INFRASTRUCTURE TO ENABLE AND EXPLOIT GPU ORCHESTRATED HIGH-THROUGHPUT STORAGE ACCESS

BY

ZAID QURESHI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

ABSTRACT

Graphics Processing Units (GPUs) have traditionally relied on the CPU to orchestrate access to the data storage. This approach is well-suited for GPU applications with known data access patterns that enable partitioning of their data-set to be processed in a pipelined fashion in the GPU. However, many emerging applications, such as graph and data analytics, recommender systems, or graph neural networks, require fine-grained, data-dependent access to storage. CPU orchestration of storage access is unsuitable for these applications due to high CPU-GPU synchronization overheads, I/O traffic amplification, and long CPU processing latencies. GPU self-orchestrated storage access avoids these overheads by removing the CPU from the storage control path and, thus, seems better-suited for these applications. However, existing system architectures and software infrastructure lack support for such GPU-orchestrated storage access.

In this work, we present a novel system architecture, BaM, that offers mechanisms for GPU code to efficiently access storage and enables GPU self-orchestrated storage access. BaM features a fined-grained, scalable software cache to coalesce data storage requests while minimizing I/O amplification effects. This software cache communicates with the storage system through high-throughput queues that enable the massive number of concurrent threads in modern GPUs to generate I/O requests at a sufficiently high rate to fully utilize the available bandwidth of the interconnect and the storage system. Furthermore, we provide array-based abstractions that not only make integrating BaM into GPU kernels trivial for programmers but also transparently optimize the number of BaM cache accesses by exploiting common GPU thread access patterns.

We evaluate the end-to-end performance and efficiency impact of each optimization for each layer of BaM's software stack with a variety of workloads with multiple data-sets. Experimental results show that GPU self-orchestrated storage access running on BaM delivers 1.04× and 1.05× end-to-end speed up for BFS and CC graph analytics. Our experiments also show GPU self-orchestrated storage access speeds up data-analytics workloads by 4.9× when running on the same hardware. In this work, we show that with carefully optimized systems software on the GPU, it is possible to use solid-state storage as a means to extend the GPU's effective memory capacity since it provides performance (up to 4.62×), cost (up to 21.8×), and I/O efficiency (up to 3.72×) benefits, even over much more expensive state-of-the-art solutions using fast DRAM, for important GPU accelerated applications.

To Ammi, Abbu, Salitah Baji, and Rakhshi,
for their constant prayers, love, and support.

ACKNOWLEDGMENTS

All praise is due to God.

I would like to thank my advisor, Professor Wen-mei Hwu, for granting me the opportunity to study and perform impactful research under his guidance. To this day, I am awestruck by Wen-mei's work ethic, efficiency, enthusiasm, compassion, optimism, insights, vision, and knowledge. I am extremely grateful for his patience and that he is always willing to listen and help in any way he can. He cares so much for his students that he puts their well-being before his concerns about their technical progress. His example continues to inspire me to strive for excellence in my academics, research, and character.I cannot possibly thank Wen-mei without thanking his other half, Sabrina Hwu, for not only enabling Wen-mei to be the advisor he is but also for looking after us.

I would also like to thank Dr. I-Hsin Chung for being an amazing mentor and for providing valuable insights and ideas ever since I met him during my internship at IBM in the summer of 2018. His vital industrial experience and knowledge, support, and advice have greatly affected the applicability and potential impact of my work.

I want to also thank the rest of my committee, Professors Josep Torrellas and David Padua, for being patient with me, providing invaluable feedback, and asking important questions, all of which have made my work stronger.

I would like to thank my academic twin, Vikram Sharma Mailthody, for being an exceptional role model, companion, collaborator, and, most importantly, a friend since day one of graduate school. He has always provided the often difficult but much-needed critique that has made our research stronger. His work ethic and efficiency motivate me to work harder. Our discussions, albeit noisy and dramatic at times, always taught me something new. I am sure our unique collaborative research style has greatly improved the impact of both of our works.

I would like to thank the members of IMPACT Research Group, both graduated and junior students. You gave me the fortune of being among some of the most intelligent, curious, and enthusiastic people. A special shout-out to Professor Izzat El Hajj, Dr. Carl Pearson, Dr. Simon Garcia De Gonzalo, Dr. David Min, Dr. Mohammad Almasri, and Professor Sitao Huang for being exceptional role models and friends. I am also grateful to Te-Chia Kao, Marie-Pierre Lassiva-Moulin, and Andrew Schuh for their help and support, shielding me from bureaucratic processes and enabling me to focus on my work.

I thank my collaborators across the university, IBM Research, and Nvidia Research. I am especially grateful to Dr. Isaac Gelado and Dr. Michael Garland from Nvidia Research for providing mentor-ship, asking the hard questions, and expecting excellence. Without their input and advice, my work would be nowhere near the same caliber.

I have had the fortune of having some of the most supportive and inspiring teachers, professors and mentors throughout my academic life. I am especially indebted to Ms. Mintz from elementary school, Mr. Kurt Bascom and Mr. Lucas Koehler from middle school, Professor Drew Youngren from high-school, Professors John Sterling, Luciano Medina and Lisa Hellerstein from my undergraduate studies, and my Masters advisor Professor Babak Falsafi for shaping me to be the student and researcher I am today.

I am grateful for my mentor and religious guide, Nasir Majeed Taib. He has been like an older brother to me for more than a decade. His religious knowledge, guidance and advice have always been practical and have helped me keep a level head in both my personal and professional lives.

I would like to thank my friends from my high school, college, and graduate school. A special shout-out to "The Crew": Marufur Bhuiya, Ali Nagi, Nathaniel Pierce Grammel, Shafat Uddin, Mohd Islam, and Emanuel Azcona. Their companionship over the years definitely made going through the tough times easier.

I am especially grateful to Nathaniel Pierce Grammel (NPG) for being a source of inspiration and setting the bar for me to reach in terms of academic excellence. I thank him for our zoom work sessions, which greatly improved my productivity. I thank him for being there when I needed professional or personal advice ever since we met at the start of our academic journeys in computer science as freshmen at NYU. I thank him for enlightening and inspiring me to pursue a Ph.D.

Finally, I cannot possibly thank my family enough for their unwavering love and support throughout my life and especially throughout my academic career. I am ever grateful to my mother for her consistent love and prayers, the amazing food, and of course, the kettles of coffee that got me through the many hours of work. I am thankful for my father for supporting me and always motivating me to be the best in whatever I pursue. I am thankful for my sisters for the laughs, support, and advice throughout my life and for being exceptional role models. I am grateful for my family's love, kindness and patience as I worked from home during the COVID-19 pandemic. I am sure it was not easy.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

There has been a phenomenal growth in GPU compute throughput in recent years. For example, as shown in Table 1.1, the compute throughput of NVIDIA GPUs has increased by 452×, from G80 to A100 over a 15-year span [1, 2]. As a result, the compute throughput of A100 (156 TF32 TFLOPS) is one to two orders of magnitude higher than its contemporary CPUs. Although the growth of memory bandwidth for GPUs is less dramatic, 18× as shown in Table 1.1, the memory bandwidth of A100 (1.555 TB/s) is about an order of magnitude higher than its contemporary CPUs. A similar trend is also exhibited across AMD GPU generations. With such levels of compute throughput and memory bandwidth, GPUs have become popular compute devices for HPC applications and dominating compute devices for neural network training.

Table 1.1: Four dimensions of advances from NVIDIA G80 to A100

Property	**NVIDIA G80 (2006)**	**NVIDIA A100 (2021)**	**A100/G80 Ratio**
Compute Throughput	0.345 SP TFLOPs	156 TF32 TFLOPs	452×
Memory Capacity	1.5 GB (GDDR)	80 GB (HBM)	53×
Memory Bandwidth	86.24 GB/s	1555 GB/s	18×
PCIe x16 Bandwidth	8 GB/s (Gen2)	32 GB/s (Gen4)	4×

Emerging high-value data-center workloads such as graph and data analytics [3, 4, 5, 6], graph neural networks (GNNs) [7, 8, 9], and recommender systems [10, 11, 12, 13, 14] can potentially benefit from the compute throughput and memory bandwidth of GPUs. However, these workloads must work with massive data structures whose sizes typically range from tens of GBs to tens of TBs today and are expected to grow rapidly in the foreseeable future. As shown in Table 1.1, the memory capacity of A100, in spite of a 53× increase from that of G80, is only 80 GB, far smaller than the desired capacity required for these workloads. The inability or difficulty for compute devices to run applications whose data structure sizes exceed their memory capacity is referred to as the memory capacity wall.

To mitigate the memory capacity wall, application developers and system designers use fast NVMe solid-state drives (SSDs) and rely on the application and the operating system (OS) running on the CPU to orchestrate the data movement between the GPUs, CPU memory, and SSDs [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. This ***CPU-centric design***, where the CPU manages the data orchestration and scheduling, works well for classic GPU applications, like dense matrix multiplication and neural network training and inference. Here, the data access patterns are pre-defined, regular, and independent of data values, enabling the CPU to partition the data into coarse-grain chunks and transfer them to accelerators for computation.

The use of coarse-grain chunks is necessary to amortize the high-software stack overhead and performs reasonably well for the above-mentioned classic GPU applications. However, when used with applications with data-dependent, irregular access patterns, the use of coarse-grain chunks in data transfer results in I/O amplification - the amount of data transferred is larger than the amount of data needed by the application.

I/O amplification can significantly reduce the bandwidth available for transferring data actually used by the compute engine. Since the available bandwidth for SSD accesses is already orders of magnitude lower than DRAM/HBM, and PCIe bandwidth is increasing at a meager pace of 4× in 15 years (see Table 1.1), such reduced effective bandwidth can severely impact the overall performance of applications. Thus, it is desirable to optimize the data transfer granularity to preserve as much of the available bandwidth for useful data, as shown in prior work [5].

Another approach is to rely on memory-mapped files and the GPU page-fault mechanism to request the CPU software stack to transfer data on-demand whenever the application attempts to access data not present in GPU memory. In this work, we refer to both alternatives, whether the CPU proactively or reactively orchestrates the data transfers for the GPU, as the **CPU-centric approaches**. Our profiling shows that CPU-centric approaches cause excessive CPU software bottlenecks and CPU/GPU synchronization overheads, resulting in poor performance (See §3.1).

To avoid the inefficiencies of the CPU-centric approaches, some state-of-the-art solutions use the host memory [5], whose capacity typically ranges from 128GB to 4TB today, or pool together multiple GPUs' memories [14] to hold the entire data-sets. We will refer to the use of host memory or pooling multiple GPUs' memory to extend the GPU memory capacity as the *DRAM-only* solution.

Extending the host memory to the level of tens of TBs is an extremely expensive proposition in the foreseeable future. Similarly, pooling together multiple GPUs' memory to extend the GPU memory to tens of TBs can also be a very expensive proposition as well, e.g., the memory capacity of each A100 GPU is only 80GB, so a 10TB pool would require 125 A100 GPUs. Furthermore, using the host memory and pooling GPU memories both require pre-loading the data-set into the extended memory. As a result, they incur long application start-up latencies, and they might still load the data that are ultimately not used due to conditions only known during the computation. We will later show with results from graph analytics that the performance of a host-memory DRAM-only solution is comparable to or lower than our proposed approach despite its much higher cost.

Proposal: We propose a novel system architecture called BaM to address the GPU's limited memory capacity. BaM capitalizes on the recent improvements in latency, throughput, cost, density, and endurance of storage devices and systems to realize another level of the accelerator memory hierarchy. The goal of BaM's design is to provide efficient, high-level abstractions for the GPU threads to easily make on-demand, fine-grain accesses to massive data structures in the storage and achieve much higher application performance than current state-of-the-art solutions.

In this work, we present, evaluate, and optimize the key components and the overall design of BaM to address three key technical challenges in efficiently using storage to increase the effective accelerator memory capacity for applications.

First, there is currently a lack of fast mechanisms for the GPU application code to generate on-demand storage accesses without involving and incurring the overheads of software on the CPU, e.g., OS page-fault handler. To address this issue, BaM provides a user-level library of highly concurrent submission/completion queues in GPU memory that enables GPU threads to make storage accesses in a high-throughput manner. We discuss key techniques to exploit the GPU's parallelism when using serial queues, e.g. NVMe queues, originally designed to be used by the less parallel CPUs.

Second, although storage devices provide more than 2 orders of magnitude of capacity compared to GPU memory, they provide significantly lower bandwidth and are accessed with much higher latency. Thus, BaM must leverage the GPU memory for data reuse and optimize the storage bandwidth utilization by submitting as few redundant requests as possible. However, there is currently a lack of GPU mechanisms for caching data in GPU memory and coalescing and issuing storage requests for needed data as GPUs have traditionally relied on the page-fault handler running the CPU for these functionalities. To fill this gap, BaM features a highly concurrent, high-throughput configurable software cache that is based on a novel, distributed self-service architecture and takes advantage of the massive memory bandwidth and atomic operation throughput of modern GPUs.

Finally, to hide the complexity of using a software cache and storage I/O queues, we provide the programmer with array-based high-level abstractions, consistent with array interfaces defined in modern programming languages (e.g., C++, Python, or Rust), to use in GPU kernels. We show that these abstractions can nearly transparently reduce BaM's overhead by exploiting and optimizing for common access patterns in GPU applications.

To our knowledge, BaM is the first ***accelerator-centric*** approach where GPUs can identify and orchestrate on-demand accesses to data where it is stored, be it memory or storage, without relying on the CPU to orchestrate the accesses. While the user-level implementation of storage device queues raises security concerns for traditional monolithic server architectures, the recent data centers' shift toward zero-trust security models that provide security guarantees through trusted hardware or software services have provided the new system framework for securing accelerator-centric storage access models like BaM [24, 25, 26].

We have built a prototype BaM system through the novel organization of off-the-shelf hardware components and development of a novel custom software stack that takes advantage of the advanced architecture features in recent GPU generations. We evaluate the BaM prototype on a variety of workloads with multiple data-sets and that BaM is on-par with a 21.7× more expensive host-memory DRAM-only solution and up to 4.62× faster than a state-of-the-art CPU-centric software solution. *Overall, our evaluation shows that with carefully optimized systems software on the GPU, it is possible to use solid-state storage as a means to extend the GPU's effective memory capacity since it provided performance, cost, and I/O efficiency benefits, even over solutions using fast DRAM, for important GPU accelerated applications.*

CHAPTER 2: BACKGROUND

Due to over a decade of improvement in compute throughput and memory bandwidth, GPUs have become the defacto compute engine for HPC, machine learning training and inference, and emerging graph and data analytics applications. As the data-sets for these applications are in the order of terabytes in size and continue to grow rapidly, they are stored on fast storage devices, like SSDs. This chapter gives an overview of the GPU architecture and programming model (§ 2.1) and describes emerging storage and non-volatile memory trends and the resulting performance characteristics (§ 2.2).

2.1 GPU ARCHITECTURE AND PROGRAMMING MODEL

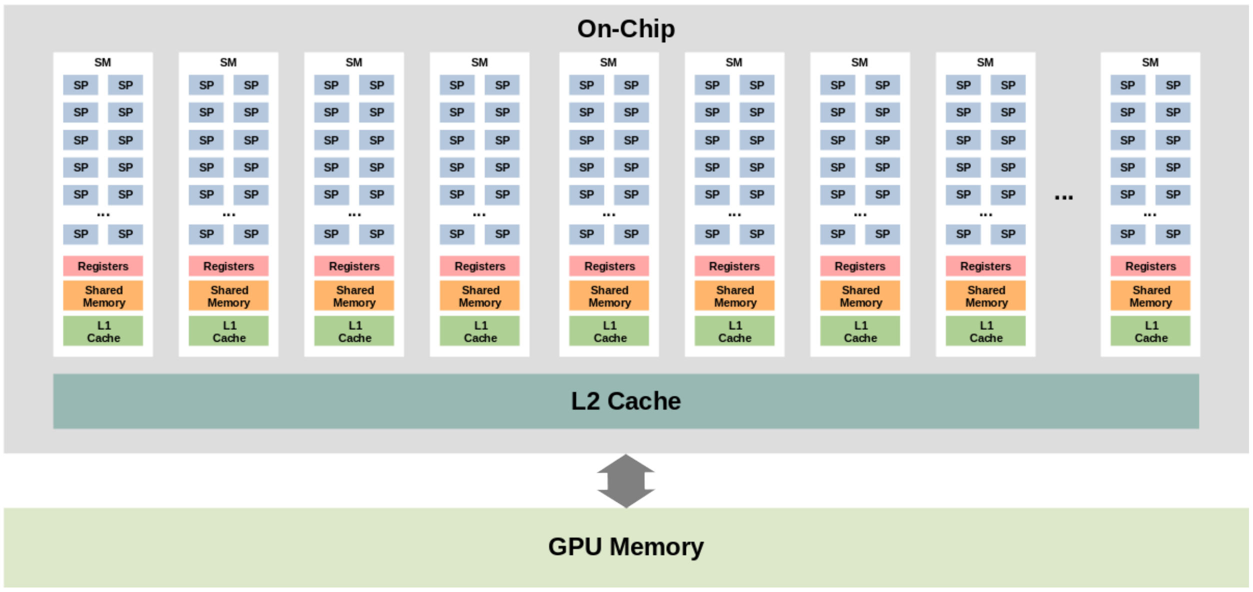


Figure 2.1: GPU architecture and memory hierarchy overview.

Whereas CPUs provide a few cores that leverage out-of-order and speculative execution to hide latencies with instruction-level parallelism, GPUs have many simple in-order cores, termed streaming processors (SPs), and leverage massive thread-level parallelism to hide latencies and provide performance. The GPU architecture, as shown in Figure 2.1, consists of many streaming multiprocessors (SMs); each of which has between 32 and 192 (SPs), a private non-coherent L1 cache, tens of kilobytes of fast registers, and tens of kilobytes of application-programmable scratchpad memory. Each SP supports a single thread of execution and at any given time an SM can have up to 2048 threads ready to be scheduled onto its SPs. SMs share an L2 cache backed by tens of gigabytes of high bandwidth global memory.

With CUDA, a GPU programmer can launch a compute kernel on the GPU with millions of threads grouped into thread blocks. The GPU schedules thread blocks on the SMs as the resource constraints permit. Once a thread block is scheduled on an SM, it remains there until all threads in the thread block have finished their execution. Threads in a thread block can synchronize and share data through a fast scratchpad memory, referred to as shared memory.

An SM further divides a thread block into warps, with each warp consisting of 32 threads. The SM uses the threads in a warp as the unit of work to schedule on the cores of the SM. Threads in a warp can synchronize and share register values with low-latency, just like lanes in traditional SIMD units. An SM can coalesce the memory accesses of a warp if its threads access the same cache-line. This enables the SM to generate larger and fewer memory requests, optimizing the GPU memory bandwidth utilization. An SM hides the latencies of memory and compute operations in warps by switching to other warps on the SM that are ready to execute. GPUs have many parallel SMs to hide latencies incurred by threads on any single SM.

2.1.1 Fine-grained memory synchronization

Since the introduction of the Volta architecture[27] and the PTX 6.0 instruction set[28] in 2017, NVIDIA GPUs and CUDA software stack have supported the C++ memory model[29, 30]. Prior to this support, if one wanted to write code where GPU threads synchronized with each other, either the undefined memory model wouldn't allow them, or the cost of synchronization would be extremely high. Now, GPU threads can synchronize among themselves using fine-grain memory reads, writes, and atomics. Furthermore, the C++ memory model allows the programmer to precisely specify the memory orderings they expect, leaving the hardware and compilers to optimize other memory operations, reordering them to hide latencies.

One of the key features of the C++, and now PTX, memory model that we leverage is that load memory operations can be marked as `acquire` and store memory operations can be marked as `release`. The definition of each is specified below.

Definition 2.1. A memory load with **acquire** ordering does not allow any subsequent memory operations, in program order within the same thread, to be reordered before it.

Definition 2.2. A memory store with **release** ordering does not allow any previous memory operations, in program order within the same thread, to be reordered after it.

In the C++ memory model, if an `acquire` load, in any thread A, reads a value in memory that was written by a `release` store operation, by any thread B, then thread A will be able to observe any and all writes, i.e., side-effects, that happened before the `release` store operation from the point of view thread B. This `acquire-release` pattern allows threads to not only communicate through atomic values but also share modified memory state with each other in a well-defined and optimized way.

In fact, CUDA GPUs support these semantics at the instruction level[28]. Loads can be marked with `weak`, i.e., no ordering guarantees, `relaxed`, i.e., program order guaranteed with other `relaxed` operations, and `acquire` orderings. Stores can be marked with `weak`, `relaxed`, and `release` orderings. Atomic operations, or read-modify-writes, can specify `relaxed` ordering, or specify `acquire` ordering on the read, or specify `release` ordering on the write, or use `acq_rel` to specify both the read as `acquire` and write as `release`.

2.2 NON-VOLATILE MEMORY AND STORAGE

Table 2.1: Comparison of different types of SSDs with DRAM DIMM.

Technology	**Product**	**RD IOPs (512B, 4KB)**	**WR IOPs (512B, 4KB)**	**Latency (μs)**	**DWPD**	**$/GB**
DRAM	DIMM (DDR4)	>10M	>10M	0.1	>1000	11.13
Optane [31]	Intel P5800X	5.1M, 1.5M	1M, 1.5M	10	100	2.54
Z-NAND [32]	Samsung P1735	1.1M, 1.6M	351K, 351K	25	3	2.56
NAND Flash [33]	Samsung 980 Pro	750K, 750K	172K, 172K	100	0.3	0.51

2.2.1 Improvement in Non-volatile memory technology

Non-volatile storage technology has come a long way from spinning hard drives. Modern non-volatile storage devices exploit great parallelism, similar to DRAM, in order to provide performance within 1-2 orders of magnitude when compared to DRAM while coming in at a fraction of the cost, as shown in Table 2.1. As the device latency continues to reduce, as a result of technological advancements like Optane or Z-NAND media, the software overhead is becoming a significant fraction of overall access latency. Figure 2.2 shows the latency breakdown of an I/O request from a highly optimized CPU software stack, `io_uring`, to three NVMe SSDs: a high-end consumer-grade SSD (Samsung 980 Pro) and two high-end ultra-low latency data-center grade SSDs (Samsung DC 1735 and Intel Optane P5800X). *As*

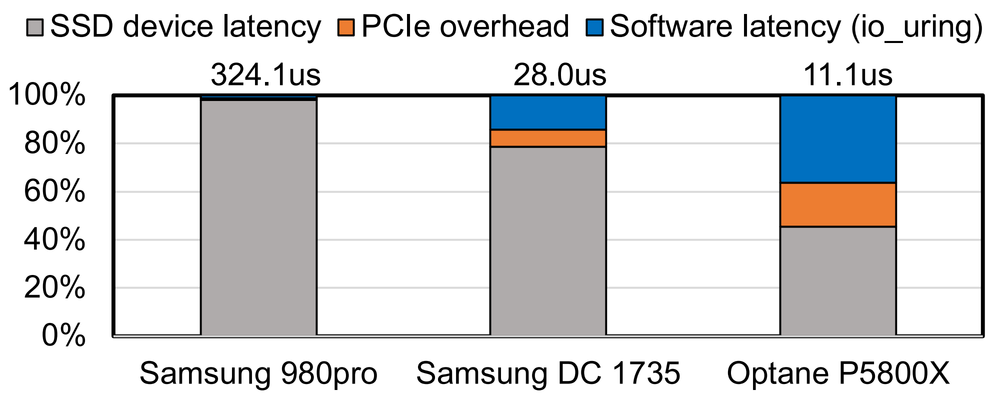


Figure 2.2: Latency breakdown of accessing a 4KB block from the storage using the CPU and `io_uring` for 3 different SSDs. Each bar is labeled with the total access latency in microseconds.

device latency decreases, the software overhead becomes a significant fraction, up to 36.4%, of the total storage access latency.

2.2.2 Storage I/O Protocol

To address this shift, emerging storage systems allow applications to make direct user-level I/O accesses to storage [26, 34, 35, 36, 37, 38, 39, 40, 41], reducing the access overhead. The storage system allocates user-level queue pairs, akin to NVMe I/O submission (SQ) and completion (CQ) queues, which the application threads can use to enqueue requests and poll for their completion. Using queues to communicate with storage systems forgoes the userspace to kernel crossing of traditional file system access system calls. Instead, isolation and other file system properties are provided through trusted services running as trusted user-level processes, kernel threads, or even storage system firmware running on the storage server/controller [26, 34, 35, 41].

NVMe as an example: The NVMe [42] protocol is the latest standard defined by the industry to enable high-throughput access for both server and consumer-grade SSDs. In contrast to traditional storage I/O interfaces that provide one pair of command submission and command completion queues, the NVMe protocol allows up to 64K parallel submission (SQ) and completion (CQ) queues, each with 64K entries, per device. The NVMe device driver allocates a pool of buffers in the memory for use by the SSDs' DMA engines for read and write requests. These queues and buffers traditionally reside in the system memory in the CPU-centric model.

An application accessing storage causes the driver to allocate a buffer from the I/O buffer pool for the request and enqueue an NVMe I/O command at the tail of an SQ with a unique command identifier. The driver then writes the new tail value to the specific SQ's write-only register in the NVMe SSD's BAR space, i.e., it rings the queue's doorbell. For improved efficiency, a driver can ring the doorbell once after enqueuing multiple requests into an SQ.

For a read request, the SSD device controller accesses its media and delivers the data into the assigned buffer using its DMA engine. For a written request, the SSD device controller DMAs the data in the buffer into its media. Once a request is serviced, the SSD device controller inserts an entry into the CQ. When the host driver detects that a valid CQ entry for a submitted command is in place, it retires the request and frees up the allocated buffers and the space in the queue for the request. We describe how the driver detects a valid CQ entry later. The completion entry also notifies the driver of how many entries in the SQ are consumed by the NVMe controller. The driver uses this information to free up space in the SQ for new I/O requests. To communicate forward progress, the driver rings the CQ doorbell with the new CQ head. For efficiency, an SSD device can insert CQ entries for multiple requests in one transaction.

Detecting a valid CQ entry for a submitted command requires the entry to contain a command identifier for a command submitted and the valid phase bit. The phase bit is the controller's mechanism to notify the driver of new CQ entries in a circular queue. Initially, since the queue is zeroed out at allocation, the driver expects the controller to flip the phase bit of a CQ entry from zero to one to notify that the entry is valid. After consuming valid CQ entries and moving the head across all entries in the queue, the driver knows that the phase bits for all entries in the CQ are flipped to one. So now when it wraps around the queue, it expects the controller to flip the phase bit of a CQ entry from one to zero to mark the entry as valid. Note, each time the head of the CQ wraps around the queue, the expected phase bit for valid CQ entries is reversed.

2.2.3 Amortizing Storage Access Overhead

The parallelism required to tolerate access latency and achieve full throughput of the storage device is fundamentally governed by Little's Law: $T \times L = Q_d$, where T is the target throughput, L is the average latency, and Q_d is the minimal queue depth required at any given point in time to sustain the throughput. If we want to achieve the full potential of the critical resource, PCIe $\times$16 Gen4 connection providing $\sim$26GBps of bandwidth, then T is $26GBps/512B = 51M/sec$ and $26GBps/4KB = 6.35M/sec$ for 512B and 4KB access granularities, respectively. The average latency, L, depends on the SSD devices used and it is $11us$ and $324us$ for the Intel Optane and Samsung 980 Pro SSDs, respectively. From Little's

Law, to sustain a desired 51M accesses of 512B each, the system needs to accommodate a queue depth of $51M/s \times 11us = 561$ requests (70 requests for 4KB) for Optane SSDs. For the Samsung 980 Pro SSDs, the required Q_d for sustaining the same target throughput is $51M \times 324us = 16,524$ (2057 for 4KB). Note that Q_d can be spread across multiple physical device queues. To sustain T over a computation phase, there needs to be a substantially higher number of concurrently serviceable access requests than Q_d over time.

