Karlsruhe Institute of Technology

Parallel NVMe Driver with Software Cache for GPU4FS

Bachelor's Thesis
submitted by

Nick Djerfi

to the KIT Department of Informatics

Reviewer:	Prof. Dr.-Ing. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Peter Maucher

May 1st 2024 – September 2nd 2024

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, September 2, 2024

Abstract

Modern filesystems and storage media put a heavy load on CPU resources, especially in servers and high performance environments containing many storage devices in one system. Advanced filesystem features such as software-level RAID, compression, encryption, and deduplication require a lot of computing power, leaving less CPU resources for other processes running on the system. To combat this issue, Maucher et al. propose GPU4FS, a novel GPU-based userspace filesystem. GPU4FS moves all filesystem-related tasks onto the GPU, including transferring data to and from the storage device. GPU4FS currently only supports byte-addressable non-volatile memory using Intel Optane. Our thesis aims to provide the foundation for supporting block-based storage media in GPU4FS using the NVMe protocol. To this end, we design and implement a general purpose NVMe driver and software cache, intended to run solely on the GPU. We base our driver on previous work by Qureshi et al. with their novel storage architecture Big accelerator Memory (BaM). BaM provides direct access to storage media for GPU applications through an array-like abstraction layer.

Our demonstrator proves to be competitive with CPU-based approaches in terms of both bandwidth and I/O operations per second. We achieve between 60 % and 90 % of the bandwidth of the Linux I/O stack in random read and write benchmarks on a consumer-grade SSD. However, our demonstrator exhibits abnormally high latency of around 290 µs in microbenchmarks and up to 10 s using a synthetic workload.

We conclude that GPU-based NVMe drivers are feasible compared to traditional CPU-based approaches. However, further research is necessary, especially towards reducing the latency of our demonstrator, to make this approach suitable in high performance environments.

Contents

Abstract **v**

Contents **1**

1 Introduction **3**

2 Background **5**
2.1 PCIe 5
2.1.1 Topology 5
2.1.2 Device Enumeration and Configuration 7
2.1.3 Base Address Registers 8
2.1.4 Link Width and Speed 8
2.2 Solid State Drives 8
2.2.1 NVMe Concepts 9
2.2.2 NVMe Protocol Basics 11
2.2.3 Atomic Operations 12
2.2.4 Performance Characteristics 13
2.2.5 SSD Trimming 15
2.3 GPUs 15
2.3.1 Execution Model 15
2.3.2 Memory Model 16
2.3.3 Programming Model 17
2.3.4 NVIDIA's GPUDirect RDMA 17
2.3.5 Peer-to-Peer Direct Memory Access on AMD GPUs . . . 18
2.3.6 NVIDIA's GPUDirect Async 18
2.4 GPU4FS 18
2.4.1 Basic structure 19
2.4.2 Extended functionality 20
2.4.3 Discussion 21

3 Related Work **23**
3.1 BaM . . . 23
3.1.1 `libnvm` Kernel Module . . . 24
3.1.2 `libnvm` Userspace Library . . . 24
3.1.3 Virtual Queues . . . 24
3.1.4 Caching . . . 26
3.1.5 Abstraction . . . 27
3.1.6 Discussion . . . 28
3.2 SPIN . . . 28
3.3 GPUfs . . . 29

4 Design **31**
4.1 Queue Management . . . 31
4.1.1 Virtual Queues . . . 31
4.1.2 Queue Arbitration . . . 38
4.2 Software Cache . . . 39
4.2.1 Basic Structure . . . 39
4.2.2 Data Structures . . . 41
4.2.3 Acquire and Release Algorithms . . . 42
4.2.4 Evict-and-Replace Algorithm . . . 43
4.2.5 Thread Coalescing . . . 45
4.2.6 Discussion . . . 45

5 Implementation **49**
5.1 Demonstrator . . . 50
5.2 NVMe SSD Peer Access . . . 51
5.3 Queue Management . . . 52
5.4 Software Cache . . . 53
5.5 Discussion . . . 53

6 Evaluation **55**
6.1 Baseline Performance . . . 55
6.2 Queue Management Performance . . . 62
6.3 Software Cache Performance . . . 67
6.4 Discussion . . . 73

7 Future Work **75**

8 Conclusion **79**

Bibliography **81**

Chapter 1

Introduction

As our storage and computation demands increase, storage devices become faster, with larger capacities, and filesystems become more advanced, with features such as software-level RAID, compression, and deduplication. In contrast, CPU performance improves more slowly, which leads to increased CPU utilization during filesystem-related tasks. This issue is especially prevalent in servers and high performance environments, which contain many storage devices in one system.

Maucher et al. [42] address this issue with GPU4FS, a GPU-based filesystem. GPU4FS is a filesystem designed to run fully on the GPU, only requiring a management process running on the CPU for initialization, teardown, and to set up inter-process communication between processes and the GPU. It aims to provide a POSIX-like filesystem API for userspace processes running on the CPU. Currently, the GPU4FS demonstrator only supports byte-addressable non-volatile memory (NVM) through Intel Optane [25] and a limited feature set.

On the other hand, Qureshi et al. [38,69,70] solve this issue for specific workloads by enabling direct storage access for GPU kernels using their novel storage architecture Big accelerator Memory (BaM). BaM differs from GPU4FS as it does not aim to implement a full filesystem on the GPU. Instead, BaM provides an array-like interface to storage devices, which GPU developers can access similarly to memory, hence the name Big accelerator Memory.

While both works aim to solve this issue for different workloads, they share the same basic principle. That is, both aim to alleviate the CPU by offloading computation to the GPU and enabling direct access to storage media on the GPU, without needing the CPU to orchestrate memory transfers.

In this thesis, we aim to prove the feasibility of a general purpose NVMe driver fully implemented on the GPU. To this end, we design an NVMe driver based on design principles of previous works [17,38,69,70,75] to make full use of the parallelism available on GPUs. We intend to develop a prototype implementation of our driver design and integrate it into the current GPU4FS [41] demonstrator. With

block-based storage, a suitable filesystem cache becomes necessary for GPU4FS to guarantee data consistency. For this reason, we aim to design a filesystem cache tailored for GPU applications and block devices to integrate into GPU4FS, or to replace an existing cache implementation in the GPU4FS demonstrator.

We will evaluate our prototype implementation in comparison to common CPU-based approaches and BaM [38, 69, 70] and aim to answer how competitive our implementation is regarding bandwidth utilization, I/O operations per second, and latency.

Due to the high degree of parallelization available to GPUs and possible with the NVMe protocol, we expect to see high bandwidth utilization comparable to CPU-based approaches while maintaining low CPU usage. We expect similar results to those shown in previous works on GPU4FS [33, 37, 41, 42, 71].

Chapter 2

Background

This chapter provides useful background on the technologies used in this thesis. We explain the basics of the Peripheral Component Interconnect Express (PCIe) [67] bus, as modern GPUs and storage devices mainly communicate over PCIe with the CPU and other devices. Next, we explain how modern storage devices and GPUs function and how they are programmed. Last, we introduce GPU4FS [42] in its current form.

2.1 PCIe

The Peripheral Component Interconnect Express (PCIe) [67] bus is a high-speed bus commonly used to connect third party devices to the CPU. Today, many devices such as graphics cards, USB controllers, storage devices, and network devices use PCIe [8].

In the following, we provide a general overview of the PCIe bus and details on specific parts of the PCIe specification relevant to the NVMe protocol and to the use of peer-to-peer direct memory access (P2PDMA).

2.1.1 Topology

The PCIe bus connects devices in a tree-like hierarchy with three major components. The Root Complex denotes the root of the hierarchy and connects it to the CPU and memory. It exposes multiple ports—each with its own hierarchy domain—and may optionally route transactions between them. Note, the PCIe specification also calls these ports Host Bridges [67].

A Switch functions similarly to a regular network switch, as it exposes multiple ports for additional PCIe devices and routes transactions between them. Endpoints represent single functions that can either request or service transactions.

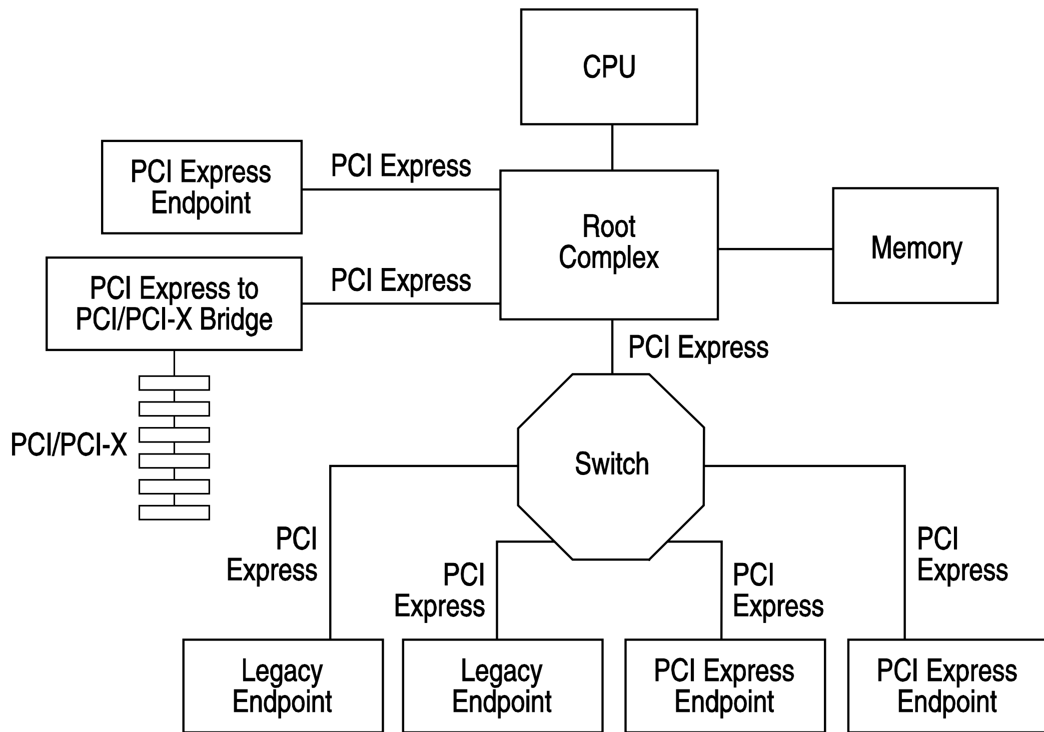


Figure 2.1: An example of a PCIe topology. The Root Complex connects the CPU and memory to its underlying PCIe bus. The Switch provides four additional Endpoints and is directly connected to the Root Complex. The topology also features legacy Endpoints to connect to older hardware using the now obsolete PCI or PCI-X standard. This figure is taken from [67].

A PCIe device may expose multiple Endpoints for different uses [67]. For instance, an Ethernet card may expose one Endpoint for every Ethernet port to allow the host to configure and use them separately. Figure 2.1 shows an example of a PCIe topology including the host system, the root complex, switches, and endpoints.

PCIe topology is relevant when the input-output memory management unit (IOMMU) handles DMA requests. The IOMMU is an abstraction layer between DMA-capable I/O devices and main memory. It intercepts and translates memory requests from I/O devices directed at main memory, providing similar services as the processor's MMU, including page-level protection [10].

Previous works [38, 69, 70] utilizing GPUDirect RDMA state, that the use of the IOMMU may massively degrade performance of peer-to-peer DMA accesses. Furthermore, previous works recommend installing devices participating

in peer-to-peer DMA under the same PCIe switch and not across hierarchy domains of a Root Complex, as this may further degrade performance. Our NVMe driver already reaches the maximum bandwidth capable on our test system without disabling the IOMMU. Therefore, we choose not to evaluate the difference in performance when disabling the IOMMU. Still, as we only use one consumer-grade SSD for our benchmarks, our maximum achievable bandwidth is less than the achievable PCIe bandwidth. As such, the effects of the IOMMU on performance require further research, as the performance degradation mentioned by Qureshi et al. may only take effect at a higher bandwidth.

2.1.2 Device Enumeration and Configuration

The PCIe specification provides the Enhanced Configuration Access Mechanism (ECAM) to enumerate and configure devices and their device functions on the PCIe bus. With ECAM, each device function has an associated 4 KiB page of memory-mapped I/O called the Configuration Space. The host can use the Configuration Space to gather information about the device and to interact with the device. For instance, the Configuration Space contains a class code, which the host can use to search for a specific device type [67].

ECAM uses a hierarchical addressing scheme for PCIe devices. The address for a device function is split into a segment number, a bus number, a device number, and a function number. The bus, device, and function numbers give an offset into a memory mapped I/O (MMIO) region, specified by the segment number, at which the Configuration Space for a particular device function resides [67]. Getting the base address for a specific segment number varies across platforms and will not be discussed in this thesis for the sake of brevity.

Given the base address for a specific PCIe segment, the host can enumerate all devices by simply checking the device class for each possible bus, device, and function number. The device class specifies the type of function the device performs and is stored in the Configuration Space. As invalid reads return values with all bits set and $\texttt{FFFF}_h$ is an invalid device class, the host can ignore all device functions with invalid device classes to filter out valid devices [67, 80]. For example, Linux version 3.19.8 uses this approach on x86 platforms to enumerate PCIe devices in the early boot stages [81].[1]

There are more optimized methods to enumerate the PCIe bus. For instance, recent Linux versions use the Advanced Configuration and Power Interface (ACPI) to enumerate PCIe devices [67, 80].[2]

[1] See file `/arch/x86/pci/early.c`

[2] See file `/arch/x86/pci/acpi.c`

2.1.3 Base Address Registers

Base Address Registers (BARs) are a mechanism for a device to expose internal memory to the host or other devices on the same PCIe bus. BARs are 32-bit wide addresses inside the Configuration Space of a device function, which the host can use as pointers to other regions of MMIO. Devices often use BARs to expose internal I/O registers or buffers for faster and more direct communication using DMA between the host and the device. A device function can use up to six 32-bit BARs or combine them into three 64-bit BARs, each describing memory regions of varying sizes. Regions can be up to 512 GiB in size if the device and the underlying BIOS support larger BAR sizes [67].

For instance, GPUs use BARs to expose parts of their internal VRAM, enabling technologies such as Unified Virtual Memory [48] or GPUDirect RDMA [52]. On the other hand, NVMe SSDs expose their Controller Properties including doorbell registers using BARs [62, 63].

