UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Systems and Algorithm Support for Efficient Heterogeneous Computing with GPUs

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Yang Liu

Committee in charge:

Professor Steven Swanson, Chair
Professor Pamela Cosman
Professor Rajesh Gupta
Professor George Porter
Professor Michael Taylor

2016

The Dissertation of Yang Liu is approved and is acceptable in quality and form for publication on microfilm and electronically:

---

---

---

---

---

Chair

University of California, San Diego

2016

# DEDICATION

To my parents who give their son
the roots of responsibility and the wings of independence.

To my wife who is always there for me.

# EPIGRAPH

Learning without thought is labor lost;
Thought without learning is perilous.

— *Confucius*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Through the many years of my graduate study journey, I have received generous help and support from many people, for which I am sincerely grateful.

I would like to thank my advisor Professor Steven Swanson for taking me, supporting me, bearing with me, and guiding me throughout my graduate research. Without his generous help, I would not be completing this dissertation.

I would also like to acknowledge all my co-authors and colleagues from the Non-Volatile Systems Lab. I feel lucky to work with them, and have learned a lot from them.

I want to thank my officemates and friends in the Computer Science and Engineering Department, and all my friends in University of California, San Diego. Special thanks to Feng Lu and Jingwei Lu for their encourage and support as friends.

Additionally, I would like to thank all my committee members for serving on my committee, and for providing valuable comments and feedback.

Finally and most importantly, I must express my deepest gratitude to my family. Thank my parents for their deep love and unconditional support for all these years. And thank my great wife Yiying for her love, patience, and support.

Chapter 2 and 3 contains material from "Hippogriff: Efficiently Moving Data in Heterogeneous Computing Systems" by Yang Liu, Hung-Wei Tseng, Mark Gahagan, Jing Li, Yanqin Jin and Steven Swanson, which appears in *ICCD '16: 2016 IEEE International Conference on Computer Design*. The dissertation author was the first investigator and author of this paper. This material is copyright This material is copyright © 2016 by the Institute of Electrical and Electronics Engineers (IEEE).

Chapter 4 contains material from "SPMario: Scale Up MapReduce with I/O-Oriented Scheduling for the GPU" by Yang Liu, Hung-Wei Tseng, and Steven Swanson, which appears in *ICCD '16: 2016 IEEE International Conference on Computer Design*. The dissertation author was the first investigator and author of this paper. This material is

Chapter 5 contains material that is submitted for publication as "Griffin: Improving GPU-Based Web Search with Faster Algorithms and Help From the CPU" by Yang Liu, Jianguo Wang, and Steven Swanson. The dissertation author was the first investigator and author of this paper.

## VITA

| | |
|---|---|
| 2006 | Bachelor of Engineering<br>Beihang University (BUAA), China |
| 2010 | Master of Science<br>Tsinghua University, China |
| 2016 | Doctor of Philosophy<br>University of California, San Diego |

## PUBLICATIONS

Yang Liu, Hung-Wei Tseng, and Steven Swanson. "SPMario: Scale Up MapReduce with I/O-Oriented Scheduling for the GPU", In *ICCD'16: Proceeding of the IEEE 34th International Conference on Computer Design*, October 2016

Yang Liu, Hung-Wei Tseng, Mark Gahagan, Jing Li, Yanqin Jin and Steven Swanson. "Hippogriff: Efficiently Moving Data in Heterogeneous Computing Systems", In *ICCD'16: Proceeding of the IEEE 34th International Conference on Computer Design*, October 2016

Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. "Willow: A User-Programmable SSD", In *OSDI'14: Proceeding of the 11th USENIX Symposium on Operating Systems Design and Implementation*, October 2014

Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. "Be Conservative: Enhancing Failure Diagnosis with Proactive Logging", In *OSDI'12: Proceeding of the 10th USENIX Symposium on Operating Systems Design and Implementation*, October 2012

Jiaqi Zhang, Weiwei Xiong, Yang Liu, Soyeon Park, Yuanyuan Zhou, Zhiqiang Ma. "ATDetector: Improving the Accuracy of a Commercial Data Race Detector by Identifying Address Transfer", In *MICRO'11: Proceeding of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2011

ABSTRACT OF THE DISSERTATION

Systems and Algorithm Support for
Efficient Heterogeneous Computing with GPUs

by

Yang Liu

Doctor of Philosophy in Computer Science

University of California, San Diego, 2016

Professor Steven Swanson, Chair

Heterogeneous computing has seen a great rise in the age of big data. In particular, heterogeneous computing systems with GPUs are able to deliver exceptional performance with better energy efficiency, thus equip us with great power to deal with the enormous yet fast growing volume of data.

In order to exploit the power of such heterogeneous systems with GPUs, we have to address the problems from three closely related aspects. First, we need to design and implement algorithms best suited to the CPU and the GPU, and schedule workloads to the best processors. Second, we have to maximize the utilization of the system resources

for better performance and higher power efficiency. Third, we should provide efficient I/O management and data transfer mechanisms to accommodate the increasing amount of data.

This dissertation explores the systems and algorithm support for efficient heterogeneous computing with GPUs to respond to the above three problems. We first present a system called Hippogriff, which enables efficient direct data transfers between the SSD and the GPU. Hippogriff is able to dynamically choose the best data transfer route for the GPU to improve the overall system performance and efficiency.

We then focus on improving the resource utilization in the context of MapReduce, and present SPMario, a system to scale up MapReduce with the GPU via optimized I/O handling and task scheduling. SPMario proposes I/O Oriented Scheduling to coordinate concurrent task execution in a way that minimizes the idle time of the system resources while avoiding I/O contention.

Last, we present a heterogeneous search engine called Griffin, which explores new parallel algorithms and task scheduling to meet the rigorous requirement of query latency in Web search. Griffin uses a dynamic intra-query scheduling algorithm to break a query into sub-operations, and adaptively schedules them to the state-of-the-art CPU search implementation and to our new GPU-based search kernels, based on the ever-changing runtime characteristics of the queries.

Overall, we demonstrate our systems address the above three problems, and can improve the performance and efficiency of the heterogeneous computing systems with GPUs.

# Chapter 1

# Introduction

Heterogeneous computing has seen a great rise in the age of big data. In particular, heterogeneous computing systems with GPUs are able to deliver exceptional performance with better energy efficiency, with flexibility and wide applicability to various applications. As a result, heterogeneous computing systems equip us with great computing power to deal with the enormous yet fast growing volume of data.

The trend of adopting heterogeneous computing has spread over multiple areas. High performance computing systems have become more heterogeneous and hierarchical, integrating both multi-core CPUs and accelerators like GPUs together [89]. For example, 52 out of the Top 500 world's fastest supercomputers in 2015 utilize NVIDIA GPUs, and they are able to achieve better performance and energy efficiency by coupling high-performance host processors with GPUs [1]. In addition, cloud computing starts to embrace heterogenous computing. As a powerful new paradigm, the combination of the two provides the compute power and data throughput that meet the requirement of many applications [2], and enables the processing ability for workloads like bioinformatics, which are previously limited by conditions such as electricity, cooling, floor space, money, etc., if using standard clusters or parallel processing [20].

The strength of heterogeneous computing comes from combining the best features of different processors. Modern multi-core CPUs have very fast and large caches, and

explore instruction-level parallelism (ILP) with mechanisms such as fine-grained branch prediction, speculative execution, instruction pipelining and prefetch. Therefore, they are suitable for supporting task parallelism. On the other hand, GPUs shine in massive (data) parallelism and high memory bandwidth. A GPU consists of hundreds or even thousands of smaller, simpler but more efficient cores, and can execute in wide SIMD (single instruction, multiple data) manner. It has fast context switch, which can hide the latency caused by instruction stalls and memory accesses, and can achieve memory bandwidth as high as several hundred GB/s.

In order to exploit the power of heterogeneous computing systems with GPUs, we have to address the problems from three aspects that are closely related to each other. The first problem is to design and implement algorithms best suited to GPUs, and to schedule workloads to the best processors that can execute them efficiently. This is the basic and fundamental problem in heterogeneous computing, and still remains an active area. For one thing, we have to develop new algorithms to adapt new workloads, to parallelize existing workloads on GPUs, and to reflect the progress in the research of parallel algorithms for better performance. For another, not all workloads are suitable to run on GPUs; some of them can run faster on CPUs. Even for a single workload, not all parts of it can be efficiently parallelized on GPUs. As a result, we need to differentiate the parts and types of the workloads based on their characteristics, and perform them on CPUs and GPUs accordingly for best fit.

The second problem is to maximize the utilization of the system resources. GPUs are precious resources that provide high performance at the cost of high power consumption. Therefore, we have to keep GPUs busy on two levels. From the perspective of individual programs, their GPU code has to utilize as many GPU cores as possible, i.e., achieving high occupancy of the GPU, for maximum parallelism and performance. From the perspective of the overall system, multiple processes of the same program, or

even multiple programs should find a way to share the GPU efficiently, enabling GPU multiplexing. Indeed, besides the GPU, we ought to improve the utilization of all system resources by keeping them busy working on meaningful jobs cooperatively, including CPUs and other components such as I/O subsystems.

The third problem is to provide efficient I/O management and data transfer mechanism. However, efficient I/O handling is a long-neglected issue in heterogeneous computing systems with GPUs. GPUs, even nowadays, are still deemed as co-processors attached to the host. They do not have the ability to handle I/O operations, and have to rely on the host to load the inputs from and store the results back to the I/O devices. As a result, the traditional GPU programming model either assumes the data has been loaded in GPU's device memory, or leaves the hassle of I/O handling to the host operating system. However, in order to deal with the very large and ever-increasing volume of data today in the real world, heterogeneous computing systems can no longer skirt around the I/O handling problems of GPUs. We have to take into consideration I/O handling when designing and implementing such heterogeneous systems for better overall performance and efficiency.

This dissertation explores the systems and algorithm support for efficient heterogeneous computing systems with GPUs to address the above three problems. The organization of the dissertation is like this. Chapter 2 first introduces the background on heterogeneous computing, discussing the trend and necessity of the heterogeneous computing. It also provides a thorough introduction to the architecture and features of the GPU, explaining the advantages of doing heterogeneous computing with GPUs.

Chapter 3 introduces the design and implementation of a system called Hippogriff, which provides efficient data transfers for GPUs in heterogeneous computing. Conventionally, GPUs have to rely on the host CPU for I/O data transfer, as we discussed earlier. To read the input from the I/O devices such as hard disk drives (HDDs) and solid

state drives (SSDs), the GPU has to ask the host to read the data into a temporary buffer allocated in the host DRAM, and then move the data further to the device memory on the GPU. To write the output to the I/O devices, again the data has to go to the buffer in the host DRAM first before eventually reaching the I/O devices. Keeping the host CPU and DRAM as man in the middle of the GPU data transfer route will unnecessarily occupy CPU cycles and DRAM that can otherwise be used to do more useful jobs. Moreover, it will increase the latency and decrease the throughput of data transfer for GPUs, and lower the power efficiency.

In Hippogriff, we first extend NVMe interface [37] to enable direct data transfer between the SSD and the GPU, bypassing the host. Then, we propose a set of simplified high-level interfaces to ease the users' job of managing the GPU device memory and specifying the data movement requirements. Hippogriff can dynamically choose the best GPU data transfer route between the direct one and the traditional CPU-involved one to improve the overall performance and efficiency of the system. In Chapter 3, we evaluate Hippogriff on multiple workloads. Our results show that Hippogriff can speedup single program workloads by 1.17x, reduce the energy consumption by 17%, and improve the energy-delay by 26%. For multi-program workloads, Hippogriff can achieve a 1.25x speedup. Hippogriff also improves the performance of a GPU-based MapReduce framework by 10%.

In Chapter 4, we focus on the problem of improving the resource utilization of heterogeneous systems in the context of MapReduce, and present SPMario, a system to scale up MapReduce with the GPU. Using GPUs to scale up MapReduce seems a natural and attractive option in existing systems, given that both GPUs and MapReduce provide parallelism. Yet most existing GPU MapReduce systems neglect the impact of the bandwidth mismatch between the I/O devices and the GPU, and only target offloading computation from CPU to GPU. However, lack of efficient I/O handling

and scheduling results in underutilization of system resources. Simply increasing the number of processes/tasks will not solve this problem; instead, it will lead to resource contention [93] among processes/tasks. In fact, our profiling of a state-of-the-art GPU MapReduce framework reveals that, even with multiple processes, the GPU utilization can be lower than 20%, while the I/O device is idle for almost 50% of the time. This inefficiency will undermine the potential performance gains from scaling up MapReduce with GPUs.

In SPMario, we scale up MapReduce with optimized I/O handling and task scheduling, to address the above problems. SPMario runtime includes a scheduler and multiple containers. The scheduler dispatches tasks to containers, while the containers provide the execution environment for the tasks and enforce the execution order. To further improve the performance and resource utilization, SPMario proposes I/O Oriented Scheduling to coordinate task execution in a way that minimizes idle time of the resources, and pipelines different types of operations, including the I/O operations, the data transfers between the host and the GPU, and the GPU kernel execution. We implement the SPMario prototype on a server with an NVIDIA Tesla K20 GPU and a high-end PCIe SSD. Our experiment results show that for the single job cases, SPMario can achieve up to a 2.28x speedup over the baseline in job execution time, and yield up to 2.12x GPU utilization and 2.51x I/O utilization. When scheduling two jobs together, SPMario with I/O Oriented Scheduling outperforms with round-robin by up to 13.54% in execution time, and 12.27% and 14.92% in GPU and I/O utilization, respectively.

Chapter 5 presents the design and implementation of a heterogeneous search engine called Griffin, to explore new parallel algorithms and task scheduling for better performance and utilization of the overall system. Web search engines have to respond to tens or hundreds of thousands of queries per second, over tens of billions of pages with the total size of multiple terabytes [63], all while keeping response times under 300ms [83].

To meet their latency goals, they rely on clever, highly-optimized algorithms that exploit intra-query parallelism. Existing studies also leverage the parallelism that GPUs [63, 44] can provide and can obtain impressive speedup compared to CPU solutions [44]. Because queries may have different characteristics, some queries may run better on GPU, while others run better on CPU. So, a heterogeneous system with CPU and GPU can achieve better overall performance by running a query either on the CPU or the GPU based on its characteristics [63].

However, the characteristics of a query can change as the query executes. While the early stages of a query's execution may run well on the GPU, the later stages are often a better fit for the CPU. Therefore, running an entire query solely on the CPU or the GPU may not achieve the best query processing performance. This suggests that a dynamic fine-grained approach that can schedule operations within the processing of individual queries to the suitable processors will lead to better performance.

So, in Griffin, we address these challenges with a dynamic intra-query scheduling algorithm that breaks a query into sub-operations and schedules them to the GPU or to the CPU based on their runtime characteristics. Griffin uses this scheduling algorithm to divide work between a state-of-the-art CPU-based search implementation and new GPU-based search kernel called Griffin-GPU. Griffin-GPU is the second key innovation in Griffin. Griffin-GPU combines two components. The first is a parallel implementation of the Elias-Fano compression [128] algorithm that provides fast decompression and a high compression ratio. The second is a load-balancing merge-based implementation [3] of parallel list intersection.

Our experiments on the real world query dataset show that Griffin speeds up the query processing by 10x and 1.5x compared to a highly optimized CPU-based search engine and Griffin-GPU running alone, respectively. Griffin also reduces tail latency: it reduces the 95th, 99th, and 99.9th percentile latencies by 10.4x, 16.1x and 26.8x,

respectively, compared the CPU-only implementation.

Finally, Chapter 6 concludes the dissertation and summarizes our findings from the preceding chapters.

# Chapter 2

# Background

This chapter provides the background information about the heterogeneous computing and the GPU for the rest of the dissertation. It first discusses the trend of parallelism and heterogeneity, as well as the motivation of heterogeneous computing with GPUs. It then provides a thorough introduction to the architecture of GPUs and the key technologies used in modern GPUs.

## 2.1 The Trend of Parallelism and Heterogeneity

This section discusses the emergence and the trend of the heterogeneous computing and the heterogeneous computing systems with GPUs.

### 2.1.1 The Failure of Traditional Scaling and the Rise of Heterogeneous Computing

For the past decades, Moore's Law [98] and Dennard scaling [62] have guided chip design to improve the performance of processors. CPU vendors like Intel and AMD were able to boost the clock rates continuously, enjoying the increased transistor budgets and clock speeds granted [131]. However, since around 2006, Dennard scaling began to fail, and Moore's Law hit the power wall and utilization wall [127], starting the era of Dark Silicon [67]. Consequently, even though the size of transistors might continue

to shrink, the increase in the clock rates can hardly progress further, due to the fact that limited by the power constraints, the energy demand would not scale down accordingly.

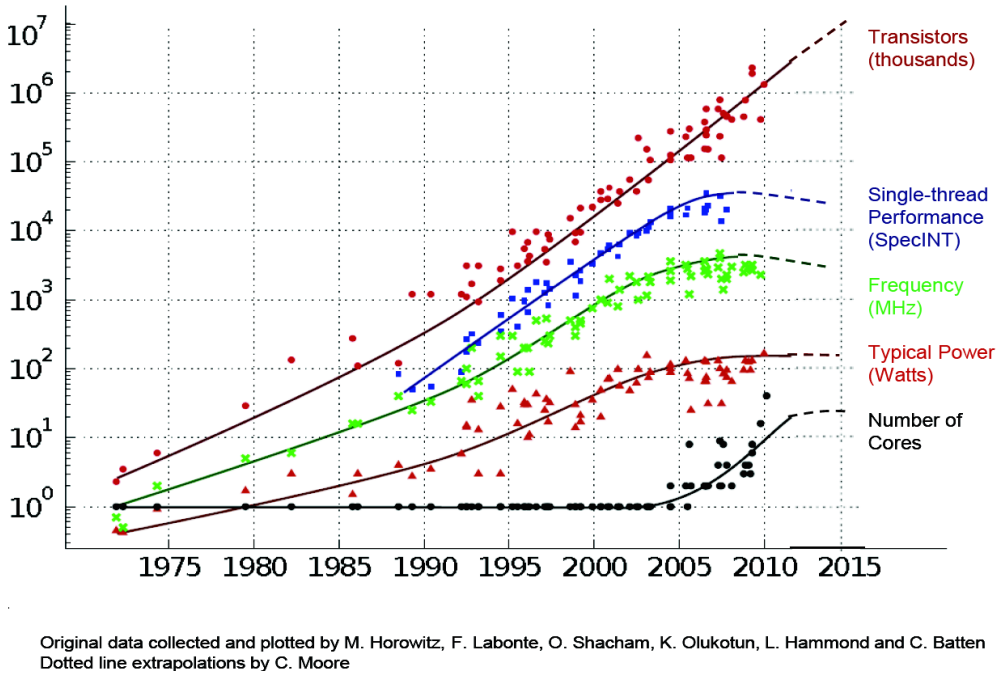



**Figure 2.1.** 35 years of microprocessor trend data [18].

Figure 2.1 demonstrates such trend with 35-year data of microprocessors. From the figure we can observe that since 1970s, the number of transistors on a chip keeps increasing, thanks to the continuous reduction in the transistor size. But the frequency and thus the single-thread performance of the chips begin to stagnate after around 2006, associated with the stagnation in the typical power (Watts) per chip, indicating the failure of Dennard scaling.

In fact, even the shrinking trend of the transistor size may not retain anymore in the near future. According to the report of 2015 International Technology Roadmap for Semiconductors (ITRS) [17], the shrinking in transistor could stop in five years [1],

[1] ITRS chair Paolo Gargini says that some further scaling may be possible after transistors go vertical

which may come sooner than we originally expected. Figure 2.2 compares the ITRS 2015 prediction with the previous more optimistic 2013 one, in which the shrinking trend was predicted to last until some time around 2018.

To continue to boost performance despite the failure of the traditional scaling, both the industry and academia have turned to the multi-core architecture. While not being the final solution, the multi-core architecture does provide the increase in performance within the power constraints by accommodating multiple cores on the same chip. Figure 2.1 reflects such trend, showing that the number of cores on a chip starts to increase since 2005.

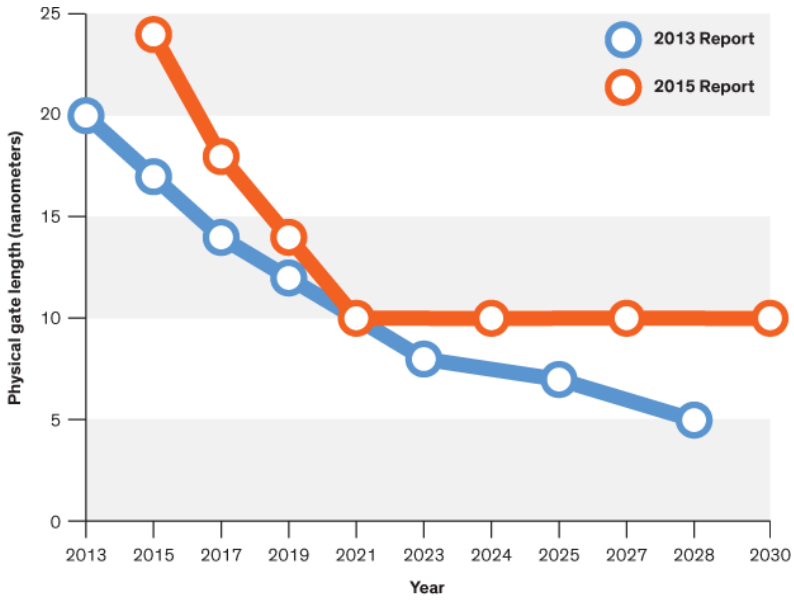



**Figure 2.2.** Comparison between the itrs 2013 and ITRS 2015 prediction in physical gate length of transistors. The previous 2013 ITRS report predicted that the physical gate length of transistors would continue to shrink until at least 2028 (blue line). But ITRS recently modified its prediction in its 2015 report, showing the transistor size will become flat sooner than expected [30].

though [30], but that would be another story.

To achieve their full performance potential, multi-core CPUs require multi-threaded applications to run for parallelism. As a result, only applications that manifest parallelism and can be parallelized easily could benefit from multi-core CPUs. Fortunately, many real world applications show that this is the case.

Furthermore, the drive to improve performance and the continuing constraints on power and scalability in multi-core CPU development have led to the efforts of incorporating specialized hardware (DSPs/FPGAs/GPU etc.) for particular tasks. The rationale behind this is that the characteristics of the tasks may be different, and different types of tasks may have different degrees of demands for parallelism. For example, in heterogeneous systems with CPUs and GPUs, CPUs usually run the operating systems and perform traditional serial tasks that may have complex control logic or branches, while GPUs can execute workloads that require massive parallelism and are mathematically compute intensive.

Indeed, the heterogeneity in characteristics of the workloads and Amdahl's Law result in heterogeneity in the hardware. Compared to CPUs that have more complex cores that include advanced control and compute units as well as large cache, GPUs have a larger number of much simpler cores. So, when Moore's Law drives up the complexity in circuit, GPUs can benefit directly from it in terms of performance thanks to the increase in the degrees of parallelism, while CPUs do not see the gain that much.

### 2.1.2 Heterogeneous Computing with GPUs

Based on how the GPU is coupled with the CPU, there are two types of heterogeneous computing systems: the one with integrated GPUs and the one with discrete GPUs. In the systems with integrated GPUs, the CPU and the GPU reside on the same chip and share the memory bus. A representative example with such architecture is Heterogeneous System Architecture (HSA) [24] developed by HSA Foundation. HSA provides a unified

framework for the integration of the CPU and the GPU, to reduce the communication latency between different processors and to reduce the users' efforts in programming the systems.

HSA aims to break the barriers between the CPU and the GPU by sharing the resources between the two processors. For example, HSA implements the Shared Virtual Memory (SVM) by supporting the shared page table. Both the CPU and the GPU can use the single set of page table entries, and can access the memory via the same virtual address. This simplifies the task of page table management for the operating system. In addition, unlike the discrete GPUs that can only access the pinned host memory thus preventing these memory pages from being swapped out, HSA enables the page faulting for the GPU for better performance and flexibility. Figure 2.3 depicts the architecture of the HSA APU from AMD.

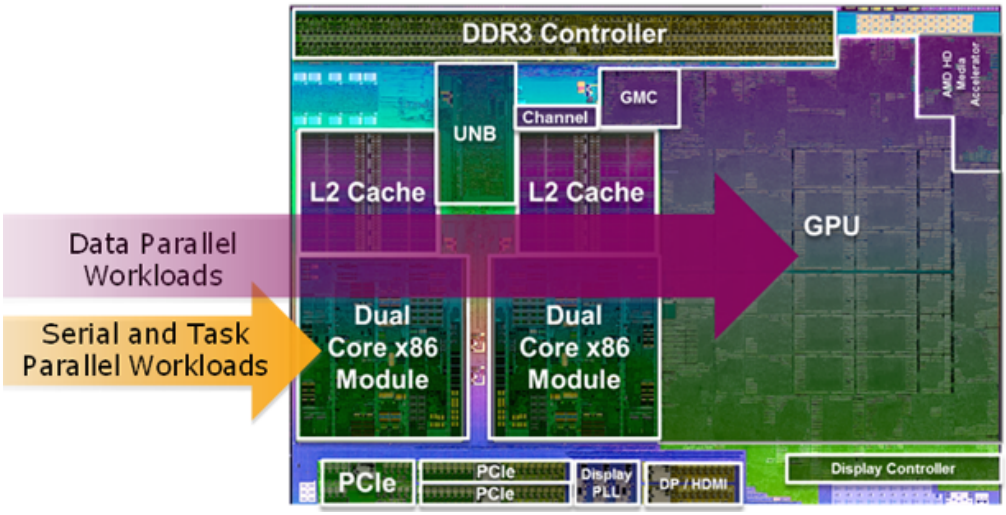



**Figure 2.3.** HSA Accelerated Processing Unit from AMD [25].

Currently, the HSA architecture is mainly popular among desktop processors (e.g., AMD APU, or Accelerated Processing Unit [19]) and mobile processors (e.g., some processors from ARM). The biggest issue of such architecture is that, unlike its discrete counterpart, the integrated GPU usually contains a very limited number of cores (e.g.,

several hundreds) due to the stricter limitation in the die size and power supply, thus can only provide modest computing power. Moreover, the bandwidth of the DRAM shared by both the CPU and the GPU is much lower (e.g., $\sim$ 60GB/s) than that of those discrete GPUs (e.g., $\sim$ 300GB/s or higher).

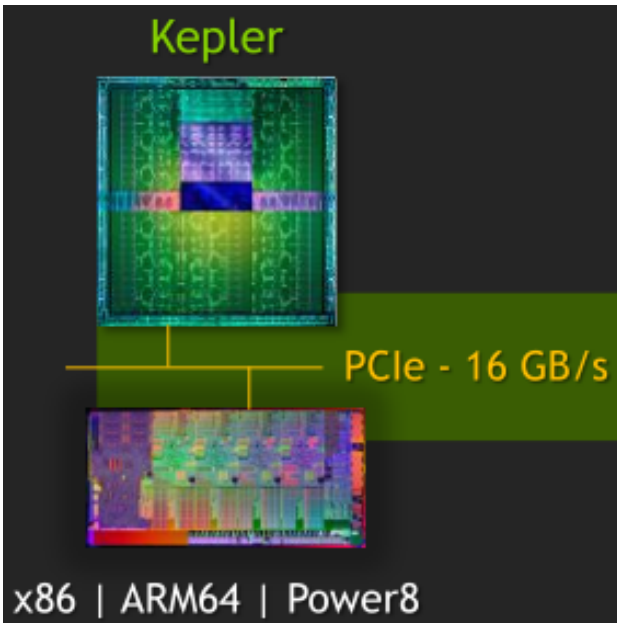



**Figure 2.4.** A system containing a discrete NVIDIA Kepler GPU [4]. The GPU connects to the host (x86/ARM64/Power8) CPU via PCIe, which has a theoretical bandwidth of 16GB/s.

In the systems with the discrete GPUs (Figure 2.4), the GPU is on a standalone, separate chip or card, and usually connects to the host CPU via PCIe bus and has its own cooling system. Therefore, the discrete GPUs can include more cores, and provide much higher computing power with the help of high throughput device memory (e.g., $\sim$ 288GB/s in NVIDIA K40). The advantage of the discrete GPUs in performance is the main reason many of the TOP 500 super computers today are using these GPUs in their systems. However, the disadvantage of the discrete GPUs is also obvious. The GPU has its own device memory that is separated from the host main memory, but does not have the ability to handle I/O independently. As a result, if the GPU requires input data from

the I/O device, it has to rely on the host CPU to load the data from the I/O device into the host DRAM first, and then transfers the data to the device memory of the GPU via the PCIe bus, which has much lower bandwidth (e.g., 16 lane PCIe gen3 has a theoretical bandwidth of 16GB/s) compared to the memory bus.

In this dissertation, we only target the heterogeneous systems with discrete GPUs, due to their huge advantage in performance and their popularity in applications that demand high computing power.

## 2.2 The Basics of the GPU

This sections introduces the basic architecture and the memory hierarchy of the GPU. It also discusses the key technologies used in GPUs for better performance and efficiency. We use a discrete NVIDIA Tesla K20 card (Kepler microarchitecture) with CUDA[2] to implement all the systems in this dissertation, so we will use CUDA terminology to describe NVIDIA GPUs of Kepler microarchitecture. We only focus on the discrete GPU without loss of generality, because an integrated GPU like AMD APU [19] shares the similar basic high level architecture with a discrete GPU, and has less complex memory hierarchy.

### 2.2.1 The Overview of the GPU

The GPU is a kind of multicore processor designed to process problems with massive data parallelism. It comprises multiple SIMD stream multiprocessors, or SMs. Each SM is further composed of many simple execution units called CUDA cores, on which CUDA threads execute sequentially. Threads are organized into thread blocks (containing up to 1024 threads on current GPUs) and grids of thread blocks. The threads inside a thread block run concurrently, and can cooperate with each other via

[2]CUDA [23] is the hardware and software architecture that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages [27].

barrier synchronization and shared memory. Each thread block has a block ID inside its corresponding grid.

Thread blocks are supposed to execute independently from each other, but threads inside the same thread block execute the same instruction at the same time. Every 32 threads within the same thread block are grouped into SIMD execution and basic scheduling units called warps, which are multiplexed onto the same SM. At any time, only threads from one warp execute in lockstep on a SM (but multiple warps can be scheduled), and super fast context switching can hide both arithmetic and memory latency as well as avoid dependency stalls. When the execution meets conditional branches, some cores are disabled for conditional operations.

In CUDA, a program executed by thread blocks on the device is called a CUDA kernel. Each thread inside the thread block executes an instance of the current kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results [27]. The execution of a kernel on the GPU cannot be preempted on the current hardware. Once a kernel starts to run, others will not proceed until the current one exits.

Currently a GPU is still deemed as a coprocessor attached to the host CPU, and there is no direct support for I/O operations. As a result, before the computation starts, a discrete GPU will have to load the input data from the host DRAM to its own device memory. If the input data is not buffered in the host DRAM, then the host will have to load the data from the storage device to the host DRAM before sending it to the GPU's device memory, costing more system resources.