The emerging queue-based storage systems [26, 35, 41] present major challenges to sustaining T in massively parallel execution paradigms. Take as an example using NVMe queues to make I/O requests. After requests are enqueued into an NVMe SQ, the queue's doorbell must be rung with the updated queue tail to notify the storage controller of the new request(s). As these doorbell registers are write-only, when a thread rings a doorbell, it must make sure that no other thread is writing to the same register and that the value it is writing is valid and is a newer value than any value written to that register before. Implementing queue insertion as a critical section, while simple, imposes significant serialization latency, which might be fine for the CPU's limited parallelism but can cause substantial overhead for thousands of GPU threads. Although the serialization overheads can be avoided by having many parallel queues, recall the NVMe spec supports 64K submission queues, due to storage controller cost, form factor, memory, and power constraints, real devices generally have a limited number of submission queues, with the Intel Optane P5800X supporting the most at 135 submission queues [43].

CHAPTER 3: MOTIVATION

To overcome the limited GPU memory capacity, state-of-the-art systems have to either (1) rely on complex application code to partition and proactively load data from storage to GPU memory in tiles, (2) provide transparent mechanisms that re-actively access storage to service page faults from the GPU, or (3) load the entire data-set into abundant CPU or pooled multi-GPU memory from where the GPU threads can directly access it. This chapter discusses the pitfalls of these state-of-the-art systems in the face of emerging applications and their large data-sets.

3.1 PROACTIVE TILING

A common approach to use GPUs for processing data-sets that don't fit in GPU memory requires the programmer to explicitly decompose and partition the data and computation into tiles that fit into the GPU memory. The CPU application code orchestrates data movement between the storage and the GPU memory to *proactively* pre-load tiles into GPU memory, launches compute kernels for each tile, and aggregates the results from processing the individual tiles.

Although proactive tiling works well for some classical GPU applications with predefined, regular, and dense access patterns, its proactive accesses to storage are problematic for emerging applications with dynamic, data-dependent, irregular access patterns, such as data analytics. The execution time overhead of synchronization, CPU orchestration, and I/O access latency compels the programmer to resort to coarse-grain tiles, which exacerbates I/O amplification.

Take executing analytics queries on the NYC taxi ride data-set as an example. The NYC Taxi data-set consists of 1.7 billion taxi trip records from 2009 to 2021. It is organized as a table where each row is a trip record and each column stores a metric such as pickup location, distance, total fare amount, taxes, surcharge, etc.

Suppose we ask the question, "Q1: What is the average cost/mile for trips that are at least 30 miles?". As the data-set has almost 2 billion rows and exceeds the GPU memory capacity, the programmer must process the data in smaller row groups that fit in GPU memory by (1) finding and loading each row group (i.e., tile) into the GPU memory, (2) performing the query on the GPU over the row group, and (3) aggregating results across row groups. During query computation over the row group, the trip distance column of the row group is scanned, and, for trips that meet the criteria of being at least 30 miles, the

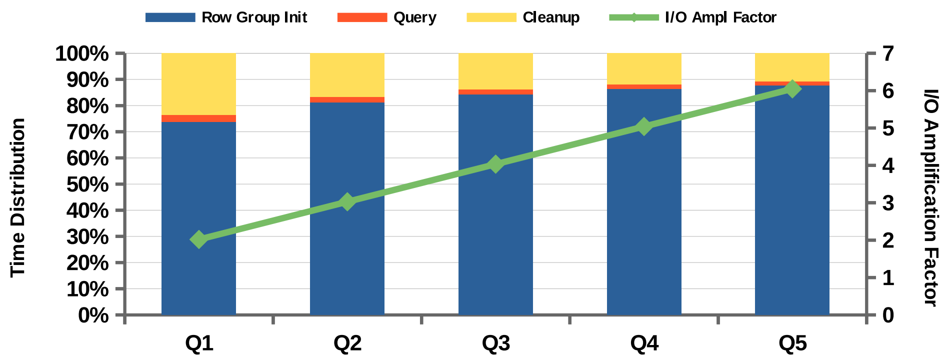


Figure 3.1: Execution time breakdown and I/O amplification in GPU-accelerated data analytics application using the state-of-the-art RAPIDS [6] system.

total cost value of the corresponding row must be aggregated. Figure 3.1 shows the profiling result for executing this query on the state-of-the-art GPU accelerated data analytics framework, RAPIDSv21.12 [6]. The CPU code to initialize the row group, which involves finding, allocating memory for and loading each row group of the columnar metric arrays into GPU memory, and the CPU code to clean up the row group accounts for more than 73% and 23%, respectively, of the end-to-end application time, reflecting the high driver and synchronization overhead.

Furthermore, since accesses to the trip distance column are dependent on values in the pickup location column, the CPU cannot determine which trip distance rows are required. Thus, to leverage the bandwidth of the storage, RAPIDS fetches all rows of both columns into the GPU's memory for each row group. As only 511k trips are at least 30 miles, only 0.03% of the second column is used. Thus, the tiling approach incurs 2.02× I/O traffic amplification for this query.

The query can be further extended to answer a more interesting question: "Q5: What is the average \$/mile the driver makes for trips that are at least 30 miles?". To answer this query we need 4 more data-dependent metrics from the data-set and for each metric added, we create a new intermediate query. We have to add the surcharges (Q2), hail fee (Q3), tolls (Q4), and taxes (Q5) metrics to get the penultimate query. But doing so *linearly scales the I/O amplification suffered by the CPU-centric model to over 6*× as the number of data-dependent metrics increases, as shown in Figure 3.1.

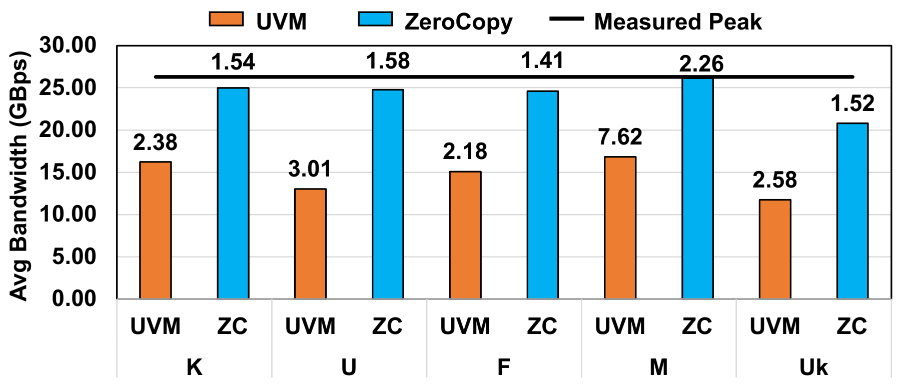


Figure 3.2: Average bandwidth in GBps (bar graph) and execution time in seconds (numbers) when running BFS graph traversal using UVM and Zerocopy mechanism. UVM page-faulting scheme adds significant overhead.

3.2 REACTIVE PAGE FAULTS

Some applications, such as graph traversal, do not have ways to cleanly partition their data-sets and thus prefer keeping whole data structures in the GPU's address space. For example, assume that a graph is represented in the popular compressed sparse row (CSR) format, where the neighbor-lists of all nodes are concatenated into one large edge-list array. An array of offsets accompanies the edge-list, where the value at index `i` specifies the starting offset for the neighbor-list of node `i` in the large edge-list. Since any node, and its corresponding neighbor-list, can be visited while traversing a graph, traversal algorithms prefer to keep the entire edge-list in the GPU's address space [5].

Starting with the NVIDIA Pascal architecture, GPU drivers and programming model allow the GPU threads to access large virtual memory objects that may partly reside in the host memory using Unified Virtual Memory (UVM) abstraction [44]. Prior work shows that the UVM driver can be extended to interface with the file system layer to access memory-mapped files [45]. This enables a GPU to generate a page fault for data that is not in GPU memory, which the UVM driver *reactively* services by making I/O request(s) for the requested pages on storage.

However, this approach introduces significant software overhead in page fault handling mechanisms when the accessed data is missing from GPU memory. Figure 3.2 shows the measured host-memory-to-GPU-memory data transfer bandwidth for an A100 GPU in a PCIe Gen4 system executing BFS graph traversal on six different data-sets (See Table 9.1) where the edge-lists are in the UVM address space and initially in the host memory. Note that there is no storage access in this experiment and the measured data bandwidth is an upper bound of what a page-fault-based approach might potentially achieve.

From Figure 3.2, the average PCIe data transfer bandwidth achieved by the UVM page faulting mechanism is ∼14.52GBps which is only 55.2% of the measured peak PCIe Gen4 bandwidth. Profiling data shows that the UVM fault handler on the CPU is 100% utilized and the maximum UVM page fault handling rate saturates at ∼500K IOPs. Such a rate can't even saturate the throughput of consumer-grade SSDs, as noted in Table 2.1. With these limitations, the UVM page fault handling mechanism simply cannot fully utilize the bandwidth of even one consumer-grade SSD for page sizes of 8K or smaller.

3.3 SCALING MEMORY

To avoid page faulting completely, applications can leverage CPU memory or even pool together the memory of multiple GPUs to host large data structures. In the graph traversal example, the programmer allocates pinned (or zero-copy) buffers in these memories, loads the edge-list data from storage into the buffers, and maps the buffers into the address space of the GPU(s) executing the computation. The GPU threads can then access these buffers in 128-byte cache-line granularity and the GPU provides sufficient memory-level parallelism to saturate the interconnect between the GPU and these memories and can out-perform the UVM solution for important graph traversal applications, as shown in Figure 3.2 and by previous works [5].

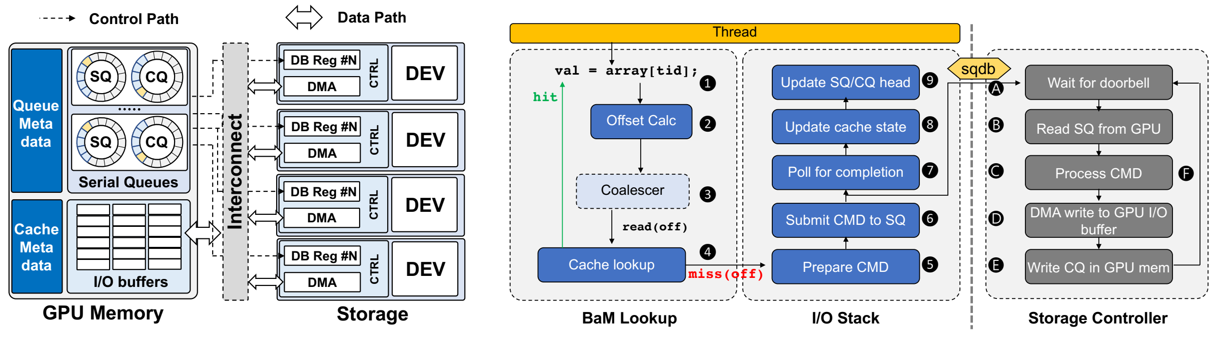
However, regardless of which memory, CPU or pooled GPU, is used to host these data structures, this approach suffers from two major pitfalls. First, *data must still be loaded from the storage to the memory before any GPU computation can start.* Often this initial data loading can be the main performance bottleneck, as CPU and pooled GPU memory can be accessed at much higher bandwidth and lower latency than the storage (see Target (T) system in § 9.5.1). Second, *hosting the data-set in CPU or pooled GPU memory requires scaling the available memory*, by either increasing the CPU DRAM size or the number of GPUs in the system, with the data-set size, *and thus can be prohibitively expensive for massive data-sets.*

CHAPTER 4: INCREASING THE EFFECTIVE GPU MEMORY CAPACITY WITH STORAGE

BaM proposes an accelerator-centric computing model in which GPU threads can directly access data where it is stored, be it memory or storage, increasing the GPU's effective addressable memory capacity. To this end, BaM's design provides high-level abstractions for accelerators to make on-demand, high-throughput access to storage while enhancing the storage access performance. BaM leverages the massive parallelism GPUs expose to applications to naturally maintain the queue depths needed to saturate the storage bandwidth. BaM provisions storage I/O queues and buffers in the GPU memory and maps the storage doorbell registers to the GPU address space. Although doing so enables the GPU threads to access terabytes of data on storage, BaM must address three key challenges in providing an efficient and effective solution.
Challenge 1: As storage protocols and devices exhibit significant latency, BaM must leverage the GPU parallelism to keep many requests in flight, efficiently tolerate such latency, and overlap it with computation. To this end, BaM I/O stack is designed to reduce the size of and contention on critical sections so that many parallel threads can efficiently enqueue I/O requests, poll for their completion, and clean up I/O queues for forward progress.
Challenge 2: As storage devices have relatively low bandwidth and GPUs have limited memory capacity, BaM must optimize the utilization of these resources. BaM's software-managed cache in the GPU memory supports highly concurrent accesses by many threads, keeps at most one copy of any particular block of data from storage, allows threads to reuse fetched bytes in the fast GPU memory, and reduces the I/O traffic.
Challenge 3: As GPU kernels generally do not expect to manage a cache and make storage accesses, BaM's high-level abstractions must hide BaM's complexity and make it easy for the programmer to integrate BaM into their GPU kernels.

4.1 BAM OVERVIEW

Figure 4.1a shows an overview of the BaM system architecture. We provision I/O queues for storage devices in the GPU memory, enabling GPU threads to make I/O accesses.A BaM cache is initialized in the GPU memory allowing GPU threads to efficiently cache data blocks fetched from the storage devices. BaM also provides the `bam::array` high-level programming abstraction that presents a memory-like interface over the cache and storage, making it easy for programmers to use BaM in their existing GPU applications. An application can call BaM host APIs to initialize the `bam::array` with data backed by storage, akin to `mmap`'ing



(a) Logical view of BaM design. (b) Life of a GPU thread accessing data through BaM.

Figure 4.1: Logical view of the BaM system and the life of a GPU thread using BaM to access data.

a file.

Figure 4.1b shows how a GPU thread uses BaM to access data. When a GPU thread makes a data access with the `bam::array` abstraction $\boxed{1}$, it uses the abstraction to determine the offset, i.e. cache-line, for the data being accessed $\boxed{2}$. The abstraction allows threads in a warp to coalesce their accesses $\boxed{3}$ if multiple threads access the same cache-line. For each unique cache-line being accessed, a single thread probes the BaM software cache $\boxed{4}$ with the offset on behalf of the rest of the threads improving cache access efficiency.

If an access hits in the cache, the thread can directly access the data in GPU memory. If the access misses, the thread needs to fetch data from the storage devices using the I/O queues in the GPU's memory. The BaM software cache is designed to optimize the bandwidth utilization of the storage devices by eliminating redundant requests to the backing memory

If the GPU thread needs to fetch data from the storage devices, the GPU thread enters the BaM I/O stack to prepare a storage I/O request $\boxed{5}$, enqueue it to a submission queue $\boxed{6}$, and then waits for the storage controller to post the corresponding completion entry $\boxed{7}$. The BaM I/O stack aims to amortize the software overhead associated with the storage submission/completion protocol by leveraging the GPU's immense thread-level parallelism, and enabling low-latency batching of multiple submission/completion (SQ/CQ) queue entries to minimize the cost of expensive doorbell register updates and reduce the critical sections in the storage protocol.

On receiving the doorbell update $\boxed{A}$, the storage controller fetches the corresponding SQ entries $\boxed{B}$, processes the command $\boxed{C}$ and transfers the data between SSD and the GPU memory $\boxed{D}$. At the end of the transfer, the storage controller posts a completion entry in the CQ $\boxed{E}$. After the completion entry is posted, the thread updates the cache state $\boxed{8}$ for the key, updates the SQ/CQ state $\boxed{9}$ and then can access the fetched data in GPU memory. BaM allows any GPU thread to access any data supported by BaM, just like memory, without any synchronization or access granularity requirements.

4.2 COMPARISON WITH CPU-CENTRIC SYSTEMS

Now we discuss how BaM compares to and overcomes the pitfalls of the CPU-centric systems for addressing the GPU's limited memory capacity.

4.2.1 Proactive Tiling

When compared to the proactive tiling CPU-centric approach, BaM has three main advantages. First, with proactive tiling, the CPU ends up copying data between the storage and GPU memory and launching compute kernels multiple times to cover a large data-set. This is done with driver code that has very limited thread-level parallelism and thus limited performance. Furthermore, each kernel launch and termination incurs costly synchronization between the CPU and the GPU. Since BaM allows GPU threads to both compute and fetch data from storage, the GPU doesn't need to synchronize with the CPU as frequently, and more work can be done in a single GPU kernel. This shift overlaps storage access latency suffered by some threads with the compute of other threads, thus improving the overall performance.

Second, because the compute is offloaded to the GPU and the data orchestration is managed by the CPU in proactive tiling, it is difficult for the CPU to determine which parts of the data are needed and when they are needed, thus it ends up fetching many unneeded bytes. With BaM, a GPU thread fetches the specific data it needs only when it requires it, reducing the I/O amplification overheads from which proactive tiling suffers.

Third, in proactive tiling, programmers expend effort to partition the application's data and overlap compute with data transfers to hide storage access latency. BaM enables the programmer to naturally access the data through the array abstraction and harness GPU thread parallelism across large data-sets to hide the storage access latency.

4.2.2 Reactive Page-Faults

Compared to the reactive page-fault approach[44], BaM does not rely on software on the CPU to manage the GPU memory or orchestrate storage accesses and data movement. As explained in §3.2, the reactive page-fault approach severely bottlenecks overall performance due to the meager throughput of the page-fault handler running in the operating system on the CPU. Recall that the current UVM page-fault handler cannot process page-fault fast enough to saturate the throughput of even a single consumer-grade SSD.

BaM leverages the massive GPU parallelism to manage a fixed-sized software-managed cache in GPU with high concurrency. GPU threads can probe the cache for their needed data and access the data from the GPU if the access is a hit, completely in parallel. On a cache miss, the requesting GPU thread can directly fetch the needed data from the backing storage using BaM's highly-concurrent I/O stack. BaM's software-managed cache and ability to make high-throughput I/O requests completely forgoes the page-fault mechanism and thus avoids its limitations. Furthermore, BaM frees up the CPU and interconnects between CPU and GPU so that they can be leveraged for other purposes.

4.2.3 GPUDirect Storage

GPUDirect Storage [15] is a great way to optimize data movement in a CPU-centric system. It allows application code running on the CPU to request that data be directly transferred between the GPU memory and a file on an SSD using Direct Memory Access (DMA) to/from the GPU memory. This reduces extra data copies and system interconnect traffic imposed by the bounce buffers needed in CPU memory for systems that do not use GPUDirect Storage. Prior works have shown the resulting performance benefits of such a system for different workloads [16, 20, 21, 46]. In BaM, we enable the same direct data path between GPUs and SSDs. However, we also move the control, i.e. orchestration, to the GPU as well. We do this in order to leverage the GPU's massive parallelism to expose more I/O parallelism, hide storage access latency, and improve software stack throughput.

4.3 COMPARISON WITH PRIOR ACCELERATOR-CENTRIC APPROACHES

Prior works have attempted to bring system calls [47] and file system file access APIs [17, 18, 47] to the GPU threads. These works use the CPU as a proxy for executing these API requests on behalf of the GPU threads, often just translating the request to the appropriate POSIX system call. Since files are a way of implementing interprocess communication in UNIX, these works bring the capability to the GPU. Although these systems enable the GPU threads to initiate storage accesses, they incur overheads when accessing data as (1) the CPU is used to process requests for the significantly more parallel GPU, (2) the accesses incur system call overhead, which can be significant compared to fast storage access latency of modern devices, and (3) the data in GPU memory and the operating system page cache in the CPU must be kept consistent for the abstractions to work. Furthermore, the GPU APIs impose restrictions on synchronization barriers, limiting the utility of the system to very specific application data access patterns. Although more recent work [17] has enabled more memory-map-like abstraction on top of these systems, it still suffers from the performance and functionality limitations of the underlying system.

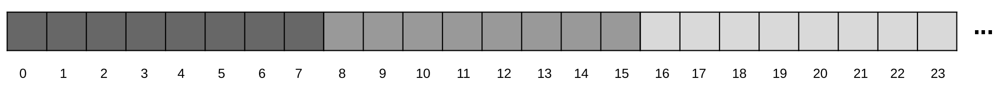
In BaM we provide a memory abstraction to the GPU to allow any GPU thread to read and write to any part of the data backed by BaM. This abstraction is consistent with how GPU applications are written to access data in the GPU memory; so porting GPU kernels to BaM is often trivial. Furthermore, generally, GPU's kernels don't use file abstractions for inter-process communication, thus, although enabling such a feature is impressive, the performance impact limits the effectiveness of using storage as a memory extension. BaM's goal is to use storage like a memory, so it must aim to achieve and expose peak storage access bandwidth for applications.

CHAPTER 5: EXPLOITING PARALLELISM FOR SERIAL I/O QUEUES

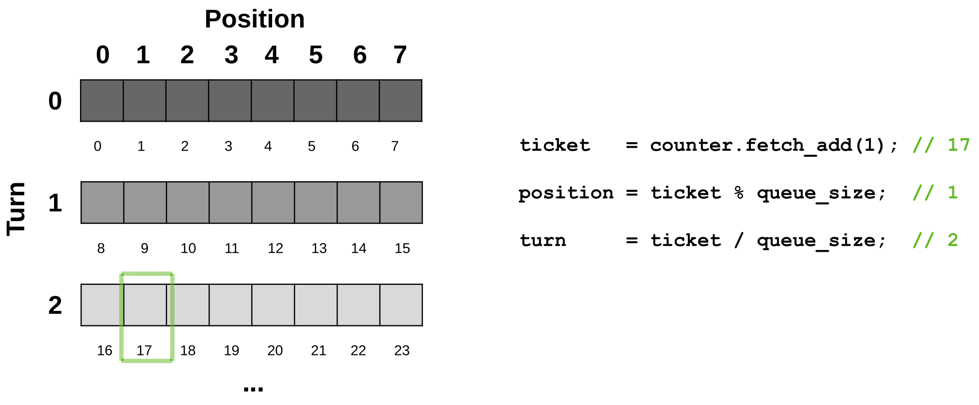
BaM leverages the GPU's massive thread-level parallelism and fast hardware scheduling to maintain the queue depths needed to hide storage access latency and to fully utilize the storage and interconnect throughput. However, ringing doorbells, say after enqueuing commands or cleaning up submission queue (SQ) entries, in the existing storage I/O protocols requires serialization. A critical section that encompasses the process of enqueuing a command and ringing the doorbell, albeit simple, would result in poor throughput and significant serialization latency when thousands of threads enqueue I/O requests concurrently. Instead, BaM uses fine-grain memory synchronization, enabling many threads to enqueue I/O requests in the SQ, poll the collection queue (CQ), or clean up the SQ and CQ in parallel. Threads enter critical sections for moving the SQ's tail and CQ's head and ringing the doorbell. Since threads that enter a critical section move the head or tail as far as possible, the overhead of these critical sections is amortized over multiple requests. In this section, we describe how BaM uses fine-grain memory synchronization to enable many threads to enqueue to and dequeue from NVMe queues in parallel.

5.1 ENQUEUING NVME REQUESTS

As NVMe queues are circular, for a physical queue of size `qs` we can map any particular instance of it to a contiguous segment of size `qs` in an infinitely sized virtual queue, as shown in Figure 5.1a. Enqueuing into the virtual queue implies picking a position in the physical queue and a turn on that position. We can get a position in the virtual queue by atomically incrementing a global counter. The returned `ticket` value is an index into a virtual queue and can be divided by the physical queue size to assign a `position` in the physical queue, the remainder, and the `turn` on that position, the quotient, as shown in Figure 5.1b. The `turn` value effectively creates `qs` number of parallel queues, each of size one, out of the physical queue entries. All incoming requests that receive the same `position` value keep monitoring the Turn Counter value of their position in the physical queue until the Turn Counter value becomes the `turn` value they received. The matching between the Turn Counter value of a position and the `turn` value received by an incoming request assigned to the position signals that the incoming request can be safely inserted into the physical queue position. In this way, the `turn` values received by the incoming requests are used to enforce an unambiguous total order for all requests that need to go into each position in the physical queue. The use of `turn` values allows the decoupling of identifying the physical queue position for an incoming



(a) Virtually infinite queue for a physical queue of size 8.



```
ticket   = counter.fetch_add(1); // 17
position = ticket % queue_size;  // 1
turn     = ticket / queue_size;  // 2
```

(b) Mapping the virtual queue to entries in the physical queue.

Figure 5.1: Physical circular queue virtualized as an infinite queue.

request from the actual insertion of the incoming request into the assigned position. The atomic operations performed for inserting the incoming requests are distributed across the `qs` Turn Counters, which improves the throughput of enqueuing incoming requests.

Figure 5.2 illustrates the use of the position-turn technique to order I/O requests from many GPU threads onto a fixed-size physical NVMe submission queue. As shown in step $\boxed{0}$, threads first atomically increment the ticket counter to get a place in the virtual queue. The returned `ticket` can be divided by the physical queue size to assign each thread an `position` in the physical queue and its `turn` on that position. Each thread uses its `position` to index into the `turn_counter` array, and polls on the location, using an acquire-load, until the counter equals the thread's `turn` ($\boxed{1}$). That is, the `turn_counter` array tracks the up-to-date mapping of each physical queue entry to a virtual queue entry.

When it is the thread's turn, i.e., the virtual queue entry assigned to the thread becomes active, the thread can copy its I/O access command into its assigned position in the queue ($\boxed{2}$). Afterwards, each thread sets the position's corresponding bit in the `tail_markers` bit-vector ($\boxed{3}$) with a release-store. The `turn_counter` array enables as many threads as the size of the queue to enqueue their requests in parallel, with the only synchronization between them being the atomic increment of the `turn_counter`.