2.1.4 Link Width and Speed

A PCIe link consists of multiple lanes. Each lane consists of two signaling pairs, one for transmission and one for reception, establishing a bidirectional serial connection. With each generation of the PCIe standard, transfer rates per lane and data direction roughly doubles, starting with 2.5 GT/s for the first generation, followed by 5 GT/s, 8 GT/s, 16 GT/s, and 32 GT/s for the following generations. For PCIe, a transfer consists of a single bit transferred over the serial connection. As such, a transfer rate of 2.5 GT/s directly corresponds to a raw bandwidth of 2.5 Gbit/s. Due to signal encoding, the useable data bandwidth is slightly lower than the transfer rate. For instance, a PCIe 4.0 link operating at 16 GT/s only has an effective bandwidth of around 1.969 GB/s instead of 2 GB/s [67, 74].

A PCIe link's width is the number of lanes used and is denoted by xN where N is the number of lanes. The PCIe specification restricts lane width to powers of two up to 32 lanes. As lanes work independently, the bandwidth scales linearly with the number of lanes. For example, a x4 PCIe 4.0 link has a combined transfer rate of 64 GT/s [67].

2.2 Solid State Drives

Flash-based solid state drives (SSDs) have become an increasingly popular choice in both the consumer and the enterprise market. As they have no moving parts, they exhibit a high degree of parallelism and high speeds in both sequential and random accesses with lower latency compared to hard disk drives (HDDs) [66].

Initially, SSDs used the Serial ATA (SATA) protocol, similarly to HDDs. Over time, the SATA protocol could no longer keep up as SSD bandwidth increased. Consequently, recent SSDs switched to the newer Non-Volatile Memory Express (NVMe) protocol based on the PCIe bus [63, 66].

The NVMe protocol is designed with high parallelism in mind, using multiple independent command queues and allowing SSD controllers to reorder and execute commands for better performance. As such, NVMe SSDs have become a popular choice in both consumer-grade systems and high-performance servers [41, 66].

This section explains select components of the NVMe protocol useful for the development of our NVMe driver. In addition to the fundamentals of the protocol, we focus on atomic operations to guarantee crash consistency for the filesystem and on performance optimizations the protocol offers.

2.2.1 NVMe Concepts

NVMe controllers are the interface the host uses to interact with the underlying storage medium. The NVMe specification differentiates between two major types of controllers for PCIe-based communication. I/O controllers support commands used to access the data stored on the storage medium and optionally support management commands. Administrative controllers only support management commands. For PCIe-based NVMe devices, controllers expose their Controller Properties in an MMIO regions used for configuration and enumeration. The host can access these properties using the address provided by the PCIe BAR0 and BAR1 registers [62, 63].

Note, in addition to the memory-based PCIe transport model, the NVMe specification also defines a message-based transport model, which can be used to access NVMe devices over a TCP connection. Additionally, it defines a mixed transport model which uses Remote Direct Memory Access (RDMA) to access the NVMe device over the network. The message-based transport model encapsulates commands in Capsules containing both the command and the data, while the mixed transport model uses Capsules for commands and memory for data [63].

The NVMe specification provides a hierarchical view on the underlying storage medium. An example of which can be seen in Figure 2.2. At the bottom of the hierarchy, namespaces are formatted areas of NVM storage, which the host can access directly through attached NVMe controllers. The controllers use Namespace IDs (NSIDs) to identify and access specific namespaces in the system. Namespaces can be thought of as the actual storage devices the host reads and writes from.

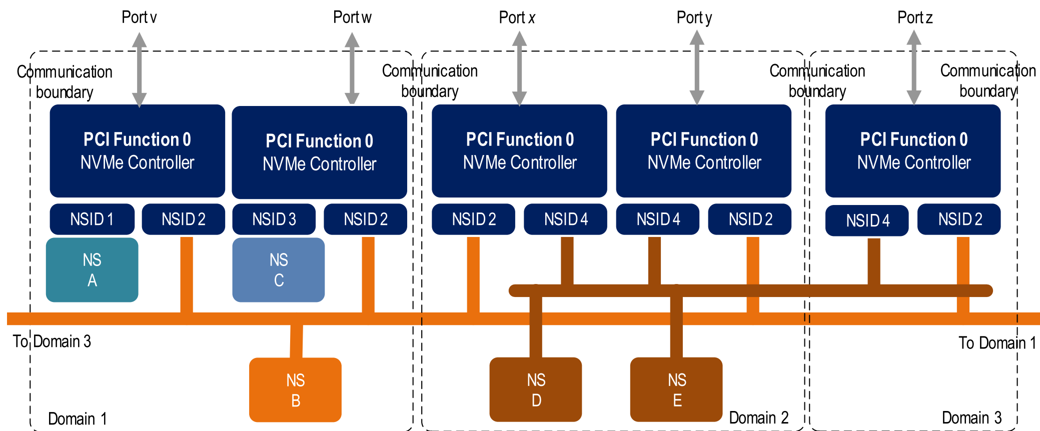


Figure 2.2: An example of a complex NVMe subsystem. It contains multiple controllers across multiple domains, each individually connected to possibly distinct PCIe busses through ports. The domains contain multiple namespaces, of which namespaces B, D, and E are shared across multiple domains. This figure is taken from [63].

Multiple controllers may share namespaces using an optional capability called multi-path I/O. With multi-path I/O multiple controllers can simultaneously and independently access data on the same namespace. Note, concurrent accesses of the same namespace require manual coordination by the hosts [39, 63].

Next, NVM Sets are logically and potentially physically separate collections of NVM storage. NVM Sets may contain one or more namespaces, which inherit attributes from the NVM Set. Most notably, these attributes include an optimal write size for which the storage device operates most efficiently [63].

Further up, Endurance Groups are collections of NVM Sets, which share endurance information. Collected information includes the number of bytes written and read from the group and an endurance estimate, which estimates the number of bytes that may be written to the group over its lifetime [63].

Next up, domains are the smallest indivisible units containing zero or more Endurance Groups and controllers. Domains are independent, but may cooperate using communication boundaries between them. For instance, namespaces in one domain can be shared across multiple domains and accessed from multiple controllers. If a domain fails to communicate with other domains, the system becomes divided, which may affect access to namespaces and system information. For instance, if the first domain in Figure 2.2 fails, the shared namespace B will no longer be visible to the controllers in the other two domains. Note, NVM Sets, Endurance Groups, and domains all have unique identifiers used in specific commands [63].

At the top, an NVM subsystem consists of one or more controllers and PCIe ports used to communicate with different hosts. Additionally, an NVM subsystem may contain an NVM storage medium with associated domains and an interface between the controllers and the storage medium [39, 63].

While the NVMe specification mandates support for multiple namespaces, support for multiple NVM sets, Endurance Groups, and domains is optional. NVMe devices with only a single domain, a single Endurance group, or a single NVM set may not support commands specific to these concepts [63].

2.2.2 NVMe Protocol Basics

At its core, the NVMe protocol uses multiple pairs of Submission and Completion Queues (SQ, CQ) to process commands. The controller processes these queues independently of each other, providing a highly parallelized interface to the SSD. The NVMe specification models queues as circular buffers with head and tail pointers that wrap around to zero if they move past the end of the queue. Submitters write entries at the tail of the queue and consumers take entries from the head of the queue. The driver notifies the controller of updated head and tail pointers by writing their addresses to the corresponding doorbell registers in the controller's Controller Properties [62, 63].

Each NVMe controller exposes an admin queue pair for the driver to manage the underlying namespaces. The driver can enqueue special commands on the Admin Submission Queue to create and delete I/O queue pairs and namespaces. Furthermore, the driver can request useful information about namespaces, such as their logical block size and overall capacity, along with supported optional features [60, 62, 63].

To interact with the data stored on disk, the driver issues commands to the I/O Submission Queues. I/O commands can only target contiguous ranges on the disk, specified by a starting linear block address (LBA) and the number of logical blocks to access. The source or destination buffer in host memory, on the other hand, can be non-contiguous as described by one of the two following formats [60, 63].

Physical Region Page Lists. A physical region page (PRP) entry describes a single contiguous physical memory page in host memory. PRP lists are sets of PRP entries stored in a single physical memory page, similar to common page address translation models. For larger data transfers, these lists form a linked list where the last entry in a PRP list points to the next PRP list in the chain [63].

Scatter Gather Lists. A Scatter Gather List (SGL) consists of one or more SGL segments forming a linked list. Each SGL segment holds one or more SGL descriptors which describe an arbitrary contiguous memory region using an address and a length in bytes. In contrast to PRP lists, SGLs have less strict alignment requirements, as the starting address and length of an SGL descriptor only require at most four byte alignment. While the NVMe specification mandates support for PRP lists, SGLs are an optional feature [63].

Discussion. SGLs provide better flexibility in many cases, as they permit four byte alignment for both the address and the size, while PRP lists only allow page aligned memory addresses after the first PRP entry. PRP lists incur a large overhead for large contiguous transfers, as they require a PRP entry for every memory page. In contrast, most large contiguous transfers can be described by a single SGL descriptor as the length field is 32 bits wide [60, 63].

We choose to use PRP lists for our demonstrator for two reasons. First, as we use a software cache with a fixed page size, it is preferable to restrict the page size to a multiple of the NVMe controller's page size. If we use at most two controller pages per transfer, we can describe the region using both PRP entries in the NVMe command data structure, without using additional PRP lists. Second, SGL support is optional and the SSDs in our test system do not support SGLs. As such, we need to support PRP lists regardless [60, 63].

We aim to implement support for SGLs in future iterations of our NVMe driver. SGL support enables us to use larger software cache lines with only a single SGL entry embedded into the NVMe command data structure [60, 63].

2.2.3 Atomic Operations

Atomic operations on block devices are essential to efficiently ensure crash consistency of a filesystem, as they ensure the device does not partially overwrite important metadata, potentially invalidating filesystem invariants [18]. The NVMe protocol provides information on the requirements for commands to be executed atomically by the NVMe controller [60].

For instance, namespaces expose three values called Namespace Atomic Write Unit Power Fail (NAWUPF), Namespace Atomic Boundary Size Power Fail (NABSPF), and Namespace Atomic Boundary Offset (NABO) through the Identify Namespace command. The combination of these three values indicate whether the controller guarantees a write operation to be written atomically to NVM even during a power fail or error condition. The controller atomically executes write operations whose sizes are at most NAWUPF logical blocks large and which do not cross atomic boundaries as specified by NABSPF and NABO [60].

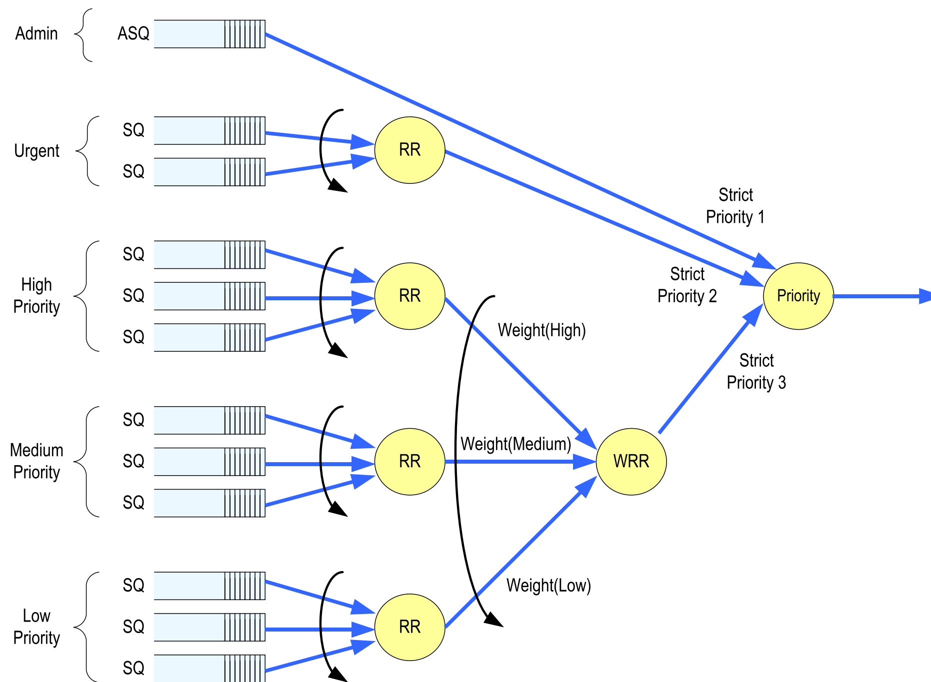


Figure 2.3: An example of the weighted round-robin arbitration mechanism with three weighted priority classes. The controller processes commands in the Admin and Urgent classes before any lower priority commands. Classes with multiple attached queues are processed using the standard round-robin arbitration mechanism (RR). The resulting commands are interleaved using the weighted round-robin arbitration mechanism (WRR) with the specified weights to determine the number of commands to process for each priority. This figure is taken from [63].

2.2.4 Performance Characteristics

The NVMe protocol provides useful ways to optimize the NVMe controller and the driver for higher throughput and lower latency.

Command Arbitration. Traditionally, NVMe controllers execute commands using a round-robin arbitration mechanism, where the controller fetches a fixed number of commands from every Submission Queue each time [63].

More modern NVMe controllers optionally support a more sophisticated arbitration mechanism with weighted round-robin queues and an urgent priority class. Figure 2.3 shows an example of the weighted round-robin arbitration mechanism. With the weighted round-robin arbitration mechanism, commands fall into three strict priority classes: the Admin class, the Urgent class, and the Weighted Round-Robin class [63].

The Admin Submission Queue falls into the Admin class, which the controller processes before all lower priority classes. The driver can assign I/O Submission Queues either to the Urgent class or to the Weighted Round-Robin class with specified priority. The controller processes commands in the Urgent class before any lower priority commands, which may lead to starvation if not configured properly. Finally, the controller chooses commands from the Weighted Round-Robin class according to weights specified by the driver in a round-robin manner [39, 63].

Preferred Values. As the internals of NVMe devices can vary greatly between different manufacturers and generations, their performance also varies under different modes of operation. The NVMe protocol provides a way for the NVMe controller to inform the driver of its preferred configuration. For instance, namespaces can expose preferred size and alignment values for write operations using the Identify Namespace command. Additionally, NVMe controllers can provide a recommended arbitration burst size for which command arbitration is most efficient [60, 63].

LBA formats. The NVMe protocol provides a list of LBA formats with varying logical block sizes to pick from when formatting a namespace. The driver can query available LBA formats with additional information on their relative performance to another and select the best format accordingly [60, 63].

