s based on an I/O or external event. This has a unique set of challenges, since the running application is only partially preempted to execute the small *PGW* task, which requires very low preemption latencies. *EDGE* also evaluates a form of partial CTA draining for event kernels (similar to [162]), where CTAs are blocked only until enough resources are available to begin executing the event kernel. For the application evaluated, Memcached, the individual event kernels require significantly fewer resources than are available on the GPU. As such, we evaluate a partial preemption mechanism, which preempts only as many resources as required to execute the event kernels. This reduces the latency to execute the high-priority event kernel and reduces the amount of state to be saved/restored from the preempted kernel. Additionally, by suspending the execution of the background (preempted) kernel, instead of fully saving/restoring it, *EDGE* improves performance for the higher-priority event kernel, while minimizing the preemption latency and impact on the background kernel. If the event kernels are expected to be much larger, requiring full preemption of a running kernel, the techniques described in prior work could be employed. Furthermore, the interrupt and *PGW* techniques in *EDGE* can be used to initiate preemption in a multiprogrammed environment instead of support from the CPU driver, self-preempting CTAs, GPU

persistent thread runtime, or compiler specified preemption points.

Multiple works have evaluated using CPU/GPU persistent thread runtimes for for fine-grained kernels and GPU I/O [33, 38, 96, 158, 178, 182, 183]. Silberstein et al. [158] propose a POSIX-like API and software framework, GPUfs, for GPU programs to access the host's file system. GPUfs belongs to the same family of work as GPUnet, GPUrdma, and GPUpIO described above. GPUfs issues CTA-level file operations to a CPU polling framework, which performs the I/O operation and returns the results back to the polling CTA. *EDGE* was initially designed for supporting external task launches, not necessarily for the CPU RPC operations in these works. However, the mechanisms proposed in *EDGE*, such as the wait-release barrier and *PGW*, could be used to avoid polling while waiting for long-latency I/O operations. Yeh et al. [182] propose Pagoda, a CPU-GPU persistent runtime framework that manages GPU tasks and resources at the warp-level, instead of at the CTA-level as most other persistent thread frameworks, to improve performance and efficiency for very small GPU tasks. Persistent thread frameworks can also be used to reduce the GPU's dependence on a CPU for GPU task management, as any device can communicate control information via the in-memory persistent thread task queues. However, if task launch rates are low, the GPU polling threads reduce energy-efficiency. Furthermore, the GPU polling threads monopolize the GPU resources, removing the ability for other GPU tasks to use the standard CUDA interface and GPU hardware task scheduling mechanisms. In contrast, *EDGE* aims to remove the requirement for persistent GPU-side runtimes through our proposed *PGW* framework, which use the fine-grained warp-level preemption mechanisms and pre-registered event kernels to reduce kernel launch overheads and dependence on a CPU for initiating GPU tasks.

There are many other works that have pushed towards evolving GPUs as first-class computing resources to support efficient GPU multiprogramming [64, 91, 93, 152, 156, 159, 171]. These works are related to the overarching goals of this dissertation for improving system-software support on GPUs. Rossbach et al. [152] propose PTask, a set of OS abstractions to support a dataflow programming model for accelerators, which enables the OS to provide fairness and isolation guarantees from a single management point. PTask also abstracts away many of the low-level GPU programming complexities. Kato et al. [93] propose Gdev, an OS GPU re-

source management runtime, which unifies interactions with the GPU from both user-space and OS applications. Gdev virtualizes GPU resources for efficient and isolated multi-tasking, and supports sharing memory across GPU contexts. Suzuki et al. [159] propose GPUvm, which virtualizes the GPU at the Hypervisor for efficient multi-tasking support in enterprise and cloud computing environments. GPUvm aggregates GPU management in the Hypervisor to remove direct virtual machine access, requiring several techniques and optimizations to virtualize GPU memory management, page mappings, communication channels, and task schedulers. Shi et al. [156] propose virtualizing the GPU across virtual machines at the host OS using the standard CUDA runtime, which intercepts and redirects CUDA calls from guest OSes to the GPU. Vijaykumar et al. [171] propose Zorua, a compiler/software/hardware virtualization framework, which virtualizes GPU resources internally on the GPU (e.g., threads, registers, and scratchpad memory) to improve programmability, portability, and performance. Zorua lends itself to supporting multitasking and preemption by efficiently managing shared GPU resources at a fine-grained level. Kato et al. [91] present an OS framework for efficiently managing GPU resources, such as GPU communication channels, kernel contexts, and memory. This work also discusses multiple different challenges to support GPU resource management in the OS, providing insights for future research directions.

The works described above are complimentary or orthogonal to the work in this dissertation. Many of the virtualization or resource management techniques could be integrated with *GNoM* to enable efficient sharing of the network interface and GPU resources for high-priority event kernels. While the event kernels proposed in *EDGE* enable direct task management from external devices using the GPU, the OS could be involved in configuring fair resource allocation for different event kernels. The *PGW* provides support for executing privileged software routines directly on the GPU and can be used to provide higher-level scheduling and resource sharing decisions in software running directly on the GPU SMs. Furthermore, the host OS (with greater visibility into the multitasking environment) and *PGW*s (with tighter integration with the GPU hardware) can cooperatively work together to guide scheduling decisions between concurrently executing kernels.

Chapter 7

Conclusions and Future Work

This chapter concludes the dissertation and discusses potential directions for future research.

7.1 Conclusions

Datacenters are important and ubiquitous computing environments with strict requirements for high performance, efficiency, and generality. With the decline in single-threaded performance scaling and power concerns surrounding multi-core scaling, the computing industry has looked towards alternative, heterogeneous systems to continue improving performance and efficiency; for example, heterogeneous CPU cores, specialized application-specific hardware designs, reprogrammable hardware systems, or parallel accelerators. Graphics processing units (GPUs) are an example of a massively parallel architecture capable of providing high levels of performance and efficiency for applications with large amounts of structured parallelism, such as high-performance computing and scientific applications. GPUs have recently made their way into the datacenter, commonly acting as accelerators for machine learning algorithms. However, there are other classes of applications with ample parallelism running on servers in the datacenter, such as server-based network applications, that are not typically considered as being strong candidates to offload to the GPU. Server applications belong to a class of highly economical and irregular applications. Accelerating server applications

on energy-efficient architectures can significantly lower power consumption, and hence cost, in the datacenter. Actually obtaining the potential benefits provided by the GPU in a heterogeneous environment, however, is challenging for multiple different reasons related to both the GPU architecture and system-level integration.