### 2.2.2 The Memory Hierarchy of the GPU

Generally, there are two types of memories in an NVIDIA GPU. One is on-chip memory, and the other one is off-chip memory. The GPU organizes these memories

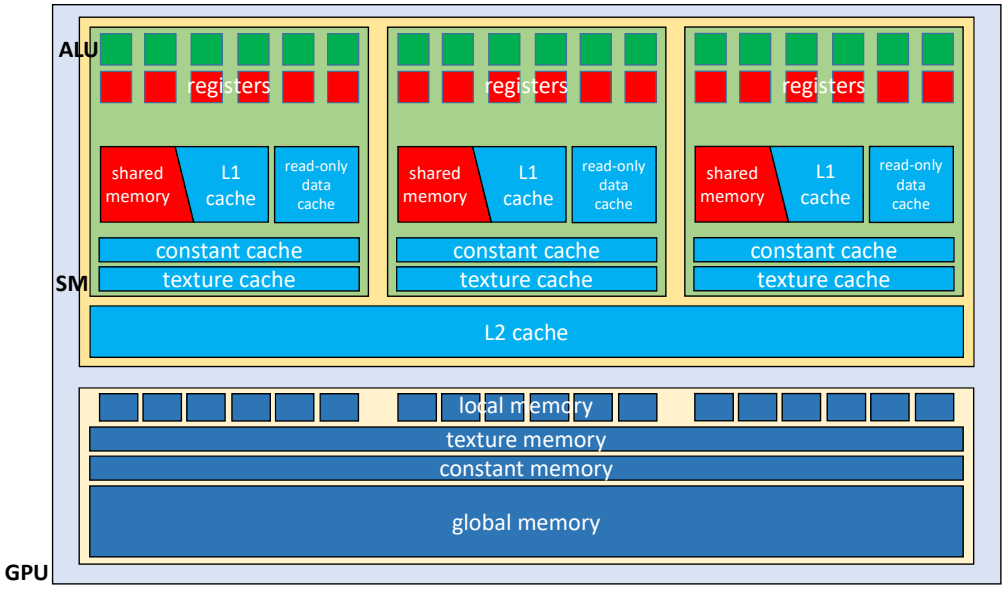



**Figure 2.5.** CUDA GPU memory hierarchy.

hierarchically, as depicted in Figure 2.5.

On-chip memory includes registers, scratchpad memory, and cache. Each SM owns a large register file, which is the fastest on-chip memory. The register file is divided evenly into multiple partitions among different CUDA threads. These partitions are further mapped into the thread-local private memory space of the threads, so that each thread has its own registers to save its states. If the threads do not get enough registers (e.g., for large arrays), they have to use the much slower off-chip local memory within their address space.

Each SM also has a small (64KB) indexable scratchpad memory with high bandwidth and low latency. This memory is partitioned into two different parts: per-thread-block shared memory and L1 cache. Kernels can use the shared memory as a software-managed data cache for inter-thread communication, while L1 cache will automatically cache the data the kernels have accessed. Users can configure this 64KB memory into three configurations: 48KB of shared memory with 16KB of L1 cache, 16KB of shared memory with 48KB of L1 cache, and 32KB of shared memory with

32KB of L1 cache.

Besides the scratchpad memory, we can also find an optimized read-only 48KB cache on GPUs with Kepler microarchitecture. It can improve the system performance because it allows the read-only accesses to bypass the shared/L1 cache path, and can support full speed unaligned memory access patterns with its higher tag bandwidth [29].

The on-chip memory also includes constant cache (for broadcasting of reads from a read-only memory) and texture cache (for aggregating bandwidth from texture memory), as well as the L2 cache of 1536KB, which is the primary point of data unification among different SMs. The L2 cache serves all load, store, and texture requests and providing efficient high speed data sharing across the GPU [29].

The off-chip memory includes global memory and local memory. The global memory (also referred as the device memory) has the largest size (e.g., 5GB in the Tesla K20 we use) in the GPU memory hierarchy, but also has the longest access latency (accesses may take hundreds of cycles). Grids of thread blocks can share the results in the global memory after the global synchronization among kernels.

Although called "local memory", the local memory locates off the chip instead of on the chip, because "local" here means thread local. The local memory is as slow as the global memory, and threads use local memory for the register spilling purpose as we mentioned previously.

Other types of off-chip memory include the constant and texture memory. The constant memory stores constants and kernel arguments, while the texture memory is optimized for 2D spatial access pattern. These two types of memory are cache optimized with the corresponding on-chip constant and texture cache we have introduced above.

Thanks to the multi-level memory hierarchy, the GPU is able to tolerate the very expensive accesses to the global/device memory, where the GPU computation usually starts. In addition, the GPU also uses a high degree of multithreading and fast context

switching to hide the memory access latency for better utilization, by keeping more active threads than the number of cores. For example, the Kepler GPUs allow up to 64 active warps on each SM. If a warp is waiting for the memory accesses, the execution will switch to another active warp that is ready to run.

### 2.2.3 Memory Address Space and Data Transfer

#### Unified Virtual Addressing (UVA) and Unified Memory

The discrete GPU has its own device memory, which is separated from the host DRAM. In the early ages of CUDA, the GPU and the CPU also have separate virtual memory address space, thus they cannot access each other's memory address directly. To access the data on the host, the GPU has to memcpy (via cudaMemcpy calls) the data from the host DRAM to its device memory explicitly; after the computation, the GPU has to memcpy the results back from its device memory to the host DRAM explicitly, if the results are needed on the host.

CUDA 2.0 introduces "zero-copy memory", which allows the GPU threads to directly access the data on the host DRAM (must be pinned host memory), as long as they obtain the device pointers to the data. Zero-copy memory maps the host DRAM address into the GPU address space so that it is visible to the GPU threads, but the actual data transfer from the host to the GPU still happens via the PCIe bus when the data is accessed. Unlike using cudaMemcpy, however, zero-copy does not need users to do explicit data transfers, and only data that is needed instead of the whole data chunk will move over the PCIe bus.

Sharing data via zero-copy memory requires synchronization of memory accesses between the host and device memory. Otherwise, undefined behaviors will happen when simultaneous modifications to the data in zero-copy memory occur from both the CPU side and the GPU side. When multiple accesses to zero-copy memory take place, these

accesses will result in multiple small (and probably synchronous) transfers via the PCIe bus with low bandwidth. In this case, an explicit data transfer in advance is more efficient than multiple zero-copy accesses.

CUDA 4.0 further introduces Unified Virtual Addressing (UVA) to provide a single unified virtual address space for both the CPU and the GPU, when applications run as 64-bit processes [21]. In UVA, there is no distinction between the host and device pointers, and CUDA runtime will determine if a pointer is on the host CPU or on the GPU. Thus, the CUDA kernels can directly access the pointers to the data on the host DRAM (must be pinned host memory though). Thanks to this single virtual address space, UVA enables simplified data accesses between the host CPU and the GPU, as well as between two different GPUs. The direct data access between the GPUs is also referred as "Peer-to-Peer" direct access.

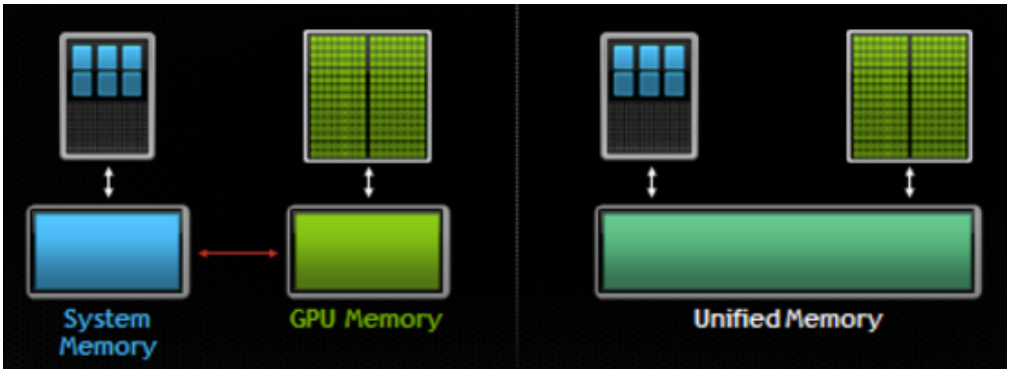



**Figure 2.6.** The separate memory space versus the unified memory space [31]. There was a clear physical and logical distinction between the host and device memories before CUDA 6.0 (on the left). UVA provides a unified virtual memory address space, and UM provides systematic support via automatic on-demand data transfers (on the right).

While zero-copy memory and UVA simplify GPU data accesses, they still require users to explicitly allocate the pinned host memory, and each access across devices will result in data transfer via the PCIe. NVIDIA adds a new feature called Unified Memory (UM) [31] since CUDA 6.0, as shown in Figure 2.6, in addition to the single address

space provide in UVA, to further improve the performance and efficiency of data transfers. Unlike in UVA, where users have to allocate the pinned host memory and the device memory before transferring data, UM automatically allocates the managed memory only on the device. When the CPU touches the data via the address, it will trigger page faults, and the CUDA runtime will migrate the part of the data from the GPU to the host via PCIe in the granularity of pages (typically 4KB). This data migration is transparent to users, and only involves dirty pages. The migration can happen both from the host to the device, or from the device to the host, and the migrated data can be moved back if needed.

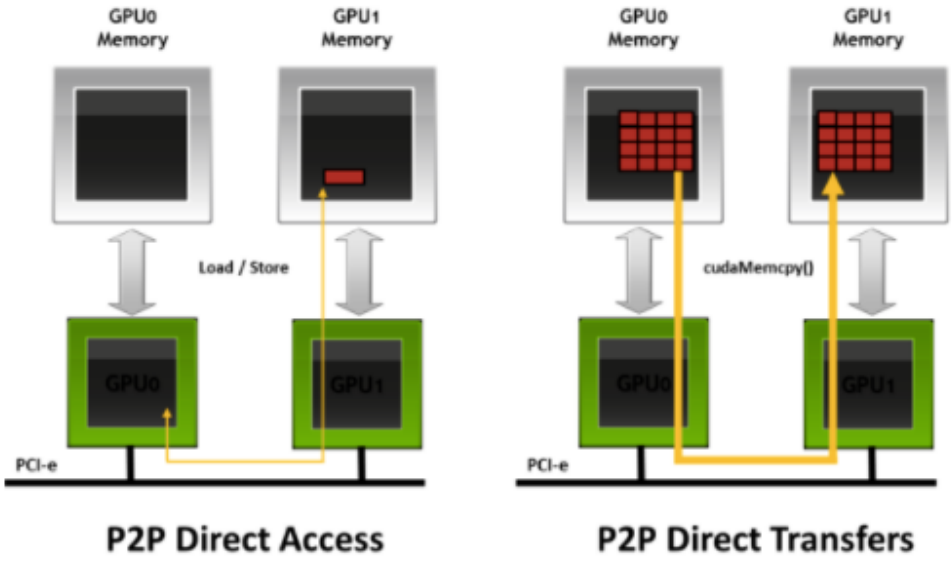



**Figure 2.7.** GPUDirect P2P [28]. A GPU can directly issue load and store to another GPU on the same PCIe root complex (on the left), and do direct data transfer from its device memory to that of another GPU (on the right).

Although UM depends on UVA, they are very different. UM can achieve better performance via the automatic on-demand page-level data transfers, while UVA only allows the kernels to have direct data access via the PCIe. Of course, users may get the best performance if they know when and where to move the data through well planned explicit data transfers. Even UM in this case may not be the best choice since it does not

know the intention of the users.

**GPUDirect**

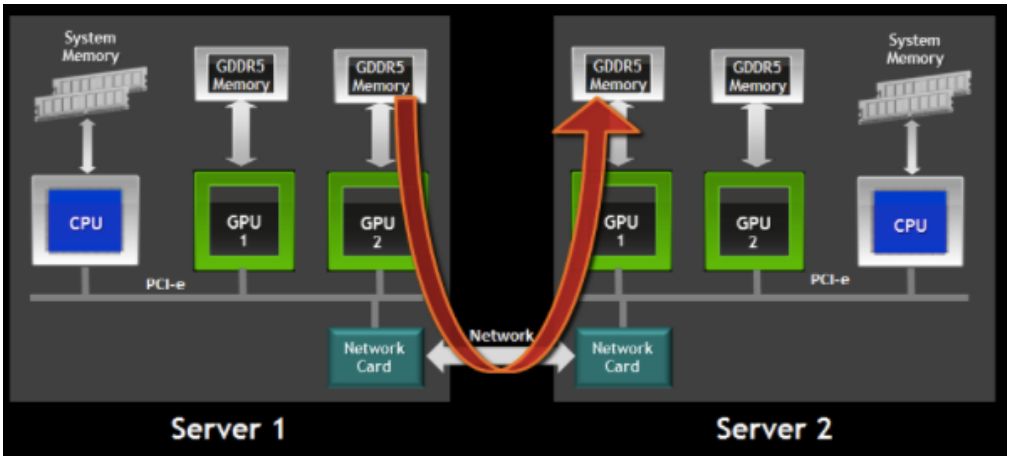



**Figure 2.8.** GPUDirect RDMA [28]. With the RDMA support, applications can do GPU-PCI-network-PCI-GPU communications bypassing the host DRAM.

GPUDirect [28] is a family of technologies that enable direct data communication among GPUs (even remote ones), and between GPUs and other PCIe devices such as NICs. This family includes three main technologies. GPUDirect Shared Access supports direct access between GPUs and PCIe devices via shared pinned memory. GPUDirect P2P (Figure 2.7) enables direct loads and stores from one GPU to another GPU, as well as direct data transfer from the device memory of one GPU to that of another, with both GPUs connected with PCIe. GPUDirect RDMA (Figure 2.8) realizes inter-node direct access between GPUs and other PCIe devices. By removing the unnecessary data transfers from and to the host DRAM, GPUDirect can improve the system performance.

### 2.2.4 Task Parallelism Support in the GPU

**CUDA Streams and Asynchronous Operations**

While the GPU computing mainly targets on massive data parallelism, CUDA GPUs also support task parallelism to efficiently use different processors in heterogeneous

computing. To enable the concurrency in executing multiple kernels and overlapping the data transfers with kernel execution and other operations, CUDA allows applications to run asynchronous commands in streams.

In CUDA, a stream is a sequence of operations executed on the device in the order they are issued from the host. The operations within the same stream will execute in the guaranteed FIFO order, and cannot overlap. The operations in different streams, however, can execute out of order, interleaving or even running concurrently with each other.

In CUDA, all operations have to run in one of the two types of streams, and they can specify the chosen stream as an argument. When no stream is specified, the operations will run in the default stream. Before CUDA 7.0, there is only one single default stream on each GPU shared by all host threads on that GPU, and all operations inside that default stream will have to synchronize with the host. This means an operation in the default stream cannot start until the previous operation finishes and returns the control to the host. Since CUDA 7.0, NVIDIA introduces per-thread default streams, to allow each thread to have its own asynchronous default thread.

In addition to the default stream, applications can create non-default streams. These non-default streams support asynchronous operations, such as kernel invocation and asynchronous memcpy. In this way, applications can run multiple kernels concurrently and overlap kernel execution with data transfers, by putting operations in different streams.

Figure 2.9 shows the examples of overlapping different operations. In the serial case, the operation of asynchronous memcpy from the host to the device (“cudaMemcpyAsync(H2D)” inside the yellow block), the kernel execution (“Kernel” inside the green block), and the operation of asynchronous memcpy from the device to the host (“cudaMemcpyAsync(D2H)” inside the yellow block) execute sequentially inside the


**Figure 2.9.** Overlapping kernel execution with data transfers [5].

default stream, thus on overlapping happens at all. In the 2-way concurrency case, kernels (K1 – K4) can overlap with data transfers from the device to the host (DH1 – DH4) in different streams, but the kernels and data transfers inside the same streams still execute sequentially. In the 3-way concurrency case, the data transfers from the host to the device, the execution of different kernels, and the data transfers from the device to the host could all overlap with each when they are in different streams. In this case, we can also see that even the data transfers from the host to the device and that from the device to the host from different streams can overlap. This is because in some NVIDIA cards (like with the Kepler architecture), there are two separate data copy engines.

As long as there are enough available resources (including the streams, the GPU cores, the device memory, etc.), more operations can run concurrently, as we see from the 4-way and 4+ way concurrency cases in Figure 2.9. Kepler GPUs can support up to 32 concurrent streams with 32 simultaneous hardware-managed work queues called Hyper-Q, and the concurrency with multiple streams improves performance.

### Multi-Process Service (MPS)

A CUDA program starts to execute by creating a CUDA context for a specific GPU, either with explicitly using the driver API or implicitly using the runtime API [22].

A context is the encapsulation of all the necessary hardware resources (e.g., the device memory) for a program to run on the GPU.

Each host process has a unique context, and at any time there is only one active context on a GPU, because the GPU uses a time sliced scheduler to schedule work from work queues belonging to different CUDA contexts [22]. As a result, work from multiple processes cannot operate concurrently, even they are from different streams (and these streams are in different processes). Only operations within different streams belonging to the same process/context can run concurrently.

To enable better sharing of the GPU resources among different processes, CUDA introduces Multi-Process Service (MPS). MPS is a binary-compatible client-server run-time implementation of the CUDA API [22], and sits between the CUDA driver and the applications. MPS includes three components. The MPS control daemon starts and stops the MPS server, and coordinates connections between clients and servers. The MPS client runtime resides inside the CUDA driver library, and applications use the runtime to communicate with the MPS server transparently to share the GPU resources with each other. The MPS server process grants the GPU resources to the clients, providing the concurrency.

Figure 2.10 and Figure 2.11 demonstrate how MPS benefits applications such as Message Passing Interface (MPI) [26] programs. In this example, the MPI application has two separate processes A and B. Without MPS (Figure 2.10), the MPI process A and B will have to take turn to run within their assigned time-slice serially. Thus, kernel A1 from process A will not be scheduled to run until kernel B1, B2, and B3 from process B run inside B's time slice. This is because each MPI process has its own context, and the time-sliced scheduler has to swap on and off different contexts to allow multiple processes to share the GPU. As a result, at any time, only one MPI process is running on the GPU.

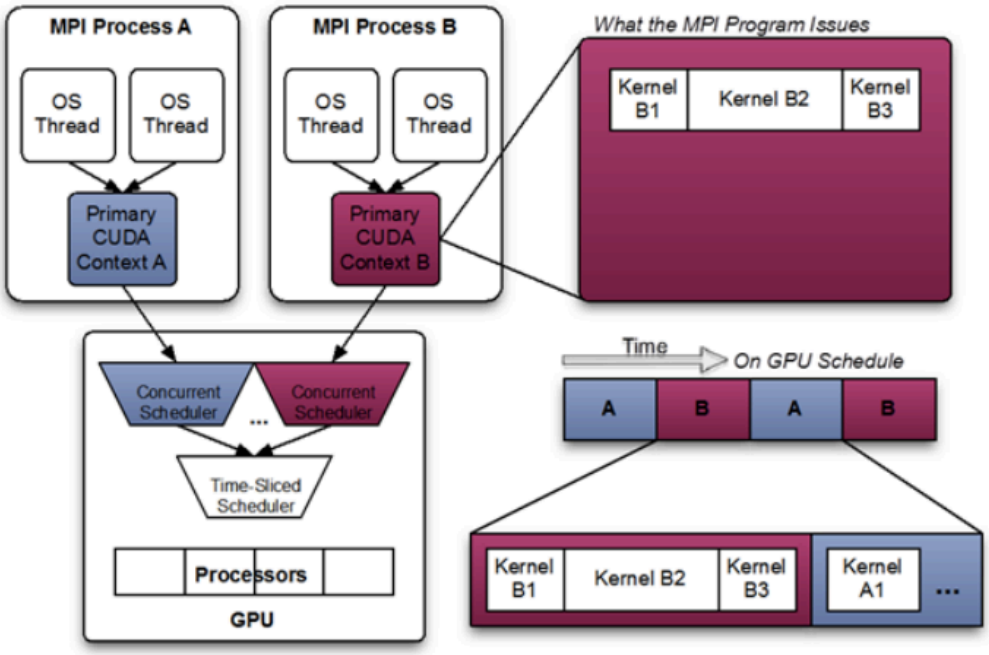



**Figure 2.10.** A likely schedule of CUDA kernels when running an MPI application consisting of multiple OS processes without MPS [22].

Things are different with MPS. The MPS server maps multiple contexts of different MPI processes into one single MPS server context, and these processes then communicate with the GPU via the shared MPS server context instead of their own individual contexts. Thus, the MPI processes can bypass the hardware limitations enforced by the time sliced scheduling to share the GPU more efficiently. In this way, MPS allows multiple tasks from different processes to run on the GPU concurrently, and to overlap the kernels and data transfers, achieving higher utilization of the system resources. In the above example, now kernel A1 from process A now can run concurrently with kernels from process B, as shown in Figure 2.11

In addition, MPS reduces the storage requirement of the GPU contexts. MPS just needs to maintain one copy of the context, and this copy will be shared across all processes. If without MPS, the GPU would have to keep a separate context including storage and scheduling resources for each process. Furthermore, MPS also eliminates the

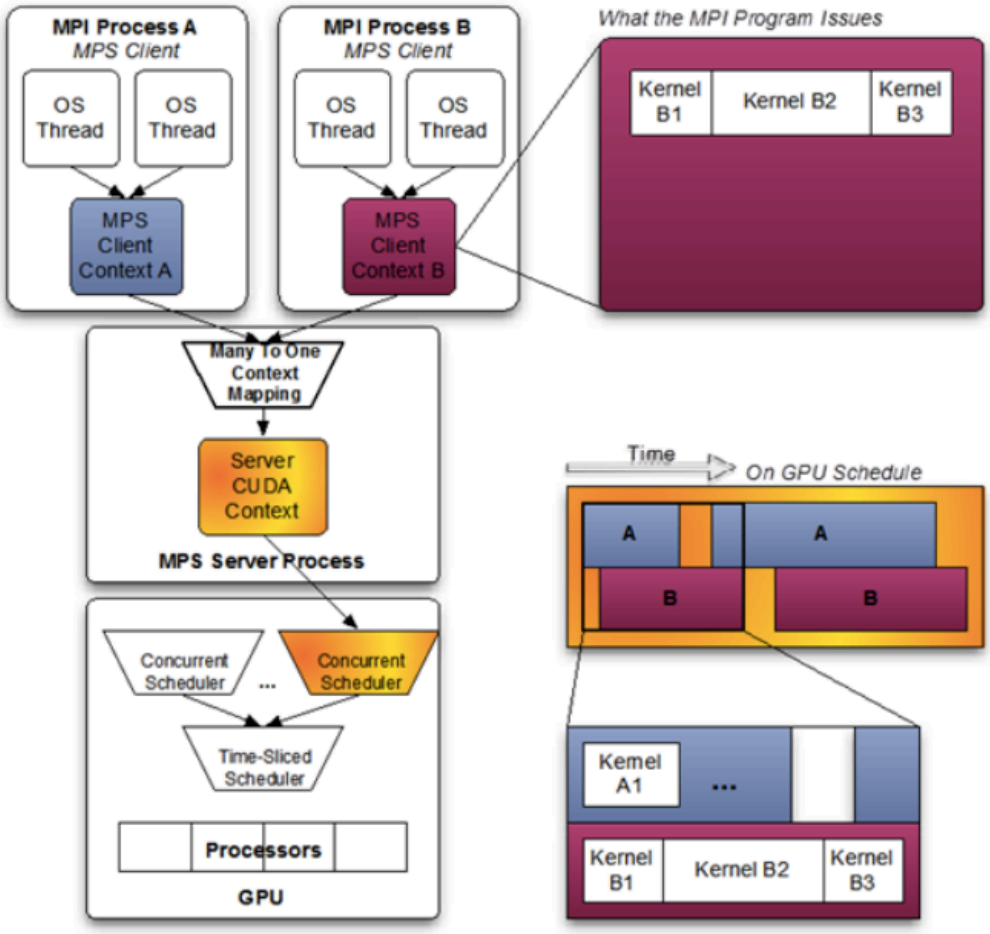



**Figure 2.11.** A likely schedule of CUDA kernels when running an MPI application consisting of multiple OS processes with MPS [22].

context switching overhead because it does not need to switch between multiple contexts of different processes.

# Acknowledgements

This chapter contains material from "Hippogriff: Efficiently Moving Data in Heterogeneous Computing Systems" by Yang Liu, Hung-Wei Tseng, Mark Gahagan, Jing Li, Yanqin Jin and Steven Swanson, which appears in *ICCD '16: 2016 IEEE International Conference on Computer Design*. The dissertation author was the first

investigator and author of this paper.

# Chapter 3

# Transferring Data Efficiently in Heterogeneous Computing Systems

The ever-growing size of application data is reshaping the design of modern high performance systems. To be able to handle such huge amount of data, a system has to excel in both processing/computing power and data transfer throughput. While heterogeneous computing systems with different types of processors (e.g., CPUs and GPUs) can provide high computing power, the efficient data transfer still remains a challenge. So in this chapter, we describe how we provide efficient data transfer in heterogeneous computing systems.

Efficient data transfer and I/O handling is a long-neglected issue in heterogeneous computing systems. Most co-processors (e.g., GPUs, FPGAs) attached to the host in a heterogeneous system do not have the ability to handle I/O operations. Thus, to access the data stored in the I/O devices, they have to adopt the CPU-centric model, and rely on the host CPU to first transfer data from the source to the main memory before to the final destination.

In addition to the entrenched CPU-centric programming and execution model, high-speed interconnects like PCIe provide fast peer-to-peer communication between devices in a computer [103, 33, 100, 130, 41, 82, 105, 102, 129, 51, 88, 139, 116]. This

brings multiple benefits. First, it reduces data transfer latency to remove the main memory and the CPU from the data path. Second, it frees the CPU from managing data movement to perform useful work or shift to the low power mode to save energy. Third, it reduces contention for TLB (Translation Look-aside Buffer) entries and memory bandwidth.

However, the existing systems require the programmers to choose either the entrenched CPU-centric or the peer-to-peer model. Thus, it is cumbersome and difficult to make the best use of heterogeneous system resources, as the programmers might not be able to accurately predict the runtime workloads and make the best choice.

To address this problem, we designed *Hippogriff* system. Hippogriff proposes a novel software interface and runtime system to manage the data transfers for the applications.

As an initial step towards the full Hippogriff design, we focused on the data transfer between the NVM-Express (NVMe) [38] solid-state disks (SSDs) [38, 80, 114] and the GPUs. We implemented *NVMeDirect*, an extension to the existing NVMe interface to support peer-to-peer transfers between the NVMe SSDs and the GPUs, in addition to the conventional way of sending data through CPU and main memory. The Hippogriff runtime system automatically selects the most efficient route of data transfers, making applications running on the Hippogriff system agnostic to the underlying interconnect technology. Thus, applications can adopt new interconnect technologies without any changes.

We evaluated the Hippogriff system on a computer containing an Intel Xeon processor, an NVIDIA K20 GPU, and a high-end NVMe SSD. Experiments demonstrate that for a single GPU-accelerated workload, Hippogriff speeds up the application end-to-end latencies by 1.17×, reduces the energy consumption by 17%, and improves the energy-delay product by 26%. With slower CPUs that consume less power, Hippogriff becomes even more beneficial, speeding up applications by 1.21× on average. As the Hippogriff

runtime system dynamically dispatches data transfers across the available paths when multiple processes are running, Hippogriff can accelerate the execution of multi-program GPU applications by 19%–31% on average. Even for a GPU-MapReduce framework that is highly optimized for heterogeneous computing systems [123], Hippogriff still improves the performance of the framework by 12%.

The remainder of this chapter is organized as follows. Section 3.1 introduces the two approaches of data transfer. Section 3.2 describes Hippogriff's design, implementation, and programming model in detail. Section 3.3 describes the system configuration and the benchmarks to evaluate Hippogriff. Section 3.4 evaluates Hippogriff's performance and energy efficiency. Section 3.5 compares Hippogriff with previous efforts. Section 3.6 summarizes this chapter.

## 3.1 Moving Data in Heterogeneous Systems

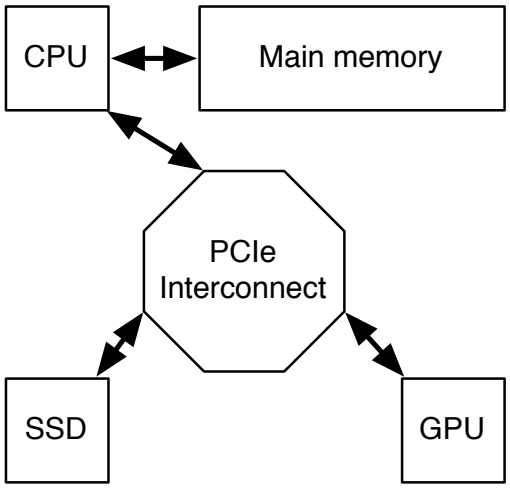



**Figure 3.1.** A typical heterogeneous computing system with GPU.

Heterogeneous computing systems bring together diverse hardware resources such as CPUs, GPUs and SSDs to improve performance, as presented in Figure 3.1. This

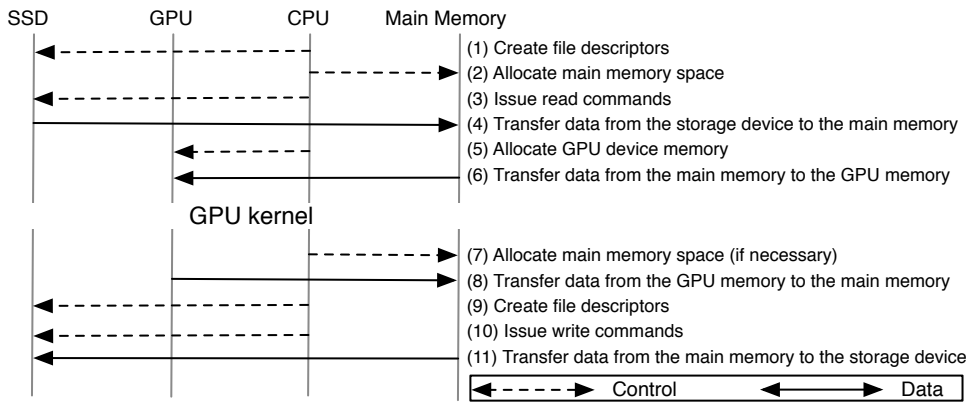



**Figure 3.2.** The conventional CPU-centric data movement between the GPU and the SSD.

section provides a brief introduction to the two types of data transfer between the storage devices and the computing resources: the conventional CPU-centric approach and the emerging peer-to-peer approach.

### 3.1.1 Conventional CPU-centric Data Transfer

In the current programming models for heterogeneous computing systems (e.g., CUDA or OpenCL), the CPU serves as both the control plane and data plane. When an application needs a data transfer from the SSD to the GPU, the SSD's driver has to first transfer the data to a DRAM buffer via DMA, and then the GPU driver delivers the data to the GPU also via DMA. Figure 3.2 illustrates this process in detail.

Steps (1–6) transfer data between the SSD and the GPU. In Steps(1–3), the application creates file descriptors, allocates DRAM memory buffers, and issues read() system calls. After the SSD directly accesses main memory in Step (4), the CPU allocates space in the GPU (Step (5)) and, in Step (6), copies the data from the main memory to the GPU memory. Steps (7–11) show the same process in reverse as the program moves the results of the GPU kernel back to the SSD.

The obvious problem is, although the application just needs the data transferred

from the SSD to the GPU, the actual data transfer has to go through the DRAM buffer first. This inefficiency wastes CPU time and memory bandwidth, pollutes TLBs and caches, and consumes DRAM unnecessarily.