To complete the insertion of their requests, threads race to acquire the `tail_lock`, as shown by steps $\boxed{4}$ and $\boxed{4}$ of the two threads in Figure 5.2. One thread will successfully acquire the lock and will attempt to move the tail by atomically resetting the `tail_markers` bits associated with newly enqueued entries ($\boxed{5}$ - $\boxed{6}$) until the queue becomes full ($\boxed{7}$) or

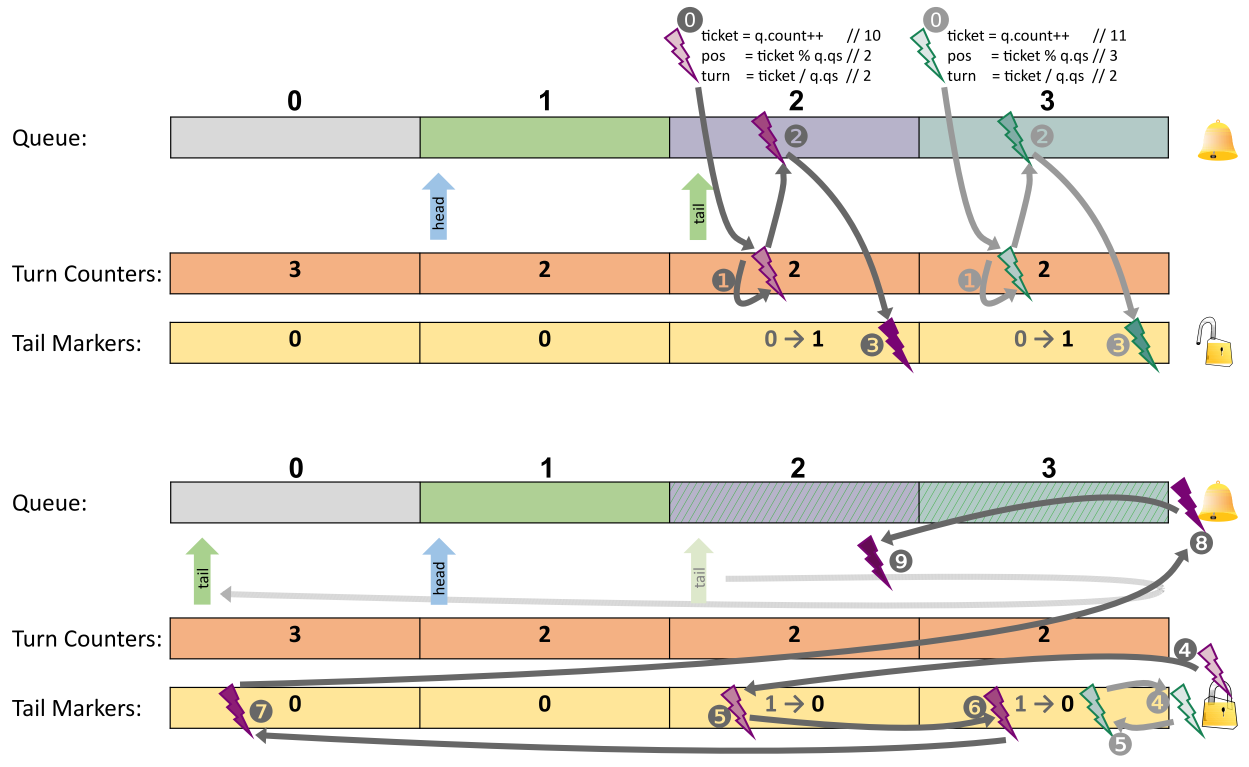


Figure 5.2: Exploiting parallelism for enqueuing commands into an NVMe submission queue (SQ). Steps for 2 threads, purple and green, attempting to enqueue requests to the SQ at positions 2 and 3, respectively.

the thread hits an unset marker. The thread then rings the queue's doorbell with the new tail (8), updates the local copy of the `tail` (9), and releases the lock. A critical section is necessary when resetting the markers as the threads must start resetting markers from the current tail and without a lock its possible to read a tail and then sometime later start the resetting process by when the previously read tail is stale and a race condition is created on the mark reset operation.

The rest of the calling threads will return if they observe that their bits in the `tail_markers` bit-vector are reset, signifying the SQ's tail will be moved past their enqueued entries. If the `tail_markers` bit for a thread has not been reset, the thread keeps trying to take the queue's `lock` and complete the insertion of its request (4 and 5), even if the thread had previously obtained the lock. This is to ensure that the SQ tail will eventually be moved past each enqueue entry.Since the one thread that enters the critical section for resetting the bits in the `tail_markers` bit-vector resets the bits for as many consecutive entries as possible, most threads that enqueue a request do not ever need to enter this critical section.

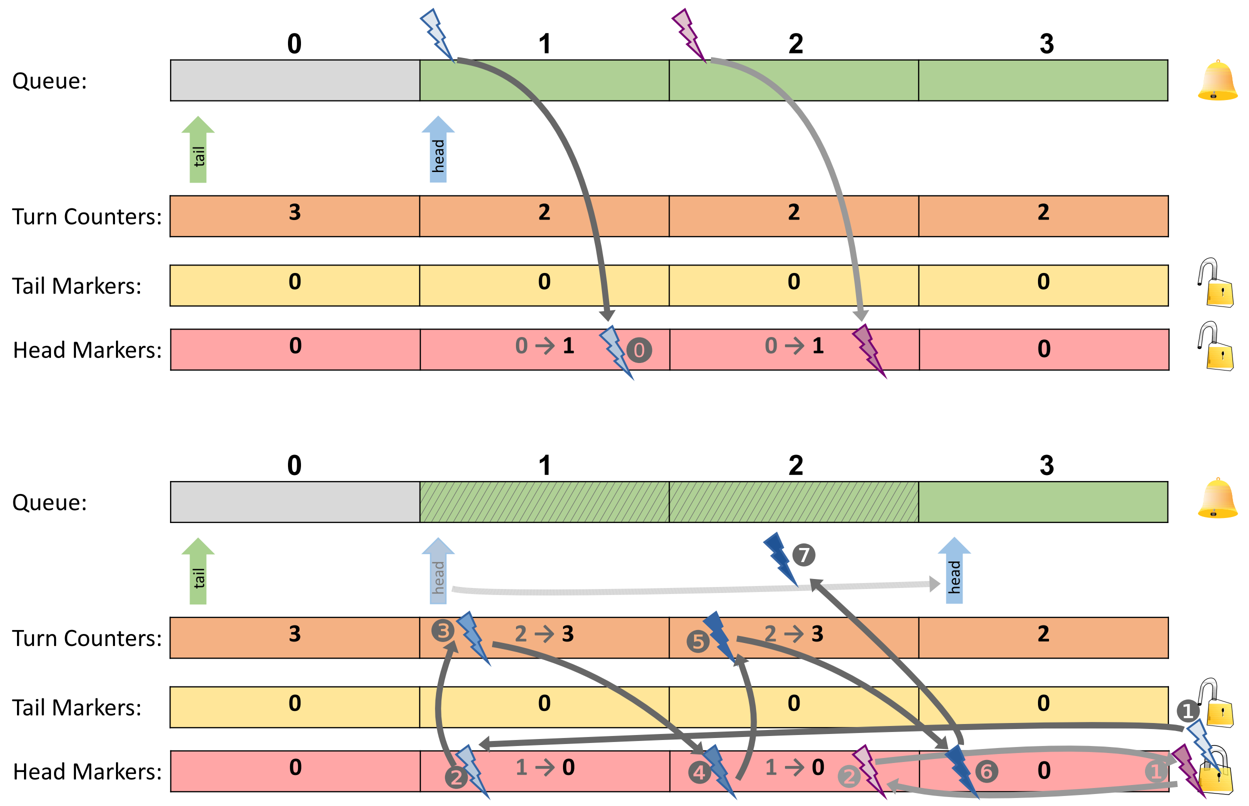


Figure 5.3: Exploiting parallelism for dequeuing commands from an NVMe submission queue. Steps for 2 threads, blue and purple, attempting to dequeue requests from the SQ at positions 1 and 2, respectively.

5.2 DEQUEUING FROM NVME QUEUES

After the thread submits its command, it can poll, without any lock, on the CQ to find the completion entry for its submitted request. When it finds the completion entry, it must mark its SQ entry and CQ entry locations to be consumed by the respective heads to communicate forward progress to other threads and the storage system. As the dequeue algorithm for the SQ and CQ are similar, we will only show the SQ dequeue process in Figure 5.3, while highlighting the key differences needed for the CQ dequeue process in our description. When a thread wants to dequeue its entry in the queue, it sets the entry's corresponding bit in the `head_markers` bit-vector ($\boxed{0}$). Then, dequeuing threads race to get the lock to move the head. One thread will successfully obtain the lock ($\boxed{1}$) and afterward, it will attempt to atomically reset bits in the `head_markers` bit-vector starting at the current `head` ($\boxed{2}$). For each entry the thread can reset a marker for, it also increments the corresponding `turn_counters` value ($\boxed{3}$), releasing the next thread waiting to enqueue into that position. Note, since GPU threads do not enqueue to the CQ, there is no `turn_counters` array for the

CQ; hence when dequeuing from the CQ, the thread with the lock does not need to update the `turn_counters` values. The thread repeats this process for each set marker it finds ($\boxed{4}$ - $\boxed{5}$) until it observes an unset marker ($\boxed{6}$). At that point, the thread updates the local `head` value ($\boxed{7}$) and releases the lock. When updating the CQ head, the thread must write the new `head` value to the CQ doorbell register to notify the storage controller of forward progress before releasing the lock.

5.3 CQ POLLING, HANDLING COMMAND IDS, AND CQ DEQUEUING

In BaM, each thread that enqueued an I/O request polls for its completion in the appropriate CQ. It does so by checking each CQ entry, from the CQ `head`, to see if (1) it is valid and (2) it corresponds to its submitted command ID. The validity of an entry is determined by the correct phase bit, as discussed in § 2.2. Recall that the phase bit value in a CQ entry is the controller's mechanism to mark valid CQ entries in a circular queue. Initially, since the queue is zeroed out at allocation, threads expect the controller to flip the phase bit of a CQ entry from zero to one to notify that the entry is valid. After consuming valid CQ entries and moving the head across all entries in the queue, it is known that the phase bits for all entries in the CQ are flipped to one. So now when the head wraps around the queue, the controller is expected to flip the phase bit of a CQ entry from one to zero to mark an entry valid. Note, each time the head of the CQ wraps around the queue, the expected phase bit for valid CQ entries is flipped.

Thus, to determine the phase expected for an entry, the threads have to determine how many times the head has wrapped around the queue and check if that count is even or odd. An even count implies the thread is expecting the phase bit of the entry to flip from zero to one to detect that it is valid. An odd count implies the thread is expecting the phase bit of the entry to flip from one to zero to detect that it is valid. To this end, we can get the entry's position in the virtual queue with 2^{32} entries by the sum of the current `head` and the offset of the entry from the `head`. We can then map the position onto the physical queue, just as described in § 5.1, and determine the expected phase by checking if the `turn` is even or odd.

If a thread sees an invalid entry in the queue before reaching the queue's end, it will assume that that is the current tail of the queue and poll again from the head. The full queue's end is exactly one less than the size of the physical queue entries from the `head`. The thread keeps polling until it observes a CQ entry with the valid phase bit and the command ID for its command.

However, incorrectly handling command ID allocation and polling from the CQ head can allow threads to assume stale entries are notifying completion for their commands. To avoid this, a command ID is locked when it is allocated to a thread, and the thread only unlocks it with a release-store after it observes the completion entry with the correct phase bit for the command **and** that the completion queue `head` is moved past that entry. This forces the next thread that acquires the command ID to start polling strictly after the previous, now stale, completion entry for that command ID in the virtual queue.

Forcing a thread to wait for the CQ's head to pass the thread's marked CQ entry creates a potential race condition. Take as an example a thread, t_i, that marked its CQ entry at position p to be dequeued, and right after reading the current `head`, needed to determine if it has moved past the entry, it is scheduled out. Meanwhile, another thread is able to reset the bit in the `head_markers` bit-vector for the CQ entry at position p, moves the head past it, and rings the CQ doorbell. Afterwards, the completion entry for the request submitted by another thread, t_k, is posted in the CQ at position p. Thread t_k polls the CQ and marks the p bit in the `head_markers` bit-vector. Now when thread t_i wakes up, it will observe a contradictory state that the p bit is set but the CQ's `head` is moved in the future. To address this race condition, we require that after a thread finds its CQ entry at a position, say position p, it takes a lock for that position. That is, there are N locks if there are N CQ entries. Only after taking the lock can the thread (1) set the bit in the `head_markers` bit-vector for the position p, (2) wait for the p bit to be reset in the `head_markers` bit-vector and (3) wait for the CQ `head` to be moved past p. Only after all three steps are done can the thread release the lock for position p. This orders the CQ dequeue operation per entry, just like the turn counters array does for the SQ. Note that threads can still poll on any position without any lock.

5.4 REDUCING SQ STALE ENTRIES

Although the SQ dequeue process described in § 5.2 maintains correctness for the queue, it causes unnecessary pollution of stale entries due to the fact that it requires the thread that enqueued an entry in the SQ to be the same one that waits for the entry to be consumed on dequeue as well. The storage controller could have read deeper into the queue than captured by the SQ `head` maintained by the GPU threads, and thus the controller communicates its progress by specifying the new SQ head in each CQ entry. Freeing up space in the SQ enables more commands to be in-flight, making it easier to hide the expensive storage-access latency.

However, using the controller's specified new SQ head presents some challenges. The controller specifies the head with respect to the physical queue and not the much larger virtual queue. Furthermore, since commands can complete and be processed out-of-order, mapping the physical queue entry to the virtual queue maintained by GPU threads is non-trivial. To address this problem, we can restrict SQ head updates to the same critical section as the one dedicated to the CQ head update. As the CQ head is moved past CQ entries in sequential order, due to the critical section, updates to the SQ head also occur in correct sequential order. Now, the thread that enters the CQ head update critical section can read the new SQ head field from the last CQ entry for which it could reset the mark bit and compare it to the current SQ's physical head. If they are the same, then the SQ's head has not moved. If they are different, then the controller has moved the SQ's virtual head by the same number of positions as the number of increments it takes to get the current SQ's physical head to the newly specified head.

Before leaving the CQ head update critical section, the thread must also increment the `turn_counters` value for each SQ position the SQ head was moved past to release the next group of threads waiting to enqueue on those positions. However, this creates a race condition with the following ordering given a thread t_0 with turn n and a thread t_1 with turn $n+1$ on position p in a SQ of size qs. After t_0 enqueues and sets the bit corresponding to p in `tail_markers`, it is scheduled out (for whatever reason), and another thread t_k enters the SQ tail critical section, resets the bit for position p and rings the queue's doorbell. The controller reads the command in position p and in the next CQ entry, processed by another thread t_u, it notifies of a new SQ head that is one position past position p, position $p+1$. Thus, t_u eventually moves the SQ head past p and increments the `turn_counters` value for p to $n+1$, allowing t_1 to writes its entry into the SQ and then set the bit for position p again in `tail_markers`. However, the bit for position p cannot be reset as the SQ head is $p+1$; recall a full queue is one where the tail is one increment away from the head. At this point if t_0 is scheduled back in then not only is t_1 waiting for the p bit in `tail_markers` to be reset, but t_0 is as well, although incorrectly. Furthermore, say the current head of the CQ is the completion entry for the command submitted by t_0. At this point, the threads cannot make forward progress as t_0 is stuck assuming its command is not submitted yet, even though it has, and thus cannot consume its completion queue entry and never moving the CQ head or the SQ head. This was not a problem in the previous implementation as the thread that was responsible for dequeuing an SQ entry was the same as the one that enqueued that entry, so an ordering was maintained. But now, any thread can dequeue any entry.

To address this, we must enforce that a thread only gets its turn on a position in the SQ when it is sure that the command enqueued at the position prior was both enqueued, i.e., consumed by the tail and submitted to the controller, and dequeued, i.e., the head moved

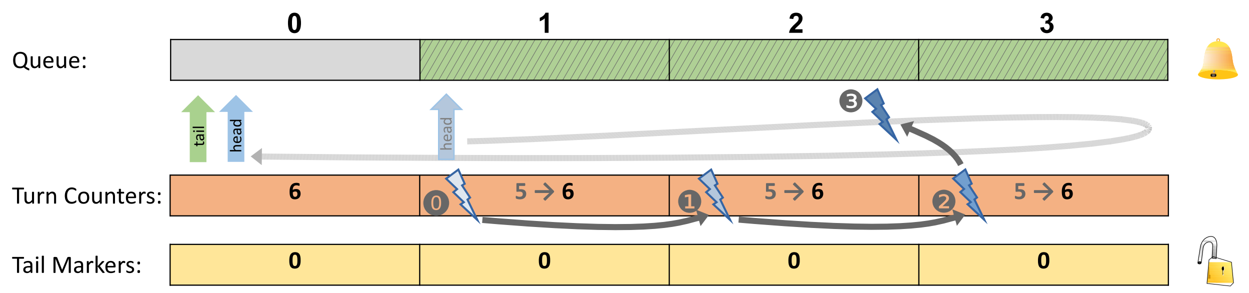


Figure 5.4: Single thread that got the CQ head lock and was able to move the CQ head, moving the SQ head based on the new SQ head specified by the last CQ entry it could move the CQ head past.

past the entry. This implies that each thread must wait for two updates (i.e., increments) to the `turn_counters` value of the position it is waiting on for each previous command submitted on that position. One of these increments happens after an enqueuing thread observes that some thread has reset its bit in the `tail_markers` bit-vector. The second happens when the thread moving the SQ head moves the SQ head past the position, as shown in Figure 5.4, by 0 - 2.

Figure 5.5 shows the new SQ enqueue algorithm. There are two differences between the SQ enqueue approach mentioned earlier and the new approach. First, instead of each thread computing its `turn` by taking the quotient of its place in the virtual queue and the physical queue size, each expected `turn` is now two times the quotient (0), allowing for the thread to wait for the each of the two updates mentioned earlier. Second, after a thread observes that the bit for its position in the `tail_markers` bit-vector is reset, it increments the `turn_counters` value for its position (10 and 6), letting other threads know that this thread's enqueue procedure is done. Since the next thread waiting for the same position waits on both increments, the order in which the two increments happen with respect to each other does not matter.

5.5 PERFORMANCE EVALUATION

Now we evaluate the performance of BaM's queue parallelization techniques. We use an NVIDIA A100 GPU with Intel Optane P5800X 1.6TB SSDs in the system described in Table 8.1 and §8.1. First, we evaluate the performance benefit of BaM's queue designs against a locking implementation. The locking implementation requires threads to lock the submission or completion queue and perform enqueuing and dequeuing as a critical section, although threads can poll on the completion queue without a lock. We evaluate

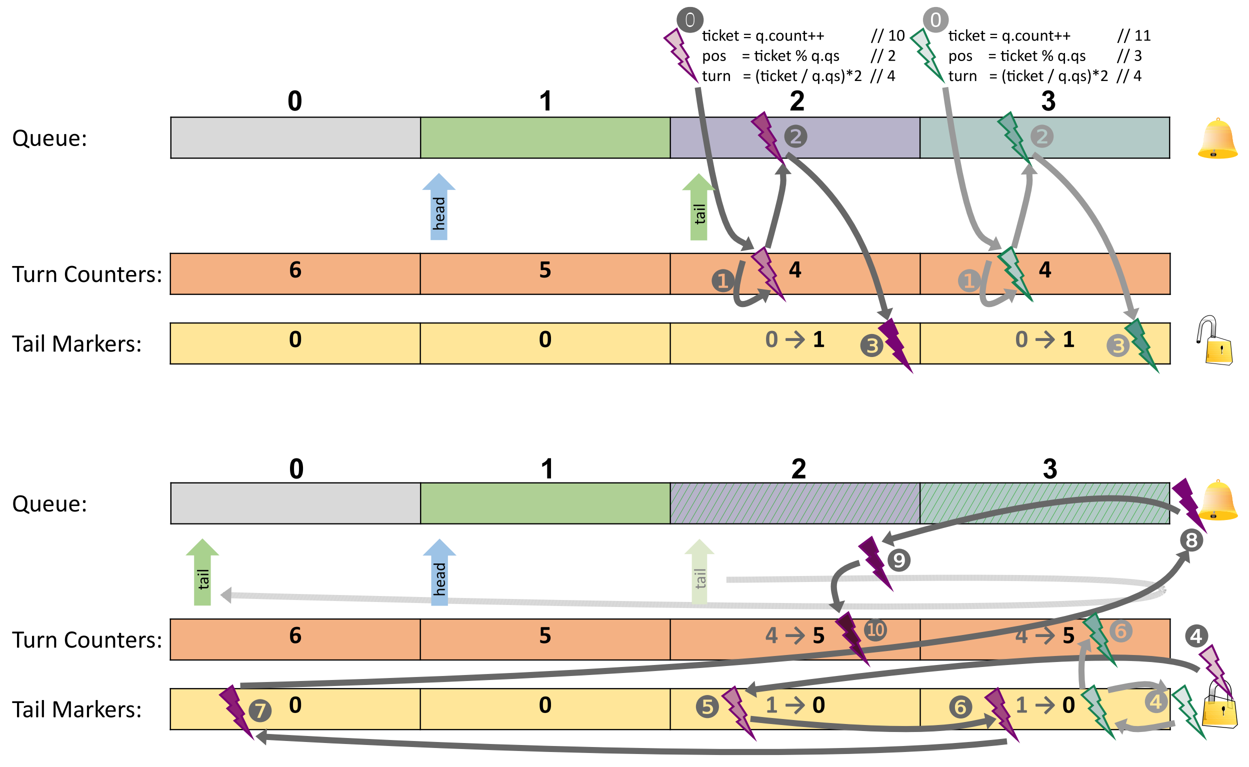


Figure 5.5: Exploiting parallelism for enqueuing commands from an NVMe submission queue with the optimized SQ dequeuing procedure. Steps for 2 threads, purple and green, attempting to enqueue requests to the SQ at positions 2 and 3, respectively.

the performance of each implementation with a single pair of submission and completion queues, with each queue of depth 1024. We use the 512-byte read I/O command as the Intel SSD provides the highest throughput with that command at a peak of 5Million I/O-operations-per-second(IOPS). We launch a GPU kernel where each thread submits an I/O request, polls for its completion, and performs the appropriate cleanup of the queues. We vary the number of threads launched to increase the contention on the queues.

As Figure 5.6 shows, the locking approach causes great serialization overhead, limiting overall peak performance to 110K-IOPS, and as contention increases beyond 4096 parallel threads, the performance of the queue implementation drastically drops to below 40K-IOPS at 8192 threads and 5K-IOPS at 32768 threads. In comparison, BaM's base queue design provides not only higher (5.15×) peak performance, 550K-IOPS for up to 4096 parallel threads, but also maintains more than 270K-IOPS even with 32768 parallel threads. Optimizing the BaM queue design even further by cleaning up stale SQ entries, as described in §5.4, the BaM queue design can reach a peak of 818K-IOPs, an improvement of 1.42×. We note that all implementations have degradation in performance after 4096 parallel threads due to the serialization enforced by the NVMe protocol for ringing doorbells.

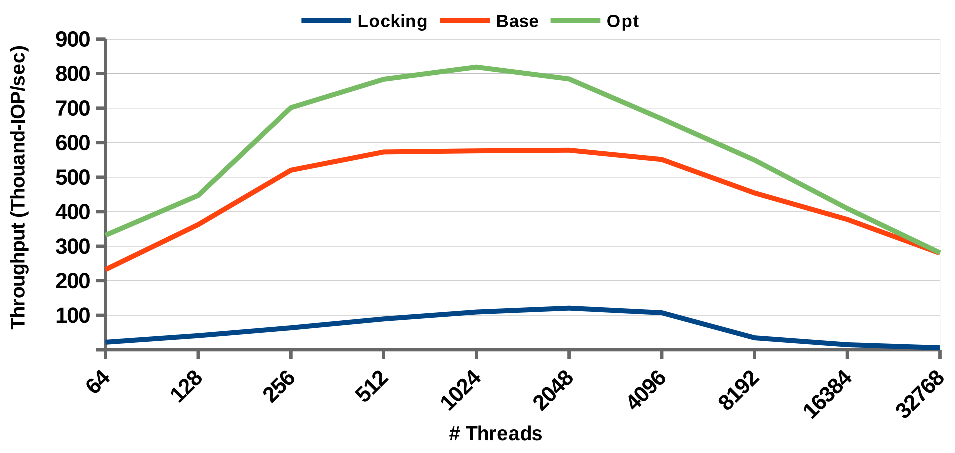
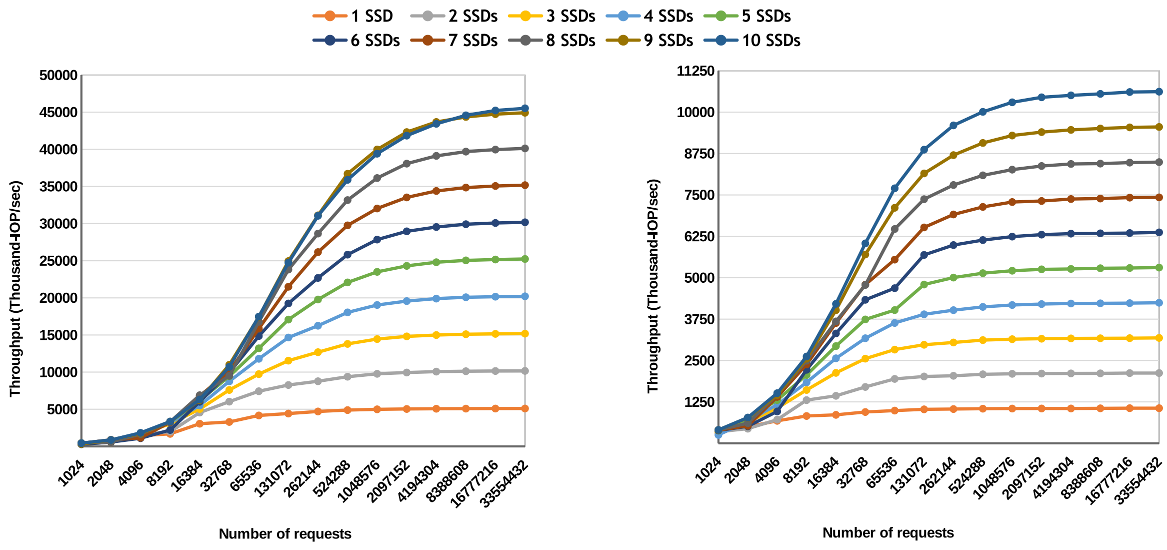


Figure 5.6: Performance of 3 implementations of NVMe submission and completion queue pairs: a `Locking` implementation, the `Base` queue design described in this chapter, and the queue design with the optimization described in §5.4(`Opt`). All queues have a size of 1024. Each thread submits 1 512Byte I/O read request.

Next, we evaluate the scalability of BaM's I/O stack by mapping up to 10 Intel Optane P5800X 1.6TB SSDs to the Nvidia A100 GPU. Each SSD provides a peak of 5-Million IOPS for read requests and 1-Million IOPS for write requests. For each SSD, we instantiate 128 submission and completion queue pairs, with each queue having a size of 1024 entries. We launch a GPU kernel where each thread submits an I/O request, polls for its completion, and performs the appropriate cleanup of the queues. We vary the number of threads launched to increase the contention on the queues. As Figure 5.7 shows, BaM can saturate a single SSD's read throughput with 65536 parallel threads and the write throughput with 8192 parallel threads. Furthermore, BaM can scale linearly as we add more SSDs providing up to 45-Million IOPS for reads with 9 SSDs, saturating the achievable bandwidth of the PCIe Gen4 x16 link into the Nvidia A100 GPU.



(a) Read Performance (b) Write Performance

Figure 5.7: Scaling the number of Intel Optane P5800X SSDs used with BaM running on a Nvidia A100 GPU.