The evolution of the GPGPU has largely been driven by the requirements of applications using these accelerators. For example, there has been a large body of research highlighting the need for reducing the impact of branch divergence on GPUs based on a growing number of GPU applications with irregular control-flow behavior [47, 56, 57, 123, 148, 150]. As a result, the latest NVIDIA Volta GPU has introduced thread-level execution states [131] to address limitations with handling irregular control-flow behavior in previous GPU architectures. While these optimizations achieve lower performance or efficiency relative to highly regular applications, they help to improve the support for more irregular applications that may benefit from some, but not all, parts of the GPU's parallel architecture. This dissertation follows a similar approach to explore the potential for using GPUs as energy-efficient accelerators for more traditional server-based applications in the datacenter through a software-hardware co-design – first understanding the behavior of network services on contemporary GPU hardware, and then evaluating enhancements to the GPU's architecture and system-level integration to better support these classes of applications.

From the software side, this dissertation evaluates a popular key-value network service, Memcached, on contemporary GPU hardware. This analysis highlights that an application with irregular control and memory access patterns can still achieve sizeable improvements in performance and efficiency on a GPU. However, the actual application processing is only a portion of the required end-to-end processing for a network request. To this end, this dissertation proposes a complete end-to-end software framework for accelerating both the network (*GNoM*) and application (MemcachedGPU) processing on contemporary GPU and Ethernet hardware. This process exposed multiple challenges and limitations with implementing such a framework in existing systems, which can not be easily solved through software alone. From the hardware side, this dissertation proposes modifications to the GPU architecture and programming model (*EDGE*) to reduce the GPU's dependence on a centralized component (e.g., the CPU) in a heterogeneous

environment. This is a step towards considering the GPU as a first-class computing resource. We expect that as the number of GPU applications with similar compute and communication requirements grows, targeted optimizations to the GPU's architecture and system-support can further improve the GPU's ability to provide significant improvements in performance and efficiency.

While much of this dissertation focuses on a single application, Memcached, the main goals of this dissertation are not specifically to highlight whether the GPU is the best architecture to accelerate Memcached. Rather, this dissertation exposes the potential of the GPU to provide benefits to applications outside of the traditional high-performance scientific computing space, such as those found in a datacenter environment. These types of applications can have varying levels of complexity and irregular behavior, which as presented in this dissertation, may be able to benefit from different parts of the GPU's parallel architecture (e.g., the large number of parallel computing resources, the SIMD execution engines, or the high-bandwidth memory systems). However, each application is different. Consequently, whether or not the GPU will be able to provide any improvements in performance or efficiency still needs to be evaluated on a case-by-case basis, or common properties need to be identified with applications already known to perform well on a GPU. This dissertation argues that an application is not guaranteed to perform poorly on a GPU just because the application does not adhere to the strict characteristics apparent in traditional GPGPU applications, using a highly irregular application, Memcached, as an example. We hope that the insights gained and the systems developed in this dissertation (such as the methodology employed to effectively port Memcached to the GPU, the NIC/CPU/GPU *GNoM* software framework, or the *EDGE* GPU programming model) will help to enable future server-based applications, or applications requiring communication between heterogeneous devices, to benefit from GPU acceleration.

Chapter 3 [69] presents an initial analysis into the behavior of Memcached on both integrated and discrete GPU hardware, as well as on a software GPU simulation framework. This chapter identifies multiple challenges with porting such an application with irregular control flow and data access patterns to contemporary GPUs, and proposes a set of optimizations to mitigate the impact of this irregular behavior on the potential performance and efficiency gains from the GPU. On an

integrated GPU system, which does not require data transfers to a separate physical memory space, we show that the GPU can outperform the CPU by a factor of 7.5× for the core Memcached key-value lookup operation. Additionally, this chapter demonstrates how an application performs on the GPU relative to a programmer's intuition of the potential performance. To this end, this chapter describes a GPU control-flow simulator, CFG-Sim, which can help to estimate the SIMD efficiency of an application prior to actually porting the application to a GPU. For Memcached, we found that the actual SIMD efficiency is approximately 2.7× higher than a naive assumption of equal branch probabilities in the code-path may suggest.

Chapter 4 identifies multiple limitations with considering only the application processing when offloading a networking application to the GPU and tackles many of the challenges with performing both the network and application processing on the GPU. Furthermore, this chapter explores how to efficiently orchestrate the communication and computation between the GPU, NIC, and CPU when implementing an end-to-end GPU networking application. To this end, this chapter proposes *GNoM* (GPU Networking Offload Manager), a CPU-GPU software framework for accelerating UDP network and application processing on contemporary GPU and Ethernet hardware. *GNoM* constructs small batches of network requests (e.g., 512 requests) and pipelines multiple concurrent request batches to the GPU to overlap the communication and computation of network batches on the GPU. Using GPUDirect, the network request data is directly transferred from the NIC to the GPU; however, a CPU software framework is required to handle the network IO and GPU task management. This chapter also describes the design and implementation of MemcachedGPU, an accelerated key-value store, which leverages *GNoM* to run efficiently on a GPU. Many of the internal data structures in Memcached are redesigned to better fit the GPU's architecture and communicating components in a heterogeneous environment. *GNoM* and MemcachedGPU are evaluated on high-performance and low-power GPUs and are capable of reaching 10 Gbps line-rate processing with the smallest Memcached request size (over 13 million requests per second (MRPS)) at efficiencies under 12 *u*J per request. Furthermore, MemcachedGPU provides a 95-percentile round-trip time (RTT) latency under 1.1ms at the peak throughputs. Together, *GNoM* and MemcachedGPU highlight the GPU's

potential for accelerating such network/server-based applications.