### 3.1.2 Peer-to-peer Data Transfer

PCIe allows devices to send data packets directly from one to another, bypassing the CPU and the main memory. Supporting PCIe peer-to-peer data transfer model requires a device to map its own device memory to PCIe base address registers (BARs) in the PCIe controller/switch. AMD's DirectGMA [33] and NVIDIA's GPUDirect [100] are technologies that allows the software to program GPUs for PCIe peer-to-peer communication.

However, the standard NVMe SSD cannot natively support peer-to-peer data transfer, as NVMe SSDs are not byte-addressable memory devices but use block addresses for its own data array.

## 3.2 Hippogriff

Hippogriff decouples the control plane from the data plane for applications, and manages the low-level details of scheduling and executing data transfers. By eliminating the main memory from the data path when using NVMeDirect, Hippogriff can reduce the data transfer latency, and free up the CPU and the main memory for other tasks.

Figure 3.3 presents the components of the Hippogriff system. Hippogriff contains three main components:

*Hippogriff API:* Hippogriff API allows the applications to specify the source and destination of the data transfer.

*Hippogriff Runtime System:* The Hippogriff runtime system selects the best route for

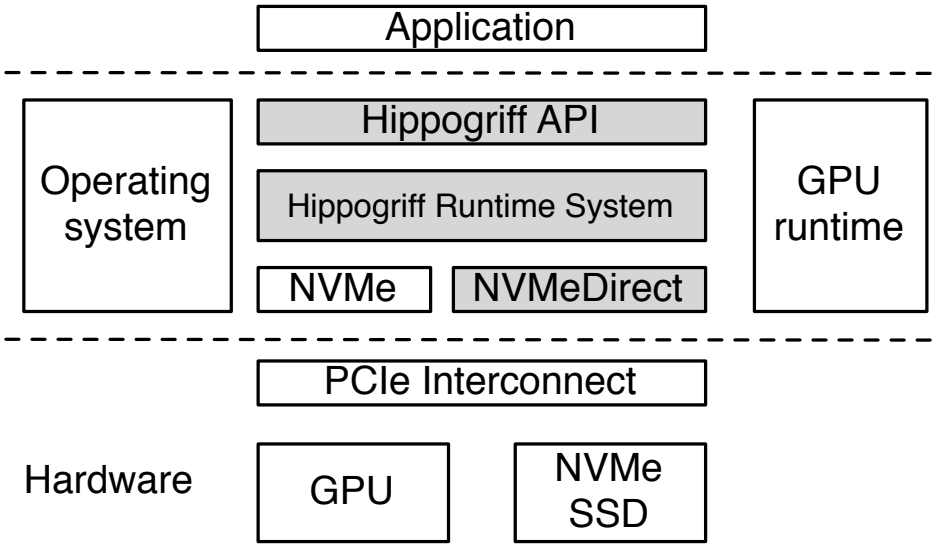



**Figure 3.3.** The system components of Hippogriff.

the data transfer based on the runtime information. It understands the file layout in the storage devices, and monitors resource usage on the destination device.

*NVMeDirect:* Hippogriff implements NVMeDirect to achieve peer-to-peer transfer between two PCIe devices (e.g, the GPU and the SSD).

### 3.2.1 Hippogriff API

The Hippogriff API allows the applications to specify only the sources and the destinations of the data transfer, while leaving the actual data movement to Hippogriff.

The applications can use the API to initialize the environment, create data access requests, perform data transfers, and release the resources when the transfer is done. The Hippogriff API interacts with the underlying file system, operating systems, and the Hippogriff kernel module to acquire permission for accessing files. Table 3.1 documents the API.

Figure 3.4 compares a simplified LUD (from Rodinia Benchmark [112]) CUDA

**Table 3.1.** The Hippogriff API.

| Synopsis | Description |
|---|---|
| `int hippogriff_init()` | The `hippogriff_init()` function initializes the Hippogriff runtime. |
| `size_t hippogriff_send(int fd, void **gpuMemPtr, size_t offset, size_t size)` | The `hippogriff_send()` function creates the Hippogriff task of sending data from file descriptor `fd` with `offset` to GPU memory location `gpuMemPtr`. If the user passes a `gpuMemPtr` containing *NULL*, Hippogriff automatically allocates the GPU memory space fits the input file or requested size. |
| `size_t hippogriff_recv(int fd, void *gpuMemPtr, size_t offset, size_t size)` | The `hippogriff_recv()` function creates the Hippogriff task of sending `size` bytes of data from GPU memory location `gpuMemPtr` to the file that `fd` links to with `offset`. |
| `int hippogriff_deinit()` | This function releases the resource that Hippogriff uses for an application. |

```
1:  fp = open(filename, O_RDWR);
2:  m = (float*) malloc(sizeof(float)*matrix_dim*matrix_dim);
3:  read(fp,m,sizeof(float)*matrix_dim*matrix_dim);
4:  close(fp);
5:  cudaMalloc((void**)&d_m,
      matrix_dim*matrix_dim*sizeof(float));
6:  cudaMemcpy(d_m, m, matrix_dim*matrix_dim*sizeof(float),
          cudaMemcpyHostToDevice);
7:  lud_diagonal<<<1, matrix_dim>>>(d_m, matrix_dim);
8:  cudaMemcpy(m, d_m, matrix_dim*matrix_dim*sizeof(float),
                                    cudaMemcpyDeviceToHost);
9:  fp = open(filename_output, O_RDWR);
10: write(fp,m,sizeof(float)*matrix_dim*matrix_dim);
11: close(fp)
```

**(a)**

```
1:  hippogriff_init();
2:  fp = open(filename, O_RDWR);
3:  hippogriff_send(fp, (void **)&d_m, 0, 0);
4:  close(fp);
5:  lud<<<1, matrix_dim>>>(d_m, matrix_dim);
6:  fp = open(filename_output, O_RDWR);
7:  hippogriff_recv(fp, d_m, 0,
                     sizeof(float)*matrix_dim*matrix_dim);
8:  close(fp);
9:  hippogriff_deinit();
```

**(b)**

**Figure 3.4.** A simplified LUD CUDA code snippet example (a) implemented with the conventional CPU-centric model and (b) with Hippogriff.

code snippet example implemented with the conventional CPU-centric model against that with Hippogriff. In the conventional implementation (Figure 3.4(a)) with CUDA, the application has to first open a file, allocates the host DRAM buffer, reads the input data into the DRAM buffer (Line 1–3). And then it will allocate the device memory on the GPU (Line 5), copies the data from the host to the device (Line 6) to do the computation (Line 7). After the computation, it will have to copies the results back to the host (Line 8), and write the results out to the SSD (Line 9–11).

As the Hippogriff API does not rely on the application to setup data transfer, the

resulting code (Figure 3.4(b)) is much simpler. After the call to `hippogriff_init()` initializes the Hippogriff runtime system, `hippogriff_send()` sends data from the SSD to the GPU and reports the location of data in the destination device using pointer `d_m`(Line 3). Once the function finishes, the kernel running on the GPU can access the memory in `d_m` (Line 5). When the kernel completes, `hippogriff_recv()` moves data directly from the GPU to the SSD (Line 7), and releases any resources Hippogriff was using with `hippogriff_deinit()` (Line 9).

### 3.2.2 Hippogriff Runtime System

The Hippogriff runtime system accepts the data transfer requests from the API, and schedules them to the optimal data transfer route. Between the two routes that the current Hippogriff runtime supports, the runtime always favors the direct data transfer enabled by NVMeDirect. However, when the GPU does not have enough available memory or is under heavy load, Hippogriff can schedule the data transfer to the main memory buffer first before the GPU is ready. This results in lower overall latency and higher bandwidth between the GPU and main memory, compared to the SSD bandwidth.

### 3.2.3 NVMeDirect

NVMeDirect extends the NVMe interface with new `ioctl`-based read and write commands to move data directly between SSDs and GPUs. As an NVMe SSD is not byte-addressable memory (but the GPU is), the NVMeDirect driver together with GPUDirect or DirectGMA prepares GPU memory for the PCIe peer-to-peer transfers and generates NVMe read and write commands that use GPU memory as DMA targets.

When the SSD receives the command, it reads or writes data directly from or to the GPU without the involvement of the CPU and the main memory. Because NVMeDirect still relies on the CPU code to issue read/write commands, NVMeDirect does not incur

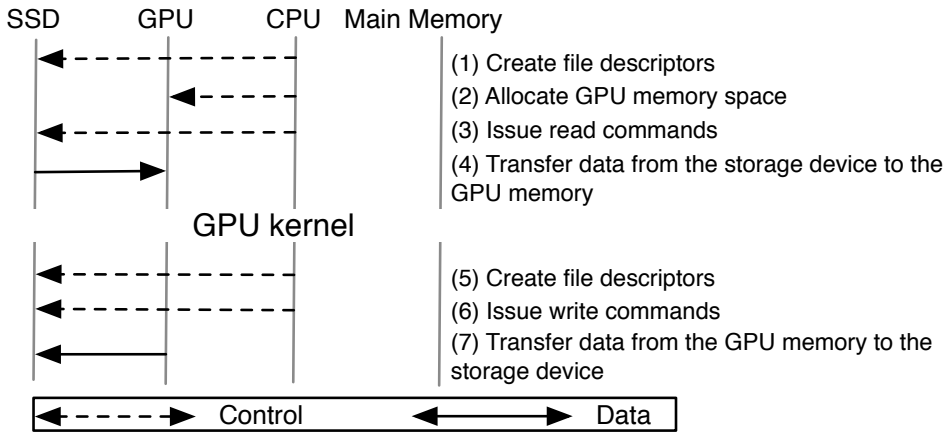



**Figure 3.5.** The process of establishing direct data transfer for NVMeDirect.

any new file system integrity issues. Figure 3.5 illustrates how NVMeDirect avoids unnecessary copies to and from main memory.

After the Hippogriff API initiates a file transfer and obtains a file descriptor from the operating system (Step (1)), the Hippogriff runtime requests a region in the GPU memory (Step (2)). In Step (3), NVMeDirect issues NVMe read commands that include the GPU memory addresses to the SSD. Finally in Step (4), the SSD pushes data directly from the SSD to the GPU using DMA, without any CPU involvement.

To write computations to the SSD, NVMeDirect follows Steps (5–7), which resemble Steps (1–4) except that NVMeDirect issues NVMe write commands instead of read commands and pulls data directly from the GPU memory.

NVMeDirect requires pinning GPU device memory to a BAR in the PCIe controller, which is a slow operation [100]. Therefore, the NVMeDirect module maintains a *pinbuffer*. When an application calls `hippogriff_init()`, Hippogriff pre-allocates a pinned memory region in the pinbuffer. Future Hippogriff transfers reuse this pinned GPU memory and avoid the overhead of manipulating PCIe BARs.

### 3.2.4 Implementation Details

Implementing Hippogriff requires the GPU to support GPUDirect that makes the GPU memory available for PCIe devices. In addition, as Hippogriff uses the NVMe command to send the GPU memory address to storage devices, Hippogriff also requires the PCIe-attached storage device to support the NVMe standard. With these hardware components, we modified the Linux drivers so that our Hippogriff library can set up direct data access channels between the GPU and the storage device. In this section, we will describe our modifications to the drivers and the implementation of our Hippogriff system library.

To create a GPU memory region that is accessible to other PCIe devices, Hippogriff first allocates a memory region in the GPU memory. Then Hippogriff uses the `nvidia_p2p_get_pages()` function that CUDA provides for GPUDirect to pin this GPU memory location in the PCI BAR. Pinning GPU device memory in the PCI BAR is a time-consuming operation that takes several milliseconds [100]. Therefore, we implemented a virtual device, called *pinbuffer*, in the system. When an application calls the `hippogriff_init()` function in the Hippogriff library, `hippogriff_init()` sends an initialization command to pre-allocate a pinned memory region in this pinbuffer device. The later function calls to the Hippogriff library will request pinned memory address from this virtual device to reuse the pinned GPU memory and avoids the overhead of performing the time-consuming pinning operation.

In terms of the NVMe driver, we extended the driver to support special ioctl read and write commands for Hippogriff. The `hippogriff_send` function and the `hippogriff_recv` function issue these special read and write commands to the NVMe driver using the ioctl system call. Upon receiving these commands, the device driver will start creating the direct data access channel between the SSD and the GPU using

NVMe commands. The extended NVMe driver treats these special read/write commands as conventional read/write request except that the driver maps the DMA memory to the GPU memory. For each special command, the NVMe driver translates these requests into standard NVMe commands, but uses the DMA addresses mapping to GPU device memory in the physical memory pages (PRP) list of the NVMe command. After the driver sends out the NVMe command, Hippogriff does not require the CPU for this data transfer. The SSD accepting the NVMe command will directly send data to the memory addresses in the PRP list without CPU's intervention.

The current NVIDIA GPU limits the size of PCI BAR memory. For example, the Tesla K20 card can only pin at most a 254MB memory region in a PCI BAR. Therefore, the Hippogriff library partitions the data into chunks that fits in the pinned memory region and issues read/write requests for each chunk of data. Then, the Hippogriff library takes the advantage of the cheap device to device memory copy (less than 5 $\mu$s in our tests) to put or pull the data to or from the global memory of the GPU device. For example, the `hippogriff_send` function will first allocate a GPU memory region (or use the memory address for the argument, if the pointer is not NULL), and copies the data from the pinned memory to the memory region once Hippogriff finishes moving a chunk of data.

Although the current Hippogriff implementation largely relies on NVIDIA's GPUDirect, the system designer can apply the same concept to other GPGPU platform as the manufacturer of the GPU provides technology similar to GPUDirect. The designer can also use the concept of Hippogriff to enable the SSD to exchange data with other accelerators in the system.

## 3.3 Experimental Methodology

This section describes our test bed, benchmark applications, and the process of evaluating these applications.

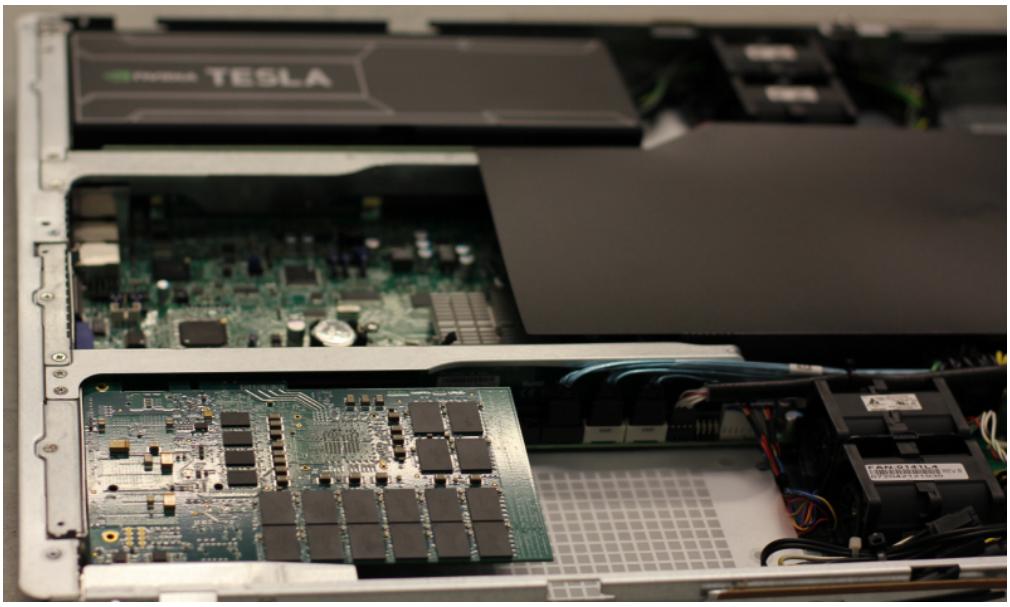



**Figure 3.6.** The test bed for Hippogriff evaluation.

### 3.3.1 Experimental Platform

The testing platform is a machine running Linux 3.16.3 on a 4-core Intel Xeon E5-2609V2 processor with 64GB DRAM. The GPU is an NVIDIA Tesla K20 card with 2496 CUDA cores and 5GB GDDR5 memory. It connects to the rest of the components through 16-lane PCIe interconnect, providing 8 GB/s I/O bandwidth. The card BIOS allows at most 192 MB of device memory to be mapped to the PCIe BARs. We use an SSD with 768GB SLC chips and a PMC-Sierra controller supporting NVMe 1.1 commands [104]. The SSD uses 4 PCIe 3.0 lanes to provide 4GB/sec bandwidth, and can sustain 2.2 GB/s for both read and write. Figure 3.6 shows our test bed.

The system uses the default "performance" governor in the Linux Intel CPU driver as the power management policy. It dynamically optimizes the frequencies of processor cores from 1.2 GHz to 2.5 GHz. To measure the power consumption, we use the Wattsup power meter to read the total system power every second. The test system consumes 117.6 W when idle.

**Table 3.2.** The applications and the input data sizes.

| | Input Size | | |
|---|---|---|---|
| Application Name | Small | Medium | Large |
| Breadth-First Search (BFS) | 37 MB | 587 MB | 1.17 GB |
| Computational Fluid Dynamics (CFD) | 16 MB | 570 MB | 713 MB |
| 2D Discrete Wavelet Transform (DWT2D) | 7 MB | 27 MB | 106 MB |
| Gaussian Elimination (Gaussian) | 67 MB | 268 MB | 1.07 GB |
| HotSpot | 34 MB | 537 MB | 2.15 GB |
| Hybrid Sort | 40 MB | 200 MB | 1.2 GB |
| Kmeans | 41 MB | 136 MB | 1.36 GB |
| LU Decomposition (LUD) | 17 MB | 268 MB | 1.07 GB |
| k-Nearest Neighbors (NN) | 71 MB | 570 MB | 1.14 GB |
| GPMR-IntCount | 128 MB | 256 MB | 512 MB |
| GPMR-Kmeans | 128 MB | 512 MB | 1 GB |
| GPMR-LinReg (Linear Regression) | 128 MB | 512 MB | 1 GB |
| GPMR-MM (Matrix Multiplication) | 128 MB | 512 MB | 2 GB |

### 3.3.2 Benchmarks

We select 9 data-movement heavy CUDA applications from the Rodinia benchmark suite [112]. They accept files as inputs, and spend 55% of execution time moving data on average. We store the input files in the binary format to eliminate the overhead of ASCII to binary translation. For each benchmark, we generate three different (small, medium, and large) sizes of input data, and rewrite the data movement part of these applications with `hippogriff_send()` and `hippogriff_recv()`, and eliminate unnecessary main memory allocations. To create multi-program workloads from Rodinia benchmarks, we slightly modify the code to allocate the required GPU memory before the GPU kernels. We also include GPMR [123], a GPU MapReduce framework, and modify only the file I/O code. We run 4 workloads—IntCount, K-Means, Linear Regression and Matrix Multiplication. Table 3.2 lists the applications and the sizes of input data in our experiments.

### 3.3.3 Benchmark Characteristics

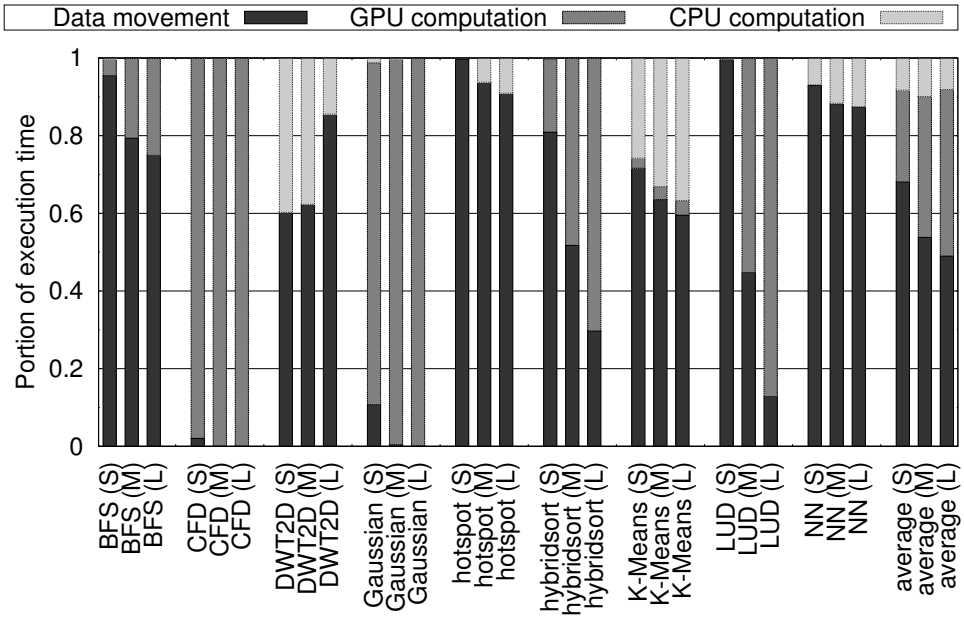



**Figure 3.7.** The execution time breakdown of Rodinia applications.

Figure 3.7 quantifies the significance of data movement in the 9 applications of the Rodinia benchmark suite. It breaks down the execution time into three different parts, and "Data movement" includes buffer allocation, data transfer, and SSD access. Our experiments show that on average, applications spend about 68%, 54% and 49% of execution time in moving data with small, medium and large input data sets, respectively.

Figure 3.8 breaks down the energy consumption into "idle energy" and "application energy". We calculate the application energy by subtracting system idle energy from the total energy consumption. The experiments show that these applications spend 94%, 87% and 85% energy in idle power for small, medium and large input sets, respectively.

Figure 3.9 breaks down compute and data movement energy usage in the same workloads. We only demonstrate the result when these applications use the large data set. Even with the large data set that data movement accounts for the least amount of execution among the three input data sets, applications still spend 48% of total system

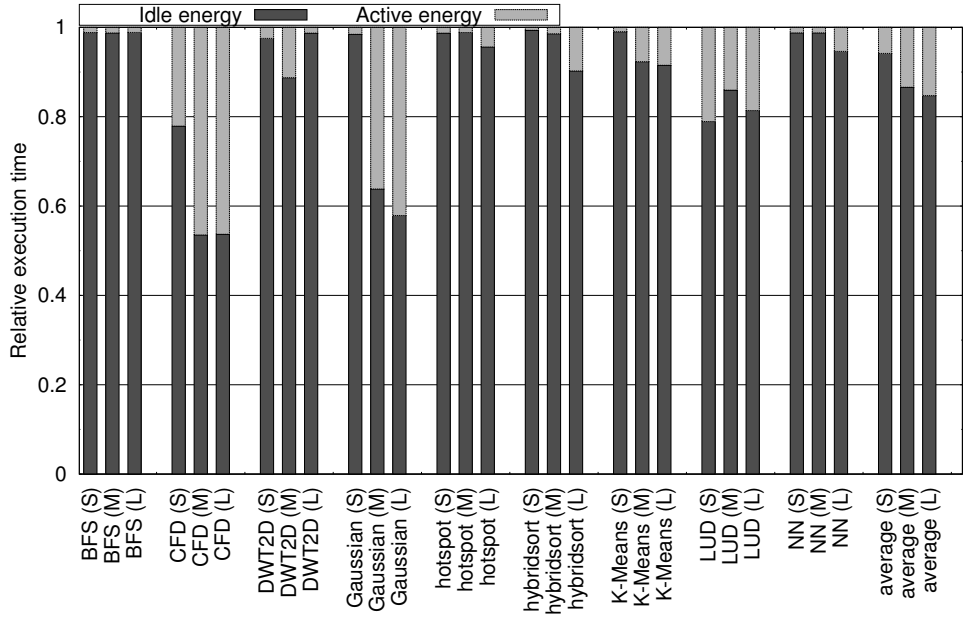



**Figure 3.8.** The energy active/idle breakdown of Rodinia applications.

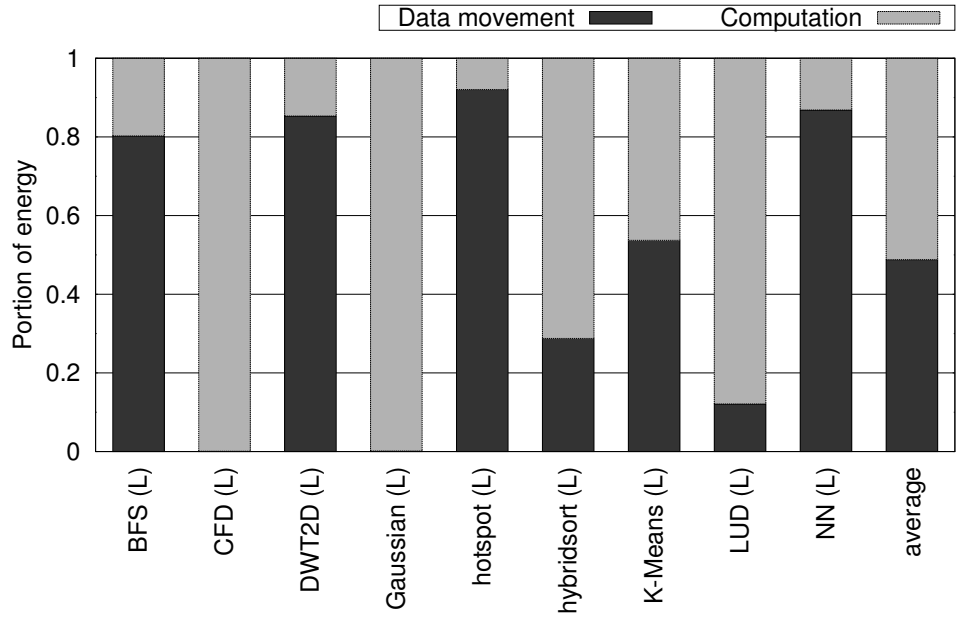



**Figure 3.9.** The compute/I/O energy breakdown of Rodinia applications.

energy in moving data.

### 3.3.4 Test Method

To compare Hippogriff with the standard CPU-centric methods, we need to integrate Hippogriff into test applications. Given a test application, we replace the traditional I/O and data movement commands with `hippogriff_send()` and/or `hippogriff_recv()`. To produce multi-program workloads from the Rodinia benchmark suite, we run them concurrently and modify the applications to ensure that the GPU kernels have sufficient input data and output memory space to complete the GPU tasks without blocking.

We then measure the end-to-end latency of applications, thus the timer starts from the launching of an application until the completion of the whole program. Before each test, we clean the operating system page caches to have accurate measurement results.

## 3.4 Results

Hippogriff improves the application performance by dynamically choosing the best data transfer route. This section first compares the two route options in Hippogriff – the conventional CPU-centric NVMe and NVMeDirect, and then demonstrates the effectiveness and trade-offs of using NVMeDirect and how Hippogriff balances these trade-offs.

### 3.4.1 Comparing NVMeDirect and conventional NVMe

As NVMeDirect removes the CPU and the main memory from the data transfer path, we expect smaller runtime overhead on the host side. To compare the runtime overhead for each request using the conventional model and NVMeDirect, we examined the number of CPU instructions and latency for read requests varying from 4 MB (the smallest input file is 7 MB) to 32 MB (the maximum data size each I/O command accepts in our SSD).

Figure 3.10 and Figure 3.11 illustrates the experimental results. We break down

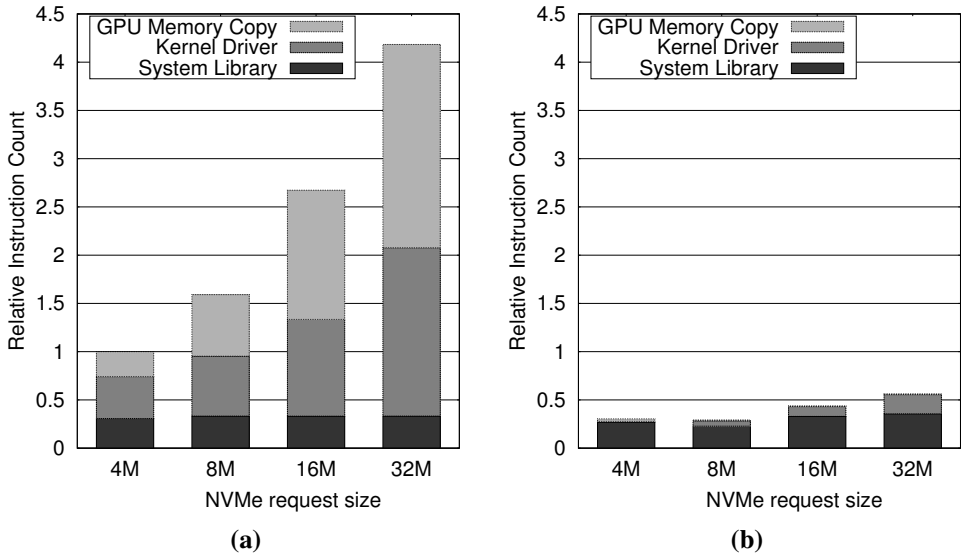



**Figure 3.10.** The instruction count breakdown for moving data from an SSD to a GPU using (a) NVMe read and (b) NVMeDirect read commands.

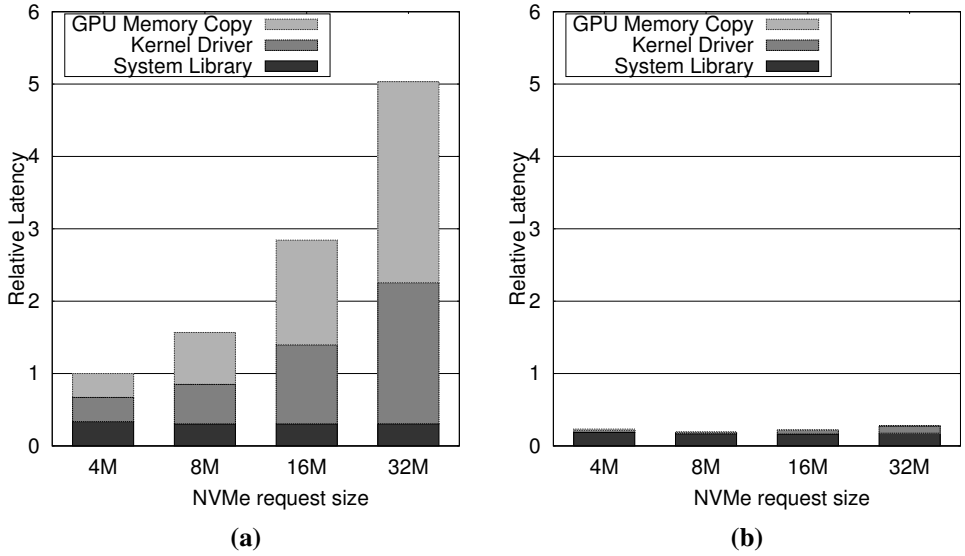



**Figure 3.11.** The latency breakdown for moving data from an SSD to a GPU using (a) NVMe read and (b) NVMeDirect read commands.

the CPU instructions and latency into three parts: “System library” covers the operations including opening file descriptors and calculating the LBAs. “Kernel driver” represents

the operations that require the processor to set up DMA and interact with the SSD. “GPU Memory Copy” includes the run-time system costs of moving data from the DRAM to the GPU. We exclude the overheads of DRAM and GPU memory allocation in these tests as they can be amortized over multiple runs. We use 4MB in conventional NVMe as the baseline in these graphs.

The conventional mechanism devotes 25%–51% of its CPU instructions to moving data from the DRAM to the GPU. This extra runtime overhead accounts for 33%–51% of the latency.

On the other hand, as NVMeDirect bypasses the CPU and the main memory when moving data to the GPU, it can eliminate the “GPU Memory Copy” overhead. NVMeDirect also avoids the overhead of setting up DMA to the host DRAM in the kernel driver. Therefore, NVMeDirect spends very few CPU instructions in GPU runtime and the kernel driver.