CHAPTER 6: BAM'S SOFTWARE CACHE

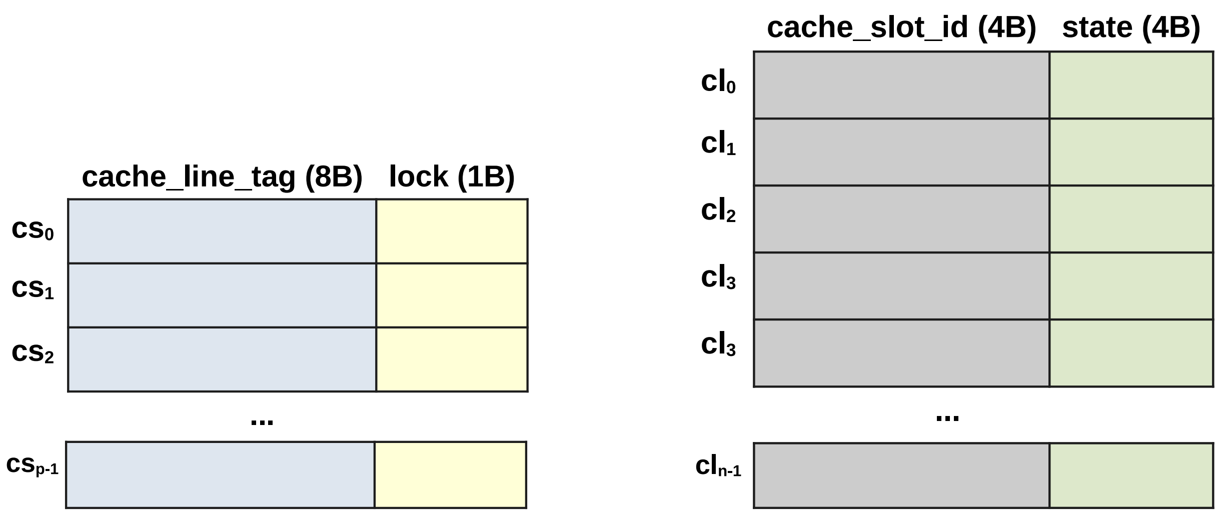
The BaM software cache is designed to optimize the storage access bandwidth utilization, leverage fast GPU memory, and provide fast data lookup. Traditional operating system memory management (allocation and translation) implementations must support diverse, legacy application/hardware needs. As a result, they contain large critical sections that limit the effectiveness of multi-threaded implementations. BaM addresses this bottleneck by pre-allocating all the virtual and physical memory required for the software cache when starting each application. This approach allows the BaM software cache management to reduce critical sections, only requiring a lock when inserting or evicting a cache-line, which allows the BaM cache to support many more concurrent accesses.

6.1 CACHE METADATA

For the rest of this thesis, we will use the term cache-line to refer to a segment of the storage data that will be mapped into the GPU memory as a unit. We use the term "cache slot" to refer to a segment of locations in the GPU memory that will be used to host cache-lines. To support fast look-ups, BaM keeps two metadata structures in GPU memory. The first, shown in Figure 6.1a, keeps two elements per cache slot in the cache: 1) a lock and 2) an 8-byte tag for the cache-line currently stored in the slot. This structure is used as a map to the victim cache-line's state on eviction, as explained in § 6.3. The second structure, shown in Figure 6.1b, keeps a 4-byte state and a 4-byte offset into the cache for each cache-line in the mapped data. Although this structure grows with the size of the data, it allows an accessing thread to check the state of a cache-line and its place in the cache in constant time, i.e., without a search.

These two structures in BaM help ensure two properties. First, if a cache-line is in the cache, then there is only one copy of it in the cache. This is important to maintain data consistency. Second, if a cache-line is not in the cache and multiple threads request the cache-line then only one of those threads will make the storage I/O request for it. This helps achieve one of the key goals of BaM, minimizing the I/O traffic.

Next, we describe how GPU threads can use these structures on cache access.



(a) Metadata kept for each cache slot in a BaM cache with p total slots.

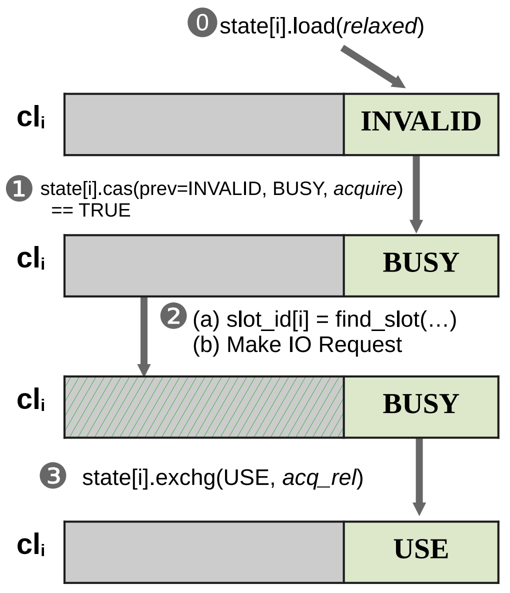
(b) Metadata kept for each cache-line in the data accessed through a BaM cache.

Figure 6.1: Metadata kept in GPU memory for BaM's software-managed cache.

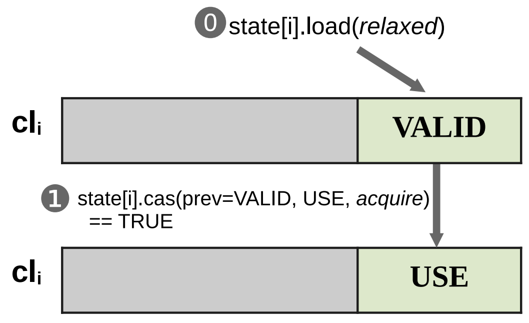
6.2 CACHE ACCESS - BASELINE DESIGN

When a thread wants to access data through the cache, it must call the `acquire_cl` routine with the cache-line index for the array element being accessed. In this routine, the thread probes the cache-line's state, step $\boxed{0}$ in Figure 6.2. If the thread observes the cache-line state is `INVALID`, i.e. the cache-line is not in the cache, it tries to lock the cache-line by attempting to change the cache-line state to `BUSY` ($\boxed{1}$) with an acquire compare-and-swap operation, as shown in Figure 6.2a. If it is successful, it can find a victim to evict, request the cache-line from the backing memory, and update the cache slot assignment for the cache-line ($\boxed{2}$). After the request to the backing memory is completed, the thread can update the cache-line's state to `USE` using an atomic exchange with an acquire-load and release-store ($\boxed{3}$), notifying other threads that the cache-line is in GPU memory and is being used by 1 thread. The acquire compare-and-swap and release-store make sure the operations in $\boxed{2}$ are not reordered around the exclusive lock that the thread must obtain.

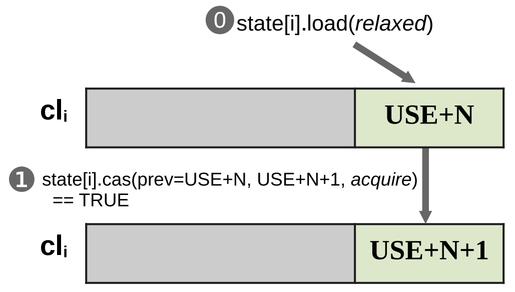
If when the thread probes the cache-line state, it observes the cache-line is `VALID`, it attempts to move the state to `USE` to notify others that the cache-line is in use with an acquire compare-and-swap operation, as shown in Figure 6.2b. If it is successful, it can use the data in the cache-line as it wills. Since the `VALID` state is encoded as an integer value strictly one less than the integer value encoding the `USE` state, the 4-bytes of the cache-line state can not only tell if the cache-line is in the cache or not or if it is busy, but it can also encode how many threads are actively using the cache-line, a reference count. That is, if


0 state[i].load(*relaxed*)
cl_i
INVALID
1 state[i].cas(prev=INVALID, BUSY, *acquire*) == TRUE
BUSY
2 (a) slot_id[i] = find_slot(...)
(b) Make IO Request
BUSY
3 state[i].exchg(USE, *acq_rel*)
USE

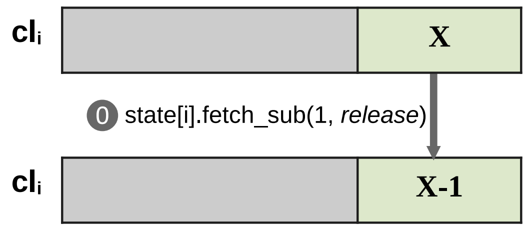
(a) Cache-line state management if the observed cache-line state is `INVALID`.


0 state[i].load(*relaxed*)
cl_i
VALID
1 state[i].cas(prev=VALID, USE, *acquire*) == TRUE
USE

(b) Cache-line state management if the observed cache-line state is `VALID`.


0 state[i].load(*relaxed*)
cl_i
USE+N
1 state[i].cas(prev=USE+N, USE+N+1, *acquire*) == TRUE
USE+N+1

(c) Cache-line state management if the observed cache-line state is `USE+N`.



(d) Releasing a cache-line after using it.

Figure 6.2: Cache-line state management on a cache-line access with `acquire_cl` and the eventual release with `release_cl`.

the cache-line state is `USE+N`, where $N >= 0$, then $N + 1$ threads are currently using the cache-line. Thus, if a thread probes the cache-line state and it observes the state is `USE+N`, it attempts to move the state to `USE+N+1` using an atomic compare-and-swap with acquire semantics, as shown in Figure 6.2c.

BaM uses reference counting to ensure cache data consistency. When finding a victim cache-line to evict as it can allow a thread to look for a cache slot that is not being used. So if threads are actively using a cache-line, the cache-line's state must be `USE+N`, where $N >= 0$. Thus, when threads are done using the cache-line they must call the `release_cl` routine that updates the cache-line state by atomically decrementing the state by 1, as shown in Figure 6.2d. This must be a release-store operation as after this state update, other threads can potentially evict the cache-line so the thread will no longer have any guarantees of the usability of the cache-line. Thus, any accesses to the cache-line by the thread must not be reordered after this state update.

If the thread's probe of the cache-line state returns that the cache-line is `BUSY`, or if any of the previously mentioned atomic compare-and-swap operations fail, the thread re-probes the cache-line state with an exponential back-off.

When a thread needs to write to data through the cache, it sets the dirty-bit through the `acquire_cl` routine. Now, when the thread probes the cache-line state and sees it is `INVALID`, after bringing the cache-line from the backing memory, the thread will set the cache-line state to `USE_DIRTY`, instead of `USE`. We leave for future work optimizing the cache such that if it is known that the entire cache-line is to be written, the storage version of the cache-line will not need to be brought in. Similarly, if when the thread probes the cache-line state and it observes the state is `VALID`, the compare_and_swap will attempt to move the cache-line state to `USE_DIRTY`. If the observed cache-line state was `USE_DIRTY`, the compare_and_swap will attempt to move the cache-line state to `USE+1_DIRTY`. As a higher-order bit is used for the dirty-bit, the `fetch_sub` can still be used on releasing the cache-line to decrement the state by 1, preserving the dirty state.

6.3 CACHE REPLACEMENT

On a miss, a thread calls the `find_slot` routine to find a victim, invalidate it, and assign the slot to the newly requested cache-line. To avoid contention among concurrent evictions, the BaM cache uses a clock replacement algorithm [48]. The cache has a global counter that gets incremented when a thread needs to find a cache slot. The returned value of the counter assigns the thread a cache slot to use, allowing concurrent threads to evict unique cache slots in parallel.

Figure 6.3 shows how a thread looking for a victim handles the cache slot k assigned to it by the clock replacement algorithm. First, the thread locks the cache slot to prevent race conditions caused by multiple threads attempting to evict the same victim and thus sue the same slot. If the thread can lock the cache slot ($\boxed{0}$), it checks the tag of the cache-line currently mapped to the cache slot ($\boxed{1}$). Using the tag it can directly check the victim cache-line's state. If the victim's state is `VALID`, and not `BUSY` or `USE+N`, the thread will attempt to evict the victim by changing its state to `BUSY` ($\boxed{2}$). If successful, it can write back the victim if it was dirty, and then mark it `INVALID` ($\boxed{3}$). Now, the thread can change cache slot k's mapping to the new cache-line ($\boxed{4}$) and release the lock ($\boxed{5}$).

If all of these operations succeeded, then the new cache-line has been assigned to cache

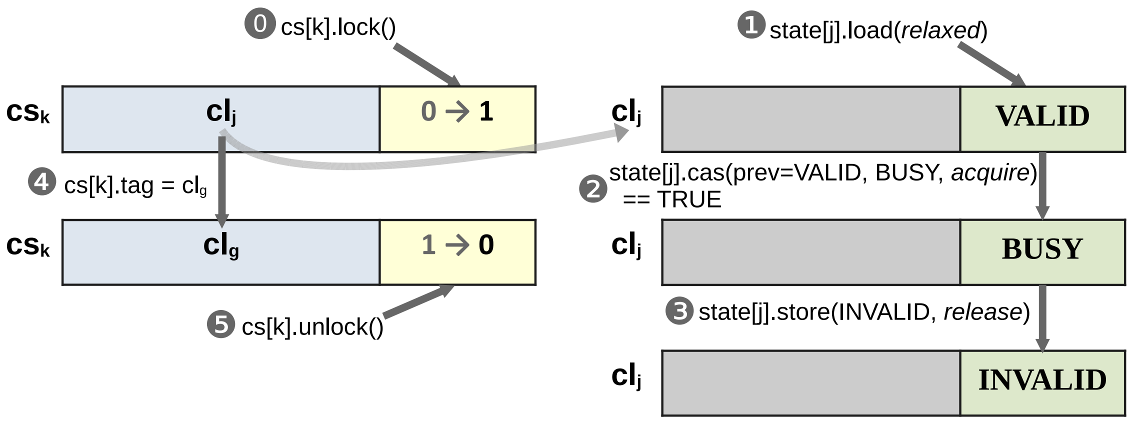


Figure 6.3: Finding and evicting a victim with the `find_slot` routine. If the compare_and_swap (2) fails, the thread unlocks the cache slot and uses the clock algorithm to attempt to evict another victim. If the thread fails to lock the cache slot (0), it uses the clock algorithm to attempt to evict another victim.

slot k and the thread can fill the slot with the cache-line from the backing memory. If any operation fails, i.e. not being able to lock the slot 0 or the victim being in `BUSY` or `USE+N` state or failing to change the victim's state to `BUSY`, the thread will release cache slot's lock (if acquired) and attempt to get another cache slot from the clock algorithm.

The thread looking for a victim can determine if the cache slot is cold, i.e. it is not mapped to any data cache-lines, by looking at the returned `lock` value when attempting to lock the cache slot (0). Recall, we use a whole byte for the `lock` field. If the returned value is the second least-significant bit set and the thread was able to obtain the lock on the slot, then the thread can deem the cache slot cold. In that case, the thread can directly update the cache slot mapping (4) and unlock the slot (5). Unlocking the cache-line un-sets the cold bit in the cache slot's `lock`.

6.4 OPTIMIZING REFERENCE COUNTING

Using compare-and-swap to manage the cache-line state, and thus the reference count can cause severe serialization latency. Take as an example a cache-line that is in the cache with a reference count of 0 and we have t parallel accesses to this cache-line. All these t accesses will be hits, so the throughput of the cache should be really high. However, using compare-and-swap to reference count can be disastrous here. First, a compare-and-swap operation is very slow and has low throughput due to its complexity. Second, use of compare-and-swap to maintain a reference count implies that if all t threads attempt to increase the reference count from 0 to 1, only one of them will succeed and the rest will fail. The threads that failed

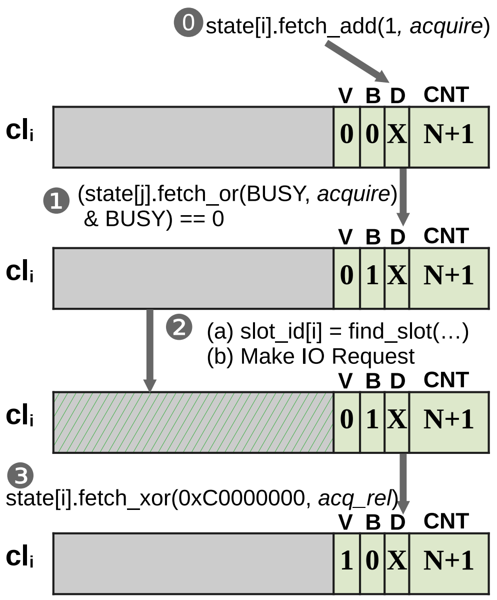
Table 6.1: Definition of cache-line states using the 3 bits.

Valid	Busy	Dirty	Definition
0	0	0	Not in cache
0	0	1	Not possible
0	1	0	Being inserted into cache
0	1	1	Being inserted into cache, marked dirty for next eviction
1	0	0	In cache
1	0	1	In cache, marked dirty for next eviction
1	1	0	Being evicted
1	1	1	Being evicted and written back

will need to retry, but again if all try the compare-and-swap in parallel then one will succeed and the rest will fail. This can repeat as many times as there are threads. As a result, if there are t total threads attempting to increase the reference count of a single cache-line by 1 each, it's possible to have $t(t+1)/2$ total number of compare-and-swap instructions executed. This does not even account for the compare-and-swap failures (and thus retries) due to the change in reference count when threads release the cache-line.

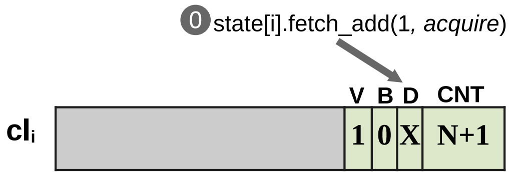
To address this, we enforce an encoding of the state that strictly separates the reference count from the cache-line state. We still use a 32-bit word to hold the state and the reference count, enabling us to atomically read, and write if needed, the state and the reference count. However, the reference count is now strictly the unsigned number resulting from the least-significant 29-bits of the word. The 3 highest-order bits are the state and specify if the cache-line is (V)alid, (B)usy, and (D)irty. The state definitions with these 3 bits are shown in Table 6.1. With this encoding, a thread can just execute `cur_state = state[i].fetch_add(1, acquire)` to both increment the cache-line's reference count by one, and read the cache-line's state. As we do not care what the previous reference count was, executing this `fetch_add` always succeeds, and thus if t threads all access the same cache-line in the cache, then only t atomic operations will execute to increment the reference count correctly. Furthermore, the atomic increment is significantly less expensive and higher-throughput than the compare-and-swap operation.

Figure 6.4 shows the cache-line state management with this new encoding and optimization for reference counting when calling the `acquire_cl` routine. As mentioned earlier, the thread probes the current cache-line state and increments its reference count using the `fetch_add` with acquire ordering ($\boxed{0}$). If the cache-line is not valid, the thread tries to set the `Busy` bit using a `fetch_or` with acquire ordering ($\boxed{1}$), as shown in Figure 6.4a. If in the returned state the `Busy` bit was not set then the thread was able to set the bit and has a lock on the cache-line to find a slot for it and bring it into the cache ($\boxed{2}$). If it was set then the thread must wait for the cache-line to not be busy. Once the cache-line has been brought into the cache, the thread that obtained the lock must unset the `Busy` bit and set the `Valid` bit. It

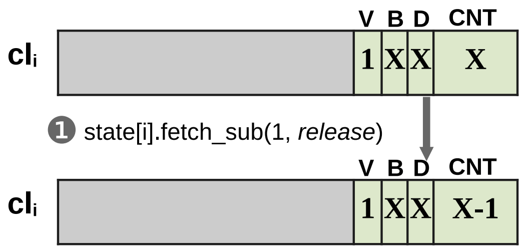


0 state[i].fetch_add(1, *acquire*)
V B D CNT
cl$_i$
0 0 X N+1
1 (state[j].fetch_or(BUSY, *acquire*) & BUSY) == 0
V B D CNT
cl$_i$
0 1 X N+1
2 (a) slot_id[i] = find_slot(...)
(b) Make IO Request
V B D CNT
cl$_i$
0 1 X N+1
3 state[i].fetch_xor(0xC0000000, *acq_rel*)
V B D CNT
cl$_i$
1 0 X N+1

(a) Cache-line state management if the observed cache-line state is `INVALID`.



(b) Cache-line state management if the observed cache-line state is `VALID`.



(c) Releasing a cache-line after using it.

Figure 6.4: Cache-line state management on a cache-line access with `acquire_cl` and the eventual release with `release_cl` with the new encoding. `X` is placed in for bits and values that do not matter.

can do this with an atomic `fetch_xor` (3). Afterward, the thread can use the data in the cache-line. As Figure 6.4b shows, if the cache-line state observed is `Valid`, then the thread can directly use the data from the cache-line as it had already incremented the reference count at the beginning of the `acquire_cl` routine. Just as before, when a thread is done using the cache-line it must atomically decrement the reference count by 1, as shown by Figure 6.4c.

This new encoding also affects how victims are evicted from the cache. With the previous state encoding, it was sufficient to move the victim from `VALID` to `BUSY` to determine if the victim was not being used and was not busy. However, with the new encoding, it is possible to have a victim with both the `Valid` bit and `Busy` bit set simultaneously. Furthermore, the victim can also have a non-zero reference count simultaneously. As shown in Figure 6.5,

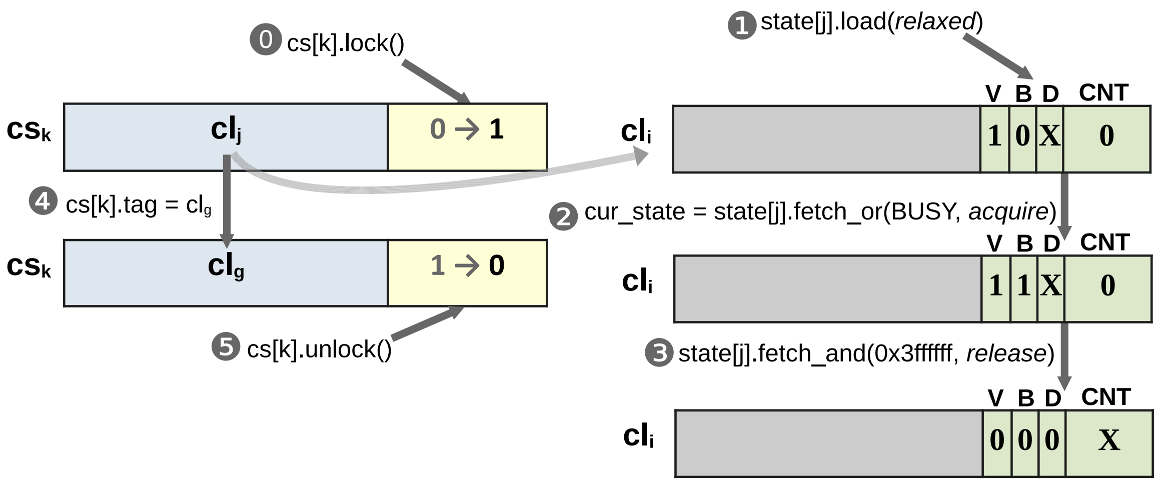


Figure 6.5: Finding and evicting a victim with the `find_slot` routine with the new encoding.

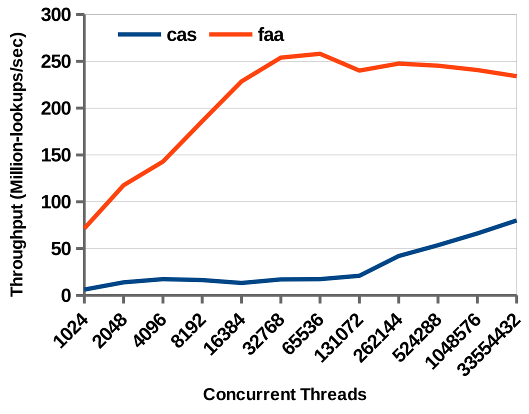
with the new encoding the evicting thread must make sure that the victim has a reference count of zero and its `Busy` bit is not set in the returned state both when the victim's state is first read (1) and when the thread executes the `fetch_or` to set the `Busy` bit (2). If at either of those points the victim has a non-zero reference count or has its `Busy` bit set by another thread, the evicting thread will abort, resetting the `Busy` bit if it was the one that set it, and get another victim assigned by the same clock replacement algorithm as before.

6.5 PERFORMANCE EVALUATION

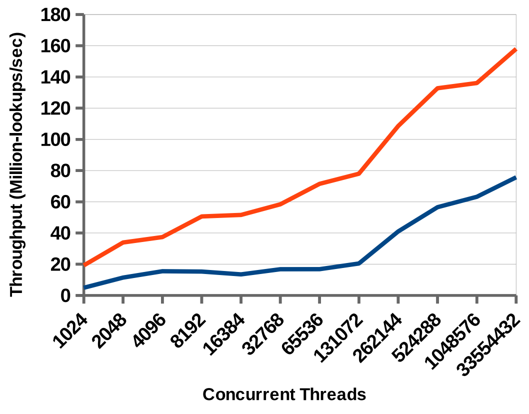
Now we compare the performance of the cache designs using `compare_and_swap` (cas) and `fetch_add` (faa) for reference counting. We use an NVIDIA A100 GPU with 4 Intel Optane P5800X 1.6TB SSDs in the system described in Table 8.1 and §8.1. For each SSD we create 128 submission and completion queue pairs with each queue having a depth of 1024. We launch a GPU kernel where each thread reads 1 8-byte data element from a 64GB data array on storage through the cache and consecutive threads read consecutive elements. We vary the number of threads from 1024 to 32 million and report both the hit throughput when all the needed bytes are in BaM's cache, and the miss throughput when the BaM cache is empty. Throughput is calculated as the total number of 8-byte data element accesses in the GPU kernel divided by the execution time of the GPU kernel. We can increase the contention on cache-lines by changing the cache-line size from 512 bytes (the smallest NVMe I/O granularity) to 4KB and 8KB. The total cache capacity is fixed to 8GB for all experiments.

As shown by the results in Figure 6.6, the `faa` cache design consistently outperforms the `cas` design in all cases. We note that the `cas` cache provides nearly the same peak throughput for each cache-line size regardless of if the accesses will be hits in the cache or a miss, signifying that the `cas` design is so slow that the storage access overhead is hidden by the significantly larger cache access overhead. In contrast, the `faa` cache design's peak hit throughput is 1.48×, 1.09×, and 1.07× faster than the `faa` design's peak miss throughput with 512-byte, 4KB, and 8KB cache-line sizes, respectively. Furthermore, as cache-line size, and thus the cache-line contention, increases, the overhead of re-executing `compare_and_swap` operations is increased and the overall performance of the `cas` design degrades. As we increase the cache-line size from 512-byte to 4KB the throughput of the `cas` design reduces 1.9× and as we increase the cache-line size from 4KB to 8KB the performance further reduces by 1.5×. In contrast, the `faa` design's performance for the miss case improves by 1.43× and 1.01× when increasing the cache-line size from 512-byte to 4KB and 4KB to 8KB, respectively. In the hit case when the cache contention overhead is most exposed, the peak throughput of the `faa` design is improved by 5% going from a 512-byte cache-line to an 8KB cache-line, while the peak throughput of the `cas` design degrades 3.26× going from a 512-byte cache-line to an 8KB cache-line.

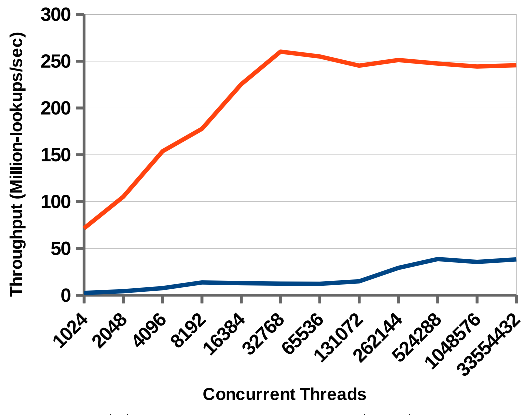
Overall, the as the `faa` design has significantly less reference-counting overhead, it outperforms the `cas` design in the miss case by 2.09×, 5.67×, and 8.64× and the hit case by 2.93×, 6.42×, and 9.98× with 512-byte, 4KB, and 8KB cache-line sizes, respectively.