Chapter 5 makes the observation that the CPU software framework portion of *GNoM* is required on contemporary GPU systems, even if no part of the application processing requires the CPU. This is a result GPUs being second-class computing resources, which rely on a host CPU to manage IO and the launching/-completion of tasks on the GPU. This chapter explores how to reduce the GPU's dependency on a host CPU for control management, without the use of continuously running GPU software frameworks, such as persistent threads, to enable any device in a heterogeneous system to efficiently execute tasks on a GPU. Specifically, this chapter proposes a novel event-driven GPU programming model, *EDGE*, which pre-registers GPU kernels (event kernels) to be launched by an external device. *EDGE* implements extensions to the CUDA API and requires minimal modifications to the GPU architecture to support storing and initiating the execution of GPU event kernels. We evaluate the opportunity to use existing GPU warps as a mechanism for launching the GPU event kernels internally on the GPU. *EDGE* exposes the GPU's interrupt interface to external devices in a heterogeneous system to trigger the execution of privileged GPU warps (*PGW*), capable of launching event kernels on the GPU, similar to CUDA dynamic parallelism (CDP). Unlike CDP, event kernels do not require dynamic configuration, which can significantly reduce the kernel launching latency. This chapter also proposes a fine-grained, warp-level preemption mechanism to reduce the *PGW* scheduling latency when the GPU does not have enough free resources available to immediately execute the *PGW*. We show that for a set of GPU benchmarks, a free warp context is available for a *PGW* 50% of the time on average, and that the preemption latency for a *PGW*, when no free warp contexts are available, can be reduced up to $\sim$36$\times$ (to $\sim$ 50 GPU cycles) with the proposed warp flushing optimizations over simply waiting for a warp to complete. Furthermore, by reducing the interference between the CPU and GPU, we estimate that *EDGE* could increase CPU throughput by 1.17$\times$ and GPU networking throughput by 3.7$\times$ when running workloads on the CPU (SPEC2006) in parallel with a networking application, such as MemcachedGPU, on the GPU.

7.2 Future Research Directions

This section discusses directions for potential future research based on the work performed in this dissertation.

7.2.1 Control-Flow Simulator

Section 3.2 presented a control-flow simulator, CFG-Sim, that simulates the behavior of a warp (wavefront) through an application's control-flow graph (CFG), to estimate the SIMD efficiency of the application when executed on a GPU. As described, this simulator considers the branch probabilities (taken or not taken) and applies this probability independently to each thread in a warp when flowing through the CFG. This is not a problem if each thread's execution is independent of all other threads in a warp. However, if the branch outcome for a thread is dependent on the other thread's in a warp, referred to here as correlated branches, then CFG-Sim may estimate a much lower SIMD efficiency than will actually result when the code is executed on a GPU. The impact of correlated branches can be seen in the Rodinia application, Ray Tracer, presented in Section 3.4.2. Consider the code in Example 3. In this example, the warp size is 32 threads (as in NVIDIA GPUs). The even warps will execute the code at block *A*, whereas the odd warps will execute the code at block *B*. If there are an even number of warps, the SIMD efficiency will be 100% and the branch probability will appear to be 50% (half of the threads execute *A*, the other half execute *B*). However, in CFG-Sim a branch probability of 50% will be applied to each thread, resulting in an estimated SIMD efficiency of roughly 50%.

Example 3 Example of correlated branches on branch probabilities.

```
1: int tid = threadIdx.x
2: if (tid % 64 < 32) { // Warp size is 32 threads
3:     // A
4: else {
5:     // B
6: }
```

Loops pose a similar challenge. Consider the code in Example 4. For each

thread in this example, the branch is taken nine times and not taken one time. Thus the branch probability appears to be 90%. CFG-Sim will apply a branch probability of 90% to each thread on each iteration, which results in a SIMD efficiency lower than the actual SIMD efficiency of 100%.

Example 4 Example of loops on branch probabilities

```
1:  for (unsigned i=0; i<9; ++i) {
2:      // C
3:  }
```

Another challenge with estimating SIMD efficiency occurs when branch outcomes are data dependent. Consider the code in Example 5. Here, all threads in a thread block will iterate through loop together; however, the number of iterations is dependent on external data. Consequently, the number of times that the threads will execute the code block *D* is not known a priori, which complicates the calculation of the final SIMD efficiency estimation. For example, the overall SIMD efficiency can be quite different if block *D* is only 10% of the static code in the GPU kernel, but 90% of the dynamic code.

Example 5 Example of data dependency on branch probabilities

```
1: N = global_buffer[blockIdx.x]
2: for (unsigned i=0; i<N; ++i) {
3:     // D
4: }
```

To generalize across multiple types of applications, CFG-Sim must consider these types of correlated branches when estimating the SIMD efficiency. However, identifying correlated branches in CPU code is challenging since the code has likely not been written for highly parallel execution or may be highly dependent in input data. An interesting research direction would be to understand how CFG-Sim could be improved to support branches related to parallel execution, the underlying GPU architecture, and data dependence.

7.2.2 Evaluating *GNoM* on Additional Applications and Integrated GPUs

The *GNoM* framework presented in Chapter 4 is evaluated on a single server application, MemcachedGPU, and a network-only GPU ping benchmark. *GNoM* should also be able to efficiently accelerate other existing GPU network packet processing applications, where different GPU threads are responsible for processing different network packets, such as IPv4/6 forwarding or packet encryption/authentication (e.g., IPSec). An obvious direction for future research is to evaluate additional applications that can take advantage of *GNoM* to accelerate the network flow from the network card directly to the GPU. MemcachedGPU was designed and evaluated for very small network packet sizes to stress the *GNoM* framework. Evaluating applications with larger packets should also benefit from *GNoM*. Additionally, the latency per Memcached kernel is relatively short, which results in larger end-to-end improvements when using *GNoM* to reduce these overheads. It would be interesting to evaluate how *GNoM* compares to the baseline network flow when the GPU kernel runtime is much longer. Furthermore, MemcachedGPU highlighted a specific requirement for the partitioning of computation between the CPU and GPU, for example, to fill the response packets with the corresponding Memcached item values on a hit in the hash table, prior to sending the response packets over the network. Other applications may place drastically different requirements on the amount and type of CPU and GPU processing, which could require rethinking parts of the *GNoM* design to efficiently support a wide range of applications.