Dataset Management. The driver can provide the NVMe controller with information on how it intends to use specific data ranges on the disk, which the controller may use for optimization internally. For example, when the driver enqueues read or write commands, it can mark the operation as part of a sequential access or as a single access. Additionally, the driver can set latency requirements for commands, allowing the controller to reorder commands more effectively. The driver can also explicitly mark regions for better read or write performance using the Dataset Management command [60].

Discussion. The weighted round-robin arbitration mechanism allows for more sophisticated mapping of commands to I/O queues. For instance, we could prioritize writing metadata or journal entries to service filesystem requests faster and use the lower priority queues to copy data from the journal to its destination.

The preferred value reporting feature allows us to optimize setting up our NVMe driver. For example, we can use the preferred write granularity and alignment to calculate an optimal page size for our software cache at startup. Additionally, this feature is easy to implement, as all necessary information is either in one of the Identify command data structures or in the Controller Properties.

The command arbitration setting and the preferred value reporting are both optional features. As such, an NVMe device may not support these, in which case we fall back to round-robin scheduling and a predetermined configuration for our driver and software cache [60, 63].

Dataset Management is difficult to implement, as it requires knowledge or heuristics on access patterns. This, in turn, adds complexity to our software cache, as metadata about the filesystem commands needs to be preserved when fetching or evicting cache lines. Additionally, as our cache is not fully-associative, a cache set contains multiple blocks. This requires us to store metadata for every block in the set, as any block may get evicted by an arbitrary thread.

As LBA formats can only be configured when formatting an NVMe namespace, this feature is out of scope for our NVMe driver. In particular, our driver may only have access to a partition of a lager namespace and should not have authority over the namespace, as namespace formatting is a destructive operation.

2.2.5 SSD Trimming

The "Trim" operation was first introduced for the AT Attachment (ATA) interface as a Dataset Management command. The operation marks the specified region on the storage device as unallocated, for which the device can optimize read commands [21]. The NVMe protocol includes this operation as the "Deallocate" Dataset Management command. Reading unallocated logical blocks on an NVMe device yields either all bytes cleared to zero or all bytes set to $\texttt{FF}_h$ [60]. It is important to know which parts of the NVMe namespace are unallocated when running benchmarks, as the performance varies between allocated and unallocated blocks.

2.3 GPUs

Originally, the main tasks of Graphics Processing Units (GPUs) are processing and displaying graphics. They are responsible for tasks such as assembling primitives, calculating projections from 3D to 2D, shading, and clipping, which are all highly parallelizable tasks. This resulted in GPUs becoming highly parallel vector processors. Today, GPUs can be freely programmed and used for general purpose computation using modern APIs like CUDA [51], OpenCL [86], or HIP [13].

2.3.1 Execution Model

Modern GPUs are highly parallel processors following the single instruction, multiple data (SIMD) model, sometimes also called the single instruction, multiple threads (SIMT) model [34].

Term	AMD equivalent	NVIDIA equivalent
Multithreaded SIMD processor	Compute unit (CU)	Streaming multiprocessor (SM)
Thread block	Work-group	Thread block
SIMD thread	Wavefront	Warp
SIMD lane	SIMD lane	CUDA core or CUDA thread

Table 2.1: Equivalent GPU terms used by AMD and NVIDIA [9,35].

GPUs group SIMD lanes together in a tree-like hierarchy. At the bottom of the hierarchy, SIMD threads commonly group 32 to 64 SIMD lanes together to execute the same instruction. Next up, thread blocks group multiple SIMD threads together with shared memory to allow communication between threads. Lastly, a multithreaded SIMD processor executes multiple thread blocks and is responsible for scheduling [23,34].

Note, the terminology introduced here is a combination of the terms introduced by Hennessy et al. [23] and by NVIDIA [35]. Table 2.1 shows equivalent terms for AMD and NVIDIA.

2.3.2 Memory Model

The memory hierarchy of NVIDIA GPUs closely matches their execution model and fits into three categories, private, shared, and global memory. Each SIMD lane has a private memory region in off-chip DRAM to store stack frames and private variables that do not fit into registers. SIMD lanes do not share private memory, but the GPU may cache private memory in its L1 or L2 cache to speed up accesses. Each SIMD processor has local on-chip memory—often called shared memory—which is partitioned and shared between thread blocks executing on the SIMD processor. Global memory is the remaining off-chip DRAM shared by all SIMD lanes on the GPU [23,35].

Traditionally, programmers had to explicitly manage GPU memory, managing memory allocations and synchronization between the GPU and CPU. With the introduction of Unified Memory on modern NVIDIA GPUs and Shared Virtual Addressing (SVA) in recent Linux version, the driver does most of the memory management. Unified memory provides a unified virtual address space for all virtual memory across all GPUs and CPUs in a system, allowing for zero-copy semantics by accessing host or device memory directly via PCIe. Additionally, the driver provides managed memory, which can be used both by the GPU and CPU, with the driver automatically migrating between the two. Today, Unified

Memory supports page faulting which allows the driver to automatically migrate non-resident memory pages on access [28, 48].

SVA allows Linux to share process address spaces with devices, simplifying the use of DMA, as applications can use the same core memory management provided by the kernel. For instance, applications can simply instruct a device to perform DMA on a buffer allocated using `malloc` [28].

2.3.3 Programming Model

Traditionally, programmers used shaders written in high-level languages like GLSL [65] or HLSL [47] to program GPUs. Shaders are usually compiled at runtime by graphics drivers like OpenGL [87] or DirectX [45] directly from source or using SPIR-V [85] or DXIL [46] as an intermediate representation. Shaders do not offer much insight on the underlying GPU architecture, providing only coarse synchronization primitives for thread blocks and fast shared memory [41, 64, 84].

With the rise of high performance computation (HPC) and general purpose computation on GPUs (GPGPU), new APIs such as CUDA [51], OpenCL [86], and HIP [13] emerged, which provide APIs for general purpose processing. These APIs provide interoperability between host and device code, allowing programmers to directly include GPU kernels written in a C or C++ dialect alongside host C/C++ code. Modern APIs, such as Vulkan [88], DirectX 12 [45], and Metal [16] provide unified APIs, which can be used for both compute and graphics-related tasks. Additionally, these APIs offer a low level view on GPU architecture, providing fine-grained synchronization primitives at the thread level and ways to group and regroup SIMD lanes to fit the application's needs [49, 64].

2.3.4 NVIDIA's GPUDirect RDMA

NVIDIA's GPUDirect RDMA [52] is a technology that enables direct memory access (DMA) between third party devices and the GPU via PCIe. It is part of NVIDIA's GPUDirect [53] family of technologies, offering different ways for the GPU to directly interact with other devices. Common uses for this technology include storage and network acceleration, and recent works such as BaM [38, 69, 70] primarily rely on it.

GPUDirect RDMA uses standard PCIe features to transfer data between the physical memory of devices under the same PCIe Root Complex. To make use of this technology, device drivers must use NVIDIA's kernel driver to pin GPU memory to the PCIe BAR space of the GPU. The device driver can then use the returned page table to transfer data to or from the GPU. Note, pinning GPU memory is an expensive operation, so optimizations to reduce the amount of pinning and unpinning are important [52].

2.3.5 Peer-to-Peer Direct Memory Access on AMD GPUs

Whereas NVIDIA provides extensive documentation on GPUDirect RDMA and its API, AMD's documentation on the matter is lacking. Recent versions of the AMDgpu [12] kernel driver include the `include/drm/amd_rdma.h` header, which provides an API remarkably similar to the GPUDirect RDMA API [14].[3]

Unfortunately, to our knowledge other works using AMD GPUs to accelerate storage or network tasks exclusively use AMD DirectGMA [15], a now likely discontinued technology introduced with the AMD FirePro™ series of GPUs. At least, new GPU models do not advertise support for DirectGMA and documentation on the technology is no longer readily available [11].

Another possibility is to use the P2PDMA API the Linux kernel offers, but the client-provider model requires modification of the AMDgpu kernel driver to function as a provider for its GPU memory [83].

2.3.6 NVIDIA's GPUDirect Async

NVIDIA introduced GPUDirect Async in CUDA version 8.0 to provide GPUs with access to memory mapped I/O of third party devices. By flagging memory as memory-mapped I/O using the `cudaHostRegisterIoMemory` flag, the GPU issues memory requests directly to the third party device instead of relying on the CPU to do so [50]. Other works use this feature to map NVMe doorbell registers into the GPUs memory space, bypassing the CPU [38, 69, 70]. Unfortunately, the HIP platform does not officially support this flag currently [13].

2.4 GPU4FS

As filesystems grow more complex with advanced features such as deduplication, journaling, software-based RAID, and compression, the amount of CPU time spent on filesystem management increases drastically. As a result, researchers [17, 38, 42, 69, 70] have found interest in the use of GPUs as filesystem accelerators, including Maucher et al. with GPU4FS [42].

Whereas previous works only move parts of the filesystem onto the GPU and reuse the existing Linux virtual filesystem (VFS), GPU4FS focuses on fully offloading the filesystem onto the GPU, minimizing CPU usage in the process [41].

[3] See file `/include/drm/amd_rdma.h:62`

2.4.1 Basic structure

As an userspace filesystem, the design of GPU4FS [41] includes a management process which configures the GPU, sets up the CPU command buffers and filesystem caches, and communicates with other processes using GPU4FS. A process wishing to use GPU4FS requests access from the management process, which then sets up a shared memory region for the process. Afterward, the process can directly communicate with the GPU and issue filesystem commands. Figure 2.4 provides an overview of GPU4FS.

GPU4FS [41] handles commands using command ring buffers in shared memory regions. The process enqueues commands alongside required data by updating the 'next' pointer of the last command in the buffer. The GPU threads repeatedly poll the 'next' pointer to wait for the next command and atomically acquire it using an atomic counter in the command data structure. After the GPU has processed the command, it signals completion by incrementing the atomic completion counter. The process may poll on this completion counter or do other work in parallel.

On-disk data structures of GPU4FS [41] are similar to those found in Ext4 [79] with inodes representing files, directories as special files, a superblock containing general metadata about the GPU4FS partition, and with data organized in blocks.

As GPU4FS [41] is designed with byte-addressable NVM in mind, blocks are either 1 GiB, 2 MiB, or 4 KiB large to reflect the page sizes used on x86-64 systems. Additionally, inodes can either be 128 B or 256 B large, depending on the configuration at build time. GPU4FS uses a single block pointer format to address both inodes and blocks. Hence, block pointers encode the size of the region they are pointing at using two bits in the seven unused lower bits. GPU4FS supports indirect blocks using a bit flag in the block pointer format. A block pointer marked as 'indirect' points to a block containing additional block pointers.

GPU4FS [41] has a block allocator, which exploits the recursive block splitting mechanism to parallelize allocation across many GPU threads. If a larger free block gets split into smaller blocks, the allocator sequentially allocates these blocks efficiently using an atomic counter. The block allocator requires locking only when all free pages run out, requiring the allocator to allocate a new list. GPU4FS handles garbage collection using a stop-the-world mechanism. It sends a stop signal to all other threads if the allocator runs out of memory and reclaims unused pages before allowing the other threads to continue.

Directories use linked lists in the original design of GPU4FS [41]. Directory entries contain a block pointer to the referenced inode, an offset to the next entry in the directory, the length of the filename, and the file name string itself [41]. Kittner [33] extended directories with a tree structure for their entries to speed up file lookup and path traversals.

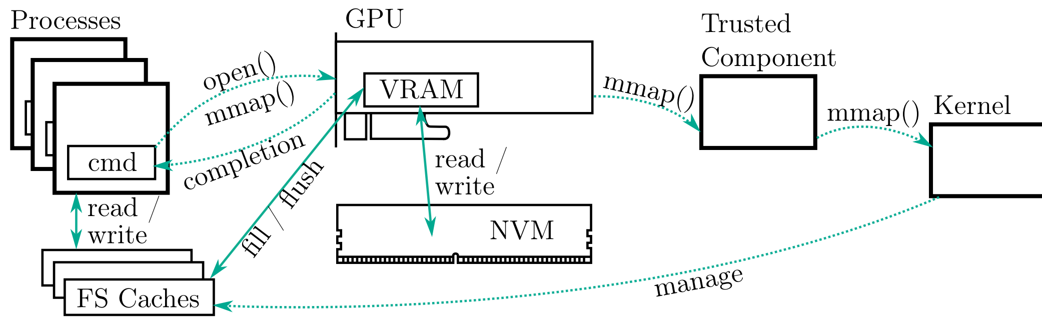


Figure 2.4: A design overview of GPU4FS. Processes write commands in GPU-accessible command buffers inside their address space. The GPU executes these commands, reading or writing from the shared filesystem caches and interacting with the storage medium. For commands requiring OS support—such as `mmap`—the GPU forwards the command to a trusted component running on the CPU, which interacts with the kernel to fulfill the requests. After a command completes, the GPU signals completion by incrementing an atomic completion counter inside the command data structure. This figure is taken from [41].

The authors of GPU4FS [41] plan to extend GPU4FS with a filesystem cache to hide the higher write latency of Intel Optane [25] memory. This is also to provide an easier way to implement the `mmap` call correctly, as defined by the Portable Operating System Interface (POSIX) [24] standard. The management process is responsible for the allocation and management of the filesystem caches [41].

2.4.2 Extended functionality

Recent works [33, 37, 71] have extended the original GPU4FS [41] demonstrator with advanced filesystem functionalities such as tree-based directories, RAID, checksumming, and deduplication.

Kittner [33] extends directories in GPU4FS with H-Trees of fixed height to speed up directory traversals. Furthermore, Kittner introduced a lookup cache correlating filename hashes with their corresponding filename, containing directory, and inode to further improve traversal. Finally, Kittner introduced full path resolution and nested directories, which the original GPU4FS demonstrator lacked.

Lucka [37] introduces filesystem-level RAID to GPU4FS. The RAID subsystem introduces a logical address space, containing logical chunks managed by the filesystem. The logical chunks inside this address space contain a list of physical chunks across multiple disks, alongside the RAID level used. This allows for different redundancy requirements on a file or block level. Lucka extended block pointers with logical and physical block pointers differentiated by a bit flag in the

lower unused bits. The RAID subsystem catches reads and writes to logical block pointers and accesses the corresponding physical blocks across multiple disks to check and preserve parity information.