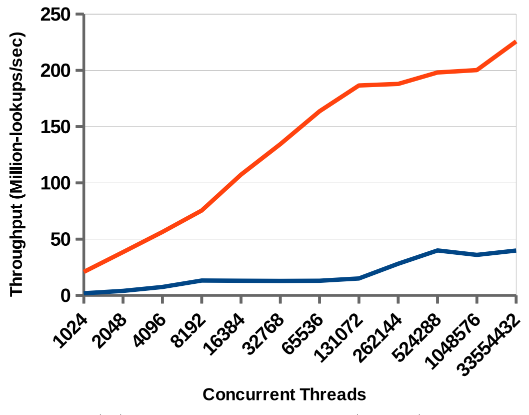
(a) 512-byte Cache-line (Hit)



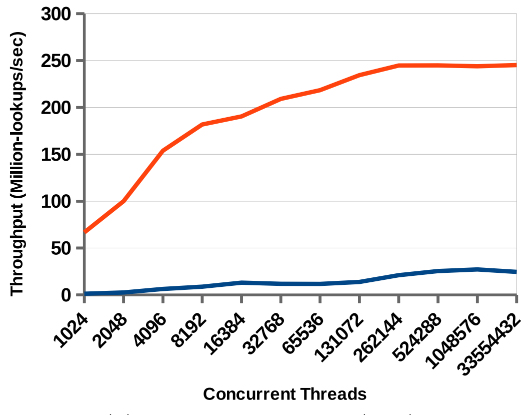
(b) 512-byte Cache-line (Miss)



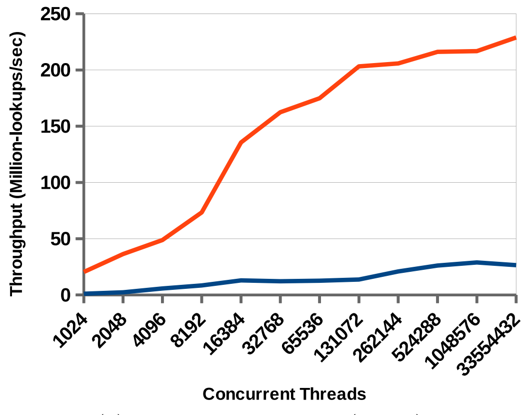
(c) 4KB Cache-line (Hit)



(d) 4KB Cache-line (Miss)



(e) 8KB Cache-line (Hit)



(f) 8KB Cache-line (Miss)

Figure 6.6: Performance of the cache designs using `compare_and_swap` (`cas`) and `fetch_add` (`faa`) for reference counting with 3 different cache-line sizes: 512-byte, 4KB, and 8KB. Each GPU thread accesses a single 8-byte element in the data through the cache. Throughput is calculated as the total number of data element accesses divided by the execution time of the GPU kernel. We report both the hit throughput, when all the needed bytes are in BaM's cache, and the miss throughput, when the BaM cache is empty.

CHAPTER 7: BAM'S USER-LEVEL GPU ABSTRACTION

Although BaM's I/O queues enable the GPU to make direct storage requests and the BaM cache allows threads to reuse data fetched over I/O, GPU programmers are not accustomed to writing GPU kernels where threads make I/O requests and use a software-managed cache. Instead, threads in GPU kernels operate on C-style arrays in memory, as shown in Listing 7.1. Mapping an application that works on arrays to one that is aware of software cache hits and misses, as well as making I/O requests for misses, imposes significant code modifications and thus burdens the programmer.

```
__global__
void kernel(unsigned* data, size_t elems_per_thread, unsigned* ret)
{
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
  size_t wid = tid / 32;
  size_t lane = tid % 32;
  size_t start = wid * elems_per_thread * 32 + lane;
  size_t end = (wid+1) * elems_per_thread * 32;
  for (; start < end; start+=32)
      *ret += data[start];
}
```

Listing 7.1: GPU linear access benchmark kernel code.

7.1 THE `BAM::ARRAY` MEMORY ABSTRACTION

To ease the software porting effort for migrating an application to use BaM, BaM's software stack provides the programmer an array-based high-level API (`bam::array<T>`), consistent with array interfaces defined in modern programming languages (e.g. C++, Python, or Rust). As `bam::array` is templated and provides APIs for accessing elements of type T with an integer index, it trivializes the programmer's effort to adapt their kernels. All it takes to adapt a GPU kernel reading from a large array to read from a large `bam::array` is to change its data type, as shown in Listing 7.2. `bam::array`'s overloaded subscript operator enables the accessing threads to coalesce their accesses, query the cache, and make I/O requests on misses; and returns the appropriate element of type T to the calling function.

We note that arguments passed to GPU kernels at kernel launch are kept in a fast read-only memory on the GPU and are shared by all threads of the GPU kernel. This implies that values of `bam::array`'s member variables cannot be modified and are constant for all threads. Thus, the abstraction keeps a pointer to the backing BaM cache as a member, which threads can use to probe and access a shared cache. We show how the abstraction overloads the subscript operator enabling the GPU threads to access the cache to get data on-demand in Listing 7.3. After finding the needed cache-line ID, the operator calls the cache's `acquire_cl` method to increment the reference count of the cache-line and get its address in memory. Then it copies the element at the index requested into a temporary variable so that the correct value can be returned after decrementing the cache-line's reference count with the `release_cl` call.

```
__global__
void kernel(bam::array<unsigned> data, size_t elems_per_thread, unsigned* ret)
{
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
  size_t wid = tid / 32;
  size_t lane = tid % 32;
  size_t start = wid * elems_per_thread * 32 + lane;
  size_t end = (wid+1) * elems_per_thread * 32;
  for (; start < end; start+=32)
      *ret += data[start];
}
```

Listing 7.2: GPU linear access benchmark kernel code ported to use BaM. Note that with BaM's `bam::array` abstraction, only the input parameter's data type needs to be changed.

```
template<T>
__device__
T bam::array::operator[](size_t i) const
{
  size_t cl_id    = get_cl_id(i);
  size_t cl_sub_idx = i % (cache->cl_size / sizeof(T));

  T* cl_addr = cache->acquire_cl(cl_id);
  T  temp    = cl_addr[cl_sub_idx];
  cache->release_cl(cl_id);

  return temp;
}
```

Listing 7.3: `bam::array`'s overloaded subscript operator that abstracts away read accesses to the cache and backing memory.

Note that since the abstraction aims to hide the cache access complexity from the user, the `acquire_cl` and `release_cl` calls must be called in the overloaded subscript operator to avoid incorrect cache-line state management. As a result, `bam::array`'s overloaded subscript operator cannot be used on the left side of assignments. Instead `bam::array` provides the `write` method shown in Listing 7.4 to enable writes to the data while the cache-line reference is held. It passes `true` to the second argument of the `acquire_cl` method which is `false` by default, so that the cache-line state management can set the cache-line's dirty bit appropriately. The `write` method requires changing kernel code that modifies array values, like `data[i] = v;`, to `data.write(i, v)`, a simple enough change.

```
template<T>
__device__
void bam::array::write(size_t i, const T& val)
{
  size_t cl_id   = get_cl_id(i);
  size_t cl_sub_idx = i % (cache->cl_size / sizeof(T));

  //set acquire_cl's second argument, the dirty flag
  //that is false by default, to true
  T* cl_addr = cache->acquire_cl(cl_id, true);
  cl_addr[cl_sub_idx] = val;
  cache->release_cl(cl_id);
}
```

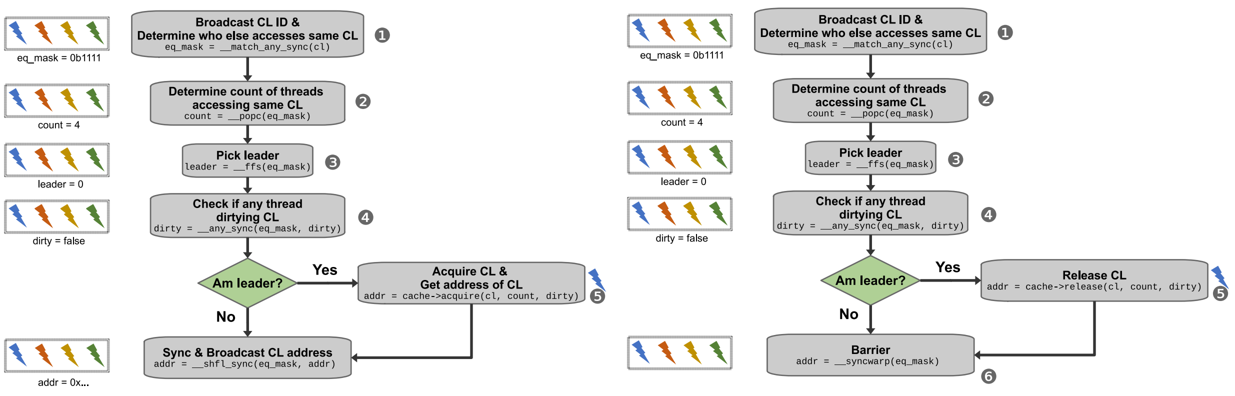
Listing 7.4: `bam::array`'s `write` method that abstracts away write accesses to the cache and backing memory.

7.2 REDUCING BAM CACHE CONTENTION

Although the `bam::array` abstraction makes it easy for a programmer to integrate BaM into their GPU application, it requires the cache-line's reference count to be incremented and decremented for each element access. Even though we optimized the reference counting in § 6.4, such a scheme increases atomic traffic, creating contention and possibly exposing serialization overhead, especially when the cache-line is in the cache. In this section, we outline two techniques BaM implements using its abstractions to reduce the per-element access overhead.

7.2.1 Warp Coalescing

In typical GPU kernel code, threads in a warp access contiguous bytes in memory. This access pattern allows the GPU hardware to coalesce the warp accesses to fewer and larger GPU cache requests, thus optimizing the GPU memory bandwidth utilization. The `bam::array` abstraction can exploit this common behavior to coalesce accesses from a warp to the cache and reduce the number of total accesses to the cache metadata by up to 32× as there are 32 threads in a warp. As shown in Listing 7.5, the BaM coalescer implements warp coalescing in software using the `__match_any_sync` warp primitive. This primitive allows threads of a warp to synchronize, share a register value (`val`), and compute a new mask (`eq_mask`) where the bit corresponding to a warp-lane is set if the thread in that lane has same register value


eq_mask = 0b1111
Broadcast CL ID & Determine who else accesses same CL
eq_mask = __match_any_sync(cl)
count = 4
Determine count of threads accessing same CL
count = __popc(eq_mask)
leader = 0
Pick leader
leader = __ffs(eq_mask)
dirty = false
Check if any thread dirtying CL
dirty = __any_sync(eq_mask, dirty)
Am leader?
Yes
No
Acquire CL & Get address of CL
addr = cache->acquire(cl, count, dirty)
Sync & Broadcast CL address
addr = __shfl_sync(eq_mask, addr)
addr = 0x...
Release CL
addr = cache->release(cl, count, dirty)
Barrier
addr = __syncwarp(eq_mask)
1
2
3
4
5
6

(a) Acquiring a cache-line (b) Releasing a cache-line

Figure 7.1: Flow diagram for warp coalescing when acquiring and releasing a cache-line. We use a warp with four threads as an example.

as the calling thread. Note that threads calling this primitive with the same `mask` and value to be compared will get the same `eq_mask` returned. From that mask, the threads know how many other threads in the warp have the same value, can easily pick a leader, and check if any thread is attempting to mark the cache-line dirty.

```
__device__
void coalesce(size_t val, const uint32_t mask, uint32_t& eq_mask, uint32_t&
  count, uint32_t& leader, bool& dirty)
{
  //mask notifying which threads have same val
  eq_mask = __match_any_sync(mask, val);
  //count of threads who have same val
  count   = __popc(eq_mask);
  //selected thread for making cache access
  leader  = __ffs(eq_mask);
  //see if any coalescing thread is going to dirty cache-line
  dirty = __any_sync(eq_mask, dirty);
}
```

Listing 7.5: Warp coalescing routine.

The `bam::array` abstraction uses this coalescer to implement new `coalesced_acquire_cl` and `coalesced_release_cl` APIs, shown in Listing 7.6 and Figure 7.1. Each thread calls the `coalesce` function with the ID of the cache-line (`cl_id`) it is accessing. This allows each thread to determine which other threads in the warp are accessing the same cache-line (1), the number of threads in the warp accessing the same cache-line (2), a leader (3), and if any thread is attempting to mark the cache-line dirty (4). When acquiring the cache-line, only the leader queries the cache and manipulates the requested cache-line's state (5), as shown in Figure 7.1a. The threads in the group synchronize using the `__shfl_sync` primitive, and the leader broadcasts the address of the requested offset in the GPU memory to the group

(6). When releasing the cache-line, only the leader updates the cache-line's reference count and state (5), and all threads wait for their leader to finish using a barrier (6), as shown in Figure 7.1b. The `cache->acquire_cl` and `cache->release_cl` calls in the `bam::array`'s subscript operator and `write` method can be replaced with the new `coalesced_acquire_cl` and `coalesced_release_cl` APIs, respectively, to get the benefit of warp coalescing.

```
template<T>
__device__
T* bam::array::coalesced_acquire_cl(const size_t cl_id, bool dirty=false)
{
  uint32_t mask = __activemask();
  uint32_t eq_mask, count, leader;
  T* base_addr;
  coalesce(cl_id, mask, eq_mask, count, leader, dirty);

  //leader makes cache access on behalf of all threads in eq_mask
  if (get_lane_id() == leader)
     base_addr = cache->acquire(cl_id, count, dirty);

  //broadcast addr
  base_addr = __shfl_sync(eq_mask, base_addr);

  return base_addr;
}

__device__
void bam::array::coalesced_release_cl(const size_t cl_id, bool dirty=false)
{
  uint32_t mask = __activemask();
  uint32_t eq_mask, count, leader;
  T* base_addr;
  coalesce(cl_id, mask, eq_mask, count, leader, dirty);

  //leader makes cache access on behalf of all threads in eq_mask
  if (get_lane_id() == leader)
     base_addr = cache->release(cl_id, count, dirty);

  //wait for leader to finish
  __syncwarp(eq_mask);
}
```

Listing 7.6: `bam::array`'s cache-line acquire and release routines that leverage coalescing.

Performance Impact: Now we evaluate the performance benefit of using the coalesced access APIs to reduce the number of cache probes with the `faa` cache design. Recall the GPU kernel used to evaluate the BaM cache in § 6.5 assigns each GPU thread a single 8-byte element to access through the cache and that threads with consecutive thread IDs access consecutive elements in the data. This access pattern is exactly the result of launching the

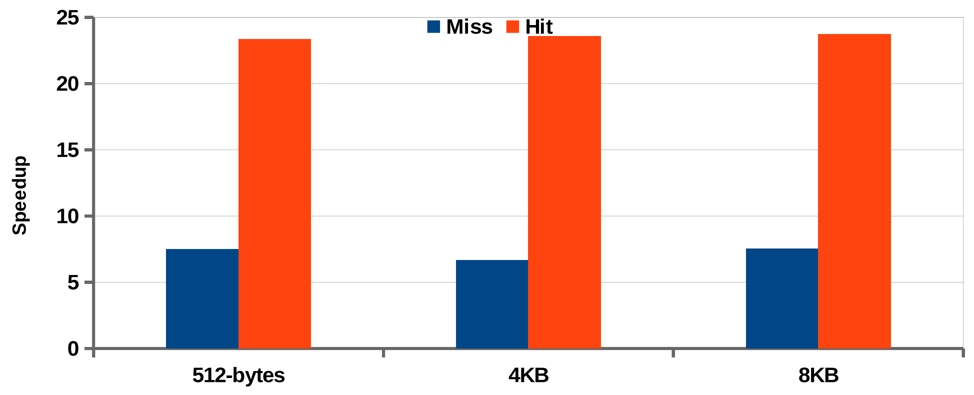


Figure 7.2: Improvement in 8-byte element lookup hit throughput and miss throughput with warp coalescing enabled over the results in § 6.5. 32-million threads are launched in this evaluation and each thread accesses a single 8-byte element.

kernel in Listing 7.1 with the `elems_per_thread` argument set to 1. Thus the threads in a warp access consecutive elements, and their accesses can be coalesced for a theoretical peak performance improvement of 32. We execute the same kernel with 32-million threads using the same Nvidia A100 GPU and 4 Intel Optane P5800X 1.6TB SSDs in the system described in Table 8.1 and §8.1 except with the coalescing optimization enabled.

Figure 7.2 shows the improvement in effective miss and hit 8-byte element lookup throughput compared to the performance of the `faa` cache design reported in § 6.5. The benefit of threads in a warp not contending with each other when probing the cache, as well as the reduction in cache probes, improves performance for both the hit throughput and the miss throughput. Miss throughput is improved to 1.19 billion look-ups-per-second (7.51× speedup), 1.51 billion look-ups-per-second (6.67× speedup), and 1.72 billion look-ups-per-second (7.53× speedup), for 512-byte, 4KB, and 8KB cache-line sizes, respectively over the performance reported in § 6.5. Hit throughput is improved to 5.47 billion look-ups-per-second (23.35× speedup), 5.79 billion look-ups-per-second (23.56× speedup), and 5.82 billion look-ups-per-second (23.74× speedup) for 512-byte, 4KB, and 8KB cache-line sizes, respectively over the performance reported in § 6.5. Speedups are higher for the hit throughput as when the data is in the cache, the cache access overhead due to contention is most exposed. Note, the coalescing does not provide the peak 32× speedup due to the overhead of coalescing in software which requires a non-trivial amount of registers and number of instructions executed.

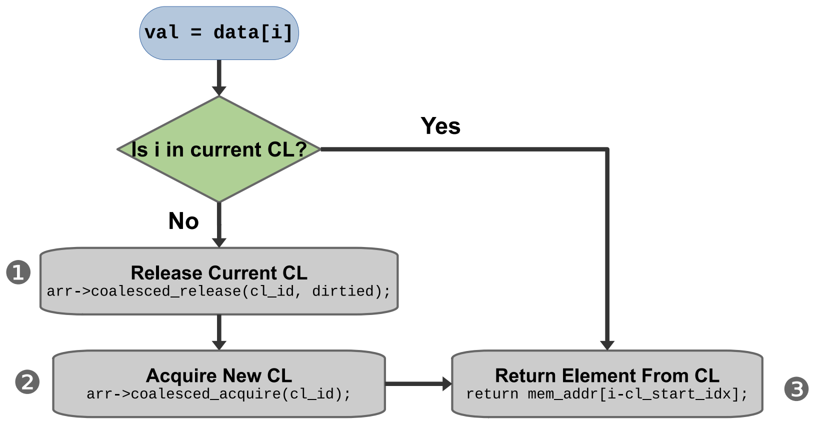

val = data[i]
Is i in current CL?
Yes
No
1
Release Current CL
arr->coalesced_release(cl_id, dirtied);
2
Acquire New CL
arr->coalesced_acquire(cl_id);
Return Element From CL
return mem_addr[i-cl_start_idx];
3

Figure 7.3: Flow diagram for an element access through the `bam::array_ref` abstraction.

7.2.2 Reusing cache-line reference in a thread

A single thread can also cause significant cache-line metadata access traffic if it accesses the same cache-line repeatedly. This can happen when a thread or a warp is assigned to traverse a contiguous chunk of data, such that each thread must access multiple elements for the whole chunk to be traversed. The GPU kernel in Listing 7.2 shown earlier is an example of such an access pattern. Since cache-lines can be large (from 512 bytes to many kilobytes in size), it's likely that many accesses of the thread will hit in the same cache-line. Thus, a thread will repeatedly acquire and release the same cache-line, possibly exposing cache access latency.

To address this problem, we propose the `bam::array_ref` reference abstraction shown in Listing 7.8, which allows each thread to keep a local reference of a cache-line in its registers. A thread can construct a `bam::array_ref`, with the address of the `bam::array` instance, in its local memory. Then the thread can access data elements at specific indices through the `bam::array_ref` using the overloaded subscript operator, just like a C-style array and `bam::array`, as shown in Listing 7.7 and Figure 7.3. The overloaded subscript operator checks if the newly accessed index falls in the cache-line to which the reference points. If it does, the operator returns the data from the address of the cache-line in memory directly (3), without having to access the cache. Otherwise, the operator releases the reference (1), acquires the reference for the new cache-line (2), and then returns the data from the newly acquired cache-line (3). Note, when releasing a cache-line (1), the abstraction must also notify the cache if the thread had written to the cache-line by passing the `dirtied` predicate to the `coalesced_release_cl` call. For an access pattern like in Listing 7.7, this new abstraction only requires 1 cache metadata access to acquire the cache-line and 1 cache metadata access to release the cache-line for all 32 accesses to the cache-line from a warp (considering perfect coalescing). As long as each thread accesses the same cache-line, i.e., it reuses the reference, it will not need to access the cache metadata for reference counting, reducing contention.

```
__global__
void kernel(bam::array<unsigned> data, size_t elems_per_thread, unsigned* ret)
{
  //create ref object
  bam::array_ref ref(&data);
  size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
  size_t wid = tid / 32;
  size_t lane = tid % 32;
  size_t start = wid * elems_per_thread * 32 + lane;
  size_t end = (wid+1) * elems_per_thread * 32;
  for (; start < end; start+=32)
      *ret += ref[start];  //access data through ref
}
```

Listing 7.7: GPU linear access benchmark kernel code using the `bam::array_ref` abstraction.

When the instance of `bam::array_ref` goes out of scope, the destructor releases the cache-line to which the reference points. This guarantee that the release will eventually be called, either if the reference gets updated or it goes out of scope, allows us to implement the subscript operator that returns the reference of an element. Thus the `bam::array_ref` abstraction enables us directly replace the array types used by GPU kernels and still use the subscript operator for both reads and writes. The subscript operator that returns the reference to the element in GPU memory, for writes, also sets the `dirtied` predicate in the `bam::array_ref` instance. This `dirtied` predicate is passed to the `coalesced_release_cl` call when releasing the cache-line (1) to make sure the cache-line's state is appropriately updated.

```
template<T>
struct bam::array_ref {
  bam::array*  arr;
  T*           cl_base_addr;
  size_t       cl_id;
  size_t       cl_start_idx;
  size_t       cl_end_idx;
  bool         dirtied;

  __device__
  array_ref(bam::array* a)
    : arr(a),
      cl_base_addr(nullptr)
  {}
```

Listing 7.8: BaM's `bam::array_ref` abstraction and its methods.

```
    __device__
    void clean()
    {
      if (cl_base_addr != nullptr) {
        //release cache-line
        arr->coalesced_release_cl(cl_id, dirtied);
        cl_base_addr = nullptr;
      }
      cl_start_idx = 0;
      cl_end_idx = 0;
      dirtied = false;
    }

    //release kept reference on destruction
    __device__
    ~array_ref() { clean(); }

    __device__
    void update_cl(const size_t i)
    {
      //release previous reference
      clean();
      cl_id  = get_cl_id(i);
      //acquire cache-line
      cl_base_addr = arr->coalesce_acquire_cl(cl_id);
      cl_start_idx = arr->get_start_idx(cl_id);
      cl_end_idx = arr->get_end_idx(cl_id);
    }

    __device__
    T operator[](const size_t i) const
    {
      //if index does not fall in current reference's cache-line
      //get new reference
      if ((i < cl_start_idx) || (i >= cl_end_idx))
          update_cl(i);
      return cl_base_addr[i - cl_start_idx];
    }

    __device__
    T& operator[](const size_t i)
    {
      //if index does not fall in current reference's cache-line
      //get new reference
      if ((i < cl_start_idx) || (i >= cl_end_idx))
          update_cl(i, true);
      dirtied = true;
      return cl_base_addr[i - cl_start_idx];
    }
  };
```

Listing 7.8 (cont.): BaM's `bam::array_ref` abstraction and its methods.

Performance Impact: We now evaluate the performance benefit of reference reuse with the coalescer to further reduce the number of cache probes compared to only the coalescer optimization results reported earlier in §7.2.1. Recall the GPU kernel, shown in Listing 7.2, used to evaluate the BaM cache in § 7.2.1 assigns each GPU thread a single 8-byte element to access through the cache and that threads with consecutive thread IDs access consecutive elements in the data. We had launched 32 million threads and thus incurred 32 million accesses to the data. We can increase the value of the `elems_per_thread` argument to coarsen the number of 8-byte accesses done per thread and launch fewer threads to maintain a total of 32-million accesses. The coarsening assigns each warp a larger consecutive region of elements to iterate over, so the larger the coarsening factor, the more a reference acquired by a thread is reused. We launch the kernel with different coarsening factors using the same Nvidia A100 GPU and 4 Intel Optane P5800X 1.6TB SSDs in the system described in Table 8.1 and §8.1 except with the `bam::array_ref` data type that exploits both warp coalescing and reference reuse. We evaluate with 3 different cache-line sizes (512-byte, 4KB, and 8KB) and coarsening factors of 1, 2, 4, 8, 16, and 32 and report the results, shown in Figure 7.4, relative to the performance achieved with just warp coalescing and 32-million threads launch with each making 1 data access shown in Figure 7.2.

First, we observe that, using the `bam::array_ref` abstraction without any coarsening, i.e. a coarsening factor of 1, provides 3-15% reduction in performance compared to using the `bam::array` abstraction. This is due to the added register usage and condition check overhead required for the `bam::array_ref` abstraction. This overhead does not get amortized as the reference is never reused in this case.

Next, looking at the 512-byte cache-line size, we observe that reusing the reference at least twice provides an improvement of 1.85×, and at most 1.9×, for the hit throughput. A cache-line of 512-byte stores 64 8-byte elements, so even though a coarsening factor of 2 reduces the total number of threads by half, it guarantees different warps do not contend on the same cache-line. Given this, the 8-byte access hit bandwidth saturates at around 75GB/s with a coarsening factor of 2 and barely improves with added coarsening due to the metadata pollution in the GPU's last-level cache, which is severe for 512-byte cache-lines. The miss throughput barely improves with the 512-byte cache-line size, with a coarsening factor of 2 providing a peak improvement of 1.08×, providing a peak effective bandwidth of 9.58GB/s, 95% of the peak bandwidth achievable with 4 Intel Optane P5800X SSDs at 512-byte access granularity.

Considering the 4KB cache-line size, we observe that increasing the coarsening factor up to 16 linearly scales the hit throughput to an effective 8-byte access bandwidth of 330GB/s, an improvement of 7.67× over just using the warp coalescing optimization without coarsening

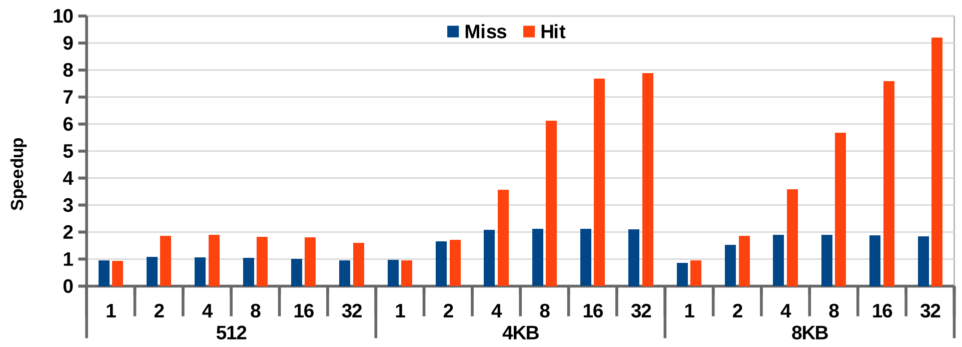


Figure 7.4: Improvement in 8-byte element lookup hit throughput and miss throughput with the `bam::array_ref` abstraction enabled over the results in Figure 7.2. We enable coarsening, allowing each thread to make 1, 2, 4, 8, 16, and 32 8-byte data accesses. A total of 32 million 8-byte data accesses are made in each case reported.

and reference reuse. Going to a coarsening factor of 32 only provides an improvement of 1.03× over the performance achieved with a coarsening factor of 16 as a 4KB cache-line stores 512 8-byte elements, provisioning 16 elements per thread in a warp to be reused with the `bam::array_ref` abstraction. The miss bandwidth improves by up to 2.11× compared to just using the warp coalescing optimization without coarsening and reference reuse, with a coarsening factor of 8, at which point the PCIe Gen4 x16 link of the GPU is saturated at 24GB/s.