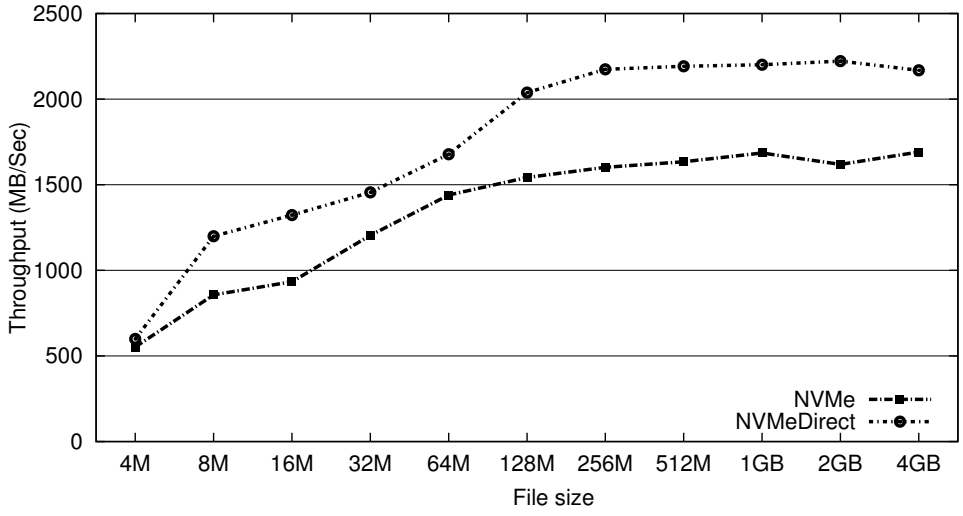



**Figure 3.12.** The throughput comparison of moving data from an SSD to a GPU.

Figure 3.12 compares the throughput of moving different sizes of file data from the SSD to the GPU with NVMeDirect against conventional NVMe. The results here exclude the overhead of allocating all necessary resources (e.g. memory buffers) along

the data paths. The file sizes are up to 4GB due to limitation of the GPU memory size.

The performance advantage of NVMeDirect becomes more significant as file size increases. When transferring a 4GB file, NVMeDirect offers up to 2221 MB/s bandwidth, while NVMe can only achieve a throughput of 1691 MB/s, which is 34% lower than NVMeDirect.

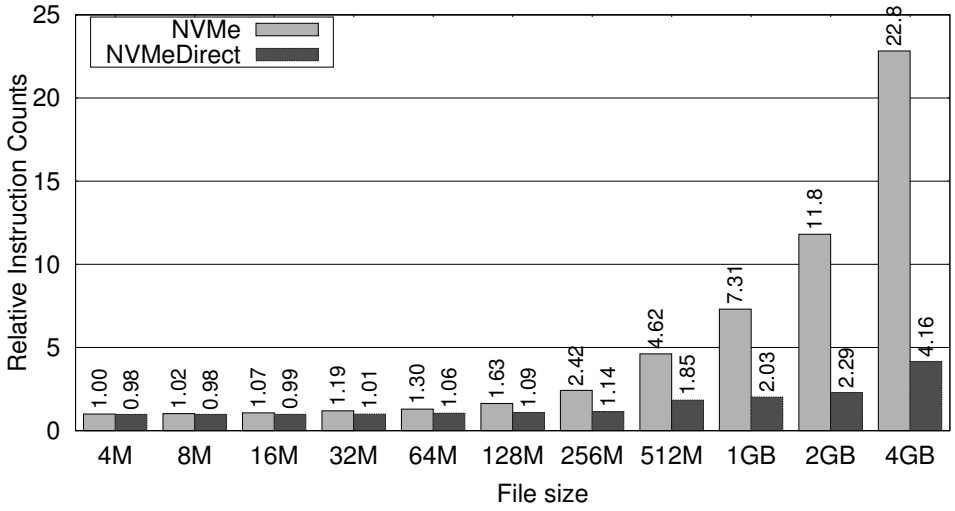



**Figure 3.13.** The comparison of number of CPU instructions when moving data from an SSD to a GPU.

To present the CPU overheads that different data movement mechanisms consume, Figure 3.13 depicts the number of CPU instructions given in the experiments in Figure 3.12. Conventional NVMe require the GPU runtime system to move data between DRAM and GPU memory, so they require more CPU instructions than NVMeDirect. To transfer a 4GB file, conventional NVMe uses 5.5$\times$ the instructions used by NVMeDirect.

### 3.4.2 Impact on single process workload

To find out the impact of Hippogriff for a single GPU application, we measure the end-to-end latency for them. In this case, Hippogriff always uses NVMeDirect to carry data as the application can have all the system resources.

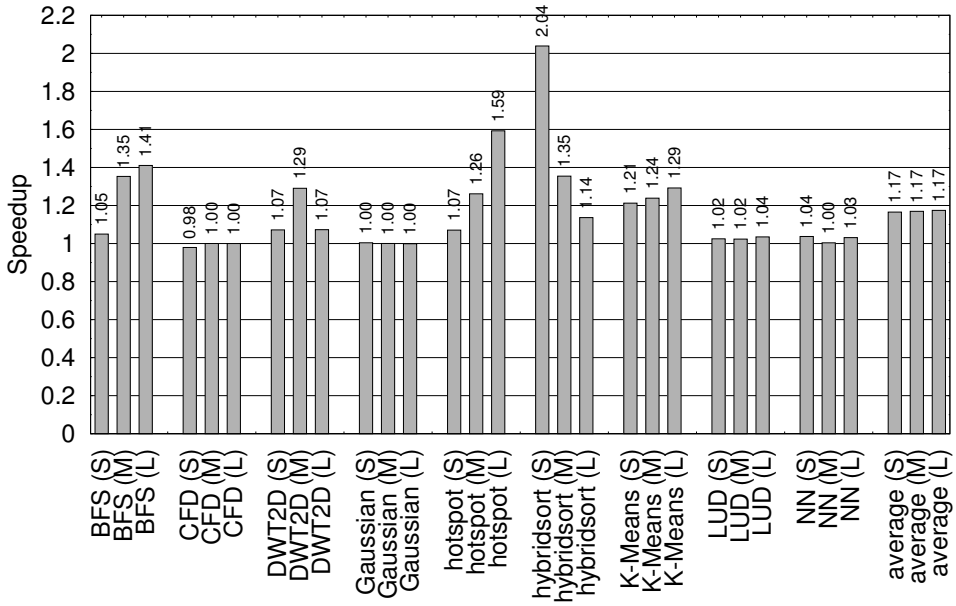



**Figure 3.14.** The speedup of applications using Hippogriff.

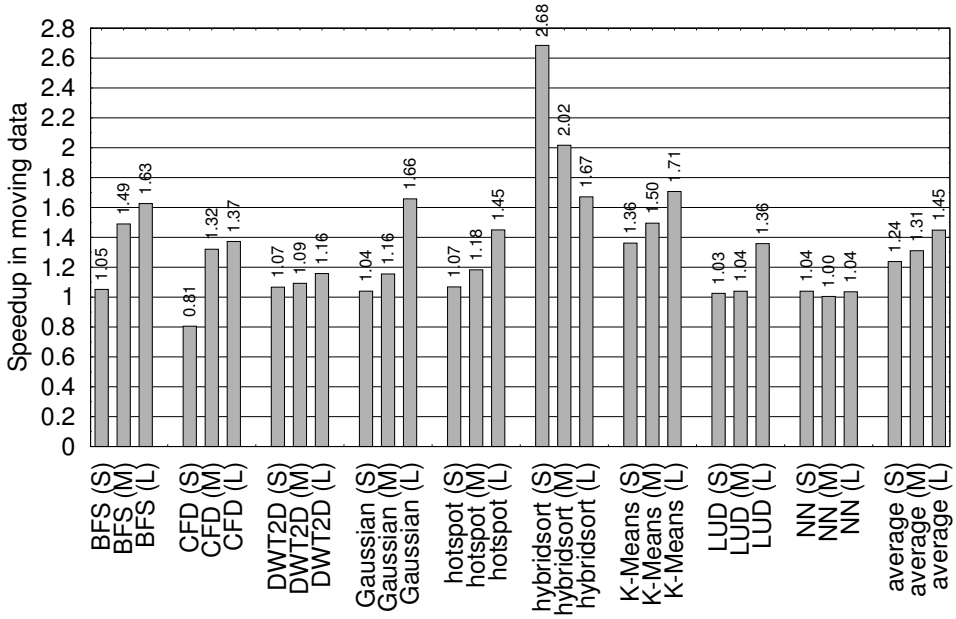



**Figure 3.15.** The speedup of data movement in applications using Hippogriff.

Figure 3.14 illustrates the overall speedup of using Hippogriff on our benchmarks. Hippogriff achieves an average speedup of 1.17× for all the input sets. If only consider the data transfer, Hippogriff speeds up the data movement by 1.45× for large inputs,

1.31× for medium, and 1.24× for small, as shown in Figure 3.15. In most cases, the effectiveness of Hippogriff becomes more obvious as the data set size increases, since larger input files help amortizing the initialization costs. Hippogriff also reduces the number of page faults by 21% for the small data sets, 26% for the medium, and 32% for the large.

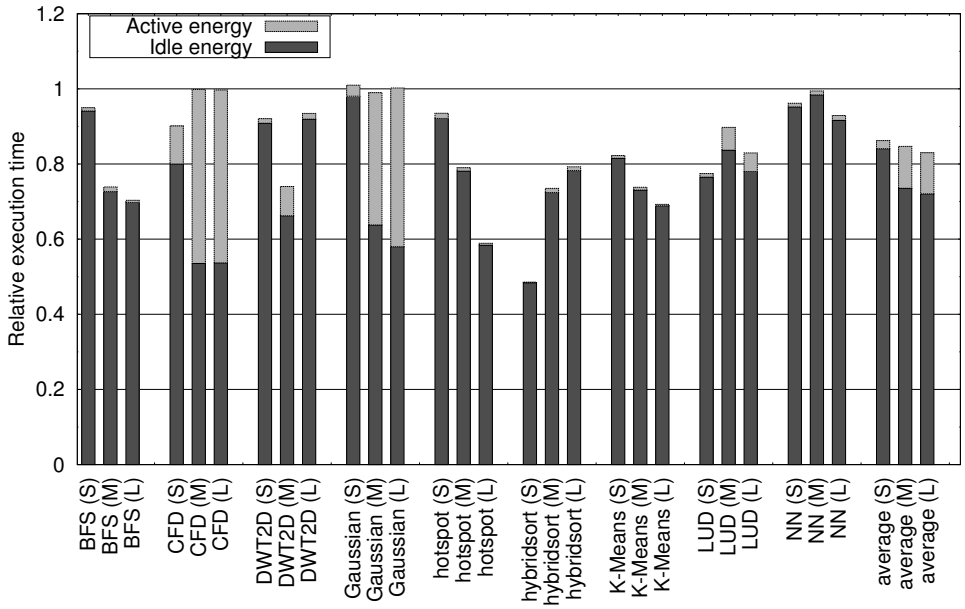



**Figure 3.16.** The relative energy consumption of applications using NVMeDirect.

### Energy and Power

Hippogriff reduces the load on CPUs during the data transfer, enabling processor cores to operate at lower clock rates or perform more useful work. To explore the potential energy and power savings using Hippogriff, we measure the total system power, and calculate the energy usage during the entire running time of these applications.

Figure 3.16 presents the relative total system energy consumption of our benchmark applications using Hippogriff against the NVMe baseline. Hippogriff can reduce total energy consumption by 17%, 15%, and 14% when running large, medium, and small input data sets, respectively. This figure also breaks down energy consumption

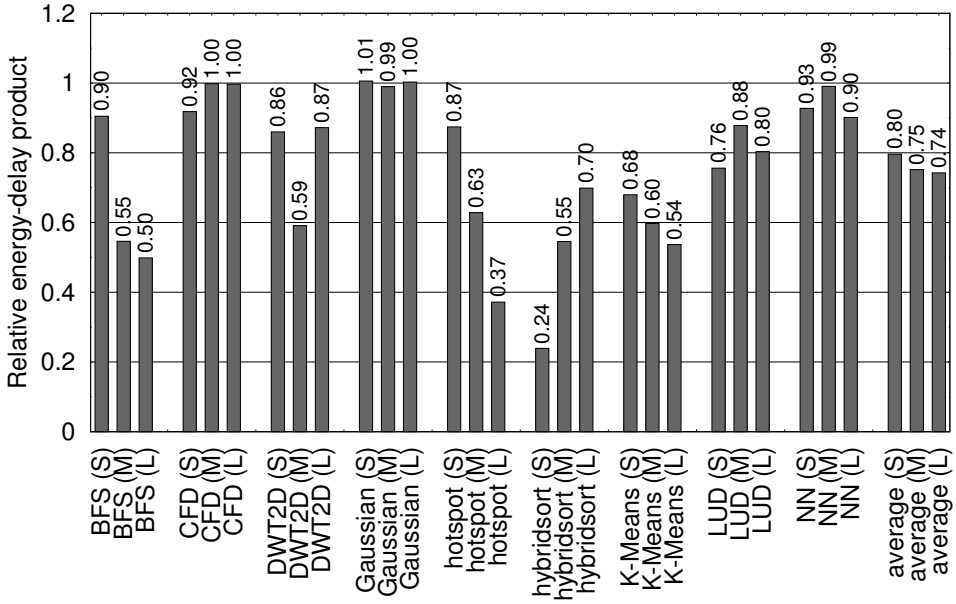



**Figure 3.17.** The relative energy–delay product of applications using NVMeDirect.

into idle energy and application energy. Since Hippogriff reduces the execution time of programs, we observe 13% reduction in idle energy. Even with an Intel CPU driver that optimizes for energy efficiency, Hippogriff still reduces total system power consumption by up to 8% and saves 30% of application energy over the baseline.

Figure 3.17 reports the energy–delay product. Hippogriff improves the energy–delay of applications by 26%, 25% and 20% for large, medium and small data sets, respectively.

**Hippogriff and Lower-End CPUs**

Hippogriff frees the CPU from transferring data between the SSD and GPU. Therefore, the system can use a simpler and more energy-efficient CPU if most computation is done on the GPU. This section explores the potential of using Hippogriff in servers with lower-end CPUs.

For this experiment, we lower the CPU frequency in our test bed to 1.2GHz, the lowest clock rate that the processor supports. Figure 3.18 examines the performance of

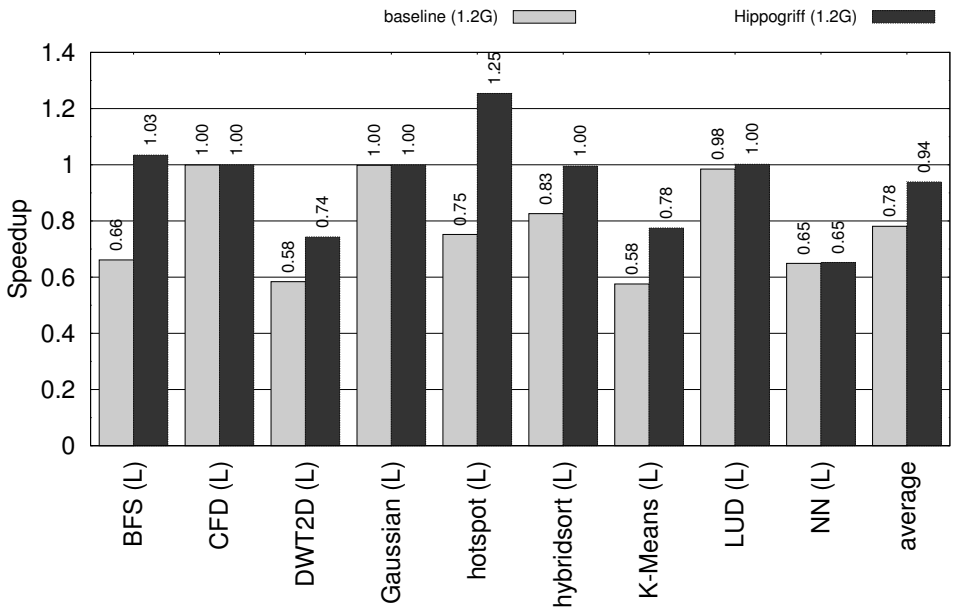



**Figure 3.18.** The speedup comparison of baseline 1.2GHz and Hippogriff 1.2GHz.

the baseline system running at 1.2 GHz ("baseline(1.2G)") and Hippogriff running at 1.2 GHz ("Hippogriff(1.2G)"). We compare these two against the baseline running at 2.5 GHz.

Lowering the CPU frequency to 1.2 GHz results in a 22% slowdown for baseline(1.2G), since the 52% slower clock rate increases the CPU computation time by 192% and data movement time by 54%.

Using Hippogriff, applications also see the increase in CPU computation time. But Hippogriff can reduce the data movement time over baseline running at 2.5 GHz by 27%. As a result, Hippogriff(1.2G) only observes 6% performance loss. In this lower-end server setup, Hippogriff shows more performance advantages. If we use execution time of baseline(1.2G) as the baseline, the Hippogriff(1.2G) achieves 1.2× speedup.

Figure 3.19 shows the relative energy consumption of the above three configurations against baseline running at 2.5 GHz. Hippogriff(1.2G) can reduce energy consumption by 4%. On the contrary, baseline(1.2G) consumes 18% more energy in total

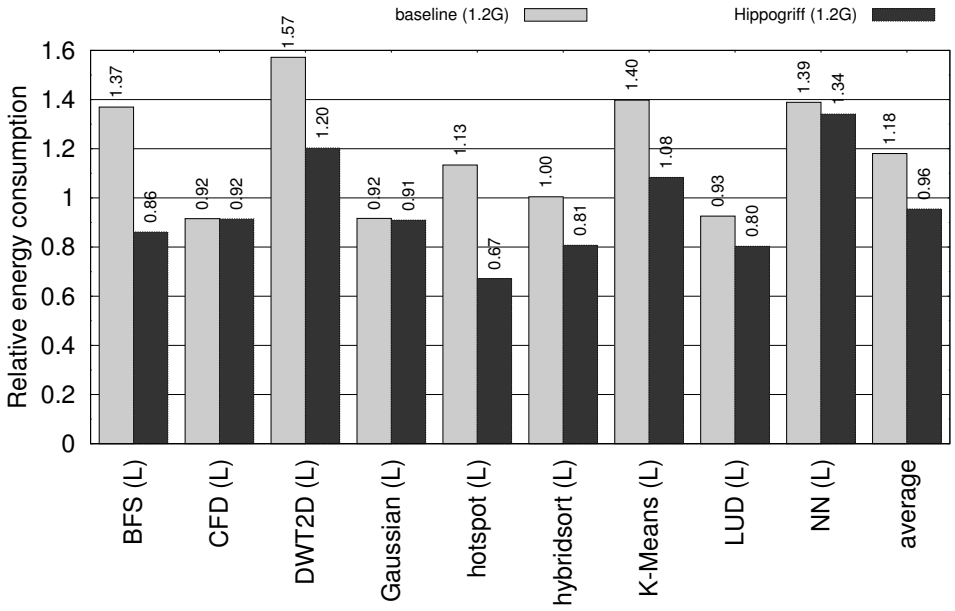



**Figure 3.19.** The relative energy consumption comparison of baseline 1.2GHz and Hippogriff 1.2GHz.

due to the increased execution time.

### 3.4.3 Multi-program workload

To demonstrate the effectiveness of the Hippogriff runtime in a multi-program environment, we compare Hippogriff and a specialized Hippogriff that always uses NVMeDirect with the baseline. We investigated rodinia benchmark applications and GPMR applications. Figure 3.20 shows the performance results. For all workloads that we examined, Hippogriff achieves average speedups of 1.25×, 1.23× and 1.15× for large, medium and small datasets, respectively.

#### Rodinia Benchmark

To create multiprogram workloads, we submit 8 homogeneous tasks simultaneously with the large, medium or small input data sets. We use the unmodified applications as the baseline and measure the end-to-end latency.

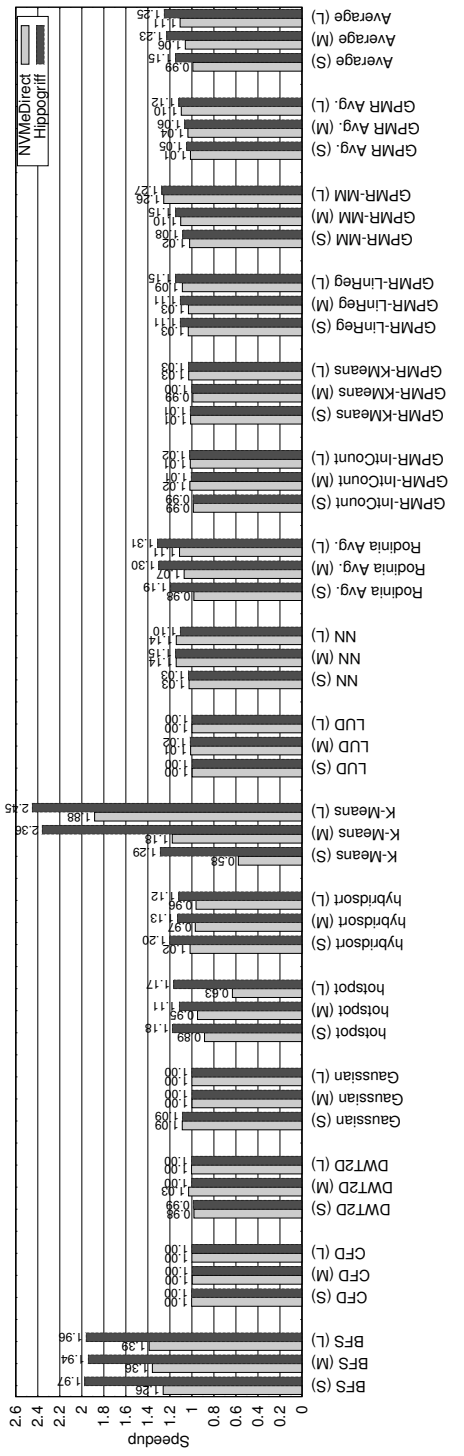



**Figure 3.20.** The speedup of Hippogriff under multiprogram workload.

Hippogriff achieves average speedups of 1.31×, 1.30× and 1.19× for large, medium and small datasets, rather than only 11%, 7% and -2% if always using NVMeDirect. Although NVMeDirect can bypass the CPU and the main memory to reduce the latency of a single task, when multiple tasks are running and one of them uses up all the GPU memory, NVMeDirect cannot initiate data transfer until the GPU task completes. In the baseline, however, the applications can send I/O data to the main memory before the GPU resource becomes available, to better utilize the I/O bandwidth.

For applications that suffers slow down with NVMeDirect only, Hippogriff can use conventional NVMe to overlap SSD I/O with GPU computation and saturate the SSD I/O bandwidth. Therefore, Hippogriff does not incur any slowdown in applications. For workloads that NVMeDirect provides speedup, Hippogriff can further improve the performance by up to 2.45×.

**GPU-MapReduce workload**

To understand the impact of NVMeDirect in highly-optimized GPU server workloads, we tailor the GPU MapReduce framework GPMR [123] to use NVMeDirect for data transfers between the SSD and the GPU. For each workload, we run 4 processes in parallel.

Even though the baseline GPMR aggressively reduces the file I/O operations and overlaps the data transfer with the GPU computation, Hippogriff can still improve the performance by up to 27%, as shown in Figure 3.20.

All GPMR workloads but Matrix Multiplication contain relatively fewer MapReduce tasks and file I/O operations. As a result, Hippogriff achieves limited performance gain. For Matrix Multiplication, the workload spawns thousands of tasks and each task reads relatively larger chunks from the input files. This allows Hippogriff to achieve 1.27× speedup.

## 3.5 Related Work

Hippogriff improves the performance of heterogeneous systems by optimizing data transfers. Most previous works considering the importance of data transfer focus on the bottleneck between CPU and GPU [54, 74, 47]. Several works [92, 87, 120, 97] also propose to mitigate the CPU–GPU bottleneck using runtime scheduling and pipelining to overlap data transfer and computation. Hippogriff leverages these efforts, and automatically select the most efficient data transfer paths.

The NVMeDirect component in Hippogriff provides direct data transfers between commercially available NVMe SSDs and GPUs, similar to Donard [121] and NVMMU [139]. However, the biggest difference lies in that, besides the direct data transfer, Hippogriff explores the benefit of choosing the best data path in the runtime, while both Donard and NVMMU might suffer from the severe performance issue as we demonstrate when using NVMeDirect alone in multi-program workloads. GPUDrive [116] provides similar functionality, but with a customized PCIe switch to access SATA SSDs. Hippogriff, NVMMU and GPUDrive leverages AMD's DirectGMA or NVIDIA's GPUDirect to program GPU memory addresses in PCIe BARs [33, 100]. Existing works focusing on providing peer-to-peer communication to improve inter-node communication within GPU clusters [130, 41, 82] or intra-node communication between GPUs or other devices [105, 102, 129, 51, 88].

Hippogriff reduces the importance of CPU performance in heterogeneous computing platforms. The result of Hippogriff encourages researchers to revisit the design of server architectures [46]. Several server systems including FAWN [43], Gordon [53] and Blade [90] also prove the concept of using low-end CPUs for servers running data-intensive applications. Our experiments show that Hippogriff is especially effective for GPU servers using low-end CPUs.

Heterogeneous System Architecture (HSA) [24] provides another approach for direct data transfer. HSA integrates the CPU and GPU on the same chip and uses a unified virtual address space to share the main memory. Several research papers demonstrate the potential benefits of HSA in analytical workloads and database operations [60, 135, 78]. However, with the limitation in the shared main memory bandwidth and the power constraint from dark silicon problem [68, 126], such integrated GPUs [40, 39, 99] can only accommodate less than 20% of the streaming units and so deliver only moderate performance compared with high-end, discrete GPUs [39, 99]. In fact, due to their streaming nature [106], many GPGPU applications require higher memory bandwidth and have different memory access patterns than most CPU applications. Hippogriff will help systems with discrete GPUs deliver better performance and energy efficiency on data-intensive and GPU-intensive applications, by significantly reducing the data transfer overhead as well as capitalizing on the superior internal GPU memory bandwidth.

## 3.6 Summary

The widespread use of accelerators, high-speed storage devices and peripherals in heterogeneous computing platforms calls for more advanced intra-computer interconnect as well as a thorough rethinking of the data transfer model.

This paper presents Hippogriff, a system that efficiently handles data transfers in heterogeneous systems. As the Hippogriff API needs only the source and destination for the data transfer, Hippogriff makes the best choice regarding data routes. We also implement NVMeDirect to provides a direct, peer-to-peer data transfer mechanism inside Hippogriff.

With NVMeDirect bypassing the CPU and main memory, Hippogriff brings 1.17$\times$ speedup for single program applications, and reduces the energy consumption by 14%–17% while improving the energy-delay by 20%–26% for different data sizes.

Hippogriff is especially effective in servers with low-end CPUs, and can speedup applications by 1.2×. With Hippogriff optimizing the data transfers by choosing the proper data route, we demonstrated an average speedup of 1.15×–1.25× for multi-program GPU workloads.

## Acknowledgements

This chapter contains material from "Hippogriff: Efficiently Moving Data in Heterogeneous Computing Systems" by Yang Liu, Hung-Wei Tseng, Mark Gahagan, Jing Li, Yanqin Jin and Steven Swanson, which appears in *ICCD '16: 2016 IEEE International Conference on Computer Design*. The dissertation author was the first investigator and author of this paper.

# Chapter 4

# Boosting the Utilization of Heterogeneous System Resources

The previous chapter describes how to provide efficient data transfer in heterogeneous computing systems. However, that is the first step towards efficient heterogeneous computing. This chapter will further introduces how to improve the system performance and efficiency by boosting the utilization of system resources (e.g., the GPU and the SSD), in the context of MapReduce.

To tackle data-intensive computing problems, the conventional MapReduce resorts to scaling out for parallelism [61, 6]. Recent research [45, 115, 96] reveals such parallelism gained by adding nodes comes at the cost of efficiency, while a scale-up configuration can achieve comparable or better performance by adding resources to a single node, given the modest dataset sizes of many analytics jobs [66, 111, 42].

An attractive way to scale up MapReduce is to utilize the massive parallelism and the high inner-bandwidth of GPUs. Existing work demonstrates that GPU MapReduce frameworks can achieve up to 160$\times$ the performance on a single GPU compared to their CPU counterparts [123].

However, there is one critical piece still missing: handling I/O efficiently. While mainly focusing on offloading computation from CPUs to GPUs, most GPU MapReduce

systems neglect the impact of the bandwidth mismatch between I/O devices and the GPU.

In addition, lack of efficient I/O handling and scheduling may lead to underutilization of system resources. For example, a GPU task will have to wait until its input is ready before its computation on the GPU starts. As a result, when the I/O device is busy with transferring the data, the GPU might be idle. Simply increasing the number of processes/tasks cannot solve this problem, and might even cause resource contention [93] among processes/tasks if without proper coordinating. In fact, our profiling of a state-of-the-art GPU MapReduce framework reveals that, even with multiple processes, the GPU utilization can be lower than 20% while the I/O device is idle for almost 50% of the time. This inefficiency will undermine the potential performance gains from scaling up MapReduce with GPUs.

In this chapter, we present *SPMario*, a GPU framework that scales up MapReduce with optimized I/O handling and task scheduling, to address the above problems. SPMario runtime includes a scheduler and multiple containers. The scheduler dispatches tasks to containers, while the containers provide the execution environment for the tasks and enforce the execution order. To further improve the performance and resource utilization, SPMario proposes *I/O Oriented Scheduling* to coordinate task execution in a way that minimizes idle time of the resources, and pipelines the I/O operations, the data transfers between the host and the GPU, and the GPU kernel execution.

We implement the SPMario prototype on a server with an NVIDIA Tesla K20 GPU and a high-end PCIe SSD. Our experiment results show that for the single job cases, SPMario can achieve up to a 2.28× speedup over the baseline in job execution time, and yield up to 2.12× GPU utilization and 2.51× I/O utilization. When scheduling two jobs together, SPMario with I/O Oriented Scheduling outperforms with round-robin by up to 13.54% in execution time, and 12.27% and 14.92% in GPU and I/O utilization, respectively.

The remainder of the chapter is organized as follows. Section 4.1 introduces the background and discusses the problem of I/O handling and resource underutilization. Section 4.2 describes the design and implementation of SPMario. We then introduce the I/O Oriented Scheduling algorithm in Section 4.3, and evaluates the SPMario prototype in Section 4.4. Section 4.5 discusses the related work. Finally, we summarize this chapter in Section 4.6.

## 4.1 Background and Motivation

This section provides a brief overview of the GPU and MapReduce, as well as GPU MapReduce. We then discuss the problem of I/O handling and resource underutilization.

### 4.1.1 The GPU and MapReduce

GPUs have gained sustained popularity in general purpose computation for their massive parallelism. The GPU is a processor comprised of multiple SIMD streaming multiprocessors, or SMs. It organizes threads into thread blocks that execute functions called kernels, and the current hardware cannot preempt a running kernel. GPU programming frameworks such as CUDA [23] and OpenCL [7] ease the process of accelerating the original serial programs.