Chapter 3 highlighted that while discrete GPUs have higher computational capacity, the ability to remove data transfers between the CPU and GPU resulted in higher overall performance from integrated GPUs. *GNoM* was developed and evaluated on discrete GPUs mainly due to the existence of GPUDirect, which enabled the NIC to transfer RX packets directly to the physical memory on high-performance discrete GPUs. The integrated GPU's memory could also be mapped into the NIC for direct packet transfers; however, discrete GPUs were selected for an initial evaluation due to the potential for higher performance when memory transfer overheads are reduced (as shown in Chapter 3). In *GNoM*, the GPU still transfers response packets on the TX path to the CPU to forward to the NIC. This is

not an inherent limitation of *GNoM* [1], but was instead a design decision to improve performance on contemporary hardware. Specifically, this enabled *GNoM* to make use of the CPU's larger memory space for storing the Memcached values without incurring additional transfers across the PCIe. Based on the results for a lower power/performance discrete GPU in Chapter 4, evaluating *GNoM* on integrated GPUs could further improve the efficiency for applications, such as MemcachedGPU, which currently require partitioning parts of the application across the CPU and GPU due to GPU memory capacity limitations. Additionally, unlike Chapter 3, which evaluated large batches of packets (e.g., 20k+ packets), *GNoM* works on much smaller batches of packets (512 packets), which places lower requirements on the GPU's computational capacity.

7.2.3 Larger GPU Networking Kernels

GNoM and MemcachedGPU were designed to support a single GPU thread per network packet [2]. Many potential GPU networking applications could require significantly more GPU threads to execute the kernel. For example, consider a GPU networking application that performs facial recognition on a network packet containing image data. Such an application requires multiple GPU threads to efficiently complete the task for a single network packet. Additionally, the larger kernel size requires fewer (potentially zero) network packets to be batched together to improve the GPU's throughput. Supporting these types of applications would require modifications to *GNoM* to handle multiple GPU threads per network packet and partition the threads between application and network processing.

7.2.4 Stateful GPU Network Processing

GNoM currently supports stateless UDP network processing. Stateless packet processing simplifies the processing requirements, as there is no hand-shaking between client and server, and no additional protocol mechanisms, such as flow control. Thus, the GPU client simply receives the packet, processes the packet, and sends the response. If the packet is dropped for any reason along the end-to-end

[1] We also implemented and evaluated an equivalent direct GPU-to-NIC TX path.

[2] As described in Chapter 4, two GPU threads are actually launched per packet to overlap parallel network and application processing tasks within the GPU kernel.

path from/to the client, it is the responsibility of the client to re-transmit the packet. GASSP [170] is the first work to evaluate stateful packet processing (TCP) on GPUs and highlights the ability for the GPU to achieve significant performance improvements over a GPU-based implementation where the packet processing occurs on the CPU. GASSP requires larger batch sizes (increases packet latency), uses main CPU memory for storing packets (does not make use of GPUDirect), and evaluates GASSP on packet processing based applications, such as firewalls, intrusion detection, and encryption. An interesting research direction would be to understand how the insights gained from both *GNoM* and GASSP could improve the performance of stateful GPU packet processing and accelerate higher levels of the application layer on a GPU.

7.2.5 Accelerating Operating System Services on GPUs

This dissertation evaluates the potential for offloading a portion of the network stack to the GPU to improve the performance and efficiency of GPU networking applications. Since the network application itself is also running on the GPU, accelerating an Operating System (OS) service, such as the network stack, has a direct impact on reducing the latency of the critical path between the NIC and GPU. A much more open-ended research direction would be to explore additional OS services, potentially those which have no existing interactions with the GPU, that could benefit from GPU acceleration.

7.2.6 Networking Hardware Directly on a GPU

The *GNoM* framework proposed in this dissertation considers a heterogeneous system in which the NIC is a physically separate component from the CPU and GPU, and is connected via an interconnect bus, such as PCIe. As highlighted in Chapter 6, many works have shown the potential for the GPU to improve the performance and efficiency of networking applications. From an architectural viewpoint, it would be interesting to evaluate how network interfaces (such as Ethernet ports) could be directly integrated into the GPU architecture, and how the network interface could be modified to better suit the GPU's vector architecture. For example, one of the inefficiencies of current packet processing on GPUs is due to the packet

layout in memory. If each GPU thread processes a separate packet, as in *GNoM*, every memory access will lead to memory divergence (*GNoM* tries to reduce the effects by loading packets into shared memory). However, a dedicated NIC on a GPU could rearrange (swizzle) the packet header contents in hardware such that the GPU threads can better take advantage of memory coalescing. An initial evaluation into the packet header swizzling technique shows up to 2.4× improvement in throughput for a GPU IPv4/6 forwarding applications. Furthermore, an integrated GPU-NIC architecture could significantly reduce the latency for ingesting and processing packets in GPU memory with separate GPU threads. Including network hardware directly on the GPU would also increase the requirement for a mechanism to independently and efficiently launch GPU kernels internally on the GPU, such as *EDGE* and the *PGW*s.

7.2.7 Scalar Processors for GPU Interrupt Handling

Chapter 5 explores how existing warps running on the GPU's SIMD pipeline can be used to handle interrupts to internally launch GPU kernels in response to an external event. However, the interrupt processing, as described in this chapter, is inherently sequential and could also be efficiently processed by a scalar processor. A direction for future research would be to evaluate how simple scalar cores integrated on the GPU die, such as scalar control processors [139] or the scalar units in AMD Compute Units [10], could be used to improve the performance and efficiency of interrupt handling for launching internal event kernels on the GPU.

7.2.8 GPU Wait-Release Barriers

Chapter 5 introduced a new type of CTA barrier, the wait-release barrier, which blocks all warps in a CTA indefinitely until a release barrier instruction is performed. The current implementation constructs wait-release barrier sets using IDs and releases all wait-release barriers belonging to the same set simultaneously. If multiple CTAs are concurrently blocked on the same wait-release barrier set, they will all be released regardless of the number of tasks pushed to the GPU. This reduces performance and efficiency, as only a subset of the released CTAs may find a valid a task to perform, while the rest are unnecessarily released from the wait-

release barrier and will immediately return to blocking. An area for future work would be to explore how to efficiently support targeted releasing of wait-release barriers. This would likely require data structures to track the status of running and blocked CTAs, and a dedicated mechanism for communicating information to these CTAs from the *PGW* responsible for releasing a blocked CTA. Additionally, given that a single *PGW* releases the wait-release barrier, the *PGW* could be used to remove synchronization on a single global work queue by distributing the pending tasks to local per-CTA work queues. The *PGW* could also be used for load-balancing the tasks across CTAs.