Similar behavior is observed with the 8KB cache-line as well. Increasing coarsening up to 32 linearly scales the hit throughput providing a peak 8-byte access effective bandwidth of 398GB/s with the `bam::array_ref` abstraction. With a coarsening factor of 4, the miss bandwidth improves by 1.89×, compared to just using the warp coalescing optimization without coarsening and reference reuse, at which point the PCIe Gen4 x16 link of the GPU is saturated at 24GB/s. Overall, coarsening and leveraging the reference reuse optimization of the `bam::array_ref` abstraction provides 2 benefits. First, it reduces contention among threads over accessed cache-lines, improving hit throughput. Second, it exposes more I/O level parallelism making it easier for the GPU to switch from storage latency bound, where many threads are waiting for a few cache-lines to be fetched from storage, to storage bandwidth bound and thus improving the miss bandwidth.

Potential Drawbacks: There are 3 potential downsides to the `bam::array_ref` abstraction. First, as shown, it requires 6 new register per thread per `bam::array_ref` instance. Although this can increase register spilling, the overheads of the I/O in BaM are so high that we have yet to observe register spilling overheads exposed. This could change in the future. Second, to make the abstraction such that the programmer does not need to be aware of the cache-line size, the abstraction has to check if each access is in the range of the cache-line it currently holds. However, this check is far less costly than the reference counting overhead to acquire and release the cache-line on each element access.Third, if threads require multiple simultaneous instances of `bam::array_ref`, then cache-line eviction could suffer from a deadlock given a small enough cache and enough parallel threads requiring the same. This is because each instance of `bam::array_ref` implies a cache-line with a non-zero reference count, which cannot be evicted. Some threads might need to reference more cache-lines before releasing any of their previously acquired cache-lines resulting in the `find_slot` routine not being able to find a victim to evict. However, in our experiments with real applications, we have yet to run into this problem, even with threads requiring 6 simultaneous references, as BaM can support caches of many gigabytes in size in GPU memory.

7.3 HOST API AND INITIALIZATION

BaM initialization on the host requires allocating a few internal data structures that are reused during the application's lifetime. These structures include data structures like NVMe controllers, queues, and BaM cache metadata. Initialization can happen implicitly through a library construction. We note that when creating instances of `bam::array`, it is the programmer's responsibility to specify, through template and constructor arguments, the mapping of the array data to blocks on the storage as well as the BaM cache instance to use.

CHAPTER 8: IMPLEMENTATION OF BAM PROTOTYPE

We use off-the-shelf hardware including NVIDIA GPUs and arrays of NVMe SSDs to construct a BaM prototype and show the benefits of allowing GPUs to directly access storage with enough random access bandwidth to take full advantage of a GPU's PCIe Gen4 x16 link. *Once this level of data access bandwidth is achieved, a storage-based solution is as good as a host memory accessed through PCIe in terms of performance but a lot cheaper.* For simplicity, we describe the prototype assuming bare-metal, direct access to the NVMe SSDs. In this section, we outline the software (§ 8.2) and hardware (§ 8.1) required to realize such a prototype.

8.1 BAM PROTOTYPE HARDWARE

Scaling BaM using the PCIe slots available within a data-center grade 4U server comes with several challenges. The number of PCIe slots available in these machines is limited. For instance, the Supermicro AS-4124 system has five PCIe Gen4 $\times16$ slots per socket. If a GPU occupies a slot it can only access 4 $\times16$ PCIe devices without crossing the inter-socket fabric. Furthermore, due to the chiplet design of modern multi-core CPUs, even when the 5 PCIe devices per socket access each other, they must cross the intra-CPU fabric. Crossing these different interconnects results in severe performance degradation as packets must be translated for each interconnect, increasing latency and limiting throughput. However, as no single NVMe SSD can provide the throughput to saturate a PCIe x16 Gen4 link, BaM hardware must scale to a large number of NVMe devices to provide the necessary throughput to saturate the $\times16$ PCIe Gen4 GPU bandwidth.

To address this, we built a custom prototype machine for the BaM architecture using the off-the-shelf components as shown in Figure 8.1. Table 8.1 provides the specification of the major components used for the prototype. BaM prototype uses a PCIe expansion chassis with a custom PCIe topology for scaling the number of SSDs. The PCIe switches provide low-latency and high-throughput peer-to-peer access. The expansion chassis has two identical drawers, both of which are currently independently connected to the host. Each drawer supports 8 $\times16$ PCIe slots. We use one $\times16$ slot in each drawer for an NVIDIA A100 GPU and the rest of the slots are populated with different types of SSDs. Currently, each drawer can only support 10 U.2 (Optane or Z-NAND) SSDs as the U.2 form factor takes up significant space. As the PCIe switches support PCIe bifurcation, a PCIe multi-SSD riser card enables more than 16 M.2 NAND Flash SSDs per drawer.

Table 8.1: BaM prototype system specification

BaM Config	**Specification**
System	Supermicro AS-4124GS-TNR
CPUs	2× AMD EPYC 7702 64-Core Processors
DRAM	1TB Micron DDR4-3200
GPU	NVIDIA A100-80GB PCIe
PCIe Expansion	H3 Platform Falcon-4016
SSDs	Refer to Table 2.1
Software	Ubuntu 20.04 LTS, NVIDIA Driver 470.82, CUDA 11.5

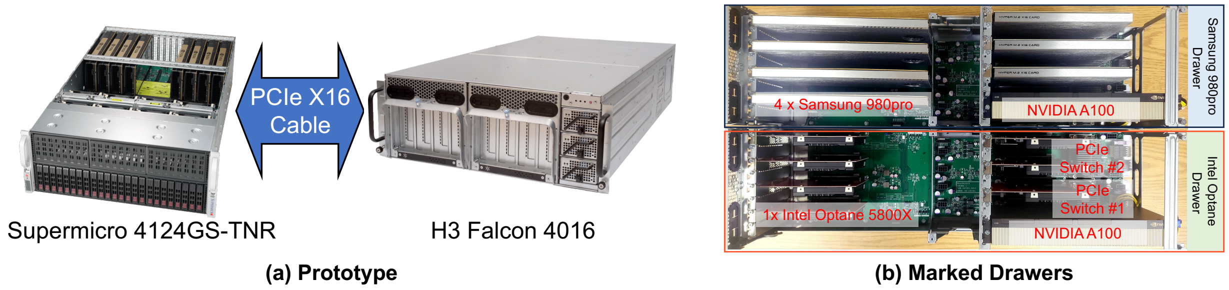


Figure 8.1: BaM prototype hardware (a) and drawers with marking (b)

SSD Technology trade-offs: Table 2.1 lists the metrics that significantly impact the design, cost, and efficiency of BaM systems for three types of off-the-shelf SSDs. The `RD IOPS` and `WR IOPS` columns show the measured random read and write throughput of each SSD at 512B and 4K granularity respectively. The `$/GB` column presents the cost per GB for each SSD type, based on the current list price per device, the expansion chassis, and the risers needed to build the system. The `Latency` column shows the measured average device latency in μs. A comparison of these metrics across SSD types shows that the consumer-grade NAND Flash SSDs are inexpensive with more challenging characteristics, while the low-latency drives such as Intel Optane SSD and Samsung Z-NAND are more expensive with desirable characteristics. For example, for write-intensive applications using BaM, Intel Optane drives provide the best write IOPs and endurance.

Irrespective of the underlying SSD technology, as shown in Table 2.1, the BaM prototype provides a 4.4-21.8× advantage in cost per GB, even with the expansion chassis and risers, over a DRAM-only solution. Furthermore, this advantage grows with additional capacity added per device, which makes BaM highly scalable as SSD capacity and application data size increase.

8.2 ENABLING DIRECT NVME ACCESS FROM GPU THREADS

Although BaM can be used with any queue-based storage system, for simplicity, we will use the NVMe SSD controllers as a simple example of storage controllers to explain the key features of the BaM prototype. In order to enable GPU threads to directly access data on NVMe SSDs we need to: 1) move the NVMe queues and I/O buffers from CPU memory to GPU memory and 2) enable GPU threads to write to the queue doorbell registers in the NVMe SSD's BAR space. To this end, we create a custom Linux driver that creates a character device per NVMe SSD in the system, like the one by SmartIO [49]. Applications use BaM APIs to `open` the character device for each SSD they wish to use.

In the custom Linux driver, BaM leverages GPUDirect RDMA features to pin and map NVMe queues and I/O buffers in the GPU memory. BaM uses the `nvidia_p2p_get_pages` kernel API to pin the pages of NVMe queues and I/O buffers in GPU memory and then maps these pages for DMA access from another PCIe device, like NVMe SSDs, using `nvidia_p2p_map_pages` kernel API. This enables the SSD to perform peer-to-peer data reads and writes to the GPU memory.

We leverage GPUDirect Async to map the NVMe SSD doorbells to the CUDA address space so GPU threads can ring the doorbells on demand. This requires the SSD's BAR space to be first memory-mapped into the application's address space. Then the BAR space is mapped to CUDA's address space using the `cudaHostRegister` API. Using `cudaHostGetDevicePointer`, the application gets the virtual address that the GPU threads can use to ring the NVMe SSD doorbell registers. We note that other storage systems can be enabled similarly.

8.2.1 GPUDirect RDMA I/O Consistency

As has been noted in prior works[50, 51], when a third-party device writes into the GPU's memory with GPUDirect RDMA over PCIe the order of these writes might not be preserved from the viewpoint of the GPU threads in the kernel concurrent to the writes. This creates an issue for BaM as GPU threads handling cache misses expect the data in the I/O buffer in GPU memory to be valid after polling for the completion queue entry for the corresponding request. Even though the SSD Controller will guarantee the writes of the requested data happen before the write of the corresponding commands completion queue entry, the GPU's internal routing can re-order these two writes such that even if the polling thread observes the completion queue entry for the submitted request, it could still read stale data, and thus incorrect, bytes when attempting to read the fetched data from the I/O buffer. Prior work

[50] has shown that this issue can be circumvented if after the thread finds a completion queue entry for its submitted command, it communicates to a CPU routine that issues a PCIe read to the GPU's memory. A PCIe read (from any device) to the GPU memory flushes all previous PCIe writes to the GPU memory so that they are all observable by the GPU threads. Only after the CPU routine finishes the read and notifies the GPU thread, can the thread be sure that the data in the I/O buffer is valid and it can proceed to mark the cache-line valid and use the data in GPU memory. However, this solution requires the GPU threads to communicate with the less parallel and far-away CPU, which consumes CPU cycles and wastes PCIe bandwidth.

We can leverage BaM's I/O stack and the NVMe protocol to address this issue without involving the CPU or any other new device. Recall, that any PCIe read issued to the GPU memory will flush all previous PCIe writes to that GPU's memory and that after a thread enqueues a command into the NVMe submission queue and rings the doorbell the controller will eventually read the command from the submission queue in GPU memory. A thread can be sure the controller has read the command when it observes a valid completion queue entry for the command posted in the appropriate completion queue. Thus, a simple solution would be that after a thread submits its I/O request to the SSD and finishes polling for its completion, it submits a second NVMe request and waits for its completion as well. Observing the completion of the second NVMe request guarantees that the controller issued the read for the second command which was submitted after the observation of the first entry's completion and it implies the previous writes were flushed and the data in the I/O buffer for the first NVMe command is guaranteed to be correctly observable by the GPU threads. However, this approach will reduce the effective throughput of BaM's I/O stack by at least half.

To reduce the total number of second NVMe requests submitted, we implement a shared global virtual queue in the BaM with two atomic unsigned integers, a `virt_head` and a `virt_tail`, and a `lock`. After a thread finishes polling for its first requests, it captures a copy of the current virtual queue head in a private variable and atomically increments the `virt_tail` to get a position in the virtual queue. Afterward, the thread uses these two values to determine if the `virt_head` has moved past the thread's assigned position, signifying that the thread's second I/O request was coalesced into a single request by some other thread. While the `virt_head` is not moved past the thread's assigned position, the thread loops attempting to take the virtual queue's `lock`, and if successful, it reads the current `virt_tail` value (`cur_virt_tail`), submits a second NVMe request and polls for its completion. After the thread finds the completion entry for the second NVMe request, it updates the virtual queue's `virt_head` to the `cur_virt_tail` value of the virtual queue's tail captured before submitting the second NVMe request. This update to the virtual queue's head effectively consumes entries in the virtual queue notifying all other threads assigned a position up to the `cur_virt_tail` value that their second NVMe request was coalesced into the single NVMe request just submitted and completed by this thread. The thread can then release the virtual queue's `lock`. When a thread observes that the `virt_head` has moved past its assigned position in the virtual queue, the thread can continue and assume all previous writes were flushed and the cache-line it fetched from storage is correctly visible to the GPU threads and can be marked valid. The virtual queue enables coalescing the second NVMe request from many threads, including those communicating with a different SSD, into a single request.

Performance Impact: We evaluate the overhead of our proposed scheme to circumvent the GPUDirect RDMA's I/O consistency problem with the Nvidia A100 GPU and 4 Intel Optane P5800X 1.6TB SSDs in the system described in Table 8.1 and §8.1. We use the BaM I/O stack and launch a kernel where each thread submits 1 needed NVMe read I/O request. We evaluate 3 different I/O request sizes: 512-byte, 4KB, and 8KB. We use a 512-byte read request as the second request submitted due to our proposed scheme to circumvent the GPUDirect RDMA's I/O consistency problem. We vary the number of threads in the kernel launch from 1024 to 131072 and measure the effective I/O throughput.

Figure 8.2 plots the slowdown in effective I/O throughput compared to using the BaM I/O stack without the fix. Issuing 512-byte reads with a low degree of I/O-level parallelism (1024 and 2048 requests) experiences a 28% overhead with the proposed scheme due to the added latency of the extra operations. Given enough I/O-level parallelism (at least 8192 requests), the overhead reduces to less than 2.5%. The overhead for 4KB and 8KB reads is always lower than 15% and given enough I/O-level parallelism (at least 4096 requests), it is reduced to less than 2.6%. Figure 8.3 shows that the number of second NVMe operations submitted for each of the evaluated cases is at most 8, significantly lower than if each NVMe request required the submission of a second request. This result shows that the coalescing of the virtual queue entries is extremely effective.

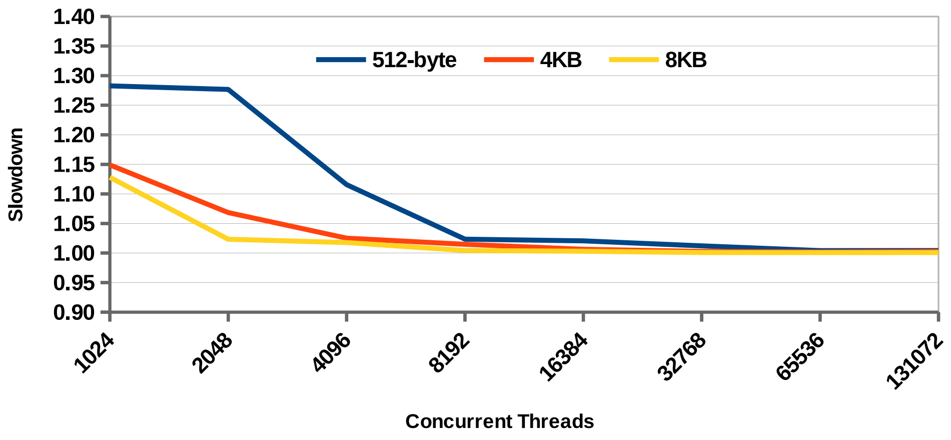


Figure 8.2: Slowdown of the BaM I/O stack with the proposed scheme to circumvent the GPUDirect RDMA's I/O consistency problem compared to the BaM I/O stack without the proposed scheme to circumvent the GPUDirect RDMA's I/O consistency problem.

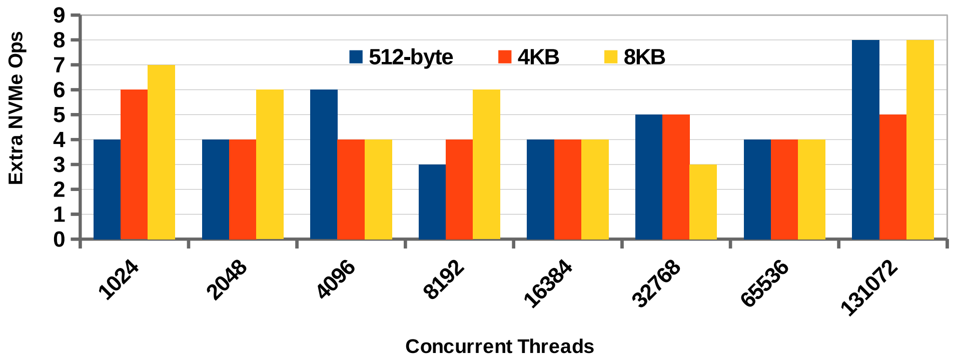


Figure 8.3: Number of extra (second) NVMe requests submitted by the BaM I/O stack because of the proposed scheme to circumvent the GPUDirect RDMA's I/O consistency problem.

CHAPTER 9: END-TO-END APPLICATION EVALUATION

As described earlier, emerging GPU applications work on data that does not fit in GPU memory. Furthermore, the state-of-the-art solutions to circumvent the limited GPU memory either suffer extreme inefficiencies in I/O amplification and/or storage and memory management where the system cannot saturate the underlying storage system. In this chapter, we show that BaM naturally excels in the emerging applications, specifically graph traversal applications and executing data-dependent analytical queries on large data-sets. We first evaluate the benefit of each component of BaM and the respective optimizations on the performance of two graph analytics applications in § 9.2. Then, we evaluate the sensitivity of BaM's application performance to the number of (§ 9.4) and type (§ 9.3) of SSDs. Finally, we compare BaM's performance against the state-of-the-art in each of the applications evaluated.

Overall our evaluation shows that BaM is a cost-effective solution to extend effective GPU memory capacity, with performance on par with, for graph analytics, or up to 4.52× better than, for data analytics applications, more expensive DRAM-based state-of-the-art GPU-accelerated solutions.

9.1 EXPERIMENT SETUP

For all experimentation we use the system described and specified in § 8.1 and Table 8.1. To truly test the limits of the BaM infrastructure components, we use the fastest storage combination we could find to serve as the backing storage for BaM. That is, all experiments with BaM unless otherwise stated are executed with 4 Intel Optane P5800X 1.6TB SSDs with an I/O access granularity of 4KB with 128 queue pairs and each queue having a depth of 1024. Each SSD provides 1.6Millions IOPS at 4KB, thus a bandwidth of 6.4GB/s, and with 4 SSDs BaM can easily saturate the PCIe Gen4 x16 link into the GPU at around 24GB/s. We replicate data across the SSDs and use round-robin scheduling to uniformly schedule I/O operations across the 4 SSDs at run-time. Using such a fast storage system helps us isolate the bottlenecks in BaM, motivating the optimizations we implement in this and future works.

Table 9.1: Graph Analytics Data-sets.

Graph	**Num. Nodes**	**Num. Edges**	**Size (GB)**
GAP-kron (K) [52]	134.2M	4.22B	31.5
GAP-urand (U) [52]	134.2M	4.29B	32.0
Friendster (F) [3]	65.6M	3.61B	26.9
MOLIERE_2016 (M) [4]	30.2M	6.67B	49.7
uk-2007-05 (Uk) [53, 54]	105.9M	3.74B	27.8

9.1.1 Graph Analytics Evaluation Setup

We use graph analytics workloads to evaluate BaM as they are becoming fundamental in many new applications. Achieving high performance for these applications is tough as the access patterns are data-dependent: visiting a node dictates which other nodes (neighbors) must be visited. We use the graphs listed in Table 9.1 for the evaluation. K, U, F, M are the four largest graphs from the SuiteSparse matrix collection [55] while the Uk is taken from LAW [56]. These graph data-sets cover diverse domains, including social networks, web crawls, bio-medicine, and synthetic graphs.

We run two graph analytics algorithms, Breadth-first-search (BFS) and Connected Components (CC), with the target system and BaM with different SSDs listed in Table 2.1. We port the state-of-the-art GPU implementations for both applications [5] to BaM. Doing so requires minimal code changes as described in §7.1. For each iteration of the traversal, we assign 8 consecutive nodes in the graph to a warp. For BFS, we report the average run time after running at least 32 source nodes with more than two neighbors. We do not execute CC on the Uk data-set since CC operates only on undirected graphs. Lastly, we fix the BaM software cache size to 8GB and the cache line size to 4KB.

9.1.2 Data Analytics Evaluation Setup

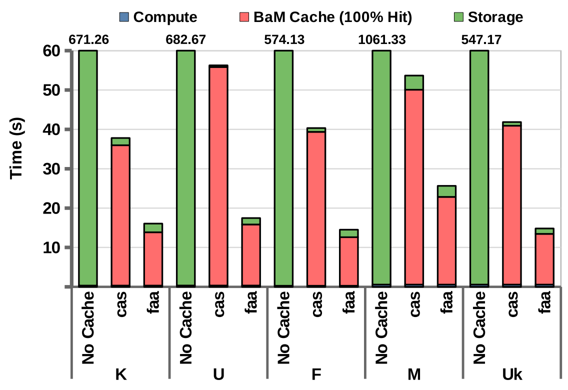
We also evaluate the performance benefit of BaM prototype for enterprise data analytics workloads. These emerging data analytics are widely used to interpret, discover or recommend meaningful patterns in data that is collected over time. As discussed in § 3.1, when GPUs are used for processing data-dependent queries, state-of-the-art solutions exhibit high I/O amplification and management overheads. As BaM enables on-demand data accesses through a fixed-size high-throughput cache, it can address both of these issues. We port the 5 data-dependent queries discussed in § 3.1 to BaM and execute them over the NYC taxi ride data-set[57]. The 200GB data-set consists of 1.7-billion trips with 49 metrics per trip and is stored in the ORC[58] format. All metrics relevant to the 5 queries are stored as 8-byte double-precision floating-point values in the ORC format. The queries look at all trips and aggregate the values of up to 5 other metrics (i.e., columns) for the trip if the `trip_distance` is within a user-specified bound. The user can change this bound to vary the sparsity of the rows for which data-dependent accesses are made. We evaluate the queries with two bounds, (1) the `trip_distance` is at least 20 miles and (2) the `trip_distance` is at least 30 miles. Of the 1.7-billion trips, 0.47% are at least 20 miles long, and 0.03% are at least 30 miles long. We also add a new query, `Q0`, that just scans the `trip_distance` in order to isolate the overhead of the data-dependent accesses. The GPU kernels to execute the queries in BaM assign a warp of 32 threads to process 512 consecutive rows of the data.

9.2 COMPONENTS OF BAM

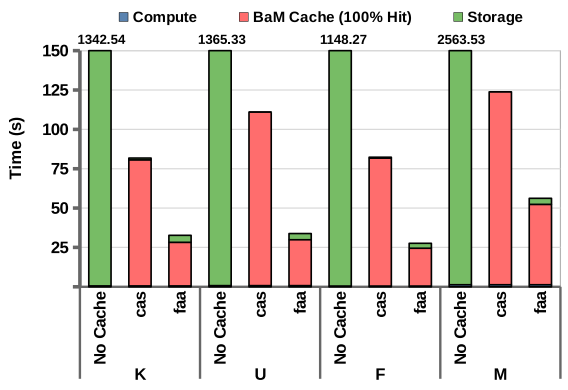
We first evaluate each component and optimization in BaM with the graph analytics applications. We use an Nvidia A100 GPU that has enough on-board memory (80GB) to host entire data-sets in order to isolate overheads of each component of the BaM and evaluate the impact of each system optimization. With this GPU and for each case, we execute 3 runs: (1) executing with all data in GPU memory (without any BaM software), (2) pre-loading all data into a large enough BaM cache in GPU memory and then executing with all data in the BaM cache, and (3) limiting the BaM cache size to 8GB and executing with a cold cache and fetching data from storage as needed. The execution time measure for (1) is the peak performance we can ever expect from the system, i.e., the BaM cache cannot be faster than the GPU memory, so the difference between the execution of (2) and (1) gives an estimate of the observable (non-overlapped) overhead of the BaM cache when the data hits in the cache. Similarly, as the optimal case for accessing storage, (3), is when there is no storage access, the difference between (3) and the sum of (1) and (2) gives us an estimate of the observable overhead of accessing storage with BaM. All evaluation graphs in this section will show all 3 components to expose how the overheads shift as the stack gets optimized. When evaluating the BaM cache and abstraction optimizations, we use BaM's most optimized storage I/O queue implementation to make sure that the effects of a slow I/O stack do not add noise to evaluating optimizations with the cache. Similarly, when evaluating the performance BaM's queues, we use the most optimized cache and abstraction with the application.

9.2.1 Benefits of Caching

First, we evaluate the effectiveness of BaM's cache to reduce storage I/O traffic. Without a cache, every data access from a thread triggers an I/O request for the appropriate data



(a) BFS



(b) CC

Figure 9.1: Execution time of BaM running BFS and CC with no cache, the `cas` cache implementation, and the `faa` cache implementation. Abstraction level optimizations are not enabled here. The no-cache bars are labeled with their execution time in seconds on top.

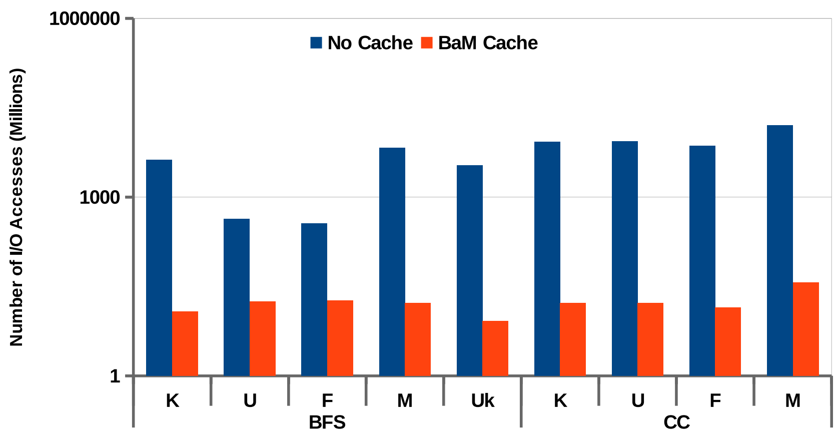
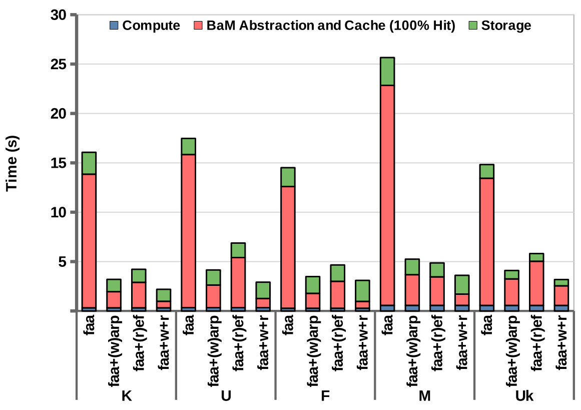


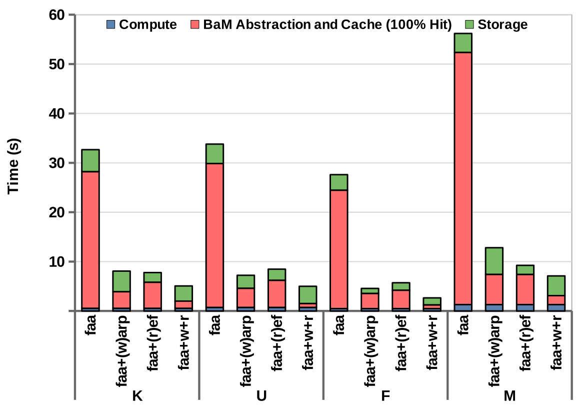
Figure 9.2: Number of I/O requests for BFS and CC with a cache and without a cache.

block. Once a data block is brought into GPU memory, it is never reused by the requesting thread or any other thread. This leads to significant I/O traffic, and thus the storage access bandwidth bounds the performance. As Figure 9.2 shows, BaM's `cas` cache reduces the number of storage I/O requests by 308× on average for BFS and 493× on average for CC. BaM's cache allows any thread to reuse any data brought from the storage by any other thread as long as it was not evicted. However, the reuse exploited by BaM's cache directly translates to contention in the cache that results in the high cache access overhead. Thus, adding BaM's `cas` cache improves the overall performance by only 15.1× and 15.5× for BFS and CC, respectively, as shown by Figure 9.1

Optimized Reference Counting: In § 6.4, we proposed optimizing the overhead of a cache access by changing the reference counting scheme for cache line state management to



(a) BFS



(b) CC

Figure 9.3: Execution time for BaM running graph workloads with and without the warp coalescing (warp) and reference reuse (ref) abstraction optimizations.

one that used much lower overhead atomic operations, `fetch_add` (`faa`), when compared to the cache management scheme using `compare_and_swap` (`cas`). Figure 9.1 also shows the effect of this optimization on the performance of the BFS and CC applications. As the optimization reduces the cache accesses overhead by 2.9× and 3.1× on average for BFS and CC, respectively, the storage overhead, mainly in the form of I/O access latency, starts being exposed. Thus, the effect of the optimization on overall execution is 2.6× and 2.7× on average for BFS and CC, respectively.