Rath [71] adds checksumming and deduplication to GPU4FS. Rath implemented checksums by adding a bit flag to block pointers, denoting a block of checksums. Files include checksum blocks interspersed between data blocks, storing the checksums of said data blocks. The GPU can then independently compute and check these checksum blocks in parallel. With this checksumming functionality, Rath proposes an inline deduplication design similar to ZFS [27]. GPU4FS maintains a deduplication tree (DDT), mapping file data to deduplicated data blocks on disk. Before any data is written to disk, GPU4FS calculates a checksum and walks the DDT to check if the data is already present on disk.

2.4.3 Discussion

As GPU4FS [41] was initially designed with byte-addressable NVM storage in mind, two issues come to mind when designing an NVMe storage backend for GPU4FS.

Keeping data consistent. In its current form, the GPU4FS [41] demonstrator depends on the byte-addressable nature of NVM storage. Integrating a block-based storage backend with command arbitration into the demonstrator may pose consistency problems. Multiple GPU threads may interact with one block of the storage medium simultaneously, as GPU4FS allows and uses block sizes smaller than 512 B. This can be mitigated by exclusively working on data already loaded into the filesystem cache and choosing a single thread to enqueue a write command into the NVMe queue. A system similar to BaM's [38, 69, 70] thread coalescing may be sufficient for this problem.

Integration into GPU4FS. Integrating block-based storage into GPU4FS [41] requires an abstraction layer between the filesystem cache and the storage backend. This abstraction layer abstracts both byte-addressable NVM storage and block-based NVMe storage behind a common interface, as both are accessed in different ways with different behavior. Whereas NVM storage acts as normal DRAM, the NVMe protocol is asynchronous in nature, making use of command arbitration mechanisms. Similarly to BaM [38,69,70], we envision a synchronous API where threads requesting cache lines from the filesystem cache are responsible for loading the data from the storage medium into the cache.

Chapter 3

Related Work

This chapter provides background on other works in the field of GPU-based filesystems. Specifically, we provide detailed information on the design of BaM [38, 69, 70], as we believe it is highly applicable for our design. Additionally, we provide an overview over other works such as GPUfs [75] and SPIN [17], which make use of GPUs to accelerate filesystem-related tasks.

3.1 BaM

Big accelerator Memory (BaM) [38,69,70] is a system architecture utilizing novel GPU technologies such as GPUDirect RDMA [52] and GPUDirect Async [50] to provide GPU applications direct access to storage devices. BaM's goal is to alleviate the complexity and overhead that big datasets, used in AI and data science workloads today, entail. It achieves this by providing a simple array-like abstraction of storage that developers can use similarly to GPU memory, hence the name Big accelerator Memory. The abstraction hides novel access coalescing and caching techniques tailored towards the degree of parallelism GPUs provide [38, 69, 70].

In this section, we provide insight on the design and implementation of BaM. Specifically, we discuss the `libnvm` [40] library and its role in providing GPUs access to NVMe SSDs. Next, we give an overview of BaM's design, focussing on its command queuing, thread coalescing, and caching technology. Finally, we explore BaMs' API design and conclude this section with a discussion on how BaM's design is helpful in designing our NVMe driver.

3.1.1 `libnvm` Kernel Module

The `libnvm` [40] kernel module provides userspace programs with a way to pin NVMe queues to GPU memory and map them for DMA access from other PCIe devices. Additionally, it allows userspace programs to map the Controller Properties into memory to configure the NVMe controllers manually [38, 69, 70].

To achieve this, the module allocates character devices for each attached NVMe controller and provides `ioctl` commands for configuration. On request, the module uses NVIDIA's GPUDirect API to pin and map GPU memory for DMA access and the Linux memory management API to map the controller properties into userspace memory [38, 69, 70].

3.1.2 `libnvm` Userspace Library

The `libnvm` [40] userspace library abstracts away most of the interaction with the character devices created by the kernel module. It provides useful abstractions for different components of the NVMe protocol such as namespaces, controllers, and queues. These abstractions include methods to allocate queues and to query information about the controller and its namespaces. Furthermore, it handles the allocation and mapping of GPU memory for the NVMe queues using the kernel module [38, 69, 70].

3.1.3 Virtual Queues

BaM [38, 69, 70] hides the latency and out-of-order execution of NVMe commands by mapping physical queue pairs onto large virtual queues. Each virtual queue has some additional associated metadata, including a global ticket counter, a turn counter array, a tail marker bit set, and a head marker bit set. BaM uses the global ticket counter to give each SIMD lane accessing the storage a unique ticket. The SIMD lane uses this unique ticket to calculate its turn and position in the queue. The turn counter array sequentially orders SIMD lanes accessing the same position in the queue. The tail marker bit set denotes which positions in the Submission Queue have valid commands ready to be submitted to the controller. Conversely, the head marker bit set denotes which positions in the Completion Queue are ready to be freed by updating the head pointer on the controller. The virtual queue uses this metadata to sequentially order SIMD lanes issuing I/O commands in parallel with minimal critical sections [38, 69, 70].

To issue an I/O command, a SIMD lane atomically reads and increments the global ticket counter to acquire a unique ticket. This ticket determines the position in the physical queue, as well as the order in which SIMD lanes enqueue commands at said position. More specifically, each SIMD lane divides its ticket

by the size of the physical queue and uses the remainder as the position in the queue and the quotient as the turn value, determining the order of SIMD lanes. Note, the quotient is multiplied by two to order both enqueuing and dequeuing on the Submission and Completion Queue, respectively [38, 69, 70].

Each position in the physical queue has an associated turn counter in the turn counter array. When this turn counter reaches the turn value of the SIMD lane, all commands of previous SIMD lanes using the same position have completed. The SIMD lane can then safely write its I/O command into the position and set the tail marker bit of this position in the bit set [38, 69, 70].

Afterward, all SIMD lanes that enqueued I/O commands alternate between polling their position's tail marker bit and trying to acquire the tail lock. The SIMD lane that acquired the lock iterates through the enqueued commands from the current Submission Queue tail pointer. It clears the tail marker bits and increments the tail counters by one until it either hits the end of the queue, or a position with a still unset tail marker bit. Then, the SIMD lane updates the tail pointer, rings the Submission Queue doorbell register of the NVMe controller, and releases the lock. The other SIMD lanes either continue, if their tail marker bit is cleared, or they keep trying to acquire the tail lock. This effectively commits as many I/O commands as possible, reducing unnecessary slow writes to the doorbell register [38, 69, 70].

After the I/O command is committed to the NVMe controller, the SIMD lane iterates through the Completion Queue from the current head to search for its corresponding completion entry with the correct phase bit and command ID. It then locks the lock associated with the position and sets the associated head marker bit in the bit set [38, 69, 70].

All SIMD lanes that try to dequeue their completion entries alternate between polling the head marker bit and trying to acquire the head lock, similarly to the enqueuing algorithm. Once a SIMD lane acquires the lock, it iterates through the completion entries from the current head, resetting the head marker bits and incrementing the turn counters by one. Once it encounters the end of the queue indicated by an invalid entry, it updates both the Submission Queue head using the head pointer provided by the NVMe controller in the entry and the Completion Queue head. Finally, it writes the updated head to the Completion Queue doorbell register and releases the lock. The other SIMD lanes continue once their head marker bit is unset and the Completion Queue head is moved past their entry, or when they successfully acquire the lock [38, 69, 70].

This algorithm minimizes both critical sections and redundant writes to the doorbell registers, while avoiding possible race conditions further detailed in the BaM paper and associated dissertations [38, 69, 70].

3.1.4 Caching

BaM [38, 69, 70] minimizes I/O requests using a reference count based write-back cache design. The design minimizes critical sections by preallocating all required memory for the cache on startup, and only requires locks when changing cache-line mappings. Furthermore, the design enforces two important invariants. First, at most one copy of a cache-line is present in the cache, maintaining data consistency. Second, if multiple SIMD lanes request a cache-line, only one SIMD lane makes the necessary I/O request, minimizing I/O requests [38, 69, 70].

The BaM cache maintains two data structures, an array of cache-lines and an array of cache-slots. A cache-line describes a contiguous region in the storage medium and consists of an index into the cache-slot array, a state containing three bit flags—'Valid', 'Busy', and 'Dirty'—and a 29-bit reference count.

A cache-slot describes a contiguous region in the cache to load a cache-line into, and consists of a cache-line tag and a lock used for insertion and eviction. This design provides constant lookup times at the cost of space efficiency, as the array of cache-lines grows with larger datasets [38, 69, 70].

When a SIMD lane requests a cache-line, it atomically fetches and increments the state of the cache-line, thereby simultaneously incrementing the reference count. If the 'Valid'-bit is set, the SIMD lane can immediately access the data in the cache using the cache-slot index. Otherwise, the SIMD lane atomically fetches and sets the 'Busy'-bit and proceeds to load the cache-line from the storage medium, if the 'Busy'-bit was not already set by another SIMD lane. If a SIMD lane writes to the cache-line, it sets the 'Dirty'-bit, marking the cache-line to be written back on eviction. Finally, if the SIMD lane finishes its work on the cache-line, it decrements the state of the cache-line, decrementing the reference count [38, 69, 70].

Cache-slot eviction uses a clock replacement algorithm with a global counter. Each SIMD lane chooses a cache-slot by fetching and incrementing the global counter and acquiring the cache-slot's lock. Afterward, the SIMD lane tries to set the 'Busy'-bit of the associated cache-line and optionally writes the modified data back to the storage medium. Then, the SIMD lane loads the new cache-line from the storage medium, updates the cache-slot and releases the lock. If the SIMD lane fails at any time, i.e., it fails to acquire the lock or to set the 'Busy'-bit, it tries another cache-slot using the global counter. Note, the clock replacement algorithm with locking allows multiple SIMD lanes to concurrently evict and insert different cache-slots [38, 69, 70].

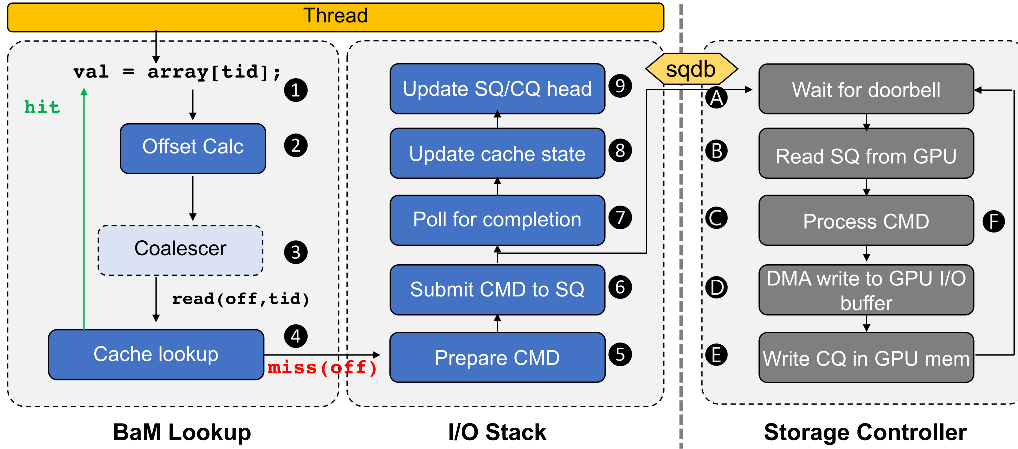


Figure 3.1: An overview of the BaM stack. BaM first coalesces SIMD lanes accessing the same cache line before reading the cache. If the data is not present in the cache, the SIMD lane goes through the I/O stack. It prepares and submits a command to the Submission Queue, polls for its completion and updates both the cache and the Submission and Completion Queue head pointers. This figure is taken from [70].

3.1.5 Abstraction

The BaM [38, 69, 70] API aims to minimize required code changes for common GPU kernels. To this end, BaM provides an array-like API with overloaded subscript operators that coalesce SIMD lanes in a thread, access the cache, and perform I/O request on behalf of the developer. Figure 3.1 provides an overview of the complete BaM I/O stack.

BaM coalesces SIMD lanes in a thread accessing the same cache-line simultaneously by computing a mask using the `__match_any_sync` primitive. Afterward, BaM determines a leader among these SIMD lanes, which performs the cache lookup and broadcasts the result using the `__shfl_sync` primitive. BaM proceeds similarly with SIMD lanes releasing the same cache-line [38, 69, 70].

The `__shfl_sync` primitive simultaneously exchanges a value between all SIMD lanes in a thread. Each lane receives and returns the value the specified source lane passes into the primitive. The `__match_any_sync` primitive compares a value across SIMD lanes in a thread. For each lane, it returns a mask of lanes that have the same value. This mask can then be used to synchronize only a specific subset of lanes in other synchronization primitives. The primitives `__match_any_sync` and `__shfl_sync` both act as barriers, which all SIMD lanes specified in the mask must reach before continuing [49].

To minimize cache lookups for consecutive accesses of the same cache-line, BaM provides a wrapper that only releases the cache-line if the SIMD lane accesses another cache-line, or if the wrapper goes out of scope [38, 69, 70].

In the optimal case, this reduces cache lookups to a single acquire and release if all SIMD lanes in a thread access the same cache-line throughout the entire GPU kernel execution [38, 69, 70].

3.1.6 Discussion

BaM's [38, 69, 70] virtual queue and cache designs are a helpful basis for our NVMe driver design. The virtual queue algorithm, in particular, fits well into a general purpose NVMe driver for the management of single queues, as it caters towards the highly parallel execution model of GPUs. However, we may extend the algorithm to manage multiple queues efficiently with weighted round-robin command arbitration provided by the NVMe protocol and stricter requirements to facilitate crash consistency.

BaM's [38, 69, 70] cache design, however, requires more adaption to be useful for our filesystem cache. Specifically, the high space requirements would have the cache take up most of the available VRAM. For example, a 2 TiB NVMe SSD with a cache-line size of 4 KiB would require 4 GiB of VRAM for the cache-line lookup table alone, as each entry takes up eight bytes. A design suitable for our filesystem cache should instead use linear search, a hash table, or a search tree to search for the correct cache-line. Still, BaM's design requirements align with our own, requiring fast lookups, minimal locking, and minimal I/O requests.

3.2 SPIN

SPIN [17] integrates peer-to-peer DMA between GPUs and SSDs into the existing Linux VFS. It uses a shim library, loaded using `LD_PRELOAD`, to intercept filesystem calls and forward them to the P-router. The P-router then determines if the request is already in the cache or if peer-to-peer DMA should be used to handle the request. To be able to perform peer-to-peer DMA requests, SPIN implements an address tunneling mechanism to transport the GPU address to the Linux NVMe driver, as the Linux VFS stack fails to correctly pin GPU memory for DMA access. SPIN uses a modified NVMe driver with additional code to unpack tunneled GPU addresses for the DMA request [17].