MapReduce [61] represents another form of parallelism to process data on a large scale. To write a MapReduce application (a "job", users just need to provide two primitive functions called *Map* and *Reduce*. The MapReduce framework will apply the Map function on input data blocks ("chunks"), and produce a list of intermediate key-value pairs. The Reduce function produces the results with these pairs.

Given the parallelism possessed by both GPUs and MapReduce, it seems natural to pursue the possibility of scaling up MapReduce systems with GPUs. For one thing,

by offloading the computation to the GPU, a single machine can drastically boost its performance. For another, the MapReduce programming model and interface can simplify GPU programming, allowing users to focus on the application logic.

### 4.1.2 I/O Handling and Resource Underutilization

While many efforts are made to improve the performance of GPU MapReduce systems [77, 79, 56, 85, 52, 50, 55, 58, 123, 86, 133] and to employ GPUs in Hadoop [70, 124, 137, 69, 32, 117], there are two important problems neglected or not well addressed by the existing solutions: the efficient I/O handling and the resource utilization.

Because the current programming model deems the GPU as a co-processor [118], most existing GPU MapReduce frameworks do not consider I/O handling. They either assume the entire datasets loaded into the host DRAM already, or leave the hassle of handling I/O to the operating system. Thus, the existing scale-up GPU MapReduce frameworks have limited scalability, and cannot handle large datasets.

In addition, few GPU MapReduce solutions aim to pursue better utilization of system resources. Indeed, it is admittedly difficult to achieve good occupancy and full utilization of the GPU with user-provided kernels, while the lack of kernel preemption prevents efficient multitasking. Thus, many systems leave individual GPUs underutilized[1]. For example, the GPU and I/O utilization are only 19.01% and 36.25%, respectively, when we run a $16K \times 16K$ matrix multiplication (MM16K) with the default single worker process in GPMR [123], an optimized GPU MapReduce library for both individual GPUs and clusters, as shown in Figure 4.1.

Could multi-tasking help solve this underutilization problem? Our *first observation* reveals that, **although simply raising the number of processes/tasks might**

[1] We define the GPU utilization as:

$$utilization = \frac{active\_time}{total\_time},$$

where *active_time* measures the accumulated time when SMs are running, and *total_time* is the total time span of the job execution. We also apply this definition to describing the I/O utilization in this chapter.

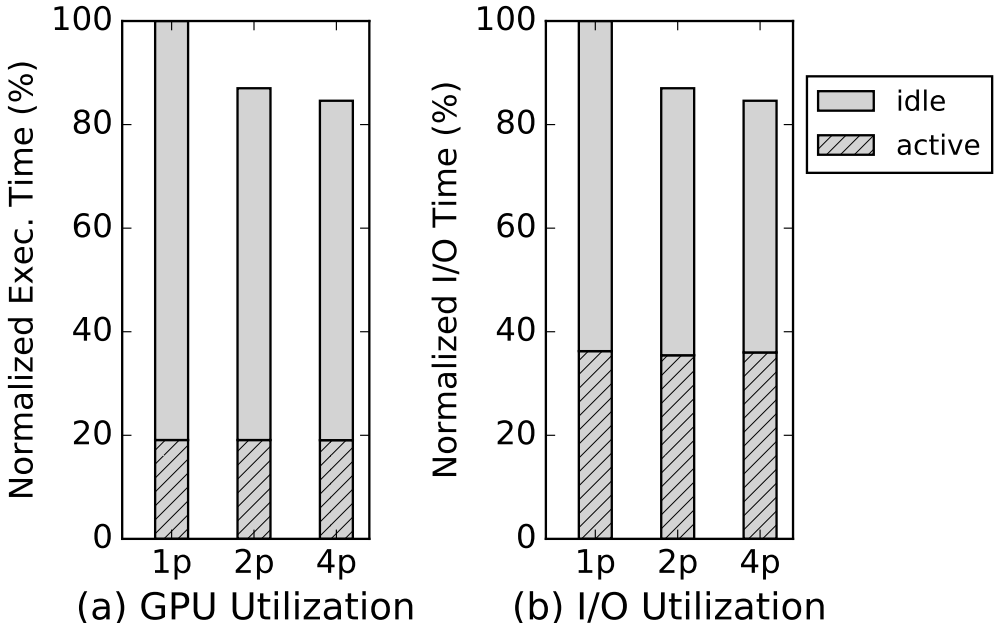



**Figure 4.1.** GPU and I/O utilization of MM16K for baseline GPMR. All data are normalized to the single process case (1p) for comparison. For 1 process (1p), the GPU and I/O utilization are only 19.01% and 36.25%, separately. Increasing the process number (from 1p to 4p) only brings 3.44% and 6.30% improvement on the GPU and I/O utilization, respectively, and reduces the execution time by 15.40%.

**marginally help, it could result in serious resource contention without proper coordination, which will in turn prevent further utilization and performance improvement**. Figure 4.1 shows the utilization of the GPU and the I/O can only improve by 3.44% and 6.30%, respectively, and the total execution time only decreases by 15.40% in MM16K, when increasing the number of processes from 1 to 4 for the same amount of the job. At the same time, however, the average per-chunk (64MB here) I/O access time increases drastically from 29.4ms to 102.8ms, as shown in Figure 4.2. The substantially increased I/O time will also lengthen the task's completion time.

In fact, the problem of resource contention is not unique in GPU MapReduce [93], yet it becomes deteriorated in a GPU scale-up configuration, because **MapReduce tasks**

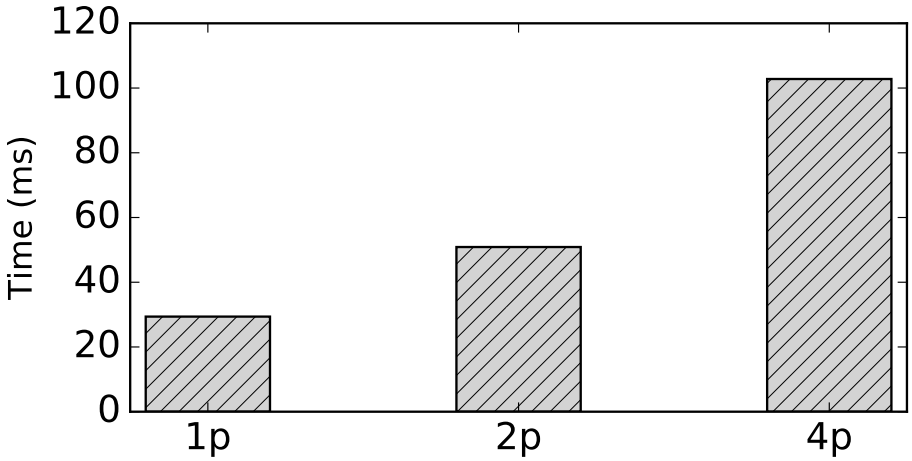



**Figure 4.2.** Avg. per-chunk I/O time of MM16K for baseline GPMR. For the same amount of job, the average per-chunk (64MB) I/O time drastically increases from 29.4ms to 102.8ms with the increase in the number of processes (from 1p to 4p), since these concurrent processes tend to compete for I/O during the same period of the time.

**come in waves** [136], and **concurrent tasks tend to compete not only for the I/O resources, but also for the non-preempted GPU**. Some NVIDIA GPUs (like K20) do support running multiple kernels concurrently when possible [5]. But this best-effort feature cannot guarantee its effectiveness. Figure 4.1(a) shows even though the execution time does decrease with more processes, the GPU active time does not shrink accordingly.

Then how to coordinate these concurrent tasks to avoid contention effectively? Our *second observation* is that, **a task kernel cannot proceed before its input chunk is ready on the GPU**. Usually users have little control over which GPU kernel to run; it is the CUDA runtime and driver that determine the concrete kernel launch sequence. But by enforcing the order of supplying the input, we can accordingly control the kernel execution sequence. In addition, our *third observation* reveals that, **for typical GPU MapReduce tasks, a single chunk with the size of 64MB or 128MB is enough to saturate the I/O bandwidth, even for a PCIe SSD**. For example, in Figure 4.2, the read bandwidth for the single process is $\sim$2.17GB/s, and our PCIe SSD can sustain $\sim$2.2GB/s.

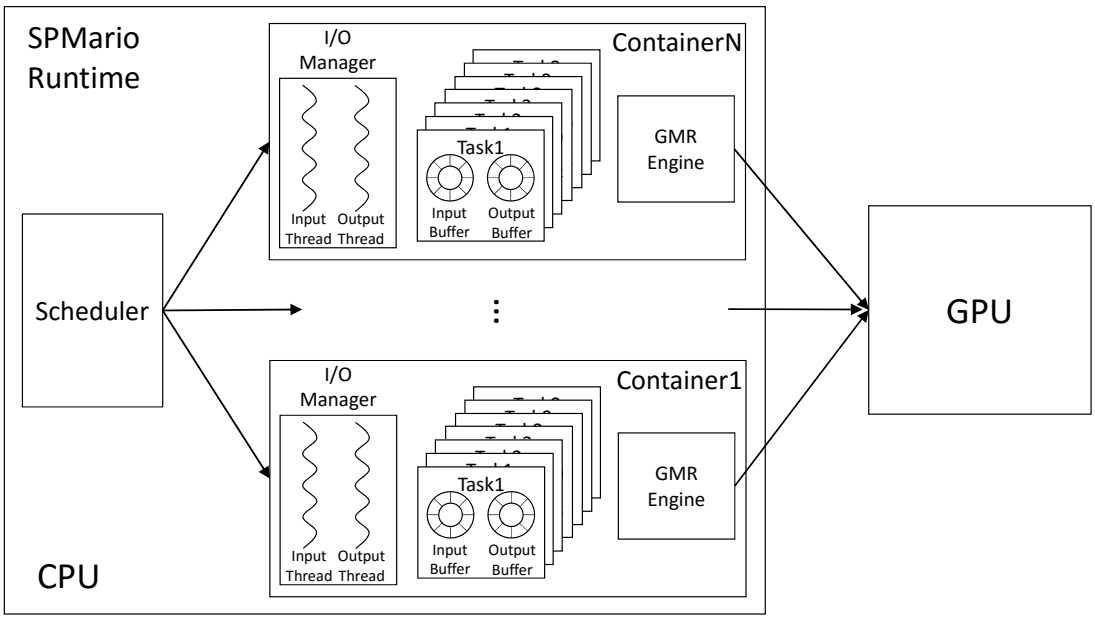



**Figure 4.3.** System architecture.

With the above three key observations, we design and implement a GPU MapReduce framework, SPMario, to improve the resource utilization and performance. By employing the proposed I/O Oriented Scheduling, SPMario can allow multiple tasks and jobs to share the GPU efficiently. We will use the term "process" and "task" interchangeably in this chapter, since a task runs in a process.

## 4.2 Design and Implementation

This section describes the design and implementation of our SPMario prototype. We first give an overview of the system architecture, and then introduce the key components and optimizations in detail.

### 4.2.1 System Overview

As depicted in Figure 4.3, SPMario includes a global *scheduler*, and one or more *containers*. The scheduler picks a container to monopolize the full I/O bandwidth within a

quota, and the chosen container decides the concrete task execution order. Each container has a single *I/O manager* to do I/Os for all its own tasks. The container collects task status, and enforces the task execution order by asking the I/O manager to supply input data to tasks following the decided order. After a data chunk is ready, the task process invokes the *GPU MapReduce Engine* inside that container to process the chunk. The SPMario runtime tracks the system resource usage including the I/O chunk buffers and the GPU device memory.

### 4.2.2 Scheduler

The centralized scheduler accepts jobs from users and dispatches tasks to different containers, based on job configurations and system resource availability. It can implement different scheduling policies like priority-based and round robin, by controlling the number of containers a job can have to run tasks and the I/O quota for each container. SPMario iteratively processes tasks, so it keeps one mapper and/or one reducer for each task in a container. Thus, having more containers for a job means higher degree of parallelism; and assigning a larger I/O quota to a container means longer run time within each scheduling cycle for the container.

The scheduler picks at most one container to run at any time. If the current container runs out of quota, the scheduler will schedule another container. The scheduler monitors the job execution and balances the system loads.

### 4.2.3 Container

A container, as shown in Figure 4.3, is analogous to a node in the scale-out configuration, and provides the execution environment for its tasks. A container accepts a list of tasks dispatched by the scheduler. The I/O manager inside each container serves I/Os to tasks. We mainly discuss the input I/O thread since the output is not on the critical

path.

Similar to other MapReduce implementations, SPMario reads inputs from SSDs into fixed-size chunks (default is 128MB and configurable). But in SPMario, it is the dedicated input thread inside the I/O manager that performs I/Os, while in other systems individual tasks fetch their own inputs independently. For example in Hadoop [6], each mapper employs its corresponding RecordReader method to fulfill I/O requests. In these systems, individual tasks are not aware of their peers' I/O behaviors. As a result, multiple I/O streams tend to compete for the I/O resource when running concurrently, leading to the resource contention problem [93].

The dedicated I/O thread in SPMario's I/O manager, on the other hand, allows scheduled tasks to take turns enjoying the full SSD bandwidth within the assigned I/O quota. This is the key to implementing the task scheduling in SPMario. By choosing the order of supplying input data, the container is able to enforce the execution order of tasks with the help of the I/O manager.

The I/O manager maintains two *chunk buffers* (for input and output) allocated on the pinned host DRAM for each *running* task. To achieve better performance, SPMario fully explores the opportunities to optimize data transfer. First, the input thread puts an input chunk into one of the available ring buffer slots, and immediately returns to perform the next I/O. If the ring buffer for the current task is full, the I/O thread will not block but seek to do I/Os for other tasks. The dedicated non-blocking I/O thread allows SPMario to overlap I/O operations with asynchronous GPU operations, so SPMario can be very efficient even when only one process is running.

Moreover, SPMario assigns a separate CUDA stream for each chunk in the chunk buffer, so multiple streams (even from different processes with the help of CUDA MPS [22]) can run in parallel, while still preserving the order between asynchronous memory copy and kernel execution inside each stream. SPMario further pipelines

operations including I/O, data transfer between the CPU and the GPU, and GPU kernel execution to reduce execution time.

### 4.2.4 GPU MapReduce Engine

After a task gets a chunk ready in its chunk buffer, it continues to invoke the GPU MapReduce Engine to process the chunk. We implement the GPU MapReduce Engine ("GMR Engine") based on GPMR [123].

Like GPMR, GMR Engine has four stages: *Map*, *Bin*, *Sort* and *Reduce*. In the Map stage, the mapper runs map kernels on inputs to emit key-value pairs, and allows doing Partial Reduction to reduce the data transfer and host DRAM usage. GMR Engine then partitions the key-value pairs into buckets, and sends them to the right reducer in the Bin stage, where a binner thread will collect and distribute the pairs. The sorter sorts the pairs, and the reducer produces outputs. The GMR Engine has a default I/O handler for simple file formats, and can accept customized handlers to deal with complex formats.

Our primary focus is the scale-up configuration, but we can easily apply our design to a scale-out configuration, by adding another level of scheduler for scheduling jobs to different nodes, while leaving the current scheduler and containers for jobs and tasks on a specific node. We leave this as future work.

## 4.3 I/O Oriented Scheduling

SPMario aims to achieve both good performance and high resource utilization, so we propose *I/O Oriented Scheduling*. There are three core aspects of the scheduling: First, it attempts to keep I/O devices (SSDs) busy to maximize the I/O bandwidth; Second, it enforces the task execution order by controlling the I/O order to reduce resource contention; Third, it tries to schedule tasks with different I/O and kernel patterns together to improve the overall resource utilization.

The concrete scheduling is twofold: the scheduler first chooses a container to monopolize the full I/O bandwidth within its assigned quota; the scheduled container then executes the chunk-based task scheduling to determine the task execution order.

**Algorithm 1:** Container Scheduling.

```
1 Given the current scheduling policy Pl:
2 for container c ∈ C do
3     pr ← getPriority(c, Pl) ;
4     quota ← getQuota(c, pr) ;
5     status ← scheduleContainer(c, quota) ;
6     updateStatus(status) ;
```

Algorithm 1 shows the process of assigning I/O quotas to containers. The scheduler loops through all containers to make sure no container is starving. SPMario is able to support priority-based scheduling policy, by changing the I/O quota assigned to each container.

After getting its quota, a container executes Algorithm 2 to do the specific task scheduling. For the convenience of the description, we assume the fixed-size chunks for now. Given a chunk of a task $t$, as well as the sum of the kernel execution time and the data transfer time between the CPU and the GPU for the chunk, we first define *Rt* (Line 2) as the ratio of this sum to the I/O time on the chunk. Within its quota, the container decides the task execution order. It always begins with selecting a task with the maximum *Rt* from the tasks with the highest priority, to make a big initial *deposit* (*Rt* unit) for the following tasks (Line 13 - 16). To decide which task to schedule next, the container first deducts 1 unit from the deposit for this upcoming task's chunk I/O (Line 18), and evaluates if the deposit left is enough for another task's chunk after this upcoming task. If it is, then the container can choose any task (with higher priority) as this upcoming task, and our algorithm selects a task that has not been scheduled before to be fair (Line 18 - 22). But if it will run short of the deposit for the task after the upcoming task, the container will try to find one with the maximum *Rt* as the upcoming task, to increase the

**Algorithm 2:** Task Scheduling inside a Container.

```
1  Define:
2  Rt = (kernelTime + dataTransferTime)/IOTime.
3  Qt, the queue for tasks not scheduled.
4  Qr, the queue for tasks scheduled to run.
5  while true do
6      quota ←getQuota();
7      deposit ← 0;
8      while quota do
9          if Qt is empty then
10             quota ← 0;
11             return the control to the scheduler;
12         else
13             if first chunk in the current round then
14                 //Select the task with
15                 //the highest priority and max Rt
16                 (T, Rt, cquota) ←getTaskRt(Qt);
17             else
18                 deposit ← deposit − 1;
19                 if deposit ≥ 1 then
20                     //Select the one
21                     //not being scheduled yet.
22                     (T, Rt, cquota) ←getTaskFair(Qt);
23                 else
24                     //Select the task with the
25                     //highest priority and max Rt
26                     (T, Rt, cquota) ←getTaskRt(Qt);
27             deposit ← deposit + Rt;
28             pushQr(T, cquota);
29             quota ← quota − cquota;
```

deposit (Line 23 - 26).

To support various-sized chunks, we just need to modify the values of *Rt* based on the smallest chunk size accordingly. In addition, the algorithm requires the knowledge of the I/O time, the time of the data transfer between the CPU and the GPU, and the kernel execution time for a given chunk. While it is easy to estimate the first two, the

kernel execution time requires more efforts. We can either get this information from the users, or SPMario can profile the kernels within the first several runs. We do not implement the profiling in our current prototype, and will leave it as future work.

## 4.4 Evaluation

We evaluate SPMario and show the experimental results in this section. We also present a case study to demonstrate SPMario's effectiveness in accelerating GPU programs.

### 4.4.1 Experimental Platform

We conduct the experiments on a server with a 4-core Intel Xeon E52609V2 CPU at 2.5GHz and 32GB DDR3-1600 DRAM. The server contains an NVIDIA Tesla K20 GPU with 5GB GDDR5 memory, which connects to the server through 16 lane PCIe 2.0 with 8GB/s bandwidth. The storage is a 768GB high-end PCIe SSD connecting to the server with 4 lane PCIe 3.0, and can sustain 2.2GB/s for both read and write. We run Ubuntu Linux kernel 3.16.3 with CUDA Toolkit 7.0.

### 4.4.2 Benchmarks

We run the experiments with four different applications on SPMario, including: Matrix Multiplication (MM), K-Means Clustering (KMC), Linear Regression (LR), and Image Grep (IG).

The first three are representative GPU MapReduce applications: MM has longer I/O time than the kernel, but the I/O and the kernel time are comparable. KMC is a compute-intensive application with longer kernel time than I/O. For LR, its kernel is much shorter than I/O. We use cuBLAS library [8] to implement MM for maximum performance, and extend KMC and LR from GPMR to support big datasets.

**Table 4.1.** Dataset sizes and chunk sizes of the benchmarks. For MM16K and MM32K, their dataset sizes are smaller than the actual input sizes, because we do not assume any caching.

| | MM16K | MM32K | KMC2K | KMC4K | LR2K | LR4K |
|---|---|---|---|---|---|---|
| Total Input Size (GB) | 17.4 | 135.2 | 32 | 64 | 16 | 32 |
| Chunk Size (MB) | 64 | 128 | 128 | 128 | 128 | 128 |

We test each application on two sizes of datasets, as shown in Table 4.1, with 1, 2, and 4 processes on the baseline[2], while with 1, 2, and 3 on SPMario, because the SPMario scheduler occupies one of the four CPU cores. We also run four pairs of these applications to show the effectiveness of SPMario's scheduling policy.

To demonstrate SPMario's ability in parallelizing GPU applications, we port a well written GPU application *imgrep* [9] (IG) to SPMario. IG takes a given image as the target, and tries to i"grep" the target image from a pool of candidates. The mapper function performs grep to emit "1" if finds the target image, or "0" otherwise, and the reducer collects the count.

### 4.4.3 Intra-Job I/O Oriented Scheduling

We evaluate SPMario's I/O Oriented Scheduling for individual jobs against GPMR [123] as the baseline, since very few other GPU MapReduce systems are both open sourced and with good code quality. To be fair, we optimize the default GPMR with streams and MPS enabled.

To better understand the results of all applications, we first discuss the results of $16K \times 16K$ Matrix Multiplication (MM16K) as an example.

We compare the baseline and SPMario in the execution time and GPU utilization for MM16K with different processes in Figure 4.4(a). Even when only one task is running, SPMario effectively overlaps the task's GPU kernel execution with I/O, reducing the

[2]We did run up to 16 processes but reached the peak performance and utilization with 4 processes for all the applications, so we only report the results for up to 4 processes here.

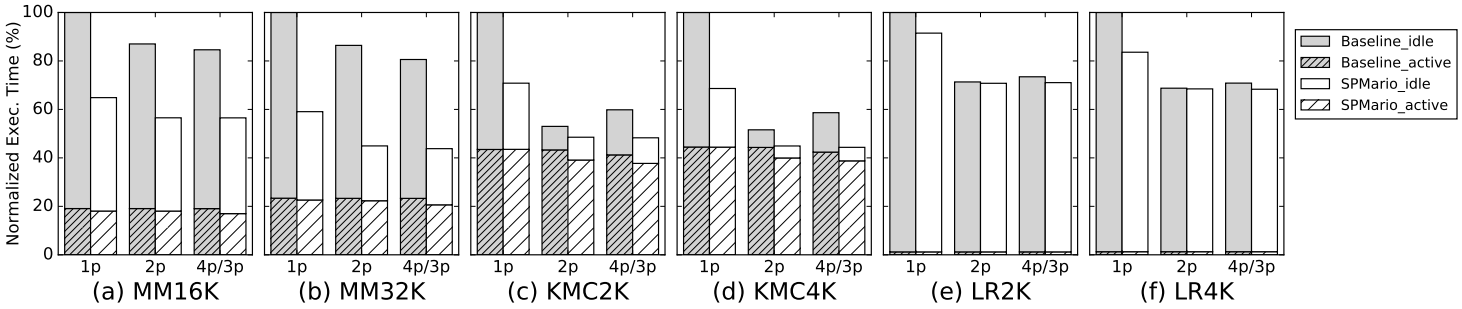



**Figure 4.4.** GPU utilization comparison of the benchmarks with different numbers of processes. All data are normalized to the execution time of the corresponding baseline's single process case (1p) for comparison. Note we use 4p/3p to denote the comparison of 4p for baseline and 3p for SPMario. Both the baseline and SPMario decrease the execution time and improve the GPU utilization by shrinking the idle time with more processes (baseline from 1p to 4p, SPMario from 1p to 3p), but SPMario outperforms the baseline for all cases.

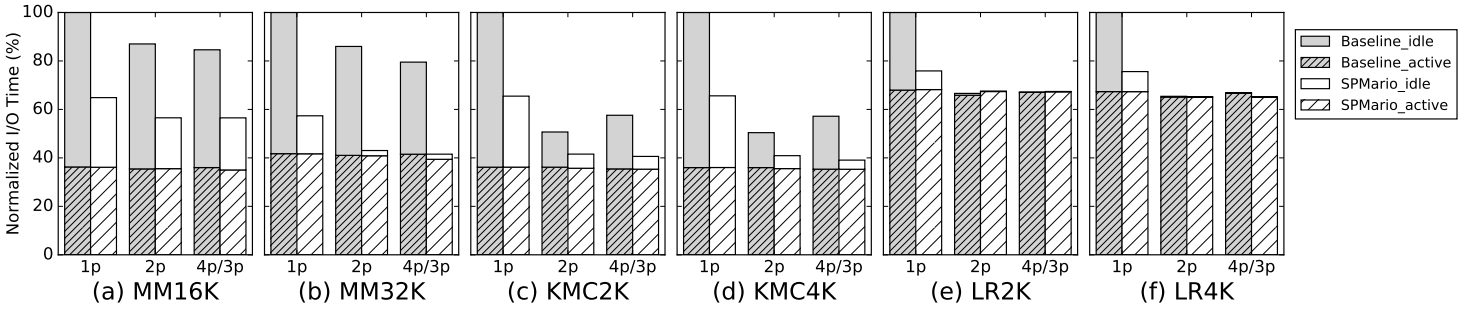



**Figure 4.5.** I/O utilization comparison of the benchmarks with different numbers of processes. All data are normalized to the I/O span of the corresponding baseline's single process case (1p) for comparison. Note we use 4p/3p to denote the comparison of 4p for baseline and 3p for SPMario. Both the baseline and SPMario improve the I/O utilization by shrinking the idle time with more processes (baseline from 1p to 4p, SPMario from 1p to 3p), but SPMario outperforms the baseline for all cases except LR2K.

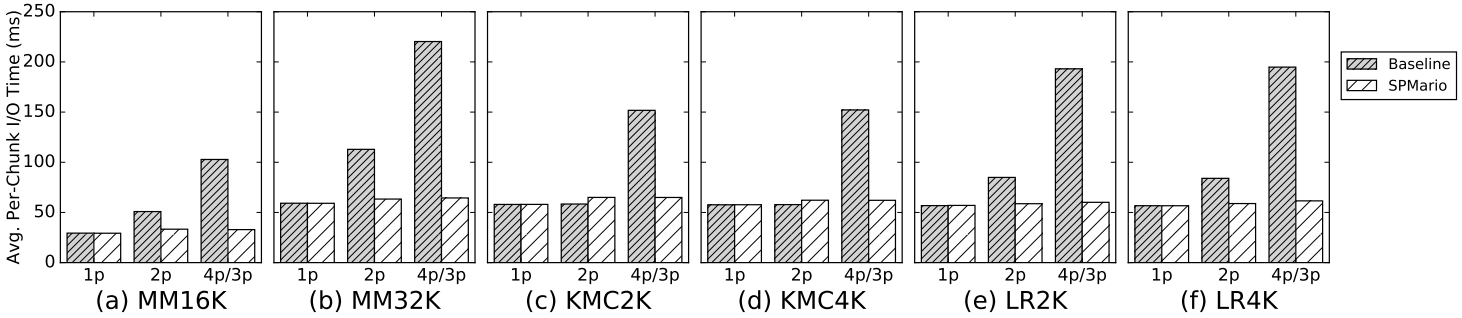



**Figure 4.6.** Avg. per-chunk I/O time comparison of the benchmarks with different numbers of processes. Note we use 4p/3p to denote the comparison of 4p for baseline and 3p for SPMario. With more processes (baseline from 1p to 4p, SPMario from 1p to 3p), the average per-chunk I/O time drastically increases for the baseline, while remains almost unchanged for SPMario.

execution time by 35.14 %. With the increased degree of parallelism, the baseline can also reduce the execution time, but SPMario keeps outperforming the baseline by 35.03 % for 2p, and by 33.2 % for 4p (3p for SPMario). Increasing the task number improves the GPU utilization for the baseline (from 19.08 % to 22.52 %), but improves more for SPMario (from 27.85 % to 31.93 %).

Figure 4.5(a) shows the breakdown of the active/idle time for the total I/O time span. For both the baseline and SPMario, the active I/O time remains the same when increasing the task numbers, because the I/O bandwidth is saturated. But compared to the baseline, SPMario drastically reduces the idle I/O time (by 73.79 % for 1p, 87.71 % for 2p, and 88.32 % for 4p/3p, separately) via its effective task scheduling.

One of the main factors causing the inefficiency of the baseline is resource contention. As illustrated in Figure 4.6(a), the average per-chunk I/O time of the baseline rises rapidly from 29.38 ms to 102.79 ms, while remains almost unchanged in SPMario, indicating severe I/O contention among different tasks in the baseline. The contention in I/O resource prevents the baseline from efficiently pipelining its GPU and I/O operations, resulting in more GPU and I/O idle time.

To demonstrate the effectiveness of SPMario, we conduct more experiments on $32K \times 32K$ MM (MM32K), KMC with 2048 million elements (KMC2K) and 4096 million elements (KMC4K), LR with 2048 million elements (LR2K) and 4096 million elements (LR4K). Figure 4.4 and Figure 4.5 depict the results of GPU and I/O utilization, and Figure 4.6 shows average per-chunk I/O time for each set of the experiments. Table 4.2 summarizes the improvements of SPMario over the baseline.

In all the experiments, we observe similar results: For the single task case, SPMario can reduce the execution time by overlapping GPU kernels and I/O operations. With the increase in the task number, SPMario can continue to improve the GPU and I/O utilization, by reducing the I/O contention and the idle I/O time through deliberate

**Table 4.2.** The percentage improvement of SPMario over the baseline for individual jobs. SPMario can reduce the execution time by up to 48.0%, and improve GPU and I/O utilization by up to 84.0% and 91.56%, respectively, compared to the baseline.

| | # of Tasks | Exec. Time ↘ (%) | GPU Util. ↗ (%) | I/O Util. ↗ (%) |
|---|---|---|---|---|
| MM16K | 1 | 35.14 | 45.9 | 53.69 |
| | 2 | 35.03 | 45.61 | 54.35 |
| | 4/3 | 33.20 | 33.50 | 45.63 |
| MM32K | 1 | 40.93 | 63.88 | 69.07 |
| | 2 | 48.00 | 84.0 | 91.56 |
| | 4/3 | 45.65 | 62.93 | 74.7 |
| KMC2K | 1 | 29.17 | 41.34 | 41.26 |
| | 2 | 8.44 | -1.3 | 7.87 |
| | 4/3 | 19.31 | 13.37 | 23.55 |
| KMC4K | 1 | 31.37 | 45.55 | 45.87 |
| | 2 | 12.92 | 3.57 | 13.51 |
| | 4/3 | 24.38 | 20.97 | 32.1 |
| LR2K | 1 | 8.52 | 9.34 | 9.66 |
| | 2 | 0.78 | 0.97 | 3.14 |
| | 4/3 | 3.28 | 2.78 | 3.75 |
| LR4K | 1 | 16.39 | 0.43 | 3.7 |
| | 2 | 0.40 | 0.27 | 3.69 |
| | 4/3 | 3.57 | 0.27 | 0.93 |

scheduling.

It is worth noting that for both KMC (in Figure 4.4(c) and (d)) and LR (in Figure 4.4(e) and (f)), the baseline execution time increases with 4 tasks, due to the rise in the idle I/O time. This indicates the effect of I/O contention has overwhelmed the benefit of parallelism (Figure 4.5(c) and (d) for KMC, and Figure 4.5(e) and (f) for LR). Also the improvement by SPMario for LR is limited, because LR is I/O dominant: it spends less than 2 % of its execution time in GPU computation, while devoting most to I/O operations. As a result, SPMario has less chances to overlap the GPU kernels with the I/O operations.