7.2.9 Rack-Scale Computing

Datacenters have recently started to shift towards rack-scale computing architectures [82], where components in a disaggregated computing [107] system reside in physically separate racks. In such a computing environment, only the resources that are actually required are allocated to the application. This increases the utilization of hardware resources in the datacenter by minimizing the amount of unused hardware resources that are indirectly reserved by application. An example of this would be an application that is allocated a full server based on its compute requirements, but only requires 50% of the storage resources installed in this server. In a rack-scale architecture, this application could be allocated the required compute resources from one rack and the required storage resources from another, leaving the unused storage resources to be allocated for another application. However, if we consider a GPU networking application, which only requires the network interface and GPU, a CPU still needs to be allocated to handle the control path from the network interface to the GPU. The *EDGE* framework proposed in Chapter 5 removes this dependency on the CPU for GPU task management in a single system; however, an interesting research direction would be to understand how such a framework may also be used in a disaggregated rack-scale environment. For example, NVIDIA's Pascal and Volta GPUs support virtual memory demand paging through unified memory [126, 131], which places a requirement on the CPU (and CPU memory) for tasks other than strictly control management.

Bibliography

[1] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Int'l Symp. on Perf. Analysis of Systems and Software (ISPASS)*, pages 163–174, April 2009. → pages 12

[2] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The Case for GPGPU Spatial Multitasking. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012. → pages 115

[3] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing Data Parallel Hardware for Server Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014. → pages 74, 180

[4] B. Aker. libMemcached. http://libmemcached.org/libMemcached.html. → pages 96

[5] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proceedings of the 17th Int'l Symp. on High Performance Distributed Computing*, HPDC '08, pages 165–174, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-997-5. → pages 180

[6] M. Alhamad, T. Dillon, and E. Chang. Conceptual sla framework for cloud computing. In *Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on*, pages 606–610. IEEE, 2010. → pages 7

[7] Altera. Fpga architecture. https://www.intel.com/content/www/us/en/processors/xeon/xeon-e7-8800-4800-v4-product-families-brief.html, 2006. → pages 4

[8] Amazon. Why machine learning on aws? https://aws.amazon.com/machine-learning, 2018. → pages 1

[9] *AMD Accelerated Parallel Processing, OpenCL - Programming Guide*. AMD, 2.5 edition, 2011. → pages 17, 43, 51, 63

[10] AMD. AMD Graphics Cores Next (GCN) Architecture. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, 2012. → pages 23, 133, 200

[11] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14. ACM, 2009. → pages 7, 175, 177

[12] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, 2014. → pages 4

[13] A. Ariel, W. W. Fung, A. E. Turner, and T. M. Aamodt. Visualizing complex dynamics in many-core accelerator architectures. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 164–174. IEEE, 2010. → pages 68

[14] ARM. Big.little technology: The future of mobile. http://img.hexus.net/v2/press_releases/arm/big.LITTLE.Whitepaper.pdf, 2013. → pages 3

[15] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. → pages 2

[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012. → pages 85, 91, 114

[17] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009. → pages 20, 50, 156

[18] P. Bakkum and K. Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, March 2010. → pages 74, 180

[19] B. Barney. Message passing interface (mpi). https://computing.llnl.gov/tutorials/mpi/, 2018. → pages 3

[20] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013. → pages 1, 6, 7, 115

[21] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In Gupta, Rajiv, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin / Heidelberg, 2010. → pages 61

[22] M. Bauer, S. Treichler, and A. Aiken. Singe: Leveraging warp specialization for high performance on gpus. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14. ACM, 2014. → pages 78, 84

[23] S. Bauer, S. Köhler, K. Doll, and U. Brunsmann. Fpga-gpu architecture for kernel svm pedestrian detection. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pages 61–68. IEEE, 2010. → pages 127

[24] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core Key-value Store. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1 –8, July 2011. → pages 39, 57, 175

[25] D. Black. Idc: Ai, hpda driving hpc into high growth markets. https://www.businesswire.com/news/home/20160309006515/en/Worldwide-Server-Market-Revenues-Increase-5.2-Fourth, November 2016. → pages 6

[26] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bar, and Z. Istvan. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *Proceedings of the 5th*

USENIX Workshop on Hot Topics in Cloud Computing, 2013. → pages 89, 90, 100, 121, 122, 123, 177

[27] D. Burger. Keynote: A New Era of Hardware Microservices in the Cloud. In *DATE*, 2017. → pages 7

[28] C. Faylor, et al. Cygwin. http://www.cygwin.com/. → pages 49

[29] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016. → pages 7

[30] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA Memcached Appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2013. → pages 177

[31] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009. → pages 157

[32] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008. → pages 21

[33] G. Chen, Y. Zhao, X. Shen, and H. Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–16. ACM, 2017. → pages 23, 130, 149, 185, 186, 187, 188

[34] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, 2010. ACM. URL http://doi.acm.org/10.1145/1807128.1807152. → pages 98, 104

[35] J. Corbet, A. Rubini, and G. Kroah-Hartman. Linux device drivers third edition. https://static.lwn.net/images/pdf/LDD3/, 2005. → pages 137

[36] A. Corporation. Implementing fpga design with the opencl standard. https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf, 11 2013. → pages 4

[37] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 186–189, New York, NY, USA, 2002. ACM. doi:10.1145/1133373.1133410. URL http://doi.acm.org/10.1145/1133373.1133410. → pages 33, 131

[38] F. Daoud, A. Watad, and M. Silberstein. Gpurdma: Gpu-side library for high performance networking from gpu kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, page 6. ACM, 2016. → pages 126, 181, 188

[39] J. Dean. Large scale deep learning. Keynote GPU Technical Conference 2015, 03 2015. URL http://www.ustream.tv/recorded/60071572. → pages 7

[40] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013. → pages 136

[41] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, February 2013. → pages 9, 109

[42] E. Demers. Evolution of AMD Graphics. AMD Fusion Developer Summit, June 2011. → pages 52

[43] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. → pages 2

[44] Deyannis, Dimitris and Koromilas, Lazaros and Vasiliadis, Giorgos and Athanasopoulos, Elias and Ioannidis, Sotiris. Flying Memcache: Lessons Learned from Different Acceleration Strategies. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*. IEEE, 2014. → pages 121, 122, 123, 179