3.3 GPUfs

GPUfs [75] is a software stack providing a POSIX-like API for GPUs to access files through the existing Linux VFS layer. In contrast to BaM [38, 69, 70], GPUs do not access the underlying storage devices themselves with GPUfs. Instead, they request the CPU to perform the access using the standard Linux VFS through an RPC protocol [75].

To allow for more advanced filesystem and POSIX features such as read-ahead and `mmap`, GPUfs employs a distributed buffer cache, accessible from the CPU, as well as from the GPU. The cache functions similarly to the Linux filesystem cache, where each page has an associated 'pframe' metadata structure with data pertaining to the cached file region. GPUfs manages the cached pages of a file in a dynamic radix tree designed for lockless traversal, only using a lock for updates or when traversals repeatedly fail [75].

Chapter 4

Design

This chapter introduces the design of our NVMe driver, explaining and discussing our design decisions and data structures used. Specifically, we will discuss queue management as well as concurrent and consistent access through software caches.

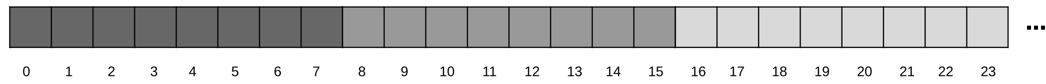
The goal of our NVMe driver is to provide GPU4FS [42] with a high-level interface to SSD hardware, designed with GPU parallelism in mind. Our design takes inspiration from BaM [38, 69, 70] for their virtual queue and software cache design and from Adas et al. [2] for their set-associative parallel software cache design.

4.1 Queue Management

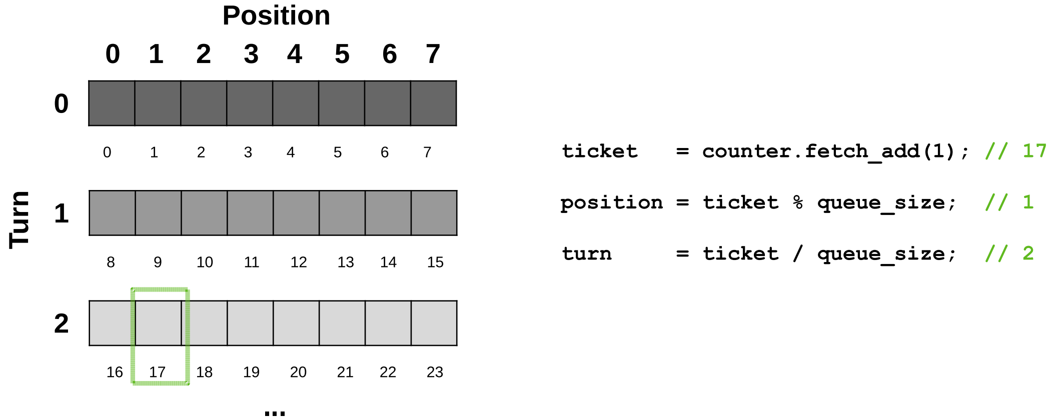
Our I/O queue management system uses BaM's [38,69,70] virtual queues with an additional abstraction layer on top to allow for more dynamic arbitration between multiple virtual queues. While BaM uses a round-robin scheme to map SIMD lanes onto virtual queues, we use a system more akin to the weighted round-robin arbitration mechanism introduced in Section 2.2.4. We use heuristics to approximate the number of outstanding commands per virtual queue to achieve a load-balancing effect with our virtual queue mapper.

4.1.1 Virtual Queues

Our virtual queue design directly copies BaM's [38, 69, 70] virtual queue design with minor optimizations and changes in implementation. A virtual queue is an infinitely large queue, used to sequentially order SIMD lanes enqueuing commands and to map them onto a slot in the underlying physical queue.



(a) Infinite virtual queue for a physical queue of size 8.



(b) Mapping virtual queue slots onto the underlying physical queue.

Figure 4.1: A visualization of the infinitely large virtual queue and how it maps onto a physical queue. This figure is taken from [69].

As the NVMe protocol uses circular command queues, mapping a slot from the virtual queue onto the physical queue can be done by dividing the slot's index by the physical queue's size and taking the remainder. Additionally, the result of the division can be used to sequentially order SIMD lanes waiting on the same physical slot. Figure 4.1 visualizes how we map the infinite virtual queue onto the physical queue.

For each slot in the physical queue, locks and markers ensure that no other SIMD lanes overwrite commands not yet submitted or completions not yet processed. Each SIMD lane synchronously enqueues its command and waits for its completion before returning to the caller.

Data Structure

We simulate an infinitely large virtual queue using a global ticket counter. Every SIMD lane atomically fetches and increments this counter to acquire a ticket. The SIMD lanes divides this ticket by the physical queue's size to calculate a position in the physical queue and a turn value.

Note, using a global counter forces the physical queue size to be a divisor of the total number of counter values. Using either a 32-bit or 64-bit unsigned integer for the counter forces us to use powers of two as our physical queue size.

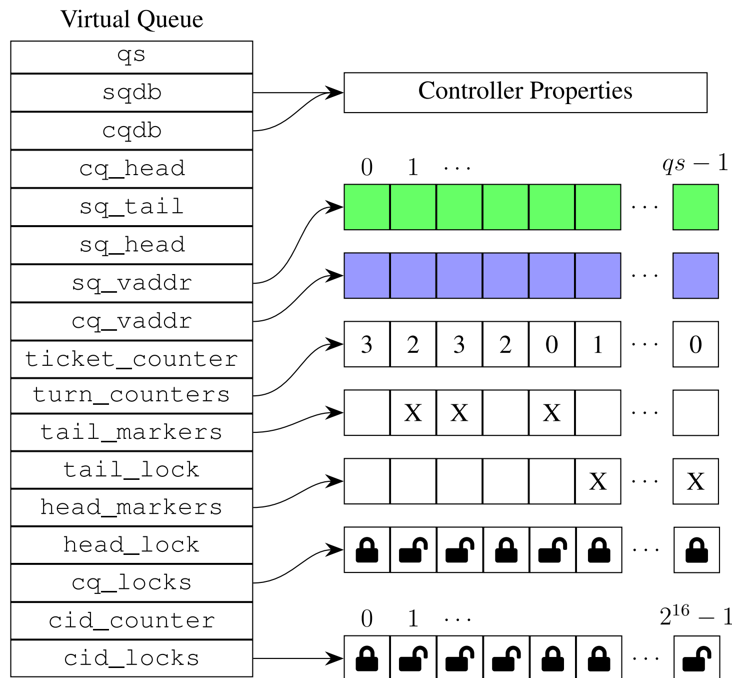


Figure 4.2: A visualization of the virtual queue data structure. The virtual queue on the left holds information about the underlying physical queue, in addition to different locks and atomic counters. Information on the physical queue includes its size (qs), its memory addresses (sq_vaddr, cq_vaddr), the doorbell registers (sqdb, cqdb), and its head and tail pointers (cq_head, sq_tail, sq_head). The remaining metadata includes atomic counters, locks, and markers used in the enqueueing and dequeueing algorithms. Most of the metadata is stored for every slot in the physical queue in additional arrays on the right. Additionally, the data structure uses an array of locks for each available command ID.

Otherwise, an integer overflow would skip entries in the physical queue. Still, for 64-bit counters this is only a theoretical issue, as it would take around half a million years operating at one million I/O operations per second to cause an integer overflow with a 64-bit counter.

The virtual queue holds metadata for every slot in the physical queue, consisting of turn counters, tail and head markers, and locks. The turn counters sequentially order SIMD lanes waiting on the same slot. The tail and head markers mark entries for submission and deletion, respectively, and the locks avoid race conditions when polling for the correct completion entry.

To prevent multiple SIMD lanes from using the same command ID, the virtual queue holds a global command ID counter similar to the global ticket counter and an array of locks for each command ID. Every SIMD lane acquires a unique

command ID by atomically fetching and incrementing the global command ID counter and locking the command ID in an array of locks.

Lastly, the virtual queue holds the state of the tail and head pointers to update the doorbell registers correctly and to handle the phase bit of completion entries correctly. The phase bit is a bit inside completion entries that the NVMe controller flips on each pass. We can calculate the expected phase bit by taking the least significant bit of the CQ head pointer stored in the virtual queue. Note, we implement both the SQ tail pointer and CQ head pointer similarly to the global ticket counter. That is, we only increment the pointers and calculate the real SQ tail and CQ head pointers in the physical queue through division by the physical queue's size.

Each virtual queue has its own set of global atomic counters, markers, and locks. This ensures that virtual queues can work completely independently of each other, providing better concurrency with more virtual queues. The only synchronization between most SIMD lanes on different queues is an atomic operation inside the virtual queue mapper to determine the virtual queue to use. A visualization of the complete data structure can be seen in Figure 4.2.

Enqueueing Commands

When a SIMD lane wants to enqueue a command into the virtual queue, it first acquires a unique ticket by atomically fetching and incrementing the global ticket counter of the virtual queue. The SIMD lane uses this ticket to calculate its position in the physical queue and a turn value on that position. Afterward, the SIMD lane waits for the turn counter for its position to equal the turn value, indicating that previous commands have been processed and dequeued from the completion queue [69].

Then, to get a unique command ID, the SIMD lane atomically fetches and increments the global command ID counter and tries to acquire the lock associated with the command ID. This is done in a loop until the SIMD lane successfully acquires one of the command ID locks. Without locking, the pseudo random order in which the GPU schedules atomic instructions between SIMD lanes may result in multiple SIMD lanes picking the same command ID on the same turn [69].

Note, locking command IDs is only necessary for sufficiently large physical queues and SIMD lane counts. It may be advisable to omit command ID locking and use smaller physical queues, depending on the capabilities of the NVMe SSD used. With smaller physical queues, unique command IDs can be calculated from the ticket without collisions on the same turn. Specifically, if the physical queue size is less than 2^{15}, half the size of the command ID space, we can assign a unique command ID for every combination of position and phase.

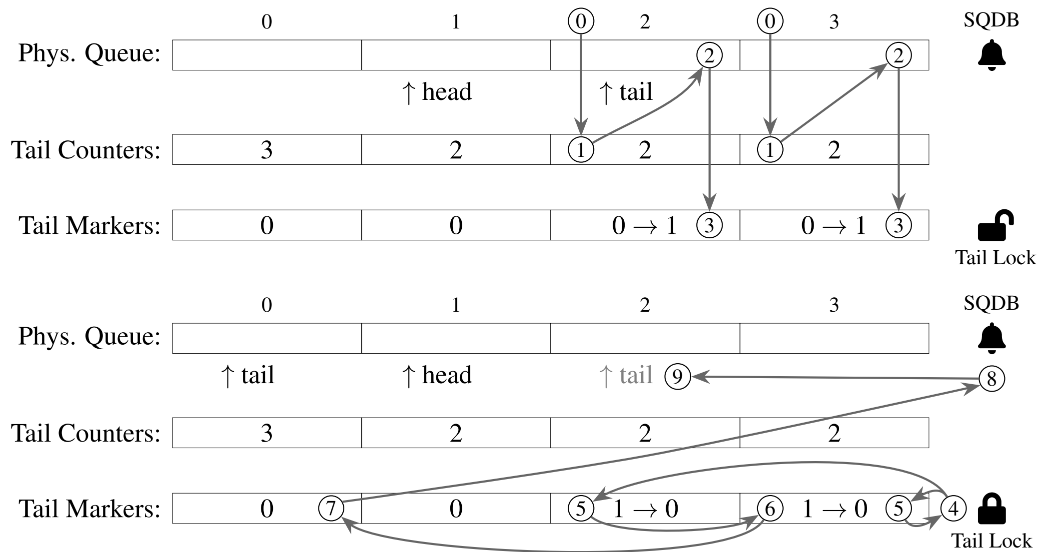


Figure 4.3: A visualization of the enqueueing algorithm using two SIMD lanes and a physical queue with four slots. After the lanes acquire their tickets (0), they wait for their turn counter (1), write their commands into their slot (2), and set their tail marker (3). Next, the lanes race to acquire the tail lock (4). The successful lane resets the tail markers from the current tail pointer (5,6) until it hits an unset marker (7). Afterward, it rings the doorbell register (8) and updates the local copy of the tail pointer (9). The other lane waits for their tail marker to reset while trying to acquire the lock (4,5). This visualization is adapted from [69].

Next, the SIMD lane copies its I/O command into its assigned position in the physical queue and marks it for submission in the tail marker array. While waiting for their tail markers to be cleared, SIMD lanes race to acquire the tail lock. The successful SIMD lane iterates from the current SQ tail pointer through the tail marker array, clearing the tail markers and incrementing the tail pointer until it either hits an unset marker or the end of the queue. Afterward, if the SQ tail pointer has changed, it writes the new value to the SQ doorbell register and releases the tail lock. This ensures, that writes to the doorbell register happen in order and only complete commands are submitted [69]. Figure 4.3 visualizes the enqueueing algorithm for two SIMD lanes.

Dequeuing Completions

SIMD lanes can poll on the Completion Queue without any locks to find the completion entry corresponding to their command. The correct completion entry can be identified by the unique command ID and the phase bit set by the controller.