9.2.2 Benefits Of Abstraction

Next, we evaluate the two optimizations provided by BaM's abstractions that reduce the number of cache probes with the means of warp coalescing and reference reuse. Note that these optimizations are orthogonal, so we will first evaluate the performance of BaM on the graph analytics applications with each one individually and then with both.

Warp Coalescing: The state-of-the-art graph traversal application implementations that we ported to BaM assign one warp of 32 threads to a node being traversed in the graph. The threads in the warp access consecutive elements in the neighbor list of the assigned node, i.e., coalesced accesses. This is efficient for both the work parallelization, as all 32 threads are available to be used to read the list, and the memory subsystem because the SM can coalesce the memory accesses of threads in a warp into fewer accesses to the memory subsystem if the threads access neighboring elements. The warp coalescing implemented in BaM's abstractions aims to get back the efficiency of this work assignment, as it reduces the number of cache probes if threads in a warp access the same cache line, thus reducing overall cache overhead. As shown in Figure 9.3, enabling warp coalescing improves BaM's overall performance, by 4.35× and 4.73× on average for BFS and CC, respectively. This performance improvement is due to the reduction of the cache overhead by 6.91× and 7.96× on average for the two applications.

Thread Reference Reuse: If the edge-list assigned to a warp is longer than 32 elements, then at least one thread in the warp will need to make at least a second access to the data. The reference reuse capability of the `bam::array_ref` abstraction can be used here to avoid the cost of reacquiring cache-lines on future accesses to the same cache-line by a thread. This optimization also improves the overall performance and reduces the caching overhead of BaM for both BFS and CC when compared to just the cache on its own. Performance is improved by 3.32× for BFS and 4.71× for CC on average, and cache overhead is reduced by 4.38× and 6.21×, respectively, over each thread just accessing the cache. However, in most cases, the improvements are not as great as with the warp coalescing optimization. This means that in these graphs, during traversal, most of the nodes touched have neighbor lists that are long enough that a warp can be used to parallelize the access to the neighbor-list but not long enough that the overhead of each thread having to acquire the same cache-line multiple times is a bottleneck.

Warp Coalescing and Thread Reference Reuse: Of course, one can use both optimizations when traversing graphs. In fact, having both optimizations not only improves the overall performance of BFS and CC across all graphs (by 5.85× and 7.75×), but also greatly reduces the caching overhead (by 15.03× and 28.06×). At this point, the performance of BaM is bottlenecked by the storage bandwidth or latency as the storage part of execution contributes to at least 53% of the execution time for both applications and across all but one graph. The exception to this is executing BFS on the Uk graph, where the cache overhead still constitutes 62% of the execution time. This unique case involves a very deep BFS traversal with many iterations, where the frontier, i.e. the number of nodes visited, of most of the iterations is very small. If the frontier is small and the neighbor lists for the nodes being accessed are also small then we get a very small number of data accesses per iteration, leading to the cache access latency being exposed.

9.2.3 Benefits of Improving Queue Performance

After all the optimizations that reduce the cache access overhead, the storage performance becomes the primary performance bottleneck for the graph analytics applications studied, as mentioned. Now, we take the graph application implementations using the optimized cache with both abstraction-level optimizations and vary the queue design to application-level

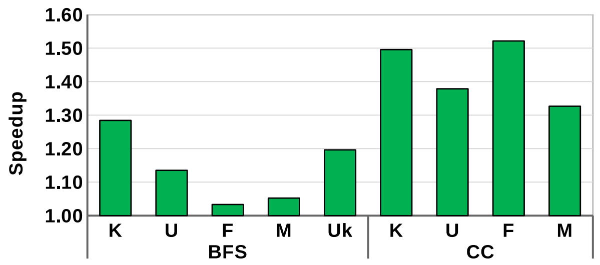


Figure 9.4: Speedup of BaM for BFS and CC with optimized queue implementation the cleans up stale entries faster.

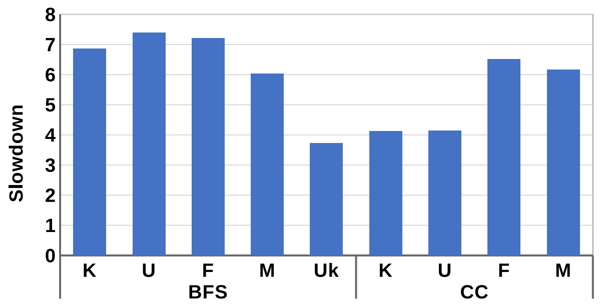
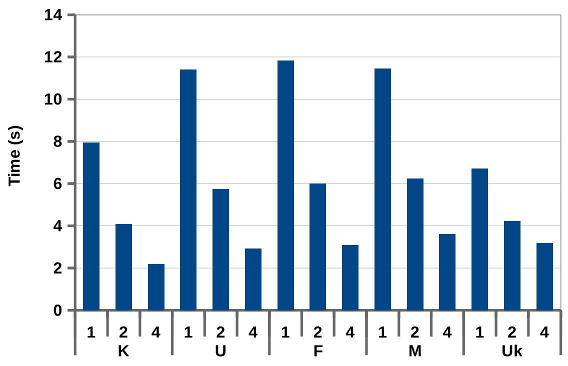


Figure 9.5: Slowdown of BaM with four Samsung 980 Pro SSDs compared to four Intel Optane SSDs for BFS and CC graph workloads.

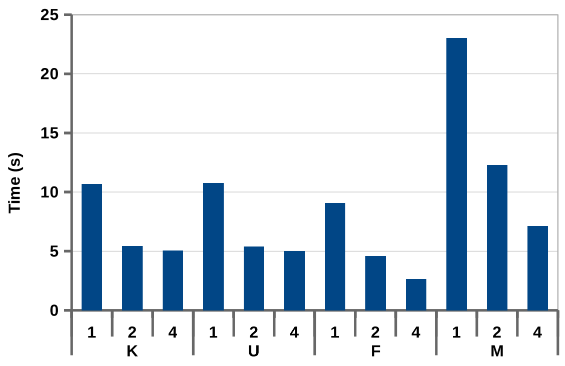
evaluate the benefit of cleaning up stale queue entries faster. Figure 9.4 shows the speedup achieved for the two applications and the different data-sets with using the optimized queue, with an average speedup of 1.14× for BFS and 1.43× for CC. In general, the CC application ends up reading a lot more data than the BFS application, as it starts by traversing every node in the graph in its first (and often second) iteration, whereas in BFS, a root node is chosen and only that node is traversed. Thus, the CC application is more I/O demanding, so any added latency, like the bubbles created with stale SQ entries, can expose the storage access latency more often.

9.3 VARYING TYPES OF SSDS

Now we evaluate the performance of BaM with consumer-grade and cheaper, compared to Intel Optane, Samsung 980 Pro SSDs for the graph workloads. As Figure 9.5 shows, BaM only slows down by 6.07× (BFS) and 5.11× (CC), on average, when using the slower SSDs. This means BaM is able to hide some of the 10× latency increase and more than 2× bandwidth reduction that we get with four Samsung 980 Pro SSDs compared to four Intel Optane SSDs.



(a) BFS



(b) CC

Figure 9.6: Execution time for BaM running graph workloads with one, two, and four Intel Optane SSDs.

9.4 SCALING SSDS

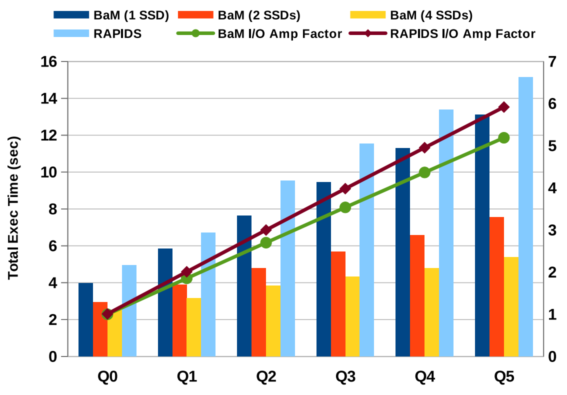
Now we see how BaM's performance scales as we vary the number of SSDs from one to four to 4. With 4 SSDs at 4KB access granularity, we can nearly saturate the PCIe Gen4 x16 link to the GPU.

9.4.1 Graph Workloads

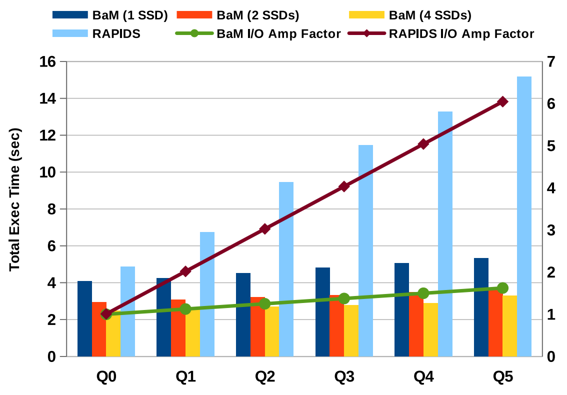
We first evaluate BaM's scaling performance for the two graph applications is shown in Figure 9.6. Generally, BaM scales linearly for both applications and all graphs from one to two SSDs. For BFS, the performance also scales linearly to four SSDs as well, except in the case of the Uk graph where there are many very small frontiers and the neighbor lists for the nodes being accessed are also small; then we get a very small number of data accesses per iteration, leading to the cache access latency being exposed. For CC, there is very minimal scaling with four SSDs for the K and U graphs where the application does not expose enough I/O level parallelism to exploit the added bandwidth as most nodes in these graphs have very short neighbor-lists, so many warps contend for the same cache-lines. Thus, for these graphs, the CC application becomes storage latency bound.

9.4.2 Data Analytics Workload

We report BaM's scaling performance for the data analytics workload in Figure 9.7. With a sparsity of 0.47%, adding a second SSD to BaM improves the end-to-end performance for all queries by up to 1.73×. Scaling from two SSDs to four SSDs provides only up to 1.4×



(a) 20 miles (Sparsity=0.47%)



(b) 30 miles (Sparsity=0.03%)

Figure 9.7: Execution time of BaM and RAPIDS for data analytics queries on NYC Taxi dataset. BaM is up to 2.82× and 4.62× speed up over the CPU-centric RAPIDS framework for 0.47% and 0.03% sparsity.

as BaM's setup overhead for pining and mapping I/O queues and I/O buffers for DMA in GPU memory required before query execution starts to be exposed. Note this overhead scales with the number of SSDs in the system as the I/O buffers must be mapped for DMA with each SSD being used to allow any SSD to perform a DMA to any I/O buffer in GPU memory.

The effect of this overhead on BaM's scaling can be even more drastic when the time to execute the query on the GPU is reduced, like in the case of executing the queries with a sparsity of 0.03%. In this case, the time to execute the query on the GPU is up to 3.45× less than with the sparsity of 0.47%. For this reduced sparsity, BaM's end-to-end time, including the setup overhead, only scales by up to 1.46× with two SSDs and 1.62× with four SSDs when compared to using BaM with one SSD.

9.5 COMPARISON TO STATE-OF-THE-ART

Next, we compare BaM with all of its optimizations against the state-of-the-art GPU systems for processing data-sets larger than GPU memory for each application.

9.5.1 Graph Analytics

A goal of the BaM is to provide competitive performance against the DRAM-based solutions. To this end, the target baseline `T` allows the GPU threads to directly perform coalesced fine-grain access to the data stored in the host-memory during graph analytics execution [5].

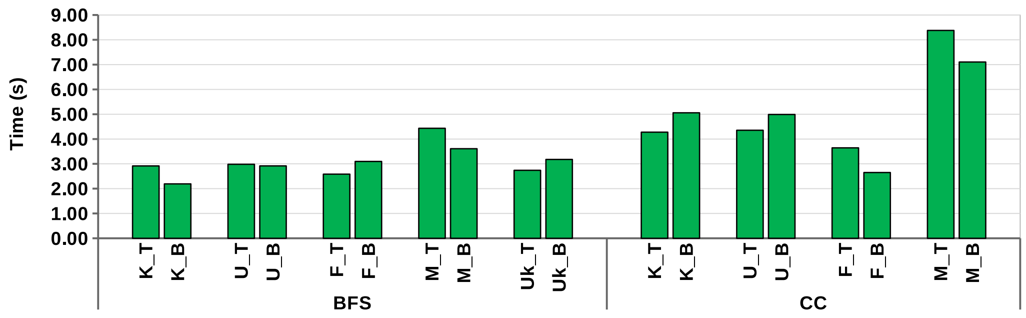


Figure 9.8: Execution time of BaM (B) and the Target system (T) running BFS and CC with 4 SSDs.

As there is abundant CPU memory for the input graphs used in this experiment, we can make a direct comparison of the performance between BaM and `T`. However, this approach requires the application to first load the data from the storage to the host memory before it can start the computation on the GPU. We use four Intel SSDs for both the target baseline and BaM and the respective execution time is shown in Figure 9.8. BaM, without using any CPU DRAM, is able to keep up with the target baseline across the board, achieving a speedup of 1.04×(BFS) and 1.05×(CC) on average.

9.5.2 Data Analytics

Now we compare the end-to-end performance of BaM for executing data analytics queries against the state-of-the-art GPU accelerated data analytics framework, RAPIDS v21.12 [6]. We execute the same queries mentioned earlier on the NYC taxi ride data-set [57] kept in the ORC format. Recall from § 3.1, that since the data-set is larger than GPU's memory, RAPIDS must partition the data and compute and aggregate the results across partitions. For this evaluation, we pin the entire ORC-encoded data-set in the Linux page cache in the 1TB of CPU memory, allowing RAPIDS to read the data directly from CPU memory without having to make any storage I/O requests. The system allows for a peak measured bandwidth of 24GB/s between the GPU and CPU memory. We do not use the CPU memory for BaM and instead replicate the ORC-encoded data-set on the four 1.6TB Intel Optane P5800X SSDs that BaM uses as backing memory. We report the end-to-end execution time and the measured data I/O amplification suffered for each query with each system in Figure 9.7.

With a sparsity of 0.03%, the I/O amplification incurred by RAPIDS increases linearly from 1.01×, when there is only one column accessed (`Q0`), to 6.04×, when there are a total of 6 columns accessed (`Q5`). This directly affects RAPIDS' execution time when executing the queries as more columns are added. In contrast, BaM suffers an I/O amplification of at most

1.6× (`Q5`), with the same sparsity. Furthermore, even when reading the data from the CPU DRAM, RAPIDS software overheads in finding needed bytes and managing GPU memory are so high that BaM, with just a single SSD (which only provides a bandwidth of 7GB/s), can outperform RAPIDS by 1.19× when accessing a single column (`Q0`) and 2.85× when accessing up to six columns (`Q5`). This is mainly attributed to BaM's significantly lower overhead in managing the GPU memory with the high-throughput software-managed cache, increasing the overall effective I/O bandwidth utilization. Overall, BaM, with four SSDs, outperforms RAPIDS, using CPU DRAM, by 1.96× when just accessing a single column (`Q0`) and 4.62× when accessing up to six columns (`Q5`).

When sparsity is reduced to 0.47%, RAPIDS suffers the same I/O amplification as with the higher sparsity since RAPIDS always moves all rows for all needed columns. With a page size of 4KB and the reduced sparsity, BaM suffers similar, though a bit lower, I/O amplification (up to 5.19×) as RAPIDS. However, BaM (with four SSDs) still consistently outperforms RAPIDS for all queries, by up to 2.82×. As both BaM and RAPIDS suffer similar I/O amplification in this case, BaM's performance advantage over RAPIDS is mainly due to lower overhead in finding needed bytes and memory management enabled by BaM's fixed-size cache. This allows BaM to expose more I/O level parallelism and to more easily saturate the GPU's PCIe Gen4 x16 link with 4 SSDs. In contrast, RAPIDS suffers significant overhead in finding data and managing GPU memory, as shown in Figure 3.1.

CHAPTER 10: BAM'S IMPACT, NEW RESEARCH OPPORTUNITIES, AND FUTURE WORK

BaM is a novel way of using GPUs (and their resources) and storage devices to expand the effective GPU memory capacity. As such, it opens up new research questions and opportunities. In this chapter, we discuss some of these directions, a couple of which are already being explored concurrently to this work.

10.1 FURTHER OPTIMIZATIONS FOR SYSTEMS SOFTWARE

BaM moves systems software, traditionally executed by the CPU, into the GPU application space. Optimizing these components for the GPU specifically was key to showing that moving them to the GPU, and thus allowing the GPU application to orchestrate fine-grain I/O accesses, was a worthwhile endeavor resulting in performance, cost, and efficiency benefits for important GPU applications. Although we have proposed optimization techniques and guidelines for each layer of the BaM stack and shown their impact on application performance, we are hoping the results of this work become the catalyst for researchers to study the layers further and optimize them further, to provide even more performance for applications. There is a plethora of research [24, 35, 36, 37, 38, 39, 47, 59] on optimizing these system's software components on the CPU, where they are traditionally run. We expect this work to be the base of a similar mountain for when the GPU (or another accelerator) needs to run systems software. We give some ideas for our future work for each layer based on the current bottlenecks and pain points of designing and implementing BaM.

Queues: As the overhead for user-level to kernel crossing increases due to issues like patching hardware bugs, many systems (including storage systems) are moving to queue-based communication with not only hardware but other systems services. In fact, previous work has even tried to support such communication between CPUs and GPUs and has relied on some sort of queues [17, 18, 47, 59]. As such, the design and performance of queues are vital for the performance of these systems. BaM took on the very challenging task of exploiting parallelism from NVMe, which is serial by definition.

However, we know the implementation can still be optimized in a few ways to provide lower storage access latency and higher I/O throughput for applications. For instance, currently, when threads poll for completion queue entries, each starts from the head and checks each valid entry to see if it is for the command the thread submitted. Clearly, there is redundant work being done here by many threads. Instead, one can create a map with an entry per command identifier that threads can use to look up the completion of their specific command before attempting to poll. Threads that find the completion of other commands will fill up this map. Thus, there is no need for every thread to search the entire CQ multiple times to find the needed completion entry. To parallelize polling even more, we can assign polling threads to poll on a specific position and only that position. With this technique, threads can poll independently on their position, and different threads attempting to poll on the same position get ordered based on their turn, similarly to how we order threads enqueue on positions in the submission queue with the `turn_counters` array. They can update the cache state for the cache-line that the I/O was for, probably before the queue is even cleaned up, releasing other threads waiting on that cache-line.

New Queue Protocols: Many challenges of exploiting parallelism from NVMe queues come from the queue's serial nature. This is, in fact, a problem for many queue protocols, even emerging ones [26, 35, 42], as they are designed to be executed on the severely less parallel CPUs. As a result, the challenges of having hundreds of thousands of threads are not considered when designing these protocols. Examples include having the doorbell registers in the BAR space of a remote device and having the doorbell registers be write-only. This incurs much overhead in managing queues and requires a lot of effort to expose parallelism. Besides optimizations for existing queue protocols to allow GPUs to communicate with other devices and services, we hope this work motivates new I/O and communication protocols designed with parallelism from the ground up. Especially since we have shown the GPU can naturally expose a lot of I/O level parallelism. Furthermore, as storage devices become faster, it is entirely possible that the communication protocol could be a limiting factor in achieving peak performance.

Caches: In BaM we aimed to make a cache with very fast lookup and proposed optimization to reduce cache contention overheads to provide high effective data access throughput for applications. However, we did this at the cost of storing much metadata in GPU memory to keep the state of the cache. As caching, especially in the context of the operating system page cache, has been around for a long time, extensive literature discusses the trade-offs between cache metadata, lookup and replacement overheads, and support for parallelism. It is unclear which cache design and algorithm are optimal for GPUs as the GPUs have an immense level of parallelism never considered before. Furthermore, GPU-specific features, like fast warp communication or use of scratchpad memory, might even optimize cache paradigms like never before. There is already a concurrent Masters's thesis [60] to this work, exploring the costs and benefits of reducing the metadata overhead using a sectored-cache design. We envision future works exploring different associativity, search algorithms and data structures, and replacement policies. There is a lot of ground to cover concerning software caches on the GPU.

Abstractions: BaM was designed to keep the programmer's effort low when integrating BaM into their applications. The proposed abstractions allow any GPU thread to access any element in the BaM backed memory and enable nearly-transparent optimizations that reduce cache access overhead. We expect that given specific application properties, new abstractions can be created that hide away new optimizations. For example, we recall that a whole thread block gets scheduled on an SM and stays there until all threads in the thread block finish their execution. Often, applications leverage the thread block semantics and fast communication within thread blocks with fast barriers and scratchpad memory to improve performance. If the application allows all threads in a thread block to access the same shared abstraction, then coalescing could potentially be done at the thread block level instead of the warp level. A thread block can have up to 32 warps so the reduced cache traffic could be substantial.

Even more interesting is that different applications can work on even higher-level abstractions, like data columns in tables for data analytics applications [6, 57]. This could lead to new abstractions optimized for those applications, as the abstraction can leverage application-specific information. The A100 GPU's memory bandwidth is 1.5TB/s [2], and our optimized abstractions with the cache provide a peak 8-byte data access bandwidth of 380GB/s. Future work in caches and abstractions should aim to lessen this gap. Of course, much of how the abstractions are designed, optimized, and used is dependent on the application, as they are the interface to the application. We discuss application level implications of BaM later in § 10.3.

Using Accelerators to Accelerate Systems Software: BaM has focused mainly on leveraging the GPU's parallelism to provide fast cache and storage system performance directly to a GPU application. However, on that path, we have optimized I/O queues and software-managed caches that can be used independently of specific GPU applications and can potentially be provided as a service to other processors. For instance, our storage stack on the GPU can easily scale to the peak 45Million IOPs of the nine Intel Optane P5800X SSDs at 512-byte granularity without contending on precious CPU memory. Thus, we can imagine using techniques proposed in BaM to accelerate storage servers, where many users could be making many I/O requests, and the server must keep up and not bottleneck the performance. Recently, GRAID [61] introduced a product that uses a custom GPU to accelerate RAID performance on a node. Such acceleration can avoid expensive CPU context switching and wasting CPU cycles. Using an accelerator to accelerate and provide systems software as a service could change how future systems are designed.

10.2 HARDWARE IMPROVEMENTS

On GPUs: It was hardware support on the GPU of a well-defined memory model, fast atomic throughput, mapping GPU memory for DMA from and SSD, and mapping the SSD's doorbell registers into the GPU threads' address space that enabled us to implement and prototype BaM. However, along the way, and even now, we run into many challenges that might present opportunities for further hardware advancements.

Due to the nature of accessing a cache and I/O, many threads poll, either for command completion or for the cache-line to be inserted into the cache. This not only wastes many execution cycles but also creates significant L2 Cache traffic. If each SM could allow more thread blocks, increasing its pool of schedulable threads, some of these overheads can be avoided. This will allow the GPU to dedicate more of those wasted cycles to potentially issuing more I/O requests, as having more threads on the GPU increases the window of data being accessed or scheduling other work. Furthermore, having a dedicated memory for threads to use for polling could reduce overheads as the polling will not take precious L2 cache bandwidth. Even more so, if the SM's thread and warp scheduler were aware of polling threads, it could optimize scheduling even more.

As we have a pretty big software stack on the GPU, we can use many registers, potentially taking registers away from applications. Some applications optimize the register usage significantly [62, 63, 64], and excessive register spilling can affect the performance of those applications and the BaM software stack. Increasing the register file of an SM can generally be very useful. It can help achieve more occupancy without worrying about register spilling and thus can help improve the performance of latency-bound applications.

Finally, accelerating the algorithms for caching or queuing in hardware might be worthwhile. Recently, GPUs have started to include dedicated accelerators for critical GPU operations, with the addition of tensor cores and ray-tracing accelerator units [2]. However, it is not clear exactly how GPU threads would communicate with or use hardware accelerators for the more complex algorithms that implement caching and queuing. This is primarily a challenge because tensor cores and ray-tracing exploit data parallelism, but it is unclear where the data parallelism is in caching and queuing algorithms.

Storage and Interconnect: While prototyping BaM with real hardware, we ran into many challenges regarding storage device performance, PCIe switches, and P2P performance that hinder the scalability of BaM. Although we were able to overcome some of these challenges by working closely with the vendors for each of the components, there are still some limitations that we think, if improved, can drastically improve BaM's overall performance. An example is the throughput of storage devices with different access granularities. The Samsung P1735 SSD provides 1.6Million IOPs for 4KB access and 1.1Million IOPs for 512Byte access, for bandwidth of 6.6GB/s and 0.6GB/s, respectively. Ideally, the storage should provide the same bandwidth at any granularity, thus providing extremely high throughput with lower granularities. However, that is not the case. This gap is reducing with newer and faster SSDs, e.g., the Intel Optane P5800X provides 1.5Million IOPs for 4KB access and 5.1Million IOPs for 512Byte access, for a bandwidth of 6.1GB/s and 2.6GB/s, respectively. However, to saturate PCIe Gen4 x16, you need more than two times the SSDs with 512Byte access granularity as with 4KB. Scaling to such a high number can be challenging. Also, byte-addressable SSDs and persistent memory have been heavily discussed in academic research. Extending BaM to such technologies could be an exciting line of research. Furthermore, the PCIe Interconnect provides pretty limited bandwidth. It will be interesting to see BaM with storage connected over faster links like NVLink.

10.3 APPLICATION LEVEL OPTIMIZATIONS

As with any new memory and storage hierarchy, even BaM's cache and I/O stack require optimal access patterns for peak performance. As the optimizations show, if accesses to the cache are not optimized, the cache can become a significant performance bottleneck [65]. On the other hand, if the application does not expose enough I/O level parallelism on the GPU at any given time, the storage access latency could be exposed, severely limiting performance. This creates an attractive design space for each application, balancing reuse in the cache (if any) and generating I/O requests for saturating the storage. A concurrent Ph.D. thesis [65] to this work focuses on addressing this exact problem for important graph and data analytics workloads, microbenchmarks with different distribution of ruse, and even traditional GPU applications like vector addition and reduction. We implore the reader to read that work [65] to learn valuable insights into how best to leverage BaM as it is a new memory system for GPUs. We expect BaM will continue to be useful for even more emerging GPU applications, even enabling new ones, and understanding how those applications best utilize a system like BaM could be vital for having a competitive implementation.