[45] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2014. → pages 178

[46] T. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 256–266. IEEE, 1992. → pages 131

[47] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt. A scaleltable multi-path microarchitecture for efficient gpu control flow. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 248–259. IEEE, 2014. → pages 191

[48] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011. → pages 3

[49] N. Express. Nvm express explained. http://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf, 2013. → pages 133, 183

[50] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3), 2008. → pages 168

[51] Facebook. Applying machine learning science to facebook products. https://research.fb.com/category/machine-learning, 2018. → pages 1

[52] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th Usenix Symposium on Networked Systems Design and Implementation (NSDI '13)*, 2013. → pages 89, 98, 101, 107, 121, 176

[53] M. Feldman. Idc's latest forecast for the hpc market: "2016 is looking good". https://www.top500.org/news/idcs-latest-forecast-for-the-hpc-market-2016-is-looking-good, November 2016. → pages 6

[54] J. Fischer, R. Majumdar, and T. Millstein. Tasks: Language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 134–143, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi:10.1145/1244381.1244403. URL http://doi.acm.org/10.1145/1244381.1244403. → pages 33, 131

[55] H. Foundation. Hsa runtime programmer's reference manual v1.1.1. http://www.hsafoundation.com/standards/, 2016. → pages 52, 132, 133, 147, 183

[56] W. W. Fung and T. M. Aamodt. Thread block compaction for efficient simt control flow. 2011. → pages 191

[57] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420. IEEE Computer Society, 2007. → pages 191

[58] E. Gansner, E. Koutsofios, and S. North. Drawing graphs with dot, 2006. → pages 47

[59] Garland, Michael and Le Grand, Scott and Nickolls, John and Anderson, Joshua and Hardwick, Jim and Morton, Scott and Phillips, Everett and Zhang, Yao and Volkov, Vasily. Parallel Computing Experiences with CUDA. *IEEE Micro*, 28:13–27, July 2008. → pages 6, 35

[60] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 347–358, New York, NY, USA, 2010. ACM. → pages 44

[61] Google. Graphics processing unit (gpu). leverage gpus on google cloud for machine learning and scientific computing. https://cloud.google.com/gpu/, 2018. → pages 8

[62] C. Gregg and K. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *ISPASS '11*, 2011. → pages 21

[63] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE, 2012. → pages 24, 183

[64] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated Virtual Machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, 2009. → pages 188

[65] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 195–206. ACM, 2010. → pages 126, 181

[66] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(EPFL-ARTICLE-168285):6–15, 2011. → pages 3

[67] T. Herbert and W. d. Bruijn. Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2017. → pages 81

[68] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In *Distributed Computing*. Springer, 2008. → pages 89, 100

[69] T. Hetherington, T. Rogers, L. Hsu, M. O'Connor, and T. Aamodt. Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems. In *Proceeding of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012. → pages vi, 192

[70] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 43–57, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3651-2. doi:10.1145/2806777.2806836. URL http://doi.acm.org/10.1145/2806777.2806836. → pages vi

[71] T. H. Hetherington, M. Lubeznov, D. Shah, and T. M. Aamodt. Edge: Event-driven gpu execution. In *2019 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019. → pages vii, 127

[72] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006. → pages 1

[73] U. Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, July/August 2010. → pages 7

[74] M. Houston and M. Mantor. AMD Graphic Core Next: Low Power High Performance Graphics & Parallel Compute. AMD Fusion Developer Summit, June 2011. → pages 50

[75] X. Huang, C. Rodrigues, S. Jones, I. Buck, and W.-M. Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *Proceedings of the 2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*, 2010. → pages 89

[76] IDC. Worldwide server market rebounds sharply in fourth quarter as demand for blades and x86 systems leads the way, February 2010. → pages 6

[77] IDC. Hpc server market declined 11.6% in 2009, return to growth expected in 2010, March 2010. → pages 6

[78] IDC. Worldwide server market revenues increase 5.2% in the fourth quarter as demand in china once again drives the market forward. https://www.businesswire.com/news/home/20160309006515/en/Worldwide-Server-Market-Revenues-Increase-5.2-Fourth, March 2016. → pages 6

[79] Intel. Intel DPDK (Data Plane Development Kit). https://https://dpdk.org, . → pages 177

[80] Intel. Intel VTune Aplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe, . → pages 72

[81] Intel. Accelerate big data insights. https://www.intel.com/content/www/us/en/processors/xeon/xeon-e7-8800-4800-v4-product-families-brief.html, 2016. → pages 3

[82] Intel. Intel rack scale design (intel rsd). https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html, 2017. → pages 8, 201

[83] Intel. Intel product specifications. https://ark.intel.com/, 2018. → pages xii, 5

[84] Intel. Intel threading building blocks (intel tbb). https://software.intel.com/en-us/tbb-documentation, 2018. → pages 3

[85] Z. Istvan, G. Alonso, M. Blott, and K. Vissers. A flexible hash table design for 10gbps key-value stores on fpgas. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013. → pages 121, 122, 123, 177

[86] B. Jenkins. Function for Producing 32bit Hashes for Hash Table Lookup. http://burtleburtle.net/bob/c/lookup3.c, 2006. → pages 38, 87, 98

[87] S. Jones, P. A. Cuadra, D. E. Wexler, I. Llamas, L. V. Shah, J. F. Duluk Jr, and C. Lamb. Technique for computational nested parallelism, Dec. 6 2016. US Patent 9,513,975. → pages xvii, 20, 145, 152

[88] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. Islam, X. Ouyang, H. Wang, S. Sur, and D. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP)*, 2011. → pages 178

[89] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. Panda. Scalable Memcached Design for InfiniBand Clusters Using Hybrid Transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012. → pages 178

[90] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*, 2017. → pages 1, 7

[91] S. Kato, S. Brandt, Y. Ishikawa, and R. Rajkumar. Operating systems challenges for gpu resource management. In *Proc. of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 23–32, 2011. → pages 188, 189

[92] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 57–66. IEEE, 2011. → pages 23, 130, 149, 185, 186

[93] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt. Gdev: First-class gpu resource management in the operating system. In *USENIX Annual Technical Conference*, pages 401–412. Boston, MA;, 2012. → pages 188