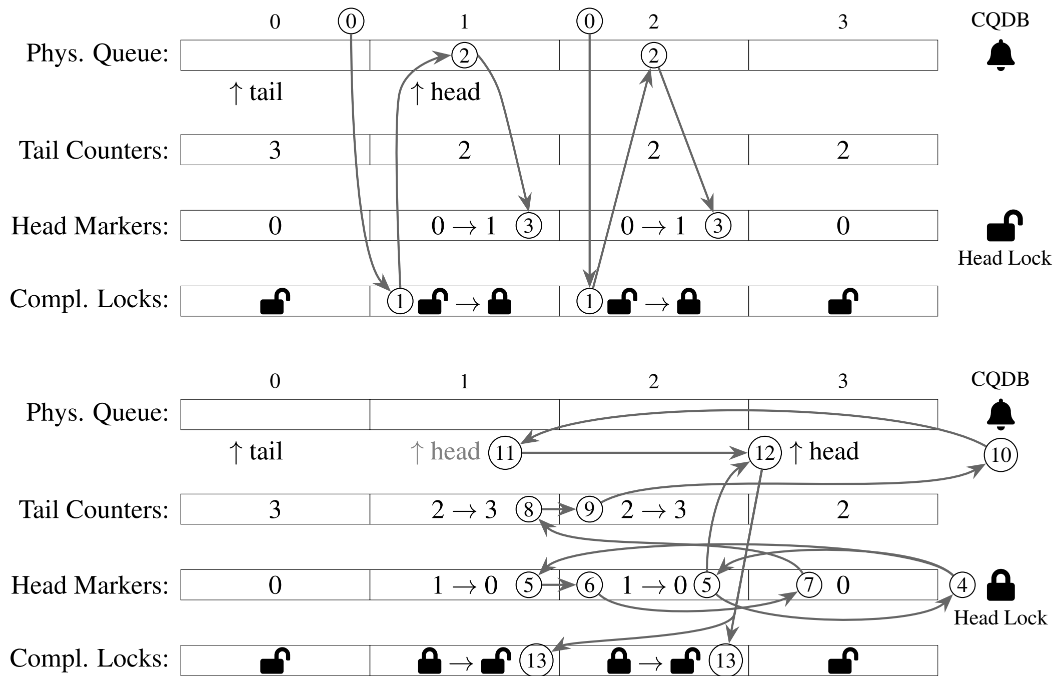


Figure 4.4: A visualization of the dequeueing algorithm using two SIMD lanes and a physical queue with four slots. After the treads find their completion entry (0), they lock the slot (1), read the command status (2), and set their head marker (3). Next, the lanes race to acquire the head lock (4). The successful lane resets the head markers (5,6) until it hits an unset marker (7) and increments the tail counters based on the SQ head stored in the completion entry (8,9). Afterward, it rings the doorbell register (10) and updates the local copy of the head pointer (11). Finally, both lanes check if the head pointer has passed their entry (12) and unlock the slots again (13). This visualization is adapted from [69].

Polling is necessary as the NVMe controller may process commands out of order. SIMD lanes poll on the Completion Queue by repeatedly iterating from the current CQ head pointer to the end of the queue. For each position, the SIMD lane calculates the expected phase bit by dividing the index by the physical queue size and taking the least significant bit. Locking is not necessary here, as completions can be uniquely identified by the unique command ID and phase bit [69].

After the SIMD lane found the completion entry for its command, it acquires a lock for the position from the CQ lock array. Locking individual CQ entries achieves the same effect as the turn counter array for the Submission Queue. It orders dequeue operations between multiple SIMD lanes on the same position in the Completion Queue, preventing them from simultaneously dequeueing entries or dequeuing entries out of order [69].

Next, the SIMD lane saves the status of the completion to return to the caller later and marks the completion entry for dequeueing in the head marker array. Similarly to the enqueueing algorithm, SIMD lanes race to acquire the CQ head lock while waiting for their head marker to reset. The successful SIMD lane iterates from the CQ head pointer, clearing head markers and incrementing the head pointer until it encounters an unset head marker. If the CQ head pointer has changed, the SIMD lane writes the new value to the CQ doorbell register. This orders writes to the doorbell register, similarly to the enqueueing algorithm [69].

The NVMe protocol includes an SQ head pointer field in completion entries to allow controllers to read and cache multiple commands and to notify the host how far the controller has read into the Submission Queue. The SIMD lane with the CQ head lock uses this field to free up slots in the Submission Queue for new incoming commands. It does so, by iterating from the current SQ head pointer to the head pointer stored in the completion entry last dequeued, incrementing the tail counters for each position. Incrementing the tail counters allows waiting SIMD lanes to enqueue their commands into the Submission Queue. Afterward, the SIMD lane releases the CQ head lock [69].

Finally, the SIMD lane waits for the CQ head to move past its position in the Completion Queue before releasing both the CQ lock on that position and the command ID lock for its command ID. SIMD lanes may only release the command ID lock after the CQ head has moved past its position. This ensures, that the next SIMD lane using the same command ID polls for its completion entry strictly after the completion entry of the previous SIMD lane [69]. Figure 4.4 shows the dequeuing algorithm using two SIMD lanes.

The CQ locks prevent a potential race condition when waiting for the head marker to be cleared. For instance, a SIMD lane, t_i, may be scheduled out right after setting the head marker for its completion entry at position p. Then, another SIMD lane acquires the head lock, resets the head marker for position p, and moves the CQ head pointer past it. Now the Controller enqueues a new completion entry on position p and a different SIMD lane, t_k, can read the new completion entry and mark the head marker bit for position p again. This leads to a contradictory state, as after t_i resumes execution, it sees the head marker bit is still set, but the CQ head pointer is already past its entry. The CQ locks order the dequeue operations, which prevents this race condition from happening [69].

Differences to BaM

Our design differs from BaM's [38,69,70] design in the use of turn counters. BaM increments the turn counter two times in the algorithm, once after a command is submitted and the tail marker is cleared and once when moving the SQ head pointer past the respective position. Therefore, BaM doubles the turn value SIMD

lanes calculate after acquiring a ticket, resulting in SIMD lanes waiting for two increments to the respective turn counter.

In contrast, our design omits the first increment to the turn counter, only incrementing them when moving the SQ head pointer past their position. As SIMD lanes always wait for the turn counter to equal their turn value, the next SIMD lane only continues after the second increment in BaM's design. Omitting the first increment and omitting to double the turn values does not change the behavior, as the next SIMD lane still waits for the later increment before continuing.

4.1.2 Queue Arbitration

To allow the use of multiple I/O queues per controller, we propose a virtual queue mapper, an abstraction layer mapping SIMD lanes onto virtual queues. The virtual queue mapper consists of an array of queue indices, a global ticket counter and virtual queues. The array of queue indices can be larger than the number of virtual queues to allow dynamic mapping of multiple slots to a single virtual queue.

To enqueue a command, a SIMD lane atomically fetches and increments the global ticket counter. The SIMD lane then divides the ticket by the length of the index array in the virtual queue mapper and uses the remainder as its index in the index array. Afterward, the SIMD lane reads the queue index from the index array and uses the specified virtual queue to enqueue its command. Figure 4.5 visualizes the mapping between the virtual queue mapper and the virtual queues.

As NVMe controllers may process commands out of order, the number of outstanding commands can differ per queue when using multiple queues. Our virtual queue mapper aims to alleviate this imbalance with dynamic remapping. Given a suitable heuristic, we can dynamically adjust the mapping between the index array and the virtual queue array, prioritizing less overloaded queues. A precise heuristic is to take the difference between a SIMD lane's ticket and the current SQ tail pointer after acquiring the ticket. As the SQ tail pointer starts at zero and gets incremented for every command, it directly correlates with the ticket counter. As such, the difference between the two indicates the number of outstanding commands the SIMD lane has to wait for. This heuristic can be stored on the virtual queue with an atomic maximum operation.

After a SIMD lane has executed its command on the virtual queue, it checks if the given heuristic has surpassed a threshold value chosen beforehand. In that case, the SIMD lane resets the stored heuristic value, selects a virtual queue to migrate to and tries to change the mapping using an atomic compare-and-swap operation with the old value. Preferably, the SIMD lane picks the virtual queue with the best—or in our case lowest—heuristic value. We use a compare-and-swap operation to allow only a single SIMD lane to change the mapping, as multiple SIMD lanes might try to change the mapping at the same time.

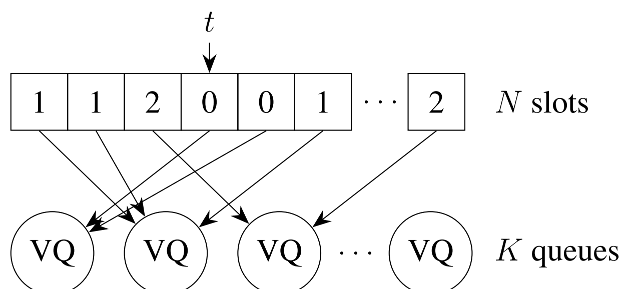


Figure 4.5: A visualization of the virtual queue mapper. The virtual queue mapper maps N slots onto K virtual queues with $N \geq K$. Each slot holds an index into the virtual queue array. A SIMD lane uses its ticket $t \mod N$ as an index into the index array to get the virtual queue to insert a command into.

4.2 Software Cache

Our software cache design adapts BaM's [38, 69, 70] software cache to reduce its memory footprint, by applying ideas from Adas et al. [2] with their set-associative parallel software cache implementation. At its core, our software cache uses the same data structures as BaM, but drops the cache line array allocated for all blocks on the storage medium. Instead, our software cache moves this information into the cache slot array to save memory, with additional logic to evict and replace cache lines correctly. With this change, our software cache no longer supports constant time lookups. As such, we further change the design from a fully-associative cache to a set-associative cache to shorten lookup times. To access the cache, we implement the same thread coalescing interface as BaM.

4.2.1 Basic Structure

Our software cache consists of two separate contiguous memory regions for cache line data and metadata, respectively. Given a cache with N cache sets and K cache lines per set, both memory regions resemble a two-dimensional rectangular $N \times K$ array. This memory layout enables easy access for both metadata and data given the cache set index n and cache line index k, as presented in Figure 4.6.

We use state management similar to BaM [38, 69, 70] for our cache sets and cache lines. Our cache lines contain a state value identical to BaM, with a 29-bit reference counter and three state bits—a Valid-bit, a Busy-bit, and a Dirty-bit—to manage the state of the cache line. First, the reference counter ensures the cache only evicts cache lines which are no longer referenced. Next, the Valid-bit signals that the cache line currently contains a valid block. Then, the Busy-bit signals that a SIMD lane is currently evicting or inserting a block into the cache line.

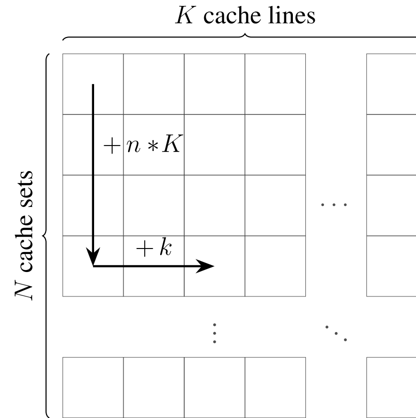


Figure 4.6: A visualization of the software cache's memory layout. The cache resembles a two-dimensional $N \times K$ array, stored contiguously in memory. We access the kth cache line of the nth cache set by converting the two-dimensional index (n, k) to a linear index $i := n * K + k$.

Valid	Busy	Dirty	Definition
0	0	0	Not in cache
0	0	1	Impossible configuration
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

Table 4.1: Definitions for possible cache line states. This table is taken from [69].

Finally, the Dirty-bit signals that the cache line is dirty and has to be written back to disk on eviction. Table 4.1 gives a more detailed explanation for each combination of state bits. The reference count together with the Busy-bit resembles a readers-writer lock. They allow multiple SIMD lanes to acquire the same cache line, but permit only one SIMD lane to evict or insert a block into the cache line when the reference count is zero.

Similarly to BaM [38, 69, 70], we use a clock replacement algorithm [19] to evict and replace cache lines in a set. We use a lock on the cache set to allow only a single SIMD lane at a time to evict and replace cache lines in a cache set. This ensures our software cache stays consistent and does not contain multiple copies of the same block in a set.

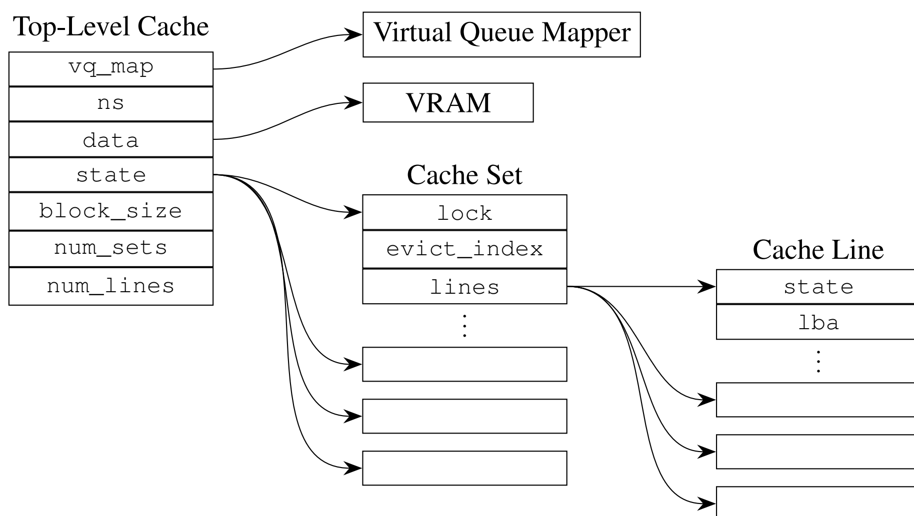
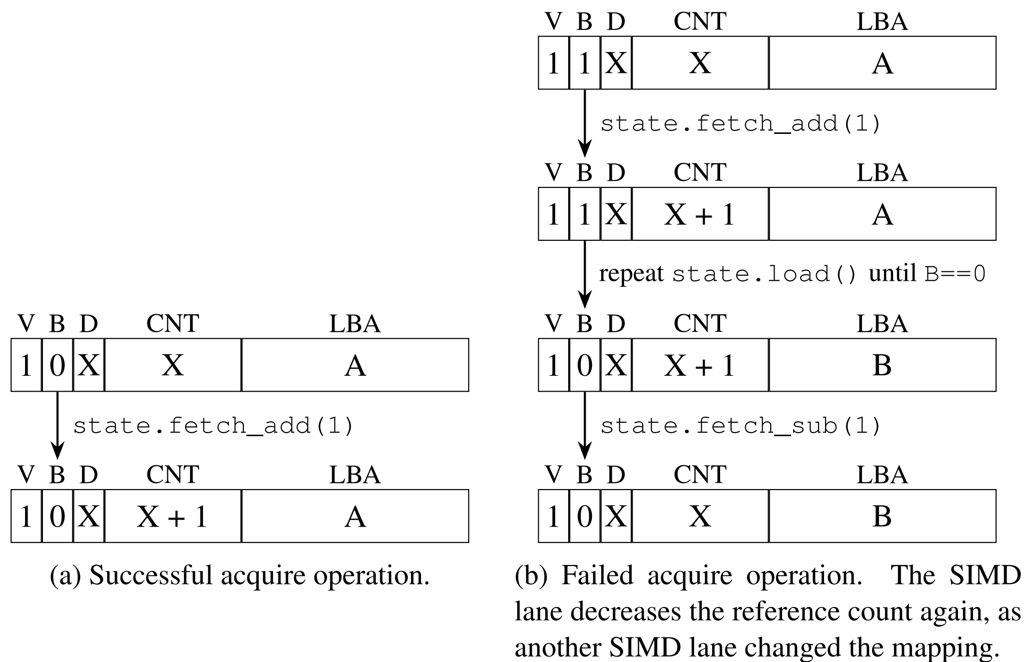


Figure 4.7: A visualization of the cache data structures and how they relate to each other. The top-level cache contains information on the NVMe namespace (ns) and a pointer to the virtual queue map (vq_map). It holds pointers to the cache line data (data) and cache line metadata (state). Additionally, it holds information on the cache layout (block_size, num_sets, num_lines). Cache sets hold a lock and the eviction index (evict_index) for the evict-and-replace algorithm, as well as an array of cache lines (lines). Cache lines hold the state value and the currently stored LBA address.

4.2.2 Data Structures

We split our software cache into three data structures, as presented in Figure 4.7. First, the top-level cache data structure holds information about the cache layout and about the NVMe namespace on which the cache operates. It holds a reference to the virtual queue map associated with the NVMe drive to read from and write to. To access the cache, the data structure holds pointers to the start of the cache line array and to the start of the metadata array, composed of cache set and cache line metadata. We can fully determine the cache layout in memory through the number of cache sets, the number of cache lines per cache set, and the cache line size stored in the top-level data structure. We use these values to calculate the offset in memory for both the cache line data and its metadata, as stated before.

Next, the cache set data structure holds information necessary for the evict-and-replace algorithm, as well as the cache line metadata for each cache line in the cache set. For the evict-and-replace algorithm, it stores a lock to allow only one SIMD lane to evict and replace cache lines and an atomic counter to choose a victim cache line to evict through the clock replacement algorithm [19].



(a) Successful acquire operation.

(b) Failed acquire operation. The SIMD lane decreases the reference count again, as another SIMD lane changed the mapping.

Figure 4.8: A visualization of the acquire algorithm on a cache line. It depicts the atomic operations on the state value and stored LBA address of the cache line. We depict the state using the Valid-bit (V), the Busy-bit (B), the Dirty-bit (D), and the reference count (CNT). Letters (X, A, B) represent unknown values. This visualization is adapted from [69].

Last, the cache line data structure holds the LBA address of the block currently stored in the cache line and a state value. Again, the state value consists of a 29-bit reference counter and three state bits, a Valid-bit, a Busy-bit, and a Dirty-bit.

4.2.3 Acquire and Release Algorithms

To acquire a block from the storage medium through the cache, a SIMD lane first calculates the cache set the block should reside in. In our demonstrator, we calculate the cache set by taking the remainder of the division of the block's index in the storage medium by the number of cache sets in the cache. This ensures neighboring cache lines fall into different cache sets.

After calculating the cache set, the SIMD lane iterates through the set's cache lines to find the cache line with the correct LBA address. If none of the cache lines have the right LBA address, the SIMD lane continues with the evict-and-replace algorithm. Otherwise, it atomically fetches and increments the state value of the cache line to simultaneously increase the reference count and fetch the state bits.

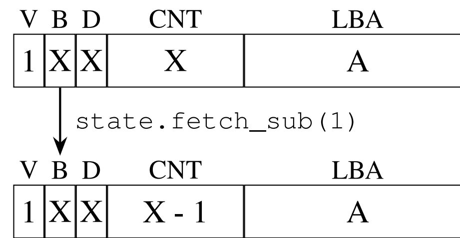


Figure 4.9: A visualization of the release algorithm on a cache line. This visualization is adapted from [69].

If the cache line currently has its Busy-bit set, the SIMD lane waits in a busy loop for the Busy-bit to reset or for the mapping to change to a different LBA address.

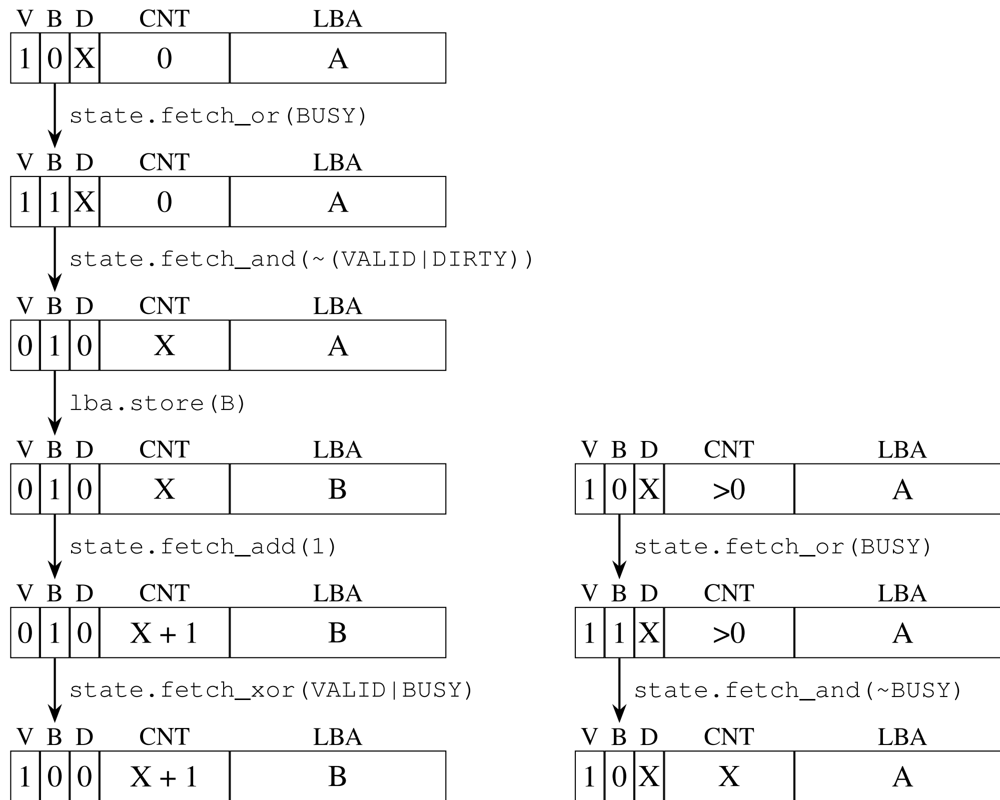
Afterward, if the cache line's Valid-bit is set and the mapped LBA address is still correct, the SIMD lane successfully acquired the cache line, as it already incremented the reference count beforehand. Before returning, the SIMD lane optionally dirties the cache line by setting the Dirty-bit using an atomic fetch-and-or operation and calculates the pointer to the cache line data using the cache set index and cache line index. Figure 4.8 presents a visualization of the acquire algorithm.

If the mapped LBA address changes at any point in the algorithm, the SIMD lane continues with the evict-and-replace algorithm instead, as another SIMD lane evicted the block in this case.

To release the cache line, the SIMD lane, again, calculates the cache set the block resides in and searches for the cache line with the correct LBA address through a linear search. After finding the correct cache line, the SIMD lane decreases the reference count with an atomic fetch-and-subtract operation to release the cache line. As the SIMD lane acquired the cache line beforehand, it is guaranteed to find the correct cache line in the set, because the non-zero reference count prevents the block from being evicted. Figure 4.9 shows a visualization of this algorithm.

4.2.4 Evict-and-Replace Algorithm

To evict and replace a cache line, the SIMD lane first locks the cache set. Before evicting a cache line, the SIMD lane searches the cache set for the correct block one more time, as another SIMD lane may have inserted the correct block into the cache already. If the correct block is still not present in the cache, the SIMD lane determines a victim cache line through a clock replacement algorithm [19]. It calculates the victim's index in the cache set by atomically fetching and incrementing the atomic counter of the set.



(a) Successful evict-and-replace operation.

(b) Failed evict-and-replace operation. The SIMD lane resets the Busy-bit and tries to evict another cache line because the reference count was not zero.

Figure 4.10: A visualization of the evict-and-replace algorithm. This visualization is adapted from [69].

Next, the SIMD lane simultaneously fetches the state of the cache line and sets the Busy-bit using an atomic fetch-and-or operation. If the cache line has a non-zero reference count or if the Busy-bit is already set, the SIMD lane tries another cache line, resetting the Busy-bit if it was not already set. Otherwise, the SIMD lane is safe to evict the cache line as other SIMD lanes trying to acquire the cache line now see that the Busy-bit is set.

To evict the cache line, the SIMD lane first resets the Valid-bit and Dirty-bit using an atomic fetch-and-and operation and writes the data back to disk, if the Dirty-bit was set.

Afterward, the SIMD lane replaces the cache line by storing the new LBA address in the cache line and loading the data into the cache line using the queue management system.

Finally, the SIMD lane increases the reference counter to acquire the cache line, simultaneously sets the Valid-bit and resets the Busy-bit using an atomic xor operation, and unlocks the cache set. Figure 4.10 visualizes evicting and replacing a cache line after selecting a victim.

4.2.5 Thread Coalescing

To reduce stress on our software cache, we implement BaM's [38, 69, 70] thread coalescing interface. Note, we use the term 'SIMD thread' instead of the term 'warp' used by NVIDIA and BaM. The thread coalescing interface groups SIMD lanes in a thread accessing the same block together to reduce the number of acquire and release operations on the software cache.

For both the acquire and the release operation, we coalesce SIMD lanes in a thread in the same way. We first split the SIMD thread into groups of SIMD lanes accessing the same block using the `__match_any_sync` intrinsic. The intrinsic calculates the mask of SIMD lanes with the same LBA address for every SIMD lane in the thread. Then, we calculate the number of SIMD lanes in a group by counting the number of bits set in the mask using the `__popcll` intrinsic. Finally, we elect a leader as the SIMD lane corresponding to the lowest bit set in the mask using the `__ffsll` intrinsic.

For the acquire operation, we check if any SIMD lane in the mask wants to dirty the cache line using the `__any_sync` intrinsic. Afterward, the leader SIMD lane acquires the cache line and broadcasts the resulting pointer to the cache line data using the `__shfl_sync` intrinsic. For the release operation, the leader SIMD lane simply releases the cache line.

In both cases, the leader SIMD lane increases or decreases the reference count by the number of SIMD lanes in the mask. This ensures that the reference count stays consistent with the number of SIMD lanes. Figure 4.11 presents the steps of this algorithm for the acquire operation.

4.2.6 Discussion

This section highlights the differences between our software cache design and BaM's [38, 69, 70] software cache. Additionally, we discuss areas where our software cache design requires further development to support important filesystem features such as software RAID and `mmap`.

Our software cache differs from BaM's [38, 69, 70] software cache in its memory layout. BaM splits the metadata into a cache slot array to store information about the blocks in the cache and a cache line array to store the state of every block on the storage medium. We omit the cache line array and instead store both the block and its state in the same array and only for blocks currently in the cache.

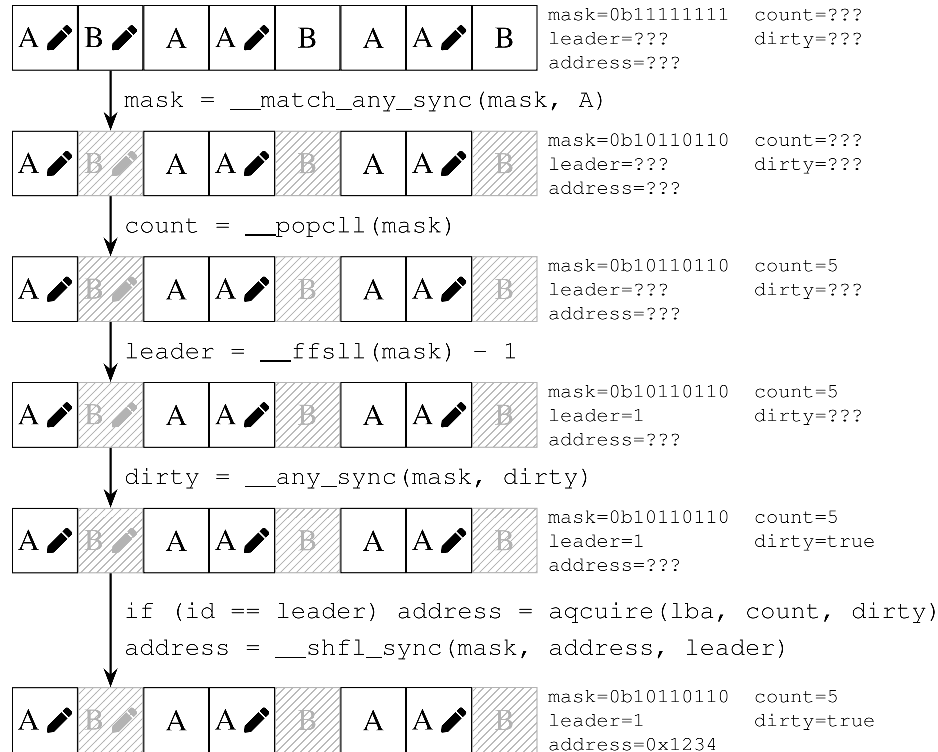


Figure 4.11: A visualization of the thread coalescing interface for the acquire operation for LBA address ‘A’. First, we create a mask of the SIMD lanes acquiring LBA address ‘A’. Next, we calculate the number of SIMD lanes in the mask and a leader SIMD lane. Afterward, we check if any SIMD lane dirties (✏) the cache line. Finally, the leader acquires the cache line and broadcasts the address.

With this change, our software cache requires a linear search to find the correct cache line instead of a constant time lookup as with BaM’s software cache. As such, we implement a set-associative cache instead of a fully-associative cache to shorten lookup times.

In its current form, our software cache cannot support important filesystem features like software RAID and `mmap`, as it only stores data in GPU memory and uses only the LBA address of the block as the cache line tag. We propose adding additional metadata to the cache line tag to support these features in future iterations of our software cache. To support software RAID, we can include the NVMe device identifier and namespace ID in the cache line tag and have the cache use the different virtual queues associated with different NVMe devices accordingly.

To enable support for `mmap`, we include residency information in the cache line tag. Residency information can include the memory address of the block as

well as an identifier whether the block is stored in GPU memory or host memory. Supporting `mmap` is more complicated, as we also have to manage pinning and unpinning cache memory for individual processes running on the system. Instead of using the cache line tag for residency information, we can integrate our software cache into a multi-level cache hierarchy with separate caches for GPU memory and host memory.