10.4 APPLICABILITY TO MULTI-CORE CPUS

Although we motivate that GPUs have significantly more parallelism than CPUs, it is undeniable that they are becoming increasingly parallel. However, vital differences in the CPU and GPU architecture could affect how a system like BaM behaves and performs on a

CPU compared to the GPU. In this section, we discuss some of these dimensions.

Atomics Implementation and Throughput: BaM's software stack leverages fine-grain memory synchronization through atomics to exploit parallelism and reduce the number and size of critical sections. It does so at the cost of executing many atomic operations. Atomics executed by GPU threads goes directly to the shared L2 (LLC) cache, as the private L1 caches are not coherent. Thus, given a large enough queue at the L2, when many atomics contend for the same word, the GPU hardware can have a very high throughput for atomics. In contrast, CPUs have coherent private caches and generally implement atomics with a coherence protocol. When executing many atomic operations, especially ones that content on the same word, there can be significant coherence traffic (on top of the data traffic). As a result, CPUs are moving to implement specialized atomics executed at the memory controller [66, 67, 68], similar to how GPU atomics are executed at the L2. This is designed to reduce coherence traffic, and atomics contending for the same sets of words can be executed at a very high throughput at the same place. With such improvements, CPUs can become interesting devices for executing the BaM software stack in the future.

Out-of-Order Speculative Execution Overheads: As noted in Chapter 2, GPUs have many simple in-order cores and leverage massive thread-level parallelism to hide latencies and provide performance. On the other hand, CPUs have a few cores that leverage out-of-order and speculative execution to hide latencies with instruction-level parallelism. However, there are two challenges with such complex cores in the context of the systems software that is accelerated by BaM. First, the memory semantics specified for the memory loads, stores, and atomics to implement the algorithms impose limitations on instruction reordering. Thus, it's possible that much of the hardware and resources in an out-of-order core designed to reorder instructions and speculatively execute instructions in the future can be wasted. Second, a release-store operation requires flushing any pending invalidations to the other cores, and an acquire-load operation requires acknowledging and processing all invalidations from other cores. On the processing of cache-line invalidations, out-of-order execution pipelines must be flushed, meaning there was work done that was wasted and the effectiveness of out-of-order execution. These challenges do not exist in in-order processors as they issue instructions in order so that ordering constraints can be enforced at the issue stage. As the overhead of atomics and memory ordering operations on out-of-order CPU cores continue to be reduced [66, 68], the techniques discussed in this work on exploiting parallelism and concurrency for high-throughput caching and I/O queue management will become relevant for the multi-core CPUs as well.

Thread Level Parallelism: BaM relies on the massive parallelism provided by GPUs to expose more memory and then I/O level parallelism, aiming to saturate the storage access bandwidth. CPUs still have a long ways to go to match the parallelism of GPUs. However, CPUs can still have many threads ready to be scheduled on their cores. However, context switch threads are expensive and can cause pollution and thrashing of caches. GPUs do not allow any thread to be context switched to any core. They limit the scheduling of the threads of a thread block to the cores of the SM they are assigned to. However, a GPU's SM can schedule between the pool of threads (up to 2048 threads) with little to no overhead. CPUs provide SMT, which helps achieve similar behavior, and modern CPUs can provide up to 8-way SMT per core [66]. Furthermore, threads running on the same core can communicate fast through the private cache (even with atomics), whereas on the GPU, private caches are read-only, so they cannot be used for communication. The benefits of SMT with the parallelization techniques used in BaM would make for interesting future research.

Hardware Cache Capacity: GPUs notoriously have tiny caches that can be used for fine-grained memory synchronization. The A100 GPU only has a 40MB L2 cache [2], and the upcoming H100 GPU [69] only improves that 1.5× with a 60MB L2 cache. In comparison, high-end CPUs have a 256MB last-level cache (LLC) [70], and on top of that, cores have coherent private caches, so the effective on-chip cache capacity is quite substantial. This is important because for a system like BaM we require the metadata for the systems software to hit in the cache because paying for memory latency for metadata checks and updates will be very expensive. Having a large cache for this metadata can impact the types of metadata one can have and the types of algorithms one can exploit for parallelism.

PCIe Considerations: Lastly, CPUs have a significant advantage over any other processor/accelerator, providing way more ports and bandwidth for I/O. Specifically for PCIe, the CPU is the root for all the devices in a node and generally provides many PCIe lanes, allowing it to scale to more storage devices and possibly get a very high aggregate storage bandwidth to the CPU DRAM. In contrast, the GPU only has one PCIe x16 link. This link easily becomes the bottleneck for performance and limits scaling to many storage devices. We hope BaM motivates these classic system designs to change.

10.5 USE IN THE CLOUD

The BaM prototype is built as a research tool to motivate, implement, and evaluate the performance, cost benefits, and feasibility of BaM with real hardware and important GPU applications. However, it does not provide storage or memory isolation and thus is only useful in a single-user system where someone wants to get the peak performance between storage and a GPU. This section discusses how a system like BaM could be used in a shared environment, especially in the cloud. We note that there are two dimensions for isolation and security guarantees: memory (I/O Buffers) and storage (SSDs). Storage is generally managed by a file system that checks access control when an application accesses data on storage. Memory is isolated with virtual memory and the MMU and IOMMU for I/O devices. Memory is a concern because if it is not protected, a user can issue a DMA request to the storage that can read or write to a location in some memory that maybe the user should not have access to. Below we discuss some of the modalities in which hardware resources (including storage) are allocated to users in the cloud and their implications on how storage and memory are managed and isolated. Each opens new challenges and potential research directions.

Bare-metal: The most basic case is when a user is allocated a bare-metal machine with SSDs and GPUs. If the user wants peak performance and is not interested in sharing the machine with other users, then they can use BaM as it is. Here the user is entirely responsible for data on the storage and reads and writes to memory regions from the GPU or the SSDs' DMA. We note this is a vital use case as many performance-critical applications (including HPC applications) are moving to the cloud and renting out bare-metal machines from the cloud provider for peak performance all the time without any contention or interruption.

Application or Virtual Machine with Dedicated Local Storage: If multiple apps or users share the same node but are provided dedicated storage, then the system provider only needs to concern themselves with memory isolation. Dedicated storage can be provided to an application or virtual machine (VM) by mapping a physical storage device to just that application or VM or virtualizing the storage with PCIe SR-IOV [71] and allocating a virtual SSD (with a dedicated virtual function) to the application or VM. Then the app or VM can manage the storage as it wills. However, the memory space must be isolated as we do not want an application of VM to be able to issue DMA requests to its dedicated storage that allows the storage to read or write memory not dedicated to the same application. The hardware IOMMU can provide this functionality, protecting DMAs from or to unwarranted regions. However, IOMMUs can have significant overheads. Another way memory isolation can be provided is if the system (possibly the hypervisor or the operating system) starts a trusted process (either on the CPU of the node or some other trusted hardware) that acts as a proxy to the storage. The key is that communicating to the proxy should not trigger a system call, so the application should talk to the proxy via queues very similar to the I/O queues used to talk to storage. The application can issue I/O requests to this service which validates the memory regions the I/O request will DMA data to or from. If the proxy determines that the request falls in a valid memory range, it will forward it to the storage system; otherwise, it will return an error completion queue entry.

Shared Storage: Finally, we look at the most strict, most general, and probably the most helpful case where the storage is shared between multiple users or virtual machines. In this case, we must rely on a proxy process that not only validates the memory range for the DMA of each I/O request but also implements or communicates to a file system that manages the blocks on the storage. Thus the proxy must also check whether the application making the file request the correct permissions and credentials to access that file. This proxy can be run as a trusted process in the node or on some trusted hardware in case the cloud implements a no-trust policy. This is the most flexible case as it completely takes the burden of managing the storage blocks away from the application or VM. The application accesses data like it would when accessing a file system. Note that the file system that the trusted proxy talks to can be remote and distributed across a data center or even across the internet. How this proxy is implemented and optimized to be as performance-transparent as possible is potentially an interesting hardware-software co-design challenge.

10.6 MULTI-GPUS

The current BaM implementation and evaluation only consider using SSDs to expand the memory capacity of a single GPU. However, most systems have many GPUs that are used by applications to improve overall system performance. When multiple GPUs share the same set of SSDs, a few challenges arise. First, with multiple GPUs contending for the same SSDs, they also contend for precious PCIe bandwidth, which can cause severe performance degradation. Second, to avoid going to storage, GPUs use the software-managed cache, which can become a significant bottleneck if it is distributed and shared by multiple GPUs. As sharing state between GPUs requires using system-level atomics, which use the coherence protocol to maintain correctness, we could run to a similar problem with atomics as on the CPUs where the coherence protocol traffic could be significant and reduce overall performance. Furthermore, just like it was required for BaM, this system would also require studying and providing guidelines for how an application best utilizes the GPUs, the shared cache, and the shared SSDs for optimal performance. We plan on tackling these significant challenges in future versions of BaM.

10.7 AS A PLATFORM FOR PROTOTYPING

We started BaM as a way to emulate a system where processors with latency hiding capabilities, like the SMs of a GPU, have local Flash-based memory and are interconnected with a fast network to share data among themselves [72, 73]. Such a drastic change in architecture generally requires implementation in a simulator to understand performance characteristics. However, we realized that a simulator might not be able to help us catch interesting parts of complex applications.Many critical applications have data-dependent behavior, which could be hard to capture in a simulator if the data sets are large. As a result, we decided to emulate a system with actual hardware to see if there was even any benefit of allowing a processor like a GPU to access a storage class memory like NVMe SSDs. We wanted to see the trade-offs, how a GPU should optimally access such a system, and how complex applications behaved throughout their entire execution. This led to the creation of BaM as presented in this work. Although BaM is designed with limitations of real hardware (e.g., PCIe, SSDs, NVMe Protocol), it has still helped us understand how a processor like the GPU can be used to access an extensive but potentially very slow memory or storage. It has helped us understand how applications can be optimized for such a system. However, we think BaM has even more potential to help computer architects and systems designers quickly prototype systems where a GPU can access large data sets to some other non-traditional memory/storage/network system and evaluate their potential benefits and trade-offs for real applications with large data-sets.

CHAPTER 11: SUMMARY OF RELATED WORK

11.1 OPTIMIZED CPU-CENTRIC MODEL

Most GPU programming models and applications were designed with the assumption that the working data-set fits in the GPU memory. If it does not, application-specific techniques like tiling are employed to process large data on GPUs [15, 16, 19, 20, 21, 22, 74, 75, 76, 77, 78].

SPIN [16] and NVMMU [46] propose to enable peer-to-peer (P2P) direct memory access using GPUDirect RDMA from SSD to GPU and exclude the CPU from the data path. SPIN integrates the P2P into the standard OS file stack and enables page cache and read-ahead schemes for sequential reads. GAIA [21] further extends SPIN's pagecache from CPU to GPU memory. Gullfoss [19] provides a high-level interface that helps in setting up and using GPUDirect APIs efficiently. Hippogriffdb [20] provides P2P data transfer capabilities to the OLAP database system. GPUDirect Storage [15] is the most recent product that migrates the data path from CPU to GPU in the NVIDIA CUDA software stack using GPUDirect RDMA technology. Similar efforts from AMD are seen in RADEON-SSG products [79]. All of these works still employ a CPU-centric model where the CPU is responsible for data orchestration. BaM provides explicit and direct fine-grain access to the storage from GPUs allowing any threads in GPUs to initiate, read and write data to the SSDs.

11.2 PRIOR ATTEMPTS OF THE ACCELERATOR-CENTRIC MODEL

ActivePointers [17], GPUfs [18], GPUNet [78], and Syscalls for GPU [47] have previously attempted to enable an accelerator-centric model for data orchestration. GPUfs [18] and Syscalls for GPU [47] first allowed GPUs to request file data from the host CPU. ActivePointers [17] added a memory-map-like abstraction on top of GPUfs to allow GPU threads to access file data like an array. Dragon [45] proposed to incorporate storage access to the UVM [44] page faulting mechanism. However, all of these approaches rely on the significantly less-parallel CPU to handle the data demands of massively parallel GPUs. Thus, as shown in §2, these approaches end up with gross under-utilization of resources and poor overall performance.

11.3 HARDWARE EXTENSIONS

Extending the support of non-volatile memories for GPUs has been proposed by directly replacing the global memory with flash memory or closely integrating it with the GPU memory system [80, 81, 82, 83, 84]. DCS [85] proposed enabling direct access between storage, network, and accelerators with the help of a dedicated hardware unit like an FPGA providing the required translation for coarse-grain data transfers. Enabling persistence within the GPU has recently been proposed [86]. We acknowledge these efforts and further validate the need to enable large memory capacity for emerging workloads. More importantly, the BaM prototype aims to use existing hardware components to provide significant performance advantages to end-to-end applications with very large real-world data-sets.

11.4 USING SSDS AS CPU MEMORY

Several prior have proposed leverage memory-mapped interface and swapping mechanism of operating systems to extend the CPU memory with SSDs [87, 88, 89, 90, 91, 92, 93]. Most of these works treat the SSD as a block device and rely on paging mechanism to migrate data from the SSD to the CPU memory. Paging mechanism is inefficient if the application exhibits data-dependent fine-grain data access to the data-set. To address this limitation, FlatFlash [94] proposed to exploit the byte-accessibility of SSD and bypasses storage software stack and focuses on fine-grain I/O operation to reduce I/O amplification. Our proposal is similar in spirit to the FlatFlash approach, and we extend the concepts of fine-grain accesses to GPUs. Compared to the CPUs, GPUs offer massive parallelism that can be leveraged to hide the long latency SSD accesses.

CHAPTER 12: CONCLUSION

In this work, we make a case for enabling GPUs to orchestrate high-throughput, fine-grain accesses to solid state storage in a new system architecture called BaM. BaM mitigates the I/O amplification problem of reading more data than needed and the synchronization overheads of CPU-centric solutions by enabling on-demand fine-grain storage accesses, as determined by the compute code running on the GPU.

BaM features a scalable software cache to coalesce data storage requests while minimizing I/O amplification effects. This software cache communicates with the storage system through high-throughput queues that enable the massive number of concurrent threads in modern GPUs to generate I/O requests at a sufficiently high rate to fully utilize the available bandwidth of the interconnect and the storage system. Furthermore, we provide array-based abstractions that not only make integrating BaM into GPU kernels trivial for programmers, but also transparently optimize the number of BaM cache accesses by exploiting common GPU thread access patterns.

Using off-the-shelf hardware components, we implement a prototype of BaM and show that with carefully optimized systems software on the GPU, it is possible to use solid state storage as a means to extend the GPU's effective memory capacity since it provides performance (up to $4.62\times$), cost (up to $21.8\times$), and I/O efficiency (up to $3.72\times$) benefits, even over much more expensive state-of-the-art solutions using fast DRAM, for important GPU accelerated applications.

REFERENCES

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” *IEEE Micro*, vol. 28, no. 2, p. 39–55, Mar. 2008.

[2] “NVIDIA Tesla A100 Tensor Core GPU Architecture,” https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[3] J. Yang and J. Leskovec, “Defining and Evaluating Network Communities based on Ground-truth,” *CoRR*, vol. abs/1205.6233, 2012.

[4] J. Sybrandt, M. Shtutman, and I. Safro, “MOLIERE: Automatic Biomedical Hypothesis Generation System,” in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1633–1642.

[5] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W. mei W. Hwu, “EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs,” *Proceedings of VLDB Endowment*, vol. 14, pp. 114–127, 2020.

[6] “CUDA RAPIDS: GPU-Accelerated Data Analytics and Machine Learning,” https://developer.nvidia.com/rapids.

[7] T. N. Kipf and M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” in *International Conference on Learning Representations (ICLR’17)*, 2017.

[8] S. W. Mi, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W. mei Hwu, “Large graph convolutional network training with gpu-oriented data communication architecture,” *Proceedings of VLDB Endowment*, vol. 14, 2021.

[9] W. Hu, M. Fey, H. Ren, M. Nakata, Y. Dong, and J. Leskovec, “OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs,” *arXiv preprint arXiv:2103.09430*, 2021.

[10] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, “Deep Learning Recommendation Model for Personalization and Recommendation Systems,” *CoRR*, vol. abs/1906.00091, 2019.

[11] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The Architectural Implications of Facebook's DNN-Based Personalized Recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 488–501.

[12] D. Mudigere, Y. Hao, J. Huang, Z. Jia, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo, J. A. Yang, L. Gao, D. Ivchenko, A. Basant, Y. Hu, J. Yang, E. K. Ardestani, X. Wang, R. Komuravelli, C.-H. Chu, S. Yilmaz, H. Li, J. Qian, Z. Feng, Y. Ma, J. Yang, E. Wen, H. Li, L. Yang, C. Sun, W. Zhao, D. Melts, K. Dhulipala, K. Kishore, T. Graf, A. Eisenman, K. K. Matam, A. Gangidi, G. J. Chen, M. Krishnan, A. Nayak, K. Nair, B. Muthiah, M. khorashadi, P. Bhattacharya, P. Lapukhov, M. Naumov, A. Mathews, L. Qiao, M. Smelyanskiy, B. Jia, and V. Rao, "Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models," 2021.

[13] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems," in *Third Conference on Machine Learning and Systems*, 2020.

[14] B. Acun, M. Murphy, X. Wang, J. Nie, C. Wu, and K. Hazelwood, "Understanding training efficiency of deep learning recommendation models at scale," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2021, pp. 802–814.

[15] "GPUDirect Storage: A Direct Path Between Storage and GPU Memory," https://developer.nvidia.com/blog/gpudirect-storage/.

[16] S. Bergman, T. Brokhman, T. Cohen, and M. Silberstein, "SPIN: Seamless operating system integration of peer-to-peer DMA between ssds and gpus," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 167–179.

[17] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: A Case for Software Address Translation on GPUs," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 596–608.

[18] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a File System with GPUs," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 485–498.

[19] H.-W. Tseng, Y. Liu, M. Gahagan, J. Li, Y. Jin, and S. Swanson, "Gullfoss : Accelerating and simplifying data movement among heterogeneous computing and storage resources," 2015.

[20] J. Li, H.-W. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson, "HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics," *Proceedings of the VLDB Endowment*, vol. 9, no. 14, p. 1647–1658, Oct. 2016.

[21] T. Brokhman, P. Lifshits, and M. Silberstein, "GAIA: An OS page cache for heterogeneous systems," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 661–674.

[22] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[23] A. Eisenman, M. Naumov, D. Gardner, M. Smelyanskiy, S. Pupyrev, K. M. Hazelwood, A. Cidon, and S. Katti, "Bandana: Using non-volatile memory for storing deep learning models," *CoRR*, 2018.

[24] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 756–771.

[25] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. Association for Computing Machinery, 2018, p. 297–312.

[26] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a True Direct-Access File System with DevFS," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, Oakland, CA, 2018.

[27] "NVIDIA Tesla V100 GPU Architecture Whitepaper," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[28] "NVIDIA PTX ISA," https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, 2021.

[29] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[30] "C++ Memory Model," https://en.cppreference.com/w/cpp/language/memory_model, 2012.

[31] "Intel® Optane™ Technology," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[32] "Samsung Z-NAND Technology Brief," https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[33] "Samsung 980 PRO SSD," https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/.

[34] Weka.io, "WekaFS Architecture Whitepaper," https://www.weka.io/wp-content/uploads/files/2017/12/Architectural_WhitePaper-W02R6WP201812-1.pdf, 2021.

[35] Y. Ren, C. Min, and S. Kannan, "CrossFS: A cross-layered Direct-Access file system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/ren pp. 137–154.

[36] J. Liu, A. Rebello, Y. Dai, C. Ye, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Scale and performance in a filesystem semi-microkernel," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 819–835.

[37] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the zofs user-space nvm file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. Association for Computing Machinery, 2019, p. 478–493.

[38] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 154–161.

[39] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "Daos and friends: A proposal for an exascale storage system," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 585–596.

[40] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A Cross Media File System," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, Shanghai, China, 2017.

[41] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 494–508.

[42] "NVMe 1.3 Specification ," http://nvmexpress.org/resources/specifications/.

[43] "Intel Optane SSD DC P5800X Series," https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html.

[44] "Unified Memory for CUDA Beginners," https://developer.nvidia.com/blog/unified-memory-cuda-beginners.

[45] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka, "DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[46] J. Zhang, D. Donofrio, J. Shalf, M. T. Kandemir, and M. Jung, "NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 13–24.

[47] J. Veselý, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt, "Generic System Calls for GPUs," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*, 2018, pp. 843–856.

[48] F. J. Corbato, "A Paging Experiment With The Multics System," *Technical Report, Massachusetts Institute of Technology, Cambridge, Project MAC*, 1968.

[49] J. Markussen, L. B. Kristiansen, P. Halvorsen, H. Kielland-Gyrud, H. K. Stensland, and C. Griwodz, "Smartio: Zero-overhead device sharing through pcie networking," *ACM Transactions on Computing System*, vol. 38, no. 1–2, jul 2021.

[50] "How to make your life easier in the age of exascale computing using nvidia gpudirect technologies," https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9653-how-to-make-your-life-easier-in-the-age-of-exascale-computing-using-nvidia-gpudirect-technologies.pdf, 2019.

[51] M. Tork, L. Maudlej, and M. Silberstein, *Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers.* New York, NY, USA: Association for Computing Machinery, 2020, p. 117–131. [Online]. Available: https://doi.org/10.1145/3373376.3378528

[52] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[53] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[54] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds. ACM Press, 2011, pp. 587–596.

[55] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[56] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: a scalable fully distributed web crawler," *Software: Practice and Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[57] The City of New York, "TLC Trip Record Data," https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[58] The Apache Software Foundation, "Apache ORC: The smallest, fastest columnar storage for Hadoop workloads," https://orc.apache.org/.

[59] J. Gómez-Luna, I. E. Hajj, L. Chang, V. García-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Peña, and W. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 43–54.

[60] A. Masood, "A Highly Concurrent Software-based Sector Cache For GPU Orchestrated Storage Access," 2022.

[61] "Storagereview independent review: Graid supremeraid™ sr-1010," https://www.graidtech.com/news-storagereview-review-of-graid-supremeraid-sr-1010/, 2022.

[62] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W.-m. Hwu, "Collaborative (CPU+ GPU) algorithms for triangle counting and truss decomposition," in *2018 IEEE High Performance extreme Computing Conference (HPEC'18)*, Boston, USA, 2018.

[63] M. Almasri, O. Anjum, C. Pearson, Z. Qureshi, V. S. Mailthody, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on k-truss Decomposition on GPU," in *2019 IEEE High Performance extreme Computing Conference (HPEC'19)*, Boston, USA, 2019.

[64] C. Pearson, M. Almasri, O. Anjum, V. S. Mailthody, Z. Qureshi, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on Triangle Counting on GPU," in *2019 IEEE High Performance extreme Computing Conference (HPEC'19)*, Boston, USA, 2019.

[65] V. S. Mailthody, "Application Support And Adaptation For High-throughput Accelerator Orchestrated Fine-grain Storage Access," Ph.D. dissertation, University of Illinois Urbana-Champaign, 2022.

[66] "Open power isa," https://www-50.ibm.com/systems/power/openpower/posting.xhtml?postingId=01F8EF905EC4A2CD85257EAF0069612D, 2022.

[67] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[68] A. Asgharzadeh, J. M. Cebrian, A. Perais, S. Kaxiras, and A. Ros, "Free atomics: Hardware atomic operations without fences," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022.

[69] "NVIDIA Tesla H100 Tensor Core GPU Architecture," https://resources.nvidia.com/en-us-tensor-core, 2022.

[70] AMD, "AMD EPYC Processors," https://www.amd.com/en/processors/epyc-server-cpu-family.

[71] "Pcie generation 5 specification," https://pcisig.com/.

[72] Z. Qureshi, V. S. Mailthody, I. Gelado, S. W. Min, A. Masood, J. Park, J. Xiong, C. Newburn, D. Vainbrand, I.-H. Chung, M. Garland, W. Dally, and W.-m. Hwu, "BaM: A Case for Enabling Fine-grain High Throughput GPU-Orchestrated Access to Storage," 2022. [Online]. Available: https://arxiv.org/abs/2203.04910

[73] W. W. Hwu, I. El Hajj, S. Garcia de Gonzalo, C. Pearson, N. S. Kim, D. Chen, J. Xiong, and Z. Sura, "Rebooting the Data Access Hierarchy of Computing Systems," in *2017 IEEE International Conference on Rebooting Computing (ICRC)*, 2017, pp. 1–4.

[74] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically Managed Data for CPU-GPU Architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 165–174.

[75] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: Association for Computing Machinery, 2010, p. 347–358.

[76] H. Seo, J. Kim, and M.-S. Kim, "GStream: A Graph Streaming Processing Method for Large-Scale Graphs on GPUs," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 253–254.

[77] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: Creating application objects efficiently for heterogeneous computing," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 53–65.

[78] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein, "GPUnet: Networking Abstractions for GPU Programs," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 201–216.

[79] AMD, "RADEON-SSG API Manual," https://www.amd.com/system/files/documents/ssg-api-user-manual.pdf.

[80] J. Zhang, M. Kwon, H. Kim, H. Kim, and M. Jung, "FlashGPU: Placing New Flash Next to GPU Cores," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[81] J. Zhang and M. Jung, "ZnG: Architecting GPU Multi-Processors with New Flash for Scalable Data Analysis," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 1064–1075.

[82] J. S. Vetter and S. Mittal, "Opportunities for nonvolatile memory systems in extreme-scale high-performance computing," *Computing in Science Engineering*, vol. 17, no. 2, pp. 73–82, 2015.

[83] J. Zhang and M. Jung, "Ohm-GPU: Integrating New Optical Network and Heterogeneous Memory into GPU Multi-Processors," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 695–708.

[84] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter, "Exploring hybrid memory for gpu energy efficiency through software-hardware co-design," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 93–102.

[85] J. Ahn, D. Kwon, Y. Kim, M. Ajdari, J. Lee, and J. Kim, "DCS: A fast and scalable device-centric server architecture," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 559–571.

[86] A. W. Baskara Yudha, K. Kimura, H. Zhou, and Y. Solihin, "Scalable and Fast Lazy Persistency on GPUs," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 252–263.

[87] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified Address Translation for Memory-mapped SSDs with FlashMap," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, Portland, OR, 2015.

[88] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11, Boston, MA, 2011.

[89] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, Washington, DC, 2009.

[90] M. Saxena and M. M. Swift, "FlashVM: Virtual Memory Management on Flash," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10, Boston, MA, 2010.

[91] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient Memory-Mapped I/O on Fast Storage Device," *Trans. Storage*, vol. 12, no. 4, May 2016.

[92] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "Improving SSD Lifetime with Byte-addressable Metadata," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17, Alexandria, VA, 2017.

[93] D.-H. Bae, I. Jo, Y. A. Choi, J.-Y. Hwang, S. Cho, D.-G. Lee, and J. Jeong, "2b-ssd: The case for dual, byte- and block-addressable solid-state drives," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 425–438.

[94] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "FlatFlash: Exploiting the Byte-Accessibility of SSDs Within a Unified Memory-Storage Hierarchy," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.