[94] *The OpenCL Specification*. Khronos OpenCL Working Group, 1.1 edition, 2011. → pages 3, 16

[95] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. Nba (network balancing act): A high-performance packet processing framework for

heterogeneous processors. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15. ACM, 2015. → pages 181, 182

[96] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. Gpunet: Networking abstractions for gpu programs. In *OSDI*, volume 14, pages 6–8, 2014. → pages 24, 126, 180, 184, 185, 188

[97] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3): 263–297, 2000. → pages 182

[98] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE micro*, 25(2):21–29, 2005. → pages 2, 8, 34

[99] A. Krizhevsky. Cuda-convnet. https://github.com/dnouri/cuda-convnet, 2015. → pages 157

[100] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26, 2007. → pages 4

[101] M. LeBeane, B. Potter, A. Pan, A. Dutu, V. Agarwala, W. Lee, D. Majeti, B. Ghimire, E. Van Tassell, S. Wasmundt, et al. Extended task queuing: active messages for heterogeneous systems. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 933–944. IEEE, 2016. → pages 134, 183

[102] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998. → pages 157

[103] K. Lee and Facebook. Introducing big basin: Our next-generation ai hardware. https://code.facebook.com/posts/1835166200089399/introducing-big-basin-our-next-generation-ai-hardware/, 2017. → pages 7

[104] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A Mapping Path for Multi-GPGPU Accelerated Computers from a Portable High Level Programming Abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics*

Processing Units, GPGPU '10, pages 51–61, New York, NY, USA, 2010. ACM. → pages 61

[105] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014. → pages 7, 107

[106] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th Usenix Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014. → pages 89, 90, 96, 100, 102, 107, 121, 122, 123, 176

[107] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 267–278, 2009. → pages 8, 201

[108] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. *SIGARCH Computer Architecture News*, June 2013. → pages 177

[109] Linux. Linux programmer's manual, pthreads - posix threads. http://man7.org/linux/man-pages/man7/pthreads.7.html, 2018. → pages 3

[110] R. A. Lorie and H. R. Strong Jr. Method for conditional branch execution in simd vector processors, Mar. 6 1984. US Patent 4,435,758. → pages 25

[111] T. P. Lottes, D. Wexler, C. Duttweiler, S. Treichler, L. Durant, and P. Cuadra. System and method for runtime scheduling of gpu tasks, Mar. 6 2013. US Patent App. 13/787,660. → pages xvii, 145

[112] N. Mathewson and N. Provos. Libevent. http://libevent.org/. → pages 30

[113] N. Mehta. Xilinx 7 series fpgas embedded memory advantages. https://www.xilinx.com/support/documentation/white_papers/wp377_7Series_Embed_Mem_Advantages.pdf, 2012. → pages 4

[114] N. Mehta. Xilinx 7 series fpgas: The logical advantage. https://www.xilinx.com/support/documentation/white_papers/wp405-7Series-Logical-Advantage.pdf, 2012. → pages 4

[115] Memcached. A distributed memory object caching system. http://www.memcached.org. → pages 6, 8, 29, 30

[116] P. Meng, M. Jacobsen, and R. Kastner. Fpga-gpu-cpu heterogenous architecture for real-time cardiac physiological optical mapping. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 37–42. IEEE, 2012. → pages 126, 134

[117] J. Menon, M. De Kruijf, and K. Sankaralingam. igpu: exception support and speculative execution on gpus. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 72–83. IEEE Computer Society, 2012. → pages 185, 186

[118] R. Merritt. ARM CTO: Power Surge Could Create 'Dark Silicon'. *EETimes*, 22 October 2009. → pages 3

[119] Michael Shebanow. ECE 498 AL : Programming Massively Parallel Processors, Lecture 12. http://courses.ece.uiuc.edu/ece498/al1/Archive/Spring2007, February 2007. → pages 25

[120] Microsoft. Azure machine learning. https://azure.microsoft.com/en-ca/overview/machine-learning, 2018. → pages 1

[121] G. E. Moore. Cramming more components onto integrated circuits, electronics,(38) 8, 1965. → pages 2

[122] S. Naffziger, J. Warnock, and H. Knapp. Se2 when processors hit the power wall (or" when the cpu hits the fan"). In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 16–17. IEEE, 2005. → pages 2

[123] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317. ACM, 2011. → pages 191

[124] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. Mcelroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, V. Venkataramani, and F. Inc. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. → pages 8, 31, 85, 90, 178

[125] ntop. PF_RING. http://www.ntop.org/products/pf_ring/. → pages 78, 84, 96

[126] NVIDIA. Nvidia tesla p100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016. → pages 21, 23, 130, 149, 201

[127] NVIDIA. Cuda c programming guide. https://docs.nvidia.com/cuda/pdf/CUDA_Dynamic_Parallelism_Programming_Guide.pdf, 2017. → pages 22

[128] NVIDIA. Gpudirect. https://developer.nvidia.com/gpudirect, 2017. → pages 21

[129] NVIDIA. Multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2017. → pages 132, 147, 183

[130] NVIDIA. Nvidia management library (nvml). https://developer.nvidia.com/nvidia-management-library-nvml, 2017. → pages 157

[131] NVIDIA. Nvidia tesla v100 gpu architecture. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. → pages xii, 5, 17, 21, 23, 52, 130, 191, 201

[132] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009. → pages 115

[133] NVIDIA Corporation. NVIDIA CUDA C Programming Guide v4.2. http://developer.nvidia.com/nvidia-gpu-computing-documentation/, 2012. → pages 3, 16

[134] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012. → pages 20

[135] NVIDIA Corporation. Developing a Linux Kernel Module using GPUDirect RDMA. http://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2014. → pages 32

[136] NVIDIA Corporation. NVIDIA GeForce GTX 750 Ti: Featuring First-Generation Maxwell GPU Technology, Designed for Extreme Performance per Watt. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, 2014. → pages 120

[137] OpenMP. Openmp application programming interface. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015. → pages 3

[138] Oracle. Oracle machine learning. http://www.oracle.com/technetwork/database/options/oml/overview/index.html, 2018. → pages 1

[139] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood. Fine-grain task aggregation and coordination on gpus. *ACM SIGARCH Computer Architecture News*, 42(3):181–192, 2014. → pages 200