Chapter 5

Implementation

This chapter introduces the implementation of our NVMe driver and software cache. First, we configure our test system to allow the NVMe SSD and the GPU to interact over P2PDMA in a containerized environment. Next, we build and test a wrapper around `libnvm`, focussed on GPU-initiated SSD access. Afterward, we implement our queue management subsystem introduced in Section 4.1. Finally, we implement our software cache introduced in Section 4.2.

Our implementation targets x86-64 Linux systems with modern NVMe SSDs and GPUs capable of P2PDMA. While our implementation targets both NVIDIA and AMD GPUs, our demonstrator only supports NVIDIA and is not tested on AMD GPUs. We use HIP [13] as our graphics API targeting C++20 [26]. Our implementation is compiled using HIPCC [5], which in turn either uses clang [36] or NVCC [55] for AMD and NVIDIA GPUs, respectively.

We test our demonstrator on a modern system equipped with

- a Supermicro H13SSL-N motherboard [77] with support for DDR5, PCIe 5.0 and PCIe 4.0 M.2 slots,
- an AMD EPYC 9124 16-core processor [4] operating at 3 GHz,
- four Kingston 16 GB DDR5 ECC memory modules [32] operating at 4800 MT/s,
- a 1 TB Samsung 980 PRO SSD [73] on which we perform our tests,
- a 1 TB Samsung 980 SSD [72] boot drive,
- an NVIDIA RTX A4500 [59] with 20 GB of GDDR6 VRAM on a x16 PCIe 4.0 connection,
- and an AMD RX 6950 XT [7] with 16 GB of GDDR6 VRAM on a x16 PCIe 4.0 connection.

```
1 $ echo -n "0000:42:00.0" >
  ↪ "/sys/bus/pci/devices/0000:42:00.0/driver/unbind"
2 $ echo -n "0000:42:00.0" > "/sys/bus/pci/drivers/libnvm
  ↪ helper/bind"
```

Listing 5.1: Unbinding the default NVMe driver and binding the `libnvm` [40] driver. In this example, the SSD has the address `0000:42:00.0`.

5.1 Demonstrator

Our demonstrator uses Docker [20] for reproducible builds on different systems. This requires us to expose both the GPU and the SSD device to the containerized application. For the SSD, we use the `--device` flag to expose the `libnvm` [40] device file to the container. To expose NVIDIA GPUs to the container, we install the NVIDIA Container Toolkit [57] and configure Docker to use the NVIDIA runtime.

Both the SSD and the NVIDIA GPU require some configuration to make full use of `libnvm` [40] and P2PDMA. We configure Linux to use the `libnvm` driver provided by the `libnvm` kernel module for our SSD. To do this, we unbind the default NVMe driver for the SSD device and bind the `libnvm` driver using special files in the sysfs [44] pseudo-filesystem. This allows us to use the `libnvm` device file to map the Controller Properties into our memory space to access the doorbell registers, for example. Listing 5.1 provides the commands to bind the `libnvm` driver to the SSD device.

To allow for multiple large I/O queues and DMA buffers, we require the GPU to expose much of its internal VRAM in its PCIe BAR space, as only memory exposed in PCIe BAR spaces can be accessed using P2PDMA. On systems and GPUs which support the Resizable BAR Capability [67], the amount of VRAM exposed in PCIe BAR space can be configured using the sysfs [44] pseudo-filesystem. On our system, the NVIDIA GPU only exposes 256 MB of VRAM by default. We configured our GPU to instead expose 16 GB of VRAM. The commands to configure the GPU's PCIe BAR space can be seen in Listing 5.2.

Both of these methods of configuring the SSD and GPU are not persistent. The NVIDIA kernel module has a parameter `NVreg_EnableResizableBar`, which resolves this issue for the GPU, allowing it to resize its BAR space itself on startup [56]. Binding the `libnvm` [40] driver for specific devices is more difficult, as it likely requires blacklisting devices for the default NVMe driver. Instead, we suggest integrating the process of rebinding the SSD to `libnvm` into GPU4FS.

```
1 $ lspci -vvvs 0000:41:00.0 | grep BAR
2 $ cat
  ↪ "/sys/bus/pci/devices/0000:41:00.0/resource1_resize"
3 $ echo -n "0000:41:00.0" >
  ↪ "/sys/bus/pci/devices/0000:41:00.0/driver/unbind"
4 $ echo 14 >
  ↪ "/sys/bus/pci/devices/0000:41:00.0/resource1_resize"
5 $ echo 1 > "/sys/bus/pci/devices/0000:41:00.0/remove"
6 $ echo 1 > /sys/bus/pci/rescan
```

Listing 5.2: Exposing more VRAM in the GPU's PCIe BAR space. In this example, the GPU has the address `0000:41:00.0`. We use `lspci` to find out, if our GPU supports Resizable BAR and which sizes each BAR space supports. Reading the `resource1_resize` file provides a bitmap corresponding to the supported sizes of the BAR space for BAR1. We unbind the driver associated with the GPU and resize the BAR space for BAR1 by writing to the `resource1_resize` file. Then, we remove the device from the device list to force the scan to reinitialize the GPU device.

5.2 NVMe SSD Peer Access

We implemented wrapper classes around `libnvm` [40] focussed on GPU access. These wrapper classes mainly handle setting up DMA buffers for P2PDMA using `libnvm` and mapping Controller Properties to be GPU-accessible. To allocate peer-accessible DMA memory on the GPU, `libnvm` uses GPUDirect RDMA [61] internally in its kernel driver. As such, our implementation only works on NVIDIA GPUs in its current form. We aim to port `libnvm` to work with AMD GPUs in future iterations of our NVMe driver. Additionally, we use an officially unsupported flag `hipHostRegisterIoMemory` when mapping Controller Properties to be GPU-accessible. This flag works when targeting NVIDIA GPUs, as HIP is a thin wrapper[1] around CUDA calls in this case, though this flag might not work when targeting AMD GPUs [6]. The flag marks the memory as I/O memory of a third-party PCIe device and marks it as non cache-coherent and contiguous [50].

Using `libnvm` [40] we see long startup times of around one to two seconds. Due to time constraints, we have not investigated the cause of the long startup time. However, we suspect the initialization of the NVMe device to be the culprit, as `libnvm` fully resets the NVMe controller on startup.

[1] See file `include/hip/nvidia_detail/nvidia_hip_runtime_api.h`

5.3 Queue Management

BaM [38, 69, 70] uses a wrapper class around atomic values similar to the struct `std::atomic<T>` from the C++ STL [26]. Internally, this wrapper class uses an internal header[2] of the CUDA C++ Core Libraries (CCCL) [54]. This internal header uses inline assembly to directly write atomic operations using Parallel Thread Execution (PTX) [58] instructions. PTX is an intermediate language similar to assembly which is directly translated to the target hardware instruction set. BaM uses this wrapper class to provide exact memory ordering semantics for each atomic operation in the algorithm. Additionally, BaM uses a special MMIO store instruction through PTX inline assembly to write to the doorbell registers. This instruction forces a non-cacheable non-combinable write to the desired memory location, which ensures the GPU writes to the doorbell register when we expect it to do so [58].

Our implementation includes a similar wrapper class and uses the same MMIO store PTX [58] instruction to write to the doorbell registers. We have tried to use standard atomic intrinsics present on both NVIDIA and AMD targets instead. These intrinsics have `std::memory_order_relaxed` [26] semantics, which is the weakest ordering requirement, providing no ordering or synchronization constraints between atomic operations [50]. Therefore, using atomic intrinsics weakens the ordering constraints imposed in BaM's [38, 69, 70] implementation. Using atomic intrinsics, we run into issues where our demonstrator fails to commit commands correctly to the NVMe controller and therefore waits for command completions indefinitely. We suspect that only the special MMIO store instruction is necessary to ensure our demonstrator commits commands correctly, however this requires further testing in future iterations of our software cache.

BaM [38, 69, 70] aligns each atomic value in their virtual queue data structure to a 32-byte boundary. The reasoning behind this decision is that NVIDIA GPUs access memory in 32-, 64-, or 128-byte memory transactions [50]. The padding ensures each atomic can be accessed using an independent memory transaction with the goal of increasing memory throughput despite high contention on the atomics. We omitted to implement this padding in our demonstrator, though we seek to test the performance difference between both implementations in future iterations of our NVMe driver.

We encountered issues with balancing while implementing the dynamic remapping feature of our virtual queue mapper. Initially, we have let SIMD lanes pick the virtual queue with the lowest number of outstanding commands as the victim for remapping. Using this scheme, we find that the system slows down after a

[2] See file `/include/cuda/std/__atomic/functions/cuda_ptx_generated.h`

short amount of time. We assume our dynamic remapping scheme causes our virtual queue map to concentrate around a small subset of virtual queues at any given time. The virtual queue map likely maps many slots at a time onto the same virtual queue until it no longer has the lowest number of outstanding commands. Due to time constraints, we instead let SIMD lanes pseudo-randomly choose a victim using the command ID of the commands the SIMD lanes executed beforehand.

5.4 Software Cache

Implementing the thread coalescing interface for our software cache requires us to know the lane ID of each SIMD lane relative to its thread. We use this value to allow only the leader SIMD lane to acquire or release a cache line. Unfortunately, HIP does not include an intrinsic to retrieve this value. For NVIDIA GPUs, we can retrieve the lane ID using the `%laneid` PTX [58] register. BaM [38, 69, 70] also uses this register for the same purpose. On AMD GPUs, we can use special GCN [3] instructions for the same purpose according to the AMD implementation[3] of the HIP API. However, we have not tested this way of calculating the lane ID on AMD GPUs.

5.5 Discussion

This section discusses the problems we encountered implementing our demonstrator, as well as the parts where our implementation deviates from our design.

A common source of problems was our use of atomics and synchronization. Initially, we tried to use only atomic intrinsics available on both NVIDIA and AMD platforms. However, this resulted in race conditions where our demonstrator would fail to correctly commit commands to the NVMe controller through the doorbell registers. We invested a lot of time into trying to fix these issues, but ultimately decided to use the same PTX [58] inline assembly BaM [38, 69, 70] uses for their atomics and for writing to the doorbell registers.

In its current form, our demonstrator makes use of features only supported by NVIDIA GPUs in four occasions. First, we use GPUDirect RDMA [52] through `libnvm` [40] to allocate peer-accessible DMA buffers on the GPU. Second, we write to the doorbell registers using a special PTX [58] MMIO store instruction. Next, we use PTX inline assembly to achieve specific memory ordering requirements for our atomics. Finally, we use the `%laneid` PTX register to acquire the lane ID of SIMD lanes relative to their thread in our thread coalescing interface.

[3]See file `hipamd/include/hip/amd_detail/amd_warp_functions.h`

To support AMD GPUs, we require GPU-agnostic alternatives or AMD-specific implementations for these features.

Our demonstrator deviates from our design in two parts. First, our demonstrator uses a pseudo-random value calculated from the command ID to dynamically remap virtual queues in our virtual queue mapper. However, according to our design the demonstrator should pick the victim virtual queue based on the number of outstanding commands of each virtual queue. Next, our demonstrator does not allocate physical queues with sizes equal to a power of two, which causes issues when the ticket counter of the associated virtual queue overflows. We omitted to fix this issue, as our benchmarks are short enough to not cause the ticket counter to overflow.

Chapter 6

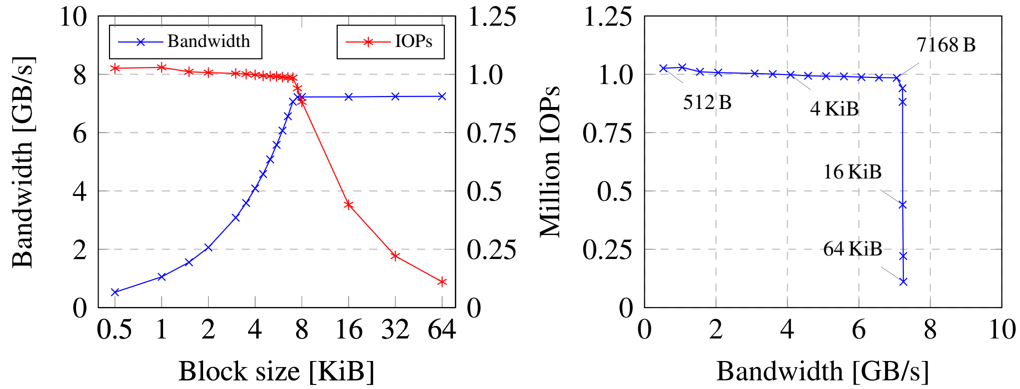
Evaluation

This thesis aims to evaluate the bandwidth and latency of our GPU-based NVMe driver in comparison to conventional CPU-based NVMe drivers and BaM [38, 69, 70]. To this end, we evaluate our NVMe driver in three ways. First, we evaluate raw performance when directly using I/O queues on the GPU without using our general-purpose data structures. Next, we evaluate our driver using only our queue management system without using our software cache, focussing on the added latency and increased noise the system introduces. Finally, we evaluate our full NVMe driver and software cache, focussing on cache hit rates and latency in different conditions and using different working sets. We compare our results to the results presented by Qureshi et al. for BaM and to performance benchmarks on Linux using `fio` [29] and the default I/O stack.

We evaluate our approach on the same system as outlined in Chapter 5. Although we have an AMD GPU installed, our demonstrator only supports NVIDIA in its current form. As such, all GPU benchmarks shown use only the NVIDIA GPU and the Samsung 980 PRO SSD.

6.1 Baseline Performance

To put our demonstrator's performance into perspective, we conducted four baseline performance benchmarks, both on the CPU and the GPU. These benchmarks purely aim to evaluate the raw performance our test system is capable of. As such, the benchmarks do not use any of our data structures on the GPU and bypass the filesystem cache using direct I/O on the CPU. The results of these benchmarks serve as a baseline for further benchmarks using our data structures to evaluate how they affect performance. The CPU benchmarks use `fio` [29], a general purpose I/O tester, while the GPU benchmarks use `libnvm` [40] to directly read from and write to the I/O queues.



(a) Bandwidth and IOPs across varying block sizes during the benchmark.

(b) IOPs plotted against bandwidth during the benchmark.

Figure 6.1: GPU read performance with random reads on a trimmed partition. The GPU achieves the advertised performance of one million IOPs and 7 GB/s bandwidth. Bandwidth increases linearly until reaching a