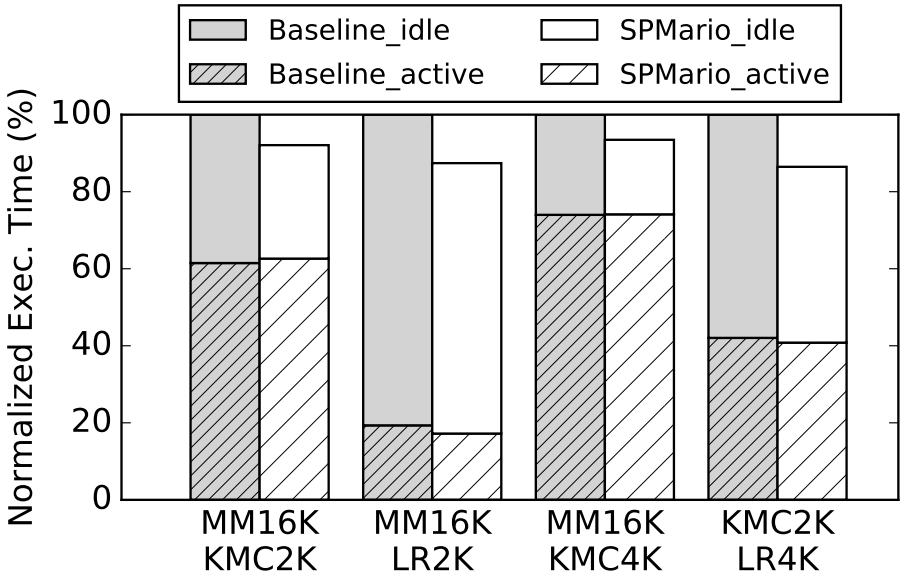
Overall, with up to three processes, SPMario is able to achieve a 2.28× speedup over the original single process baseline in job execution time, and yields up to 2.12× GPU utilization and 2.51× I/O utilization.

### 4.4.4 Inter-Job I/O Oriented Scheduling

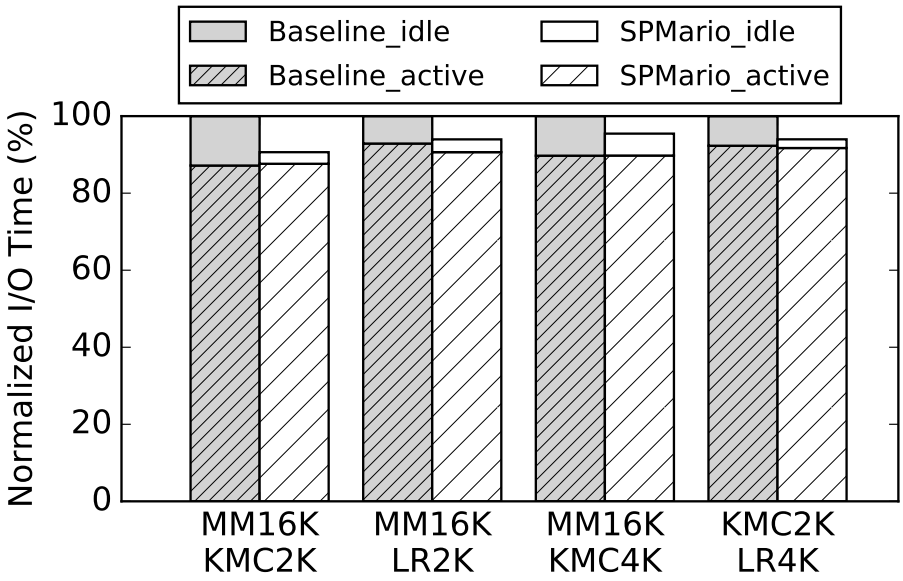
To test SPMario's effect on scheduling multiple jobs, we run four pairs of jobs, including [MM16K, KMC2K], [MM16K, LR2K], [MM16K, KMC4K], and [KMC2K, LR4K]. We choose these job pairs because: First, the jobs in each pair have close dataset sizes and execution time. Second, these combinations cover all the three different types of jobs.

Currently there is no job scheduler supporting multi-job scheduling on GPUs, and the GPMR cannot run multiple applications together either. So instead of using GPMR as the baseline, we use **SPMario with round-robin** as the baseline to compare against **SPMario with I/O Oriented Scheduling**.

It is not easy to further squeeze any improvement out of the highly optimized SPMario jobs, since SPMario has already removed most idle I/O time for individual jobs, as we see in section 4.4.3. However, **SPMario's I/O Oriented Scheduling is still able**




**(a)** GPU Utilization




**(b)** I/O Utilization

**Figure 4.7.** GPU and I/O utilization comparison of co-scheduling between SPMario with round-robin and I/O Oriented Scheduling. Even compared to the already highly-optimized SPMario with round-robin, SPMario with I/O Oriented Scheduling is still able to further shrink the idle time for higher utilization and better performance.

**to find optimization opportunities**, as shown in Figure 4.7. Table 4.3 summarizes the improvement in percentage. Limited by K20's device memory (5GB), SPMario is only tested to schedule two jobs together, but it can schedule more with bigger device memory (e.g., 12GB in K40).

**Table 4.3.** The percentage improvement of SPMario with I/O Oriented Scheduling over round-robin for co-scheduling. When scheduling two jobs together, SPMario with I/O Oriented Scheduling can reduce the total execution time by up to 13.54 %, and improve GPU and I/O utilization by up to 12.27 % and 14.92 %, respectively, compared to with round-robin. The most gain is from the job pair [KMC2K,LR4K], due to the fact that KMC has longer kernel execution than I/O, while LR has a very short kernel.

| | Exec. Time ↘ (%) | GPU Util. ↗ (%) | I/O Util. ↗ (%) |
|---|---|---|---|
| [MM16K,KMC2K] | 7.92 | 10.63 | 9.17 |
| [MM16K,LR2K] | 12.59 | 1.80 | 11.65 |
| [MM16K,KMC4K] | 6.53 | 7.18 | 7.03 |
| [KMC2K,LR4K] | 13.54 | 12.27 | 14.92 |

### 4.4.5 A Case Study

To demonstrate the effectiveness of SPMario, we run the SPMario version of IG against the original one. The original IG includes improvements such as supporting multiple loading threads, and using a ring buffer to overlap I/O with GPU computation. We configure the original IG with four loading threads, and compare it with SPMario running with three tasks. We generate 1200 TIFF files with the average size of 30 MB. The results in Figure 4.8 show that SPMario can reduce the total execution time by 33.4 %, while improve the GPU and I/O utilization by 50.73 % and 55.02 %, respectively.

SPMario allows users to focus on the job-specific logic while leaving the optimizations, such as overlapping I/O with computation, and automatically increasing the number of tasks to provide more parallelism to SPMario.

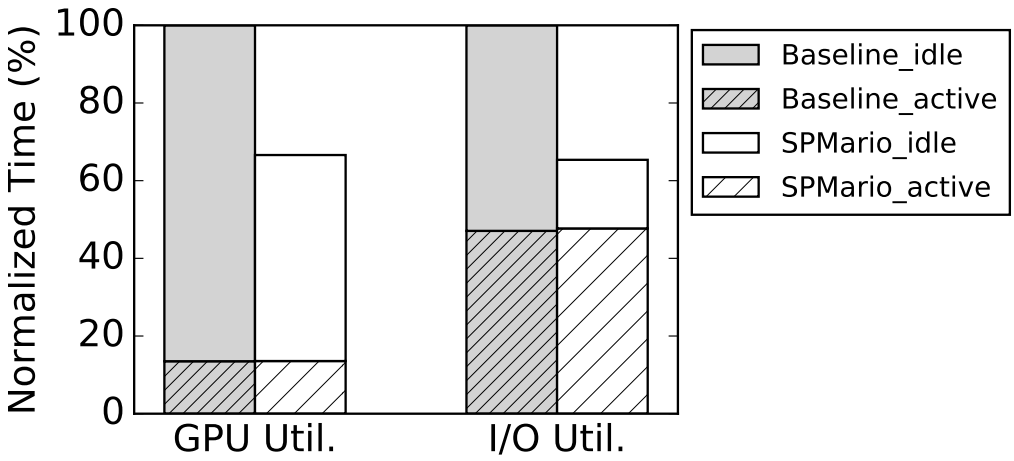



**Figure 4.8.** GPU and I/O utilization of imgrep.

## 4.5 Related Work

Following the success of MapReduce [61] and Hadoop [6] on CPU clusters, great interests have arisen in applying the MapReduce programming model to GPU systems and employing GPUs in existing MapReduce systems.

*Stand-Alone GPU MapReduce Frameworks:* Mars [77] is among the first attempts to port the MapReduce model to GPUs. Catanzaro et al. [52] build a framework with a constrained MapReduce model for Support Vector Machine problems. MapCG [79] avoids some inefficiency of Mars with atomic operations and provides source code portability between CPU and GPU. Some efforts utilize the shared memory in a GPU [85, 55]. Grex [50] takes a step further by using parallel input data split and load-balance data partitioning. Chen et al. [56] explore different schemes to run tasks on both the CPU and the GPU on an integrated architecture. These early studies mainly focus on offloading computation to the GPU, and can only handle datasets smaller than GPU memory. On the contrary, SPMario can efficiently handle large datasets.

GPMR [123] is a GPU MapReduce library for both single node and GPU cluster.

It utilizes chunking to handle datasets bigger than GPU memory, but does not consider I/O accesses. PMGMR [57] improves its single-node predecessor MGMR [58] for bigger datasets, by loading a large chunk into the DRAM and pipelining on small blocks of that chunk. But using slow hard drive disks may underutilize the multiple GPUs. Moim [133] implements a multi-GPU MapReduce framework to explore the parallelism of both CPUs and GPUs and supports load balancing. MATE-CG [86] provides generalized reduce API to run jobs on both CPUs and GPUs. Glasswing [64] uses a 5-stage pipeline to improve the performance of a hybrid MapReduce system with OpenCL. Although it also claims to "scale vertically", its vertical scalability merely means getting better performance with a better device (e.g., "scale" from a CPU to a GPU), while still running a single task on each device. This is very different from SPMario, in which we explore to scale up GPU MapReduce with running multiple concurrent tasks and even jobs on a single GPU.

None of the above GPU MapReduce frameworks or systems explore the potential of sharing a single GPU among multiple jobs, or improving the system utilization. On the contrary, SPMario employs I/O Oriented Scheduling to efficiently overlap I/O, data transfer between the host and the GPU, and GPU kernel execution, and to schedule multiple jobs to run together to better utilize the I/O and GPU resources.

*Hadoop GPU Extensions and Scheduling:* Many previous works [70, 69, 32, 124] focus on invoking GPU kernels from Java code. Systems like [75, 137, 113] further explore to automatically generate GPU kernels.

Scheduling in Hadoop is well investigated [136, 35, 72], but these general scheduling cannot schedule GPU tasks. Shirahata et al. [117] propose a dynamic profiling based method to schedule map tasks on CPUs and GPUs, and HeteroDoop [113] uses tail scheduling scheme to achieve load balance.

These Hadoop GPU extensions and scheduling do not support GPU sharing, nor I/O optimizations for GPU execution, since the I/O handling has to go through

the standard Hadoop I/O path. Realizing the importance of I/O in scheduling, Hadoop YARN [10] has considered disk I/O as another form of resource together with CPU and DRAM [11]. It avoids over-allocation to provide tasks with guarantee in bandwidth, response time, and I/O isolation. However, statically limiting each task's bandwidth would not help to increase the overall system utilization. SPMario has direct control over I/O, so it can coordinate I/O accesses among different tasks, improving the resource utilization while avoiding the resource contention.

*Scheduling in Heterogeneous Systems:* Scheduling in heterogeneous systems is an active research area. Qilin [91] provides a new API and exploits methods to adaptively map tasks to CPUs and GPUs. Ravi et al. [108] study the problem of scheduling jobs on both the CPU and the GPU on both a single node and a cluster. TORQUE [12] is a popular scheduler for heterogeneous clusters. But they do not consider GPU sharing or I/O in scheduling. PTask [110] proposes OS abstractions to provides a system level solution for GPU task scheduling, and is complementary to SPMario. Its budget-based scheduling policy looks similar to SPMario, but it does not consider I/O and will block GPU invocation when the budget is negative, while in SPMario we can continue to schedule I/O operations even when the GPU is busy.

## 4.6 Summary

GPUs offer massive parallelism and high inner bandwidth, and provide an attractive option for scaling up MapReduce jobs. To better utilize the advanced hardware for large datasets, GPU MapReduce frameworks require efficient I/O handling and task scheduling for both performance and resource utilization. In this chapter, We present SPMario, a GPU MapReduce framework to improve the utilization of both GPU and I/O resources, and share the GPU among concurrent tasks. Experiments on three rep-

resentative jobs and a case study show SPMario is able to accelerate job execution and boost resource utilization by removing idle I/O time with the proposed I/O Oriented Scheduling.

## Acknowledgements

This chapter contains material from “SPMario: Scale Up MapReduce with I/O-Oriented Scheduling for the GPU” by Yang Liu, Hung-Wei Tseng, and Steven Swanson, which appears in *ICCD ’16: 2016 IEEE International Conference on Computer Design*. The dissertation author was the first investigator and author of this paper.

# Chapter 5

# Exploring the Algorithms and Fine-grained Scheduling for Heterogeneous Computing

In the previous two chapters, we have discussed the more general problems of I/O handling and system resource utilization in heterogeneous computing. In this chapter, we will focus on providing better algorithm and fine-grained scheduling support in heterogeneous systems for an important application, the web search engine.

In large-scale information retrieval services, search engines serve as the key gateway to rapidly growing data sets and must provide relevant results with consistently low latency. For example, web search engines have to respond to tens or hundreds of thousands of queries per second, over tens of billions of pages with the total size of multiple terabytes [63], all while keeping response times 300 ms [83]. To provide scalability, current search engines resort to massive, coarse-grain parallelism by distributing queries across large compute clusters. To meet their latency goals, they rely on clever, highly-optimized algorithms that exploit intra-query parallelism.

Previous work explores intra-query parallelism by increasing the number of CPU threads[122, 125, 83, 76] that process each query (Figure 5.1(a)). Existing studies also leverage the parallelism that GPUs [63, 44] can provide (Figure 5.1(b)) and can obtain

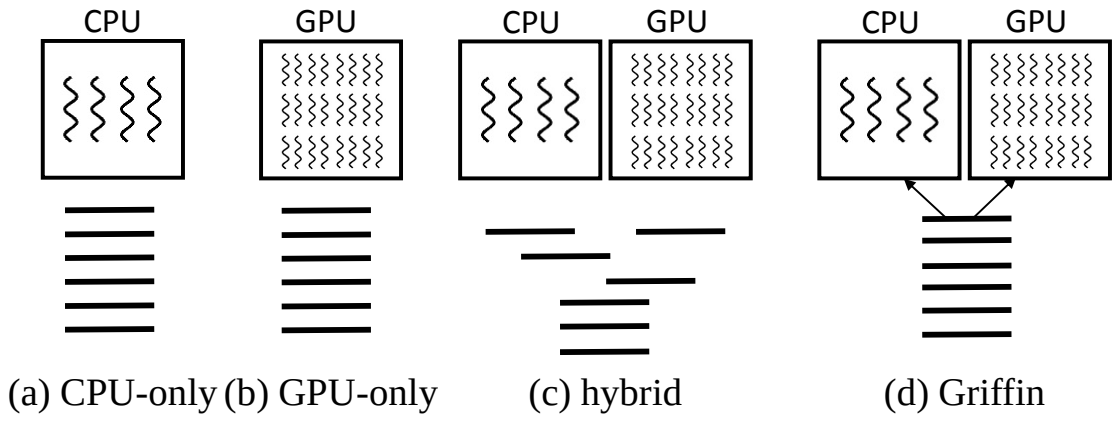



**Figure 5.1.** Intra-query parallelism schemes.

impressive speedup compared to CPU solutions [44]. Because queries may have different characteristics, some queries may run better on GPU, while others run better on CPU. So a heterogeneous system with CPU and GPU can achieve better overall performance by running a query either on the CPU or the GPU based on its characteristics [63] (Figure 5.1(c)).

However, the characteristics of a query can change as the query executes. While the early stages of a query's execution may run well on a GPU, the later stages are often a better fit for the CPU. Therefore, running an entire query solely on CPU or GPU in a (Figure 5.1(a), (b), and (c)) may not achieve the best query processing performance. This suggests that a dynamic fine-grained approach that can schedule operations within the processing of individual queries to the suitable processors (Figure 5.1(d)) will lead to better performance.

In this chapter, we present *Griffin*, a search engine that combines two innovations to provide improved performance. First, Griffin uses both the CPU and GPU to process queries and migrates executing queries between the GPU and the CPU as their characteristics change. Griffin decides when and where to execute query operation without *a priori* of the query's characteristics. To make a proper decision, Griffin considers overheads

due to data transfer between CPU and CPU and GPU memory management as well as the system load.

Griffin addresses these challenges with a dynamic intra-query scheduling algorithm that breaks a query into sub-operations and schedules them to the GPU or to the CPU based on their runtime characteristics. Griffin uses this scheduling algorithm to divide work between a state-of-the-art CPU-based search implementation and new GPU-based search kernel called Griffin-GPU.

Griffin-GPU is the second key innovation in Griffin. Griffin-GPU combines two components. The first is a parallel implementation of the Elias-Fano compression [128] algorithm that provides fast decompression and a high compression ratio. The second is a load-balancing merge-based implementation [3] of parallel list intersection.

Our experiments on the real world query dataset [13] show that, Griffin speeds up the query processing by 10x and 1.5x compared to a highly optimized CPU-based search engine and Griffin-GPU running alone, respectively. Griffin also reduces tail latency: it reduces the 95th, 99th, and 99.9th percentile latencies by 10.4x, 16.1x and 26.8x, respectively, compared the CPU-only implementation.

This remainder of this chapter is organized as follows. Section 5.1 introduces the background of query processing, and discusses the characteristics of CPU and GPU. Section 5.2 describes the design and implementation of Griffin. We then evaluate the Griffin prototype in Section 5.3. Section 5.4 discusses the related work. Finally, we summarize this chapter in Section 5.5.

## 5.1 Background

The rise of search as a “killer app” on the Internet as well as the increase in the amount data available via search has lead to highly optimized web search algorithms and implementations. These algorithms aim to maximize performance (in terms of latency

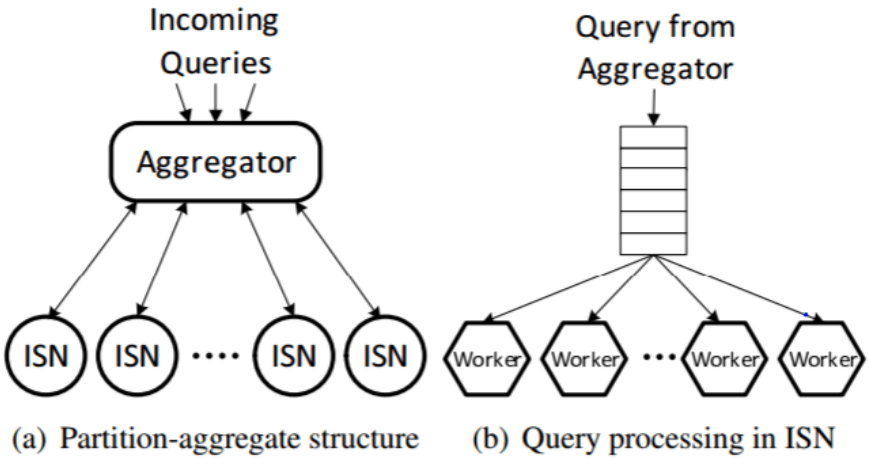



**Figure 5.2.** The Search Architecture [83].

and bandwidth) and minimize memory and/or storage requirements. As a result, these algorithms store indices in specialized compressed formats that make minimize data storage while still allowing for fast search.

The fastest and most space-efficient search engines also tailor the compressed data structures and algorithms to match the hardware that will perform the search. As a result, the best techniques for a CPU-based search engine will differ from the best techniques for a GPU-based search engine.

Below, we describe the typical search architecture in web search engines, and state-of-the-art approaches of query processing on CPU and GPU.

### 5.1.1 Search Architecture

To cope with the high volume of the web index (consisting of billions of web documents) and the search queries, the search engine usually adopts a two-level architecture [63, 83], as shown in Figure 5.2. The first level is the aggregator, and the second level are index serving nodes (ISNs). Hundreds of ISNs store the whole web index for billions of web documents, each responsible for a subset of the whole index. If the result of a query is cached, the aggregator will return the result from the result cache directly

to the user. Otherwise, the aggregator distributes the query to a group of ISNs, each of which executes query processing, and return the results to the aggregator. The aggregator then collects the results, and return the most relevant results back to the users.

### 5.1.2 Query Processing

Query processing is at the heart of search engines and it is the focus of this chapter. The most important query processing data structure is the *inverted index*, which consists of many *inverted lists*. There is one inverted list for each possible search term, and the list for a term holds the document IDs (docID) of the documents (e.g., web pages) that contain that term. The inverted lists usually store the docIDs in sorted order. The inverted index needs to be as compact as possible to minimize the cost of the memory needed to store it, so search engines usually store it in compressed form and decompress it as needed.

To process a query, the search engine first loads the inverted lists of the query terms into memory (if necessary), and decompresses them. Then, it computes the intersection of the lists, yielding the set of docIDs that contain all the search terms.

To compute the full intersection, the search engine performs a series of pair-wise intersections. The intersection usually starts with the two shortest inverted lists (i.e., the two rarest search terms) to reduce unnecessary overhead. This yields an intermediate list of docIDs, and the search engine then computes the intersection of this resulted list with the next longer inverted list. The process repeats until all the lists have been incorporated or the list of matching docIDs is empty.

The basic algorithm for computing the intersection of two sorted inverted lists is similar in spirit to merge sort: The search engine scans the sorted lists and records the docIDs they have in common. Modern search engines use sophisticated data structures like skip lists and skip pointers (Figure 5.3), to allow the algorithm to skip large portions

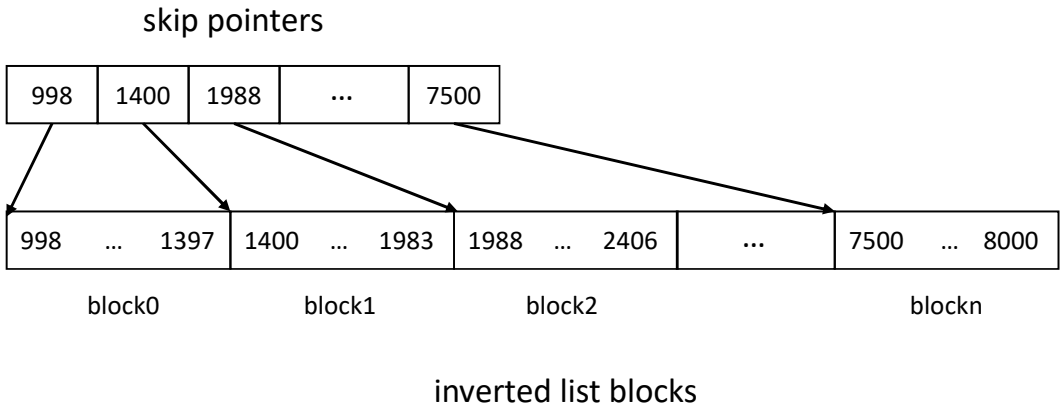



**Figure 5.3.** An example of an inverted list with skip pointers. The skip pointers store the offset and the first value of each inverted list block, and can support binary search to fast locate the required blocks.

of the lists during the scan.

Next, the search engine calculates a relevance score for each of the documents using document metadata (e.g., the frequency of term in the document or the documents popularity), sorts the documents according to this score and returns the top $K$ results (e.g., enough to fill the first page page of search results).

Below we describe these three key operations in more detail.

**Index Compression and Decompression**

Since inverted index can be large, compression is necessary to save memory and storage space and to reduce data transfer time. Aggressively-optimized search engines may even keep the entire inverted index in memory spread across several machines. Thus, higher compression ratio is important as is decompression speed. Compression speed is less critical since it only occurs when the search engine prepares a new index (usually offline).

There have been many proposed compression schemes for inverted lists [140]. Many compression techniques in modern search engines start by computing the deltas

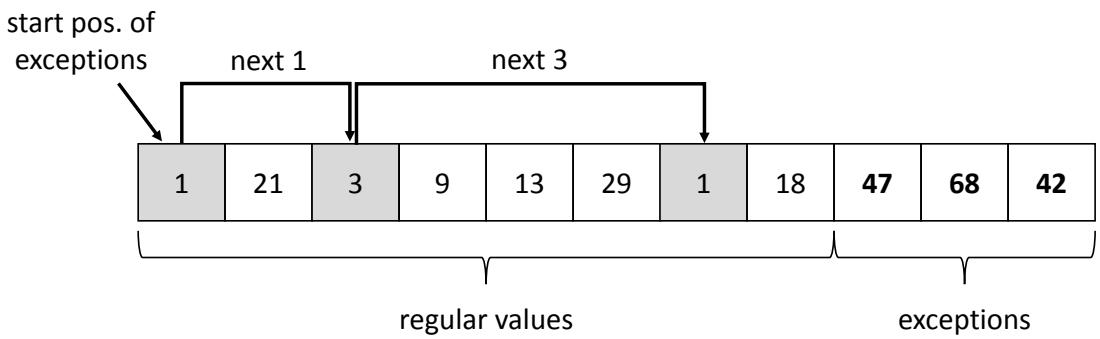



**Figure 5.4.** An example of PforDelta encoding. Given a sequence of docIDs $\ell(t) = (100, 121, 163, 172, 185, 214, 282, 300, 347)$, the corresponding sequence of d-gaps is $\ell(d) = (21, 42, 9, 13, 29, 68, 18, 47)$. Here we have $b = 5$ bits, so the exceptions are (42, 68, 47). We use 5 bits to represent each of the regular values as well as the pointer (the gray positions) to the next exception element. The value of exceptions are stored at the end of the sequence.

(a.k.a *d-gaps*) between two consecutive docIDs. PforDelta [141] is a state-of-the-art d-gap-based compression scheme that provides a good tradeoff between decompression speed and space overhead [138, 134].

Given a sequence of d-gaps $\ell(d)$ calculated from the sequence of ascending docIDs, PforDelta compresses d-gaps by choosing the smallest $b$ such that a vast majority of elements (e.g., 90%) can be encoded in $b$ bits. These are called *regular values*. It then encodes the d-gaps $|\ell(d)|$ values by allocating $|\ell(d)|$ $b$-bit slots, and leaves the values that cannot be represented in $b$ bits (called *exceptions*) at the end. Each exception takes 32 bits while each regular value takes $b$ bits. In order to indicate which slots are exceptions, it uses the unused $b$-bit slots to construct a linked list, so that the $b$-bit slot of one exception indicates the offset of the next exception. In the case where two exceptions are more than $2^b$ slots apart, it adds additional exceptions in between the two slots. Figure 5.4 shows a example.

To avoid decompressing the entire list, the algorithms usually store d-gaps into blocks of, for example, 128 elements [138]. Each block contains the range of docIDs in the block so the algorithm to quickly determine whether a block contains any docIDs of

interest.

### List Intersection

List intersection is the key operation of query processing. It scans and intersects over several inverted lists of corresponding terms to find the common docIDs.

There are many intersection algorithms. We use *SvS* [59], a popular choice. It orders the lists from shortest to longest and computes partial intersections starting with the shortest two lists. This improves performance since the run-time of the merge step depends strongly on the length of the shortest list. For example, if we search "EuroSys Belgrade 2017", the search engine may subtract and divide the terms into three inverted lists for terms "EuroSys", "Belgrade", and "2017", respectively:

$$\ell(EuroSys) = (11, 15, 17, 38, 60),$$
$$\ell(Belgrade) = (3, 5, 8, 11, 13, 15, 17, 38, 46, 60, 65),$$
$$\ell(2017) = (2, 4, 6, 11, 13, 14, 15, 19, 25, 33, 38, 60, 70).$$

And the result of the intersection is the common docIDs in the three inverted lists:

$$\ell(EuroSys) \cap \ell(Belgrade) \cap \ell(2017) = (11, 15, 38, 60).$$

### Top-$K$ Scoring

Search engines aim to return the most relevant results to users, and this process usually includes two steps. (1) Similarity computation: for each matched document $d$ in the result set, compute the similarity (or score) between $q$ and $d$ according to a ranking function; (2) Ranking: find the top ranked documents with the highest scores. In this chapter, we follow a typical ranking model, BM25 [109, 119], to determine the similarity between a query and a document. This model and its variants are widely used in many

modern search engines such as Lucene [14].

Let,

$qtf$ : term's frequency in query $q$ (typically 1)

$tf$ : term's frequency in document $d$

$N$ : total number of documents

$df$ : number of documents that contain the term

$dl$ : document length

Then,

$$Similarity(q,d) = \sum_{t \in q} (Similarity(t,d) \times qtf)$$

$$Similarity(t,d) = tf \cdot (1 + \ln \frac{N}{df+1})^2 \cdot (\frac{1}{dl})^2$$

Typically, each entry in the inverted list contains a document frequency (in addition to document ID and positional information). When a qualified result ID is returned, its score is computed by using the above equations.

### 5.1.3 Query Processing on CPU

Most search engines execute decompression, list intersection, and top-$K$ operations on CPU. CPU is good at dealing with complex logic, and its advanced prefetch and branch handling can provide high performance and efficiency. As a result, CPU is able to run both sequential merge, especially when the data accesses exhibit ample spatial locality. When the two lists involved in the intersection have similar lengths, the algorithm must access most of the items in both lists, leading to that locality. As a result, CPU performance on the merge is high. Alternately, if the length difference between the two lists is large, CPU can perform binary search with the help of skip pointers to skip large portion of unnecessary computation including decompression and comparison. In this case, the CPU clock speed and aggressive branch handling will still perform well.

On the other hand, CPUs cannot exploit the large amount of fine-grain parallelism

that exists in query algorithms [81], due to the limited number of cores. The CPU accounts for over 70% of the response time [76] in search workloads and an even larger fraction when the query has many terms, since there are more decompression, intersection, and top-$K$ computations to perform. As result, the fine-grain parallelism that CPUs cannot exploit is a significant missed opportunity.

### 5.1.4 Query Processing on GPU

GPUs offer on way to exploit the parallelism that CPUs cannot. Compared to CPUs, GPUs are able to provide massive parallelism with hundreds or thousands of simpler cores. GPU hardware multithreading and fast context switching can hide both arithmetic and memory latency, as well as avoid dependency stalls by overlapping thread execution. What is more, its SIMD execution model can amortize the overhead of instruction fetch and decode, and harness data parallelism [34]. In addition, GPU have much higher inner bandwidth (as high as 208GB/s in NVIDIA K20) than CPU.

On the other hand, GPUs incur substantial startup overheads related to data transfer and memory allocation, and they have limited memory (e.g., 5GB in NVIDIA K20). However, these costs occur just once, so running larger, more complex query operations on the GPU can amortize them.

**Decompression on GPU** A good parallel decompression algorithm should be efficient on GPU without sacrificing high compression ratio or fast decompression speed. The state-of-art CPU decompression method PforDelta is a poor match for GPU implementation, because it maintains a linked list to store the exception pointers that it must process sequentially. This leads to slow global memory accesses and thread divergence.