[140] J. Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, volume 5. San Diego, CA, USA, 1996. → pages 33, 131

[141] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2): 122–144, 2004. → pages 101

[142] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, July 2009. → pages 4, 115

[143] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. *ACM SIGARCH Computer Architecture News*, 43(1):593–606, 2015. → pages 23, 130, 149, 185, 186

[144] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990. ISBN 1-55880-069-8. → pages 90

[145] PCI Express. PCI Express 3.0 Frequently Asked Questions. https://web.archive.org/web/20140201172536/http://www.pcisig.com/news_room/faqs/pcie3.0_faq/#EQ2. → pages 21

[146] J. Power, J. Hestness, M. Orr, M. Hill, and D. Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *Computer Architecture Letters*, 13(1), Jan 2014. ISSN 1556-6056. doi:10.1109/LCA.2014.2299539. URL http://gem5-gpu.cs.wisc.edu. → pages 156

[147] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014. → pages 4, 7, 115

[148] M. Rhu and M. Erez. The dual-path execution model for efficient gpu control flow. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 591–602. IEEE, 2013. → pages 191

[149] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012. → pages 166

[150] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 99–110. ACM, 2013. → pages 191

[151] F. Rosenblatt. *The perceptron, a perceiving and recognizing automaton (Project Para)*. Cornell Aeronautical Laboratory, 1957. → pages 1

[152] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011. → pages 188

[153] D. Rossetti. Gpudirect: integrating the gpu with a network interface. In *GPU Technology Conference*, 2015. → pages 185

[154] Y. S. Shao and D. Brooks. Research infrastructures for hardware accelerators. *Synthesis Lectures on Computer Architecture*, 10(4):1–99, 2015. → pages 4

[155] M. C. Shebanow, J. Choquette, B. W. Coon, S. J. Heinrich, A. Kalaiah, J. R. Nickolls, D. Salinas, M. Y. Siu, T. Thorn, N. Wang, and other. Trap handler architecture for a parallel processing unit, Aug. 27 2013. US Patent 8,522,000. → pages xvii, 20, 22, 23, 145

[156] L. Shi, H. Chen, J. Sun, and K. Li. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers*, June 2012. → pages 188, 189

[157] L.-W. Shieh, K.-C. Chen, H.-C. Fu, P.-H. Wang, and C.-L. Yang. Enabling fast preemption via dual-kernel support on gpus. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 121–126. IEEE, 2017. → pages 23, 130, 149, 185, 186

[158] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: integrating a file system with gpus. In *ACM SIGPLAN Notices*, volume 48, pages 485–498. ACM, 2013. → pages 184, 185, 188

[159] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. Gpuvm: Gpu virtualization at the hypervisor. *IEEE Transactions on Computers*, 65(9):2752–2766, 2016. → pages 188, 189

[160] Y. Suzuki, H. Yamada, S. Kato, and K. Kono. Towards multi-tenant gpgpu: Event-driven programming model for system-wide scheduling on shared gpus. In *Proceedings of the Workshop on Multicore and Rack-scale Systems*, 2016. → pages 184

[161] T. Jablin et al. Automatic CPU-GPU Communication Management and Optimization. In *PLDI*, June 2011. → pages 61

[162] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on gpus. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 193–204. IEEE Press, 2014. → pages 23, 130, 149, 150, 185, 186, 187

[163] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling Preemptive Multiprogramming on GPUs. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014. → pages 115

[164] ThinkTank Energy Products. Watts up? Plug Load Meters. https://www.wattsupmeters.com/secure/index.php. → pages 97

[165] top500. Green500 list for june 2017. https://www.top500.org/green500/lists/2017/06/, 2017. → pages 6, 34

[166] top500. Top 10 sites for november 2017. https://www.top500.org/lists/2017/11/, 2017. → pages 6, 34

[167] J. Turley. Introduction to intel architecture. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf, 2014. → pages 3

[168] S. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In Amaral, Jos, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 2008. → pages 61

[169] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks*, 53(11): 1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html. → pages 53

[170] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis. Gaspp: A gpu-accelerated stateful packet processing framework. In *USENIX Annual Technical Conference*, pages 321–332, 2014. → pages 126, 181, 199

[171] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu. Zorua: A holistic approach to resource virtualization in gpus. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–14. IEEE, 2016. → pages 188, 189

[172] W. Fung, et al. . Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proc. 40th IEEE/ACM Int'l Symp. on Microarchitecture*, 2007. → pages 25, 51

[173] W. Fung, et al. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. Archit. Code Optim.*, 6(2):1–37, 2009. ISSN 1544-3566. doi:{http://doi.acm.org/10.1145/1543753.1543756}. → pages 25

[174] J. Wang and S. Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured gpu applications. In *Workload Characterization*

(IISWC), 2014 IEEE International Symposium on, pages 51–60. IEEE, 2014. → pages 20, 22, 156

[175] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus. *ACM SIGARCH Computer Architecture News*, 43(3): 528–540, 2016. → pages xvii, 22, 145

[176] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 358–369. IEEE, 2016. → pages 23, 130, 149, 158, 185, 186

[177] A. Wiggins and J. Langston. Enhancing the Scalability of Memcached. https://software.intel.com/sites/default/files/m/0/b/6/1/d/45675-memcached_05172012.pdf. → pages 90, 107, 175

[178] B. Wu, X. Liu, X. Zhou, and C. Jiang. Flep: Enabling flexible and efficient preemption on gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 483–496. ACM, 2017. → pages 185, 186, 187, 188

[179] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing Data Warehousing Applications for GPUs using Kernel Fusion/Fission. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2012. → pages 74, 180

[180] Xilinx. Using embedded multipliers in spartan-3 fpgas. https://www.xilinx.com/support/documentation/application_notes/xapp467.pdf, 2003. → pages 4

[181] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 230–242. IEEE Press, 2016. → pages 158

[182] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers. Pagoda: Fine-grained gpu resource virtualization for narrow tasks. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and*

Practice of Parallel Programming, pages 221–234. ACM, 2017. → pages 24, 188

[183] L. Zeno, A. Mendelson, and M. Silberstein. Gpupio: The case for i/o-driven preemption on gpus. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 63–71. ACM, 2016. → pages 79, 80, 185, 188

[184] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11), 2015. → pages 124, 179