**List Intersection on GPU** GPU list intersection algorithms often rely on parallel binary search. When the length difference between the two lists involved in the intersection is

large, parallel binary search reduces the search space quickly. In addition, since binary search can skip many blocks in the longer list, it also reduces the number of blocks that the GPU must decompress.

However, GPU binary search is not very efficient since it makes coalescing memory accesses difficult and can lead to branch divergence. This divergence leads to idle threads and reduced performance. Even worse, when the length difference between the two lists is large, the binary search is more likely to reach its worst case complexity $\log(N)$. This will lead to more frequent divergence, since the vast majority of the items from the longer list will be missing in the shorter list. In addition, as each thread is accessing a different area of the memory, there is little opportunity to coalesce accesses leading to lower memory bandwidth.

When the lengths of the two lists are close, the benefit of reducing search space in the binary search will decrease. In Section 3 we find that, empirically, the benefits fade as when the ratio of lists sizes falls below 128x. These situations are quite common: according to the real-world dataset we are using, about 64.69% *actual* intersections contain the lists with the ratio of the length less than 128.

Parallel merge-based search is an alternative to binary search that is likely to be more efficient in these cases. A merge-based algorithm will allow for cache line reuse among neighboring threads and reduce the number of global memory, since the algorithm can load data in to the faster thread-shared memory. We describe our novel merge-based algorithm in the next section.

## 5.2 Design and Implementation of Griffin

This section describes the design and implementation of our Griffin prototype. Griffin consists of three main components: Griffin-GPU, the CPU query processing component, and the scheduler. Griffin-GPU implements the advanced parallel algorithms

on the GPU. The CPU query processing component implements state-of-the-art CPU query algorithms [141, 59, 109]. The scheduler decides whether to run the current query operation on the GPU or CPU.

### 5.2.1 Griffin-GPU

Griffin-GPU executes query operations on GPU and relies on two key algorithms: a novel parallel decompression scheme called *Para-EF encoding* to compress and decompress inverted lists, and a parallel *merge-based intersection* algorithm to efficiently intersect inverted lists on the GPU.

**Parallel List Decompression**

A suitable encoding scheme for Griffin-GPU has to satisfy three requirements: (1) high decompression speed; (2) high compression ratio; (3) highly parallel.

Based on these three requirements, we adopt an encoding scheme called *EF (Elias-Fano) encoding*. The basic idea of EF encoding first appeared in 1974 [65], and Vigna "rediscovers" it in 2013 [128] and finds it has both higher decompression speed and compression ratio than the PforDelta.

To compress a sequence of integers, EF encoding divides each integer into high bits and low bits, and encodes them into the *low-bits array* and the *high-bits array*. For the list with $n$ integer elements and $U$ as the maximum possible value, the low-bits array stores the (fixed) $b = \lfloor \log \frac{U}{n} \rfloor$ bits of each element contiguously. The high-bits array then stores the remaining upper bits (with variable lengths) of each element as a sequence of *unary-coded d-gaps* of these elements. To decompress these integers, we just need to recover the high bits from the unary-coded d-gaps array, find its corresponding low-bits, and concatenate them. Figure 5.5 illustrates the basic encoding and decoding of EF scheme in detail with an example.

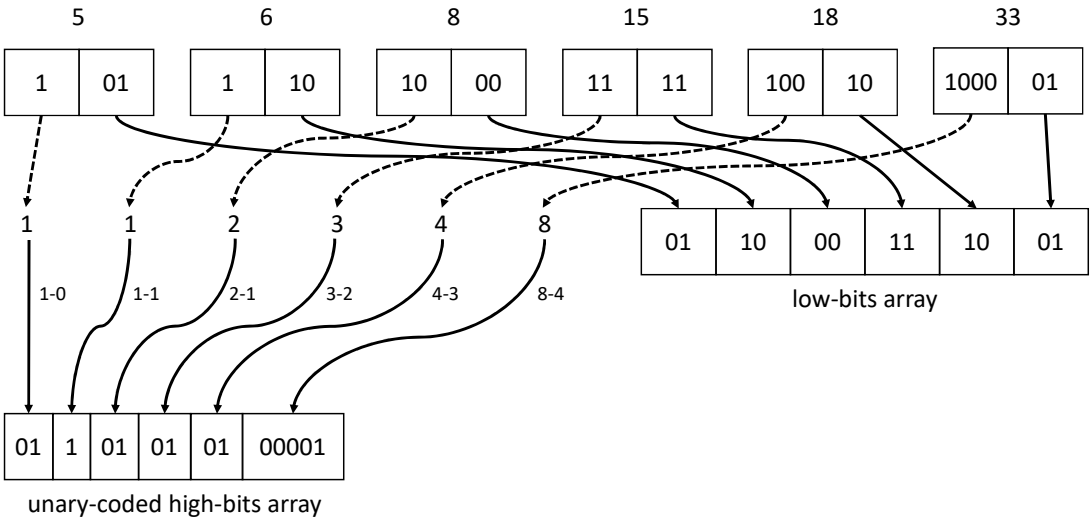



**Figure 5.5.** An EF encoding example. For the original integer sequence (5,6,8,15,18,33), upper bound $U = 36$, thus $b = \lfloor \log \frac{36}{6} \rfloor = 2$. The low-bits array on the right stores the low $b$ bits of the original sequence, while the unary-coded d-gaps array on the bottom left encodes the high-bits array. Each encoded element in the d-gaps array ends with "1", and the number of "0s" inside an element represents the value of d-gap. The accumulated number of "0s" gives the actual high-bits value.

**Algorithm 3:** Parallel EF Decompression.

**Input** : EF-compressed high-bits array *hb_array*, low-bits array *lb_array*, and the number of low bits $b$

**Output** : Decompressed array *decmp_array*

```
1 for each thread i do
2     ps_array[i] ← popcount(hb_array[i]);
3     ps_array[i] ← prefix_sum(ps_array[i]);
4     count = ps_array[i] − ps_array[i − 1];
5     while j < count do
6         index_array[j + ps_array[i − 1]] ← i;
7         j ← j + 1;
8     Recover high_bits_i from hb_array[index_array[i]];
9     decmp_array[i] ← (high_bits_i << b) | lb_array[i];
```

Griffin-GPU is the first GPU search engine to adopt EF encoding, and we provide the first parallel EF decompression implementation *Para-EF decompression*, as described in Algorithm 3.

The algorithm first computes the population count (popcount) for each element in the high-bits array *hb_array* in parallel (line 2) to know how many original (decompressed) elements each 32-bit word in the *hb_array* encodes. Then, it calculates the prefix sum (using an efficient GPU-friendly algorithm) from the popcount result and store it in the temporary *ps_array*(line 3).

Our algorithm separates the actual decompression into two phases: scheduling and decompressing. It first uses the prefix sum result to schedule and assign decompression tasks to thread (line 4-7), so that each thread will decompress a value from the corresponding word in *hb_array* and deliver it to the final decompressed array *decmp_array*. For example, if $ps\_array[0] = 13$, then $thread\_0 - thread\_12$ will decompress $element\_0 - element\_12$ from $word\_0$ of *hb_array*. If $ps\_array[1] = 20$, then $thread\_13 - thread\_19$ will decompress $element\_13 - element\_19$ from $word\_1$ of *hb_array*. After the task distribution, each thread recovers the high_bits an element and concatenate its corresponding low_bits to get the decompressed element (line 8-9).

We have implemented this algorithm in CUDA [23], which provides a popcount instruction, __popc [15]. In addition, we use the parallel prefix sum to reduce the dependency in the original serial EF decompression. We store a look-up table in the shared memory of the GPU to further improve the performance of recovering the high-bits array. We also store the temporary arrays in shared memory.

**Parallel List Intersection**

Griffin-GPU's list intersection algorithm divides list intersection operations into two class, depending on the relative sizes of the two lists. The cross-over point between

the two techniques is a parameter of the algorithm that we determine empirically in the next section.

When difference between list lengths is large Griffin-GPU uses parallel binary search with the skip pointers. Griffin-GPU first does binary search over the skip pointers instead of the long list to identify chunks that may contain that elements in the short list. After this step, it only transfers, decompresses, and processes those chunks.

When the difference between the list lengths is small Griffin-GPU uses a parallel merge-based intersection algorithm based on GPU MergePath [73], which, in turn, is based on derives from [101]. The goal of the algorithm is to reuse cache lines between neighboring threads, and reduce the accesses to the global memory by loading and accessing data in the shared memory.

GPU MergePath divides the list intersection into two stages: *partitioning* and *merging* (Figure 5.6). In the partitioning stage, we divide the list into partitions that each potentially contain common elements from both lists. GPU MergePath sizes the partitions so a pair of partitions will fit in the GPU's shared memory. And then in the merging stage, we merge the two sub-lists inside each non-overlapping partition to get the final intersected results. The merging threads can run concurrently without synchronization.

The biggest challenge and the core of implementing this algorithm on GPU efficiently is to partition lists to achieve load balancing and eliminate the need for synchronization. Figure 5.6 shows an example of even partitions and their boundaries.

Instead of blindly doing binary search or deciding the partitions statically [63, 44], we first consider the process of merging list $A$ and $B$ from a different perspective. We can visualize this process as a *merge path* from the top-left corner to the bottom-right corner of an $|A| \times |B|$ grid (Figure 5.7), allowing only right and downwards directions. Because both list $A$ and $B$ are sorted, we always have:

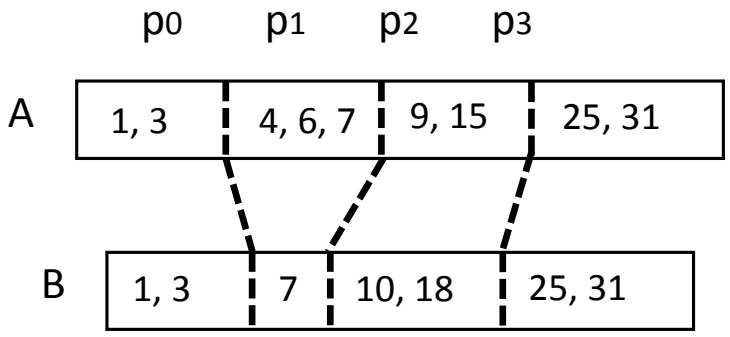



**Figure 5.6.** An Even Partition Example. The partitions divide elements from list A and B into 4 partitions ($P_0$,$P_1$,$P_2$,$P_3$). Each partition evenly contains 4 elements of both lists to do intersection independently later.

$$For\ all\ j' > j,$$

$$if\ A[i] < B[j],\ then\ A[i] < B[j'].$$

Thus, merging the two lists is transformed to the process of finding the merge path through this matrix. We can construct such a path executing the following two operations:

- Advance rightward by one step, if $A[i] \leq B[i]$;
- Advance downward by one step otherwise.

The merge path is essentially the history of decisions made during the merge, and exactly one such path exists.

We can use the existence of this path to locate the $p$ partitioning points for $A$ and $B$. We can find these points by drawing $p$ cross diagonals from the top-right to the bottom-left of the grid. Because the cross diagonals and the path are in the opposite directions, they must meet at some intersections, and they are the partitioning points.

These points and diagonals have some interesting properties:

- Given a partitioning point $P_{ij}$, we must have $A[i] = B[j]$, or $A[i]$ is the lower bound into $B$ for the current partition.

- For a point on a diagonal $P_{ij}$, $i + j = |diagonal|$, where $|diagonal|$ is the distance from the top-left origin to the position where the diagonal intersects with the axis of list $A$.

We can use the above two properties to do binary search along each diagonal (in parallel) to find the point where it crosses the merge path, and then uses these points to partition $A$ and $B$. Figure 5.7 illustrates the operation of MergePath.

The $p$ partitioning points divide the all the elements from A and B into $p$ partitions evenly, thus the process of finding these points ensures that the merge stage is perfectly load balanced. After discovering these $p$ partitioning points, we can eventually do the serial merge inside each partition. For more detail about this algorithm, please refer to [73].

Previous work [3] provides an efficient CUDA implementation of MergePath.

**Top-K Selection**

The final stage of query processing is identifying the top-$K$ query results to return to the user. We evaluated three $top-K$ ranking function in Griffin-GPU: GPU bucketSelect [36], GPU radixSort, and CPU partial_sort (provided by the C++ STL).

GPU bucketSelect is a fast parallel K-selection algorithm. We use it as the first step of $top-K$ selection by selecting the $K-max$ value, and select all the $top-K$ values. The GPU radixSort is a brute-force solution, which just sorts all values in the list in parallel, and we pick the $top-K$ values. The CPU partial_sort returns only the top-K values.

We sample the list of full results for 100 queries and ran these three top-$K$ algorithms on them. We found that the CPU implementation provides the best performance

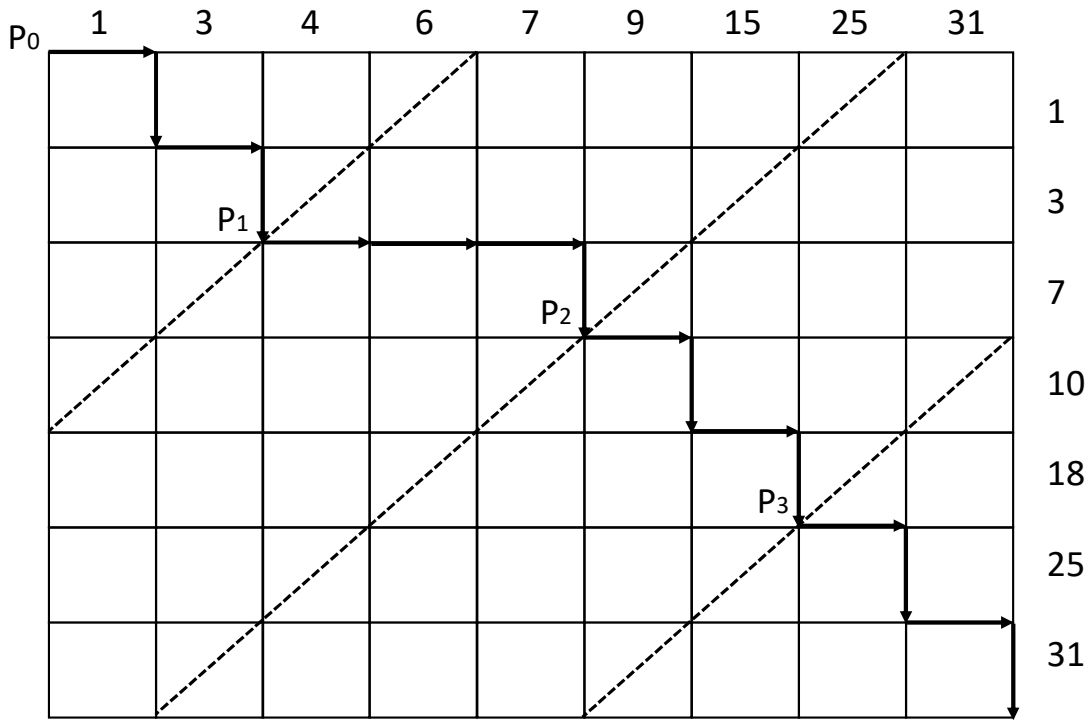



**Figure 5.7.** A Merge Path example. Say we want to intersect list A = (1,3,4,6,7,9,15,25,31) and list B = (1,3,7,10,18,25,31) with 4 partitions (so $p = 4$, and each partition has $(|A| + |B|)/p = 4$ elements evenly). In the grid, we draw 4 diagonals, and they intersect with the merge path shown with arrows at 4 points ($P_0$, $P_1$, $P_2$, $P_3$). Thus, *partition_0* contains (1,3) from A and (1,3) from B, *partition_1* contains (4,6,7) from A and (7) from B, *partition_2* contains (9,15) from A and (10,18) from B, and *partition_3* contains (25,31) from A and (25,31) from B. If we get these 4 partitions, we can run merging on them concurrently, and get the intersection results: (1,3,7,25,31).

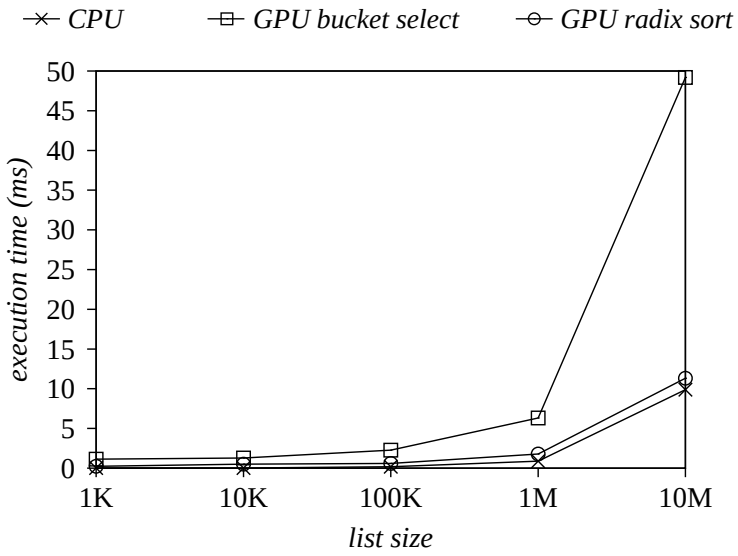



**Figure 5.8.** Top-K evaluation.

(Figure 5.8). We suspect the poor showing of the GPU algorithms is due to the small number of results (queries rarely result in more than several thousands matches). These small input sizes cannot saturate the GPU or amortize the GPU's overhead.

### 5.2.2 Hybrid Query Processing in Griffin

The performance of list intersection in Griffin-GPU relative to the CPU implementation depends strongly on the relative sizes of the lists at hand: Griffin-GPU performs best when ratio of list sizes is relatively small, and the CPU performance better the difference is large.

As query processing proceeds, the length difference between the two lists involved will increase for two reasons. First, one of the two lists constitutes the intermediate results for the query, and this list shrinks monotonically during execution. Second, the query algorithm starts with the smallest two lists, so the algorithm intersects the list of intermediate results with larger and larger lists.

To understand the performance of Griffin-GPU and our CPU implementation

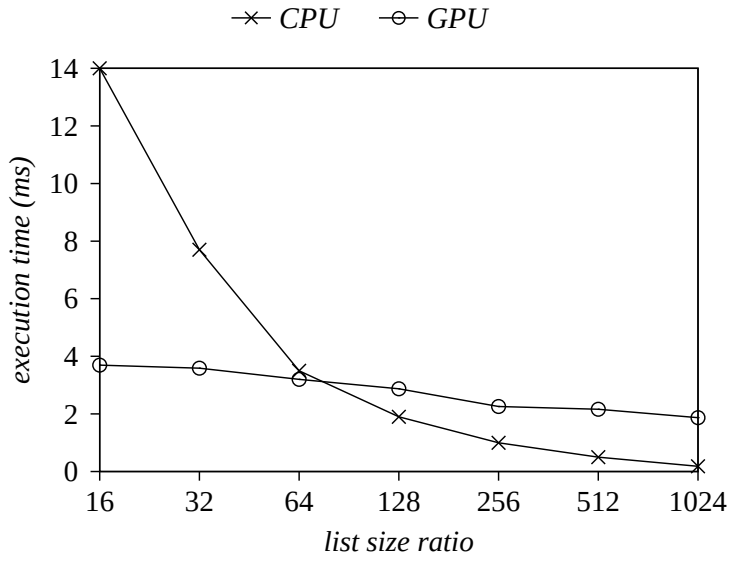



**Figure 5.9.** Observation.

on lists of different size, we randomly select 100 intersection list pairs in each of the following ratio ranges: [1,16), [16,32), [32,64), [64,128), [128,256), [256,512) and [512,1024). To have the meaningful average time, we also limit the length of the longer lists to be in the range of [1M,2M]. We measure the run time of Griffin-GPU and our CPU implementation for each set of pairs. Figure 5.9 contains the results.

Griffin-GPU outperforms the CPU version when the list lengths are with a factor 128. In these cases, almost all of the data blocks in the intersected lists need to be decompressed, leaving little chance for binary search to avoid decompression. Thus, Griffin-GPU can utilize the parallel EF decompression as well the parallel merge-based intersection algorithm to efficiently merge the two lists. In addition, the GPU-related overhead of kernel invocation, memory management, and data transfer from CPU to GPU can be well amortized.

For larger ratios, the performance of the CPU implementation surpasses the GPU. As the ratio increases, the latency of Griffin-GPU continues to drop, but the latency of the CPU implementation decreases faster. This is because when the length difference

between the two lists is large, performing binary search allows the algorithm to avoid decompressing large portions of the list. At this time, the modern CPUs with speculative execution and branch prediction can address the branch divergence effectively.

The value of 128 as a cross over point between GPU and CPU is supported both by our empirical measurements and by formal analysis. First, we compress 128 elements inside each data block with EF encoding. Let $R$ and $S$ be two lists where $|R| \leq |S|$. Let $\lambda = \frac{|S|}{|R|}$, then, we show that if $\lambda > 128$, it is more likely to skip unnecessary data blocks:

$$\lambda > 128 \Longleftrightarrow \frac{|S|}{|R|} > 128 \Longleftrightarrow |R| < \frac{|S|}{128}$$

In other words, when $\lambda > 128$, the number of elements in the short list ($R$) is smaller than the number of blocks in the long list ($S$). Thus, there exists at least one block in $S$ that is irrelevant. As an example in Figure 5.10, $R$ contains 5 elements and $S$ contains 8 blocks, and two elements in $R$ are mapped to the same block in $S$. As a result, there are 4 blocks (white color) in $S$ that can be skipped.

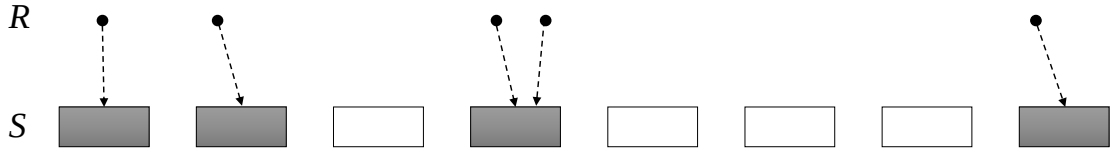



**Figure 5.10.** An example explaining the ratio choice.

Similarly, we can also show that if $\lambda \leq 128$, then all the blocks in $S$ are relevant. That is because $\lambda \leq 128 \Longleftrightarrow |R| \geq \frac{|S|}{128}$, which indicates that the elements in $R$ are likely to be mapped to all the elements in $S$.

To exploit the crossover point between the performance of the GPU and CPU implementations, Griffin uses both the GPU and the CPU to cooperatively process queries. When a query arrives in the system, the scheduler first decides if the current query is suitable for running on Griffin-GPU or on the CPU, depending on the length ratio for the

two shortest inverted lists. If the ratio is less than 128, Griffin begins execution on the GPU.

After each intersection, the system will check if the length ratio of the two lists in the next round is less than 128. If it is, processing continues on the GPU. Otherwise, Griffin transfers the intermediate results to the CPU and executes the rest of the query there.

## 5.3 Evaluation

In this section, we experimentally evaluate Griffin to answer the following questions:

- What is the performance of Griffin-GPU?
- How effective Griffin is as a hybrid GPU/CPU search engine to process queries cooperatively?
- How much can Griffin improve the tail latencies?

### 5.3.1 Methodology

In our evaluation, we run real-world queries over inverted lists generated from real web data. We compare Griffin against both a highly-optimized CPU-only implementation and GPU-only implementation with state-of-the-art algorithms. In this chapter, we assume the whole dataset resides in the main memory.

We conduct the experiments on a server with a 4-core Intel Xeon E52609V2 CPU at 2.5 GHz and with 64 GB DDR3-1600 DRAM. The server installs an NVIDIA Tesla K20 GPU with 5 GB GDDR5 memory, which connects to the server through 16 lane PCIe 2.0 with 8 GB/s bandwidth. We run Ubuntu Linux kernel 3.16.3 with CUDA Toolkit 7.0.

### 5.3.2 Benchmark

The benchmark used in our evaluation includes two parts: the queries and the web data. The queries we run are from the query logs collected from the TREC [13] 2005 and 2006 (efficiency track), which contain 150,000 real queries. The web data clueweb12 [16] is a collection of 41 million Web documents (around 300 GB) crawled in 2012. It is a standard benchmark and widely used in the information retrieval community. We parse the documents and build the inverted lists each entry of which contains a document ID and the corresponding document frequency [94].In our experiments we use 126,865 queries over about 100GB of these web documents with our limited disk space.

To have a better understanding of the benchmark, we first analyze the inverted lists and the queries we use. Figure 5.11 gives the size distribution of the inverted list involved in our experiments. Most lists are of the size between 1 $K$ elements and 1 $M$ elements. Figure 5.12 shows the distribution of the number of terms for the queries. About 27% queries contain only 2 terms, 33% contain 3, and 24% contain 4.

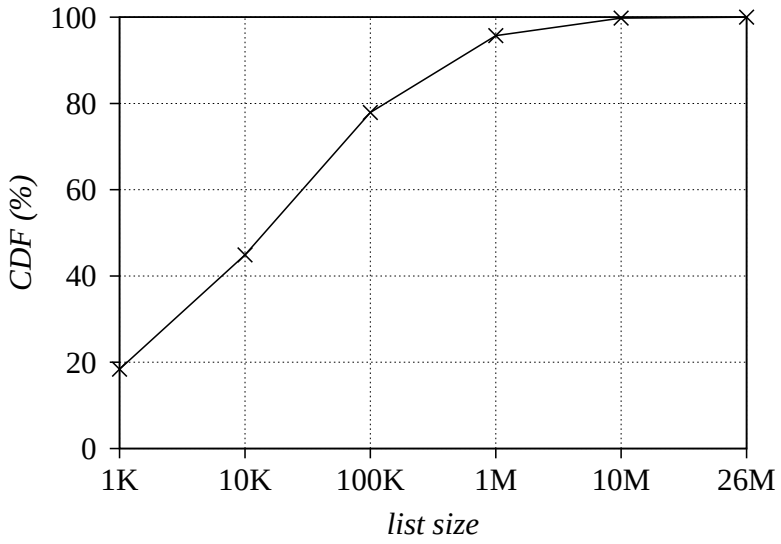



**Figure 5.11.** Inverted list size distribution.

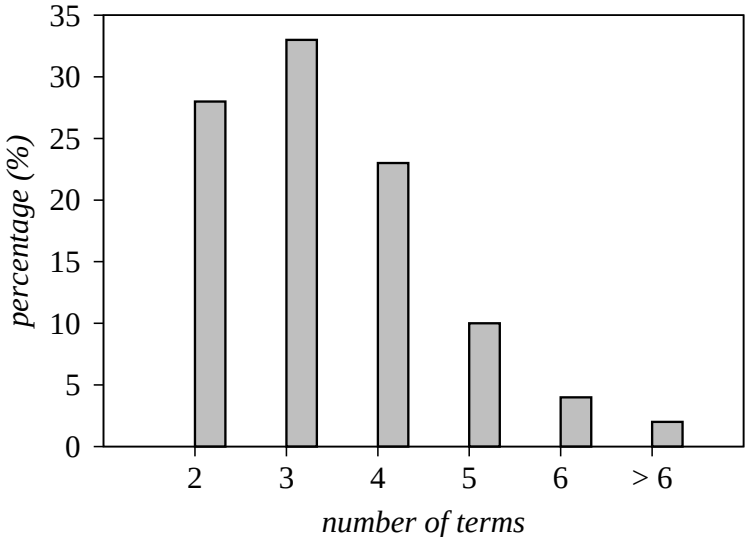



**Figure 5.12.** Number of terms distribution.

### 5.3.3 Performance of Griffin-GPU

We run several micro benchmarks to test the performance of Griffin-GPU in decompression and list intersection.

**Decompression**

Griffin-GPU implements the Para-EF decompression algorithm described in Section 3, and we compress the inverted lists into 128-element blocks. We compare the average compression ratio of the EF scheme to that of the state-of-the-art PforDelta over all the inverted lists in our tests. EF scheme can achieve an average compression ratio of 4.6, which is 1.4x better than PforDelta (Table 5.1). In addition, we also observe that the compression ratio of the EF scheme will increase with the size of the inverted list, and it can be as high as 11.6x on a list with 106 M elements.

**Table 5.1.** Compression ratio comparison.

| Scheme | PforDelta | EF |
|---|---|---|
| Compression Ratio | 3.3 | 4.6 |

To demonstrate the decompression speed of Griffin-GPU, we randomly select about 7 K lists with lengths ranging from 1 K to 10 M and group them by size. We run both the Para-EF decompression of Griffin-GPU and CPU PforDelta decompression on these lists. Figure 5.13 depicts the average decompression time for each of the groups, and shows the speedup of Griffin-GPU over the CPU PforDelta decompression. When the lists are very short (e.g., of $\sim 1K$ or $\sim 10K$ elements), the speedup is relatively low ($< 2$). With the increase in the list size, however, the speedup increases from $\sim$11x to $\sim$29.6x. There are two reasons for this result. First, a longer list with more elements is more likely to saturate the GPU with higher degree of parallelism than a short list. Second, decompressing a longer list requires more computation, and this amortizes the overhead of data transfer and GPU memory allocation.

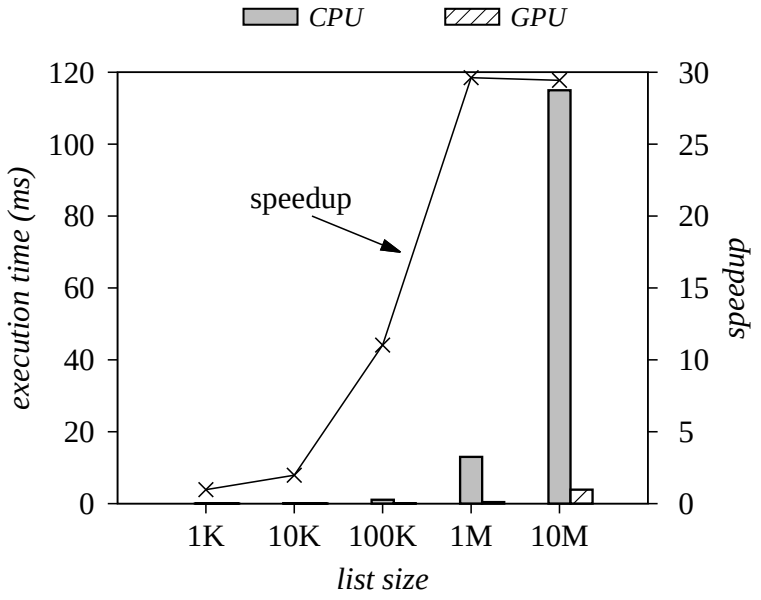



**Figure 5.13.** Decompression speed comparison.

### List Intersection

To see how Griffin-GPU's parallel merge-based intersection performs compared to CPU merge, CPU binary, and parallel binary search, we run the experiments of list

intersection with these four methods on selected pairs of lists from our dataset. The list pairs have comparable list lengths (the two lists either have similar lengths or the length of the longer list is less than 16x longer than the shorter list), and their lengths ranges from 1 K to 10 M.

Figure 5.14 shows the performance of list intersection with these different methods. We can see that with the relatively longer lists, both CPU merge and Griffin-GPU merge outperform their binary search counterparts. This is because when two lists have comparable lengths, merge-based methods can make better use of the cache and local memory of the processor. We also notice that Griffin-GPU's merge can achieve a up to 87.35x speedup over the CPU merge.

CPU binary search is slowest in these cases, while GPU binary search can achieve a speedup up to 102x over its CPU counterpart due to the parallelism. However, Griffin-GPU merge can still have up to 2.29x speedup over the very fast GPU binary search.

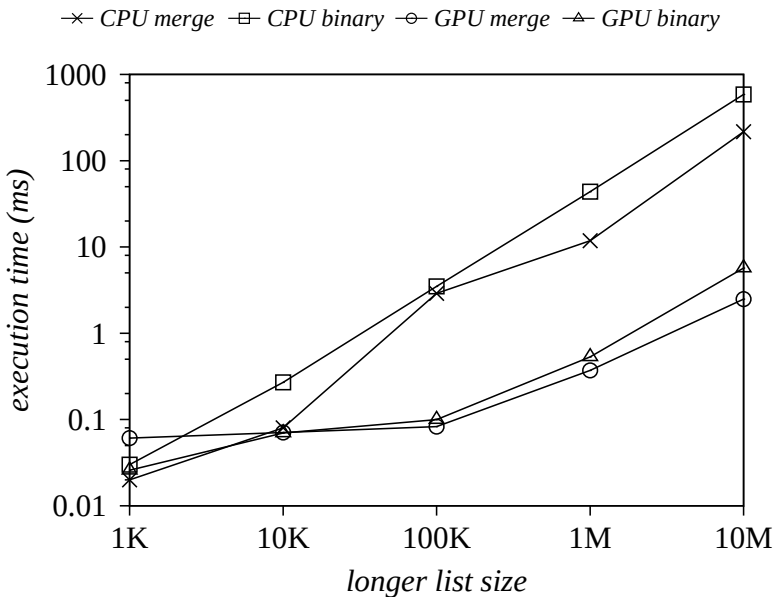



**Figure 5.14.** List Intersection Comparison.

### 5.3.4 Overall Performance

To verify Griffin's effectiveness as a hybrid GPU/CPU search engine that processes queries cooperatively, we test the end-to-end query processing latency with Griffin against a highly-optimized pure CPU search implementation and Griffin's own pure GPU implementation (Griffin-GPU). We first divide the 126,865 queries into different groups based on the number of terms present in the queries, and then run the three different configurations on each group to get the average latency. From Figure 5.15 we can see that, Griffin can consistently outperform the pure CPU search and the Griffin-GPU, with an average speedup of $\sim$ 10x and $\sim$ 1.5x, separately.

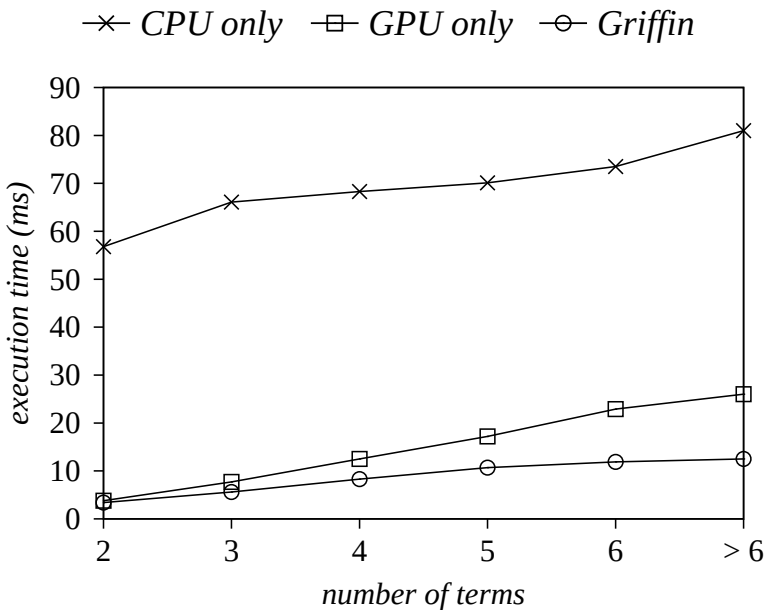



**Figure 5.15.** End-to-End query latency comparison.

### 5.3.5 Case Study: Tail Latency Reduction

Reducing tail latency is very important for interactive services like Web search, since very long tail latency will significantly affect the quality of service as well as the user experience. To see if Griffin can effectively reduce tail latency of the queries, we

compare the latency distribution of the pure CPU solution (Figure 5.16a) and Griffin (Figure 5.16b).

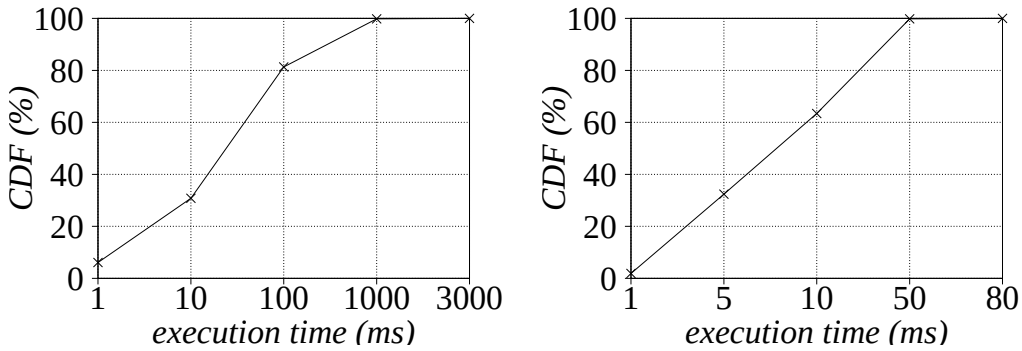



**Figure 5.16.** Query latency distribution.

Figure 5.17 compares the tail latency of the CPU search versus Griffin. As we can see, Griffin can achieve a speedup of 6.6x, 8.3x, 10.4x, 16.1x, and 26.8x, for 80th, 90th, 95th, 99th, and 99.9th percentile response time over the CPU search.

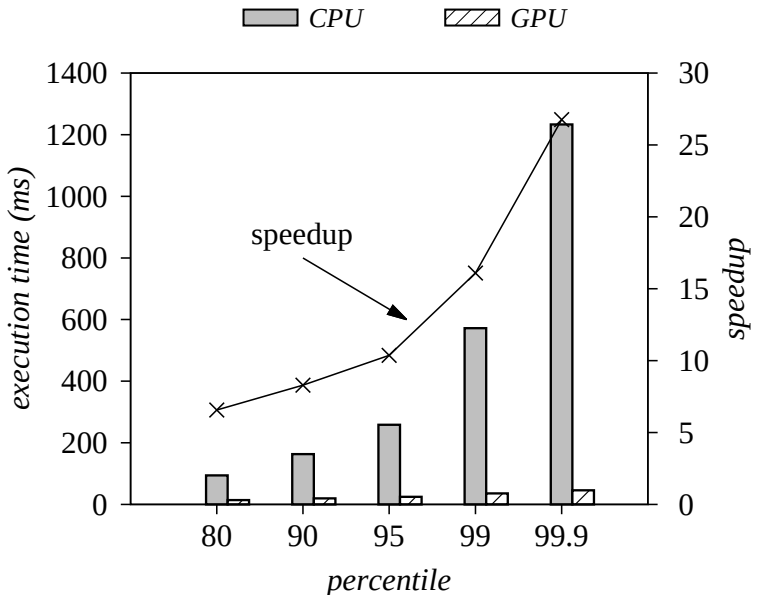



**Figure 5.17.** Tail latency reduction with Griffin.

## 5.4 Related Work

**Parallelizing query processing with GPU.** There have been some efforts in parallelizing query processing with GPU. Ding *et al.* [63] are among the first to explore the possibility of building search engines with GPU. However, the decompression algorithm they use is hard to achieve both high compression ratio and fast decompression speed at the same time, and their *Parallel Merge Find* intersection does not consider load balancing for GPU. They do propose to schedule individual queries to either GPU or CPU to improve the overall system performance. Wu *et al.* also propose a CPU-GPU cooperative computing method for list intersection [132]. But different from Griffin, both of the two only consider scheduling the whole queries, neglecting the changing characteristics of the queries during runtime.

Ao *et al.* [44] propose a new linear-regression-based compression scheme and list intersection method. But this assumes the linear properties of the datasets, and may not perform well on some datasets [44]. In addition, they also assume caching all the inverted lists in the very limited on-board memory of the GPU (in their case 1.5GB with NVIDIA GTX 480 card). This will save much overhead of data transfer between CPU and GPU, and that of GPU memory allocation. However, this may be less practical nowadays, even considering only caching the most frequently accessed data, given the rapid growing volume of data today.

**Parallelization in interactive services to reduce response latency.** Adaptive job scheduling in multiprogrammed environment has been studied [48, 49, 71, 95]. The scheduler assumes no *a priori*, and can adjust the degree of parallelism as the job executes based on the job characteristics. Similarly, Griffin can schedule part of a query to either CPU or GPU based on the changing query characteristics.

Degree of Parallelism Executive (DoPE) proposed by Raman *et al.* [107] provides an API for users to choose different parallelism options. The runtime will dynamically decide the degree of parallelism accordingly. To reduce the average response time of Web queries, Joen *et al.* [83] propose adaptive parallelism. The algorithm chooses different degrees of parallelism for requests in advance, based on the system load and the requirement of requests. Griffin could also adopt these techniques to implement flexible scheduling.

To achieve load balance among threads when applying multithreading in query processing, [122, 125, 83] explore intra-query parallel algorithms for lower response time and better load balance. While in Griffin, the merge-based intersection algorithm in Griffin-GPU can automatically achieve load balancing partitioning of indices.

To reduce tail latency, instead of directly considering the system load and workload characteristics, Jeon *et al.* [84] adopt machine learning to predict request service demand, and parallelize the predicted-to-be-long requests accordingly. Haque *et al.* [76] outperforms [84] by using *Incremental Parallelism*, which can dynamically increase parallelism to reduce tail latency. It pre-computes parallelism policy offline using information such as request service demand profiles and hardware parallelism, and adds parallelism as specified during runtime according to system load and request execution process. This method is very effective in reducing tail latency, and is complementary to Griffin. Because Griffin dynamically schedules parts of queries to CPU and GPU cooperatively, decompression and intersection of longer lists tend to run on GPU with massive parallelism and high throughput. As a result, Griffin can reduce the latency of many long queries if run on CPU. Griffin can combine the incremental parallelism techniques to further lower tail latency.

## 5.5 Summary

In the face of rapidly growing data sets and query loads in large-scale interactive information retrieval services, search engines must provide relevant results with consistent, low response times, which is very challenging. This chapter presents Griffin, a search engine that explores the intra-query parallelism by adaptively scheduling parts of a query to GPU or CPU, to reduce the average response latency. Griffin introduces two GPU algorithms in Griffin-GPU: (1) a parallel EF decompression that provides both high compression ratio and fast decompression speed; and (2) a merge-based list intersection that achieves load balancing partitioning and efficient merging. We evaluate Griffin in a big dataset with real world queries and inverted lists generated from web data. The experimental results demonstrates that Griffin's adaptive scheduling can achieve an average speedup of 10x compared to a highly-optimized CPU implementation, and 1.5x compared to Griffin-GPU. We also show that Griffin can effectively achieve a speedup of 6.6x, 8.3x, 10.4x, 16.1x, and 26.8x, for 80%, 90%, 95%, 99%, and 99.9% percentile response time over the CPU search.

## Acknowledgements

This chapter contains material that is submitted for publication as "Griffin: Improving GPU-Based Web Search with Faster Algorithms and Help From the CPU" by Yang Liu, Jianguo Wang, and Steven Swanson. The dissertation author was the first investigator and author of this paper.

# Chapter 6

# Conclusion

In the age of big data, the enormous yet ever-growing volume of data demands great compute power and I/O handling ability. Heterogeneous computing systems, especially those with GPUs, are able to deliver exceptional performance with better energy efficiency, granting us the power to deal with the huge amount of data.

This dissertation has explored the systems and algorithm support for efficient heterogeneous computing with GPUs, from three closely related aspects with three different systems, respectively. The first system, Hippogriff, targets on providing efficient I/O management and data transfer mechanisms. It identifies the importance of efficient data transfers in GPU computing, and improves the overall system performance and efficiency with two technologies. Hippogriff first enables efficient direct data transfers between the SSD and the GPU, and then dynamically chooses the best data transfer route for the GPU. By bypassing the CPU and the host DRAM for data transfers, Hippogriff achieves 1.17× speedup for single program applications, and reduces the energy consumption by 14%–17% while improving the energy-delay by 20%–26% for different data sizes. By optimizing the data transfers with automatically choosing the proper data route, Hippogriff gets an average speedup of 1.15×–1.25× for multi-program GPU workloads.

The second system, SPMario, focus on improving the system resource utilization

in the context of MapReduce. It scales up MapReduce with the GPU via optimized I/O handling and task scheduling. SPMario makes three unique observations in GPU MapReduce systems. First, simply raising the number of processes/tasks may result in severe resource contention, despite the limited performance improvement it might bring. Second, a task kernel cannot proceed before its input chunk is ready on the GPU. Third, for typical GPU MapReduce tasks, a single chunk of 64MB or 128MB is enough to saturate the I/O bandwidth, even for a PCIe SSD. Based on these observations, SPMario proposes I/O Oriented Scheduling to coordinate concurrent task execution in a way that minimizes the idle time of the system resources while avoiding I/O contention. Compared to other GPU MapReduce frameworks, SPMario is the only one that observes and solves the resource contention problem, and allows multiple processes to share the GPU efficiently.

Griffin is the third system presented in this dissertation. Griffin is a heterogeneous search engine to explore new parallel algorithms and task scheduling in heterogeneous computing with GPUs. To meet the rigorous requirement of query latency in Web search, Griffin uses a dynamic intra-query scheduling algorithm to break a query into sub-operations, and adaptively schedules them to the state-of-the-art CPU search implementation and to our new GPU-based search kernels, based on the ever-changing runtime characteristics of the queries. As far as we know, Griffin is the first heterogeneous search engine that explores the intra-query parallelism, and considers the runtime query characteristics. Griffin's adaptive scheduling can achieve an average speedup of 10x compared to a highly-optimized CPU implementation, and 1.5x compared to Griffin-GPU. Griffin can also effectively achieve a speedup of 10.4x, 16.1x, and 26.8x, for 95%, 99%, and 99.9% percentile response time over the CPU search.

In conclusion, this dissertation demonstrates we can improve the performance and efficiency of heterogeneous computing with GPUs, by utilizing the systems and algorithm

support presented in this dissertation. With our three systems, we have examined and discussed the problems that are important to the design and implementation of efficient heterogeneous computing systems with GPUs, from the underlying data transfers, to the task scheduling and resource utilization, and to the high level algorithm designs. We believe that the design and technologies uncovered and proposed in this dissertation will bring us one step further towards building heterogeneous computing systems with better performance and efficiency.

# Bibliography


[1] http://www.theregister.co.uk/2015/09/23/nine_of_the_worlds_fastest_gpu_supercomputers/.

[2] http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-phi-life-sciences-computing-paper.pdf.

[3] https://nvlabs.github.io/moderngpu/intro.html#libraries.

[4] http://wccftech.com/nvidia-pascal-volta-gpu-leaked-2017-2018/.

[5] http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf.

[6] https://hadoop.apache.org/.

[7] https://www.khronos.org/opencl/.

[8] http://docs.nvidia.com/cuda/cublas/#axzz3yw7ERpUo.

[9] https://github.com/sbates130272/libdonard/tree/master/imgrep.

[10] http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[11] https://issues.apache.org/jira/browse/YARN-2139.

[12] http://www.clusterresources.com/products/torque-resource-manager.php.

[13] http://trec.nist.gov/.

[14] https://lucene.apache.org/.

[15] http://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__INT.html#group__CUDA__MATH__INTRINSIC__INT_1g43c9c7d2b9ebf202ff1ef5769989be46.

[16] http://www.lemurproject.org/clueweb12.php.

[17] 2015 International Technology Roadmap for Semiconductors (ITRS). http://www.semiconductors.org/main/2015_international_technology_roadmap_for_semiconductors_itrs/.

[18] A Journey to Exascale Computing. http://science.energy.gov/~/media/ascr/ascac/pdf/reports/2013/SC12_Harrod.pdf.

[19] AMD Accelerated Processing Unit. http://support.amd.com/en-us/recommended/system-building.

[20] Convey Computer Tames Data Deluge, Convey Computer white paper. http://conveycomputer.com/Resources/Convey_Computer_BioIT_White_Paper.pdf.

[21] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[22] CUDA Multi Process Service Overview. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[23] CUDA Parallel Computing Platform. http://www.nvidia.com/object/cuda_home_new.html.

[24] Heterogeneous System Architecture: A Technical Review. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf.

[25] HSA Accelerated Processing Unit. http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/.

[26] MPI Forum. http://mpi-forum.org/.

[27] NVIDIA Fermi Compute Architecture Whitepaper. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[28] NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[29] NVIDIA Kepler GK110 Compute Architecture Whitepaper. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[30] Transistors Could Stop Shrinking in 2021. http://spectrum.ieee.org/semiconductors/devices/transistors-could-stop-shrinking-in-2021.

[31] Unified Memory in CUDA 6. https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

[32] Amin Abbasi, Farshad Khunjush, and Reza Azimi. A preliminary study of incorporating gpus in the hadoop framework. In *Computer Architecture and Digital Systems (CADS), 2012 16th CSI International Symposium on*, pages 178–185. IEEE, 2012.

[33] Advanced Micro Devices, Inc. FirePro DirectGMA Technical Overview. http://developer.amd.com/tools-and-sdks/graphics-development/firepro-sdk/firepro-directgma-sdk/, 2014.

[34] Sandeep R. Agrawal, Valentin Pistol, Jun Pang, John Tran, David Tarjan, and Alvin 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 '14, pages 19–34, New York, NY, USA, 2014. ACM.

[35] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 61–74, New York, NY, USA, 2012. ACM.

[36] Tolu Alabi, Jeffrey D. Blanchard, Bradley Gordon, and Russel Steinbach. Fast k-selection algorithms for graphics processing units. *J. Exp. Algorithmics*, 17:4.2:4.1–4.2:4.29, October 2012.

[37] Amber Huffman. NVM Express Revision 1.1. http://nvmexpress.org/wp-content/uploads/2013/05/NVM_Express_1_1.pdf, 2012.

[38] Amber Huffman. NVM Express Revision 1.1. http://nvmexpress.org/wp-content/uploads/2013/05/NVM_Express_1_1.pdf, 2012.

[39] AMD. AMD FirePro(TM) W9100 Workstation Graphics. http://www.amd.com/Documents/FirePro_W9100_Data_Sheet.pdf, 2014.

[40] AMD. Compute Cores White Paper. https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf, 2014.

[41] Roberto Ammendola, Massimo Bernaschi, Andrea Biagioni, Mauro Bisson, Massimiliano Fatica, Ottorino Frezza, Francesca Lo Cicero, Alessandro Lonardo, Enrico Mastrostefano, Pier Stanislao Paolucci, Davide Rossetti, Francesco Simula, Laura Tosoratto, and Piero Vicini. Gpu peer-to-peer techniques applied to a cluster interconnect. IPDPSW '13, pages 806–815, 2013.

[42] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory

caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association.

[43] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. SOSP '09, pages 1–14, 2009.

[44] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proc. VLDB Endow.*, 4(8):470–481, May 2011.

[45] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 20:1–20:13, New York, NY, USA, 2013. ACM.

[46] M. Arora, S. Nath, S. Mazumdar, S.B. Baden, and D.M. Tullsen. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE Micro*, 32(6):4–16, Nov 2012.

[47] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. ISPASS '09, pages 163–174, April 2009.

[48] Nikhil Bansal, Kedar Dhamdhere, and Amitabh Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 40(4):305–318, September 2004.

[49] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007.

[50] Can Basaran and Kyoung-Don Kang. Grex: An efficient mapreduce framework for graphics processing units. *J. Parallel Distrib. Comput.*, 73(4):522–533, April 2013.

[51] Ray Bittner, Erik Ruf, and Alessandro Forin. Direct GPU/FPGA Communication Via PCI Express. *Cluster Computing*, 17(2):339–348, June 2014.

[52] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. In *Workshop on Software Tools for MultiCore Systems*, 2008.

[53] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for*

*Programming Languages and Operating Systems*, ASPLOS '09, pages 217–228, 2009.

[54] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A Performance Study of General-purpose Applications on Graphics Processors Using CUDA. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, October 2008.

[55] Linchuan Chen and Gagan Agrawal. Optimizing mapreduce for gpus with effective shared memory usage. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 199–210, New York, NY, USA, 2012. ACM.

[56] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 25:1–25:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[57] Yi Chen, Zhi Qiao, Spencer Davis, Hai Jiang, and Kuan-Ching Li. Pipelined multi-gpu mapreduce for big-data processing. In *Computer and Information Science*, pages 231–246. Springer, 2013.

[58] Yi Chen, Zhi Qiao, Hai Jiang, Kuan-Ching Li, and Won Woo Ro. Mgmr: Multi-gpu based mapreduce. In *Grid and Pervasive Computing*, pages 433–442. Springer, 2013.

[59] J. Shane Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *TOIS*, 29(1):1–25, 2010.

[60] M. Daga, A.M. Aji, and Wu-Chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. pages 141–149, July 2011.

[61] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[62] Robert H. Dennard, Fritz H. Gaensslen, Hwa nien Yu, V. Leo Rideout, Ernest Bassous, Andre, and R. Leblanc. Design of ion-implanted mosfets with very small physical dimensions. *IEEE J. Solid-State Circuits*, page 256, 1974.

[63] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. Using graphics processors for high performance ir query processing. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 421–430, New York, NY, USA, 2009. ACM.

[64] Ismail El-Helw, Rutger Hofman, and Henri E. Bal. Scaling mapreduce vertically and horizontally. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 525–535, Piscataway, NJ, USA, 2014. IEEE Press.

[65] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[66] Khaled Elmeleegy. Piranha: Optimizing short jobs in hadoop. *Proc. VLDB Endow.*, 6(11):985–996, August 2013.

[67] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[68] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. ISCA'11, pages 365–376, 2011.

[69] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K Govindaraju. Mars: Accelerating mapreduce with graphics processors. *Parallel and Distributed Systems, IEEE Transactions on*, 22(4):608–620, 2011.

[70] Reza Farivar, Abhishek Verma, Ellick Chan, and Roy H. Campbell. Mithra: Multiple data independent tasks on a heterogeneous resource architecture. In *Cluster Computing and Workshops*, CLUSTER '09, pages 1–10. IEEE Computer Society, 2009.

[71] D. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems, 1994. Research report. IBM T.J. Watson Research Center, 1994.

[72] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 459–471, Berkeley, CA, USA, 2015. USENIX Association.

[73] Oded Green, Robert McColl, and David A. Bader. Gpu merge path: A gpu merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 331–340, New York, NY, USA, 2012. ACM.

[74] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. ISPASS '11, pages 134–144, April 2011.

[75] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop

and opencl. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1918–1927, Washington, DC, USA, 2013. IEEE Computer Society.

[76] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 161–175, New York, NY, USA, 2015. ACM.

[77] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[78] Jiong He, Mian Lu, and Bingsheng He. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endow.*, 6(10):889–900, August 2013.

[79] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: Writing parallel program portable between cpu and gpu. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.

[80] Intel. Intel(R) Solid-State Drive DC P3700 Series Specifications. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-p3700-spec.pdf, 2015.

[81] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 314–325, New York, NY, USA, 2010. ACM.

[82] J. Jenkins, J. Dinan, P. Balaji, N.F. Samatova, and R. Thakur. Enabling fast, non-contiguous gpu data movement in hybrid mpi+gpu environments. CLUSTER'12, pages 468–476, Sept 2012.

[83] Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Adaptive parallelism for web search. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 155–168, New York, NY, USA, 2013. ACM.

[84] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. In *Proceedings of the 37th International ACM SIGIR Conference on*

*Research Development in Information Retrieval*, SIGIR '14, pages 253–262, New York, NY, USA, 2014. ACM.

[85] Feng Ji and Xiaosong Ma. Using shared memory to accelerate mapreduce on graphics processing units. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 805–816, Washington, DC, USA, 2011. IEEE Computer Society.

[86] Wei Jiang and Gagan Agrawal. Mate-cg: A mapreduce-like framework for accelerating data-intensive computations on heterogeneous clusters. In *Proceedings of the 2012 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '12, pages 644–655, Shanghai, China, 2012. IEEE Computer Society.

[87] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. Gpu join processing revisited. DaMoN '12, pages 55–62, 2012.

[88] Shinpei Kato, Jason Aumiller, and Scott Brandt. Zero-copy I/O processing for low-latency GPU computing. ICCPS '13, pages 170–178, 2013.

[89] Alexey Lastovetsky. Heterogeneous parallel computing: From clusters of workstations to hierarchical hybrid platforms. *Supercomput. Front. Innov.: Int. J.*, 1(3):70–87, October 2014.

[90] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. ISCA '08, pages 315–326, 2008.

[91] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[92] D. Lustig and M. Martonosi. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. HPCA '13, pages 354–365, 2013.

[93] Siyuan Ma, Xian-He Sun, and Ioan Raicu. I/o throttling and coordination for mapreduce. *Illinois Institute of Technology*, 2012.

[94] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[95] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 11(2):146–178, May 1993.

[96] F McSherry, M Isard, and DG Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association.

[97] R. Mokhtari and M. Stumm. BigKernel – High Performance CPU-GPU Communication Pipelining for Big Data-Style Applications. IPDPS '14, pages 819–828, 2014.

[98] Gordon E. Moore. Readings in computer architecture. chapter Cramming More Components Onto Integrated Circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[99] NVIDIA. Whitepaper NVIDIA's Next Generation CUDA(TM) Compute Architecture: Kepler TM GK110/210. http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf, 2014.

[100] NVIDIA Corporation. Developing a Linux Kernel Module Using RDMA for GPUDirect. http://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf, 2014.

[101] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. Merge path - parallel merging made simple. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 1611–1618, Washington, DC, USA, 2012. IEEE Computer Society.

[102] L. Oden and H. Froning. GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters. CLUSTER'13, pages 1–8, Sept 2013.

[103] PCI-SIG. PCI-Express Specification. https://www.pcisig.com/specifications/pciexpress/.

[104] PMC-Sierra. Flashtec NVMe Controllers. http://pmcs.com/products/storage/flashtec_nvme_controllers/, 2014.

[105] S. Potluri, H. Wang, D. Bureddy, A.K. Singh, C. Rosales, and D.K. Panda. Optimizing mpi communication on multi-gpu systems using cuda inter-process communication. IPDPSW'12, pages 1848–1857, May 2012.

[106] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous System Coherence for Integrated CPU-GPU Systems. MICRO-46, pages 457–467, 2013.

[107] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. Parallelism orchestration using dope: The degree of parallelism executive. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 26–37, New York, NY, USA, 2011. ACM.

[108] Vignesh T. Ravi, Michela Becchi, Wei Jiang, Gagan Agrawal, and Srimat Chakradhar. Scheduling concurrent applications on a cluster of cpu-gpu nodes. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12, pages 140–147, Washington, DC, USA, 2012. IEEE Computer Society.

[109] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 232–241, 1994.

[110] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[111] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, pages 2:1–2:5, New York, NY, USA, 2012. ACM.

[112] M. Boyer S. Che, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. IISWC '09, pages 44–54, Oct 2009.

[113] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. Heterodoop: A mapreduce programming system for accelerator clusters. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 235–246, New York, NY, USA, 2015. ACM.

[114] Samsung. Industry's First NVMe Solid State Drive. http://www.samsung.com/global/business/semiconductor/file/product/XS1715_ProdOverview_2014_1.pdf, 2012.

[115] Michael Sevilla, Ike Nassi, Kleoni Ioannidou, Scott Brandt, and Carlos Maltzahn. A framework for an in-depth comparison of scale-up and scale-out. In *Proceedings of the 2013 International Workshop on Data-Intensive Scalable Computing Systems*, DISCS-2013, pages 13–18, New York, NY, USA, 2013. ACM.

[116] Mustafa Shihab, Karl Taht, and Myoungsoo Jung. Gpudrive: Reconsidering storage accesses for gpu acceleration. In *Workshop on Architectures and Systems for Big Data*, 2014.

[117] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Hybrid map task scheduling for gpu-based heterogeneous clusters. In *Proceedings of the 2010 IEEE*

*Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 733–740, Washington, DC, USA, 2010. IEEE Computer Society.

[118] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: Integrating a file system with gpus. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498, 2013.

[119] Amit Singhal. Modern information retrieval: a brief overview. *IEEE Data Engineering Bulletin*, 24(4):35–43, 2001.

[120] Fengguang Song and Jack Dongarra. A scalable approach to solving dense linear algebra problems on hybrid cpu-gpu systems. *Concurrency and Computation: Practice and Experience*, 2014.

[121] Stephen Bates. PROJECT DONARD: PEER-TO-PEER COMMUNICATION WITH NVM EXPRESS DEVICES. http://blog.pmcs.com/project-donard-peer-to-peer-communication-with-nvm-express-devices-part-1/, 2014.

[122] Trevor Strohman, Howard Turtle, and W. Bruce Croft. Optimization strategies for complex queries. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '05, pages 219–225, New York, NY, USA, 2005. ACM.

[123] Jeff A. Stuart and John D. Owens. Multi-gpu mapreduce on gpu clusters. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1068–1079, Washington, DC, USA, 2011. IEEE Computer Society.

[124] Yu Shyang Tan, Bu-Sung Lee, Bingsheng He, and Roy H. Campbell. A map-reduce based framework for heterogeneous processing element cluster environments. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12, pages 57–64, Washington, DC, USA, 2012. IEEE Computer Society.

[125] Shirish Tatikonda, B. Barla Cambazoglu, and Flavio P. Junqueira. Posting list intersection on multicore architectures. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 963–972, New York, NY, USA, 2011. ACM.

[126] M.B. Taylor. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. DAC'2012, pages 1131–1136, June 2012.

[127] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 205–218, New York, NY, USA, 2010. ACM.

[128] Sebastiano Vigna. Quasi-succinct indices. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 83–92, New York, NY, USA, 2013. ACM.

[129] Hao Wang, S. Potluri, D. Bureddy, C. Rosales, and D.K. Panda. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. 25(10):2595–2605, Oct 2014.

[130] Hao Wang, Sreeram Potluri, Miao Luo, AshishKumar Singh, Sayantan Sur, and DhabaleswarK. Panda. MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science - Research and Development*, 26(3-4):257–266, 2011.

[131] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[132] D. Wu, F. Zhang, N. Ao, G. Wang, X. Liu, and Jing Liu. Efficient lists intersection by cpu-gpu cooperative computing. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[133] Mengjun Xie, Kyoung-Don Kang, and Can Basaran. Moim: A multi-gpu mapreduce framework. In *Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering*, CSE '13, pages 1279–1286, Washington, DC, USA, 2013. IEEE Computer Society.

[134] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[135] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. CPU-assisted GPGPU on Fused CPU-GPU Architectures. HPCA '12, pages 1–12, 2012.

[136] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[137] Yanlong Zhai, Emmanuel Mbarushimana, Wei Li, Jing Zhang, and Ying Guo. Lit: A high performance massive data computing framework based on cpu/gpu cluster. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, CLUSTER'13, pages 1–8. IEEE, 2013.

[138] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[139] Jie Zhang, David Donofrio, John Shalf, Mahmut Kandemir, and Myoungsoo Jung. Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. PACT 2015, 2015.

[140] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.

[141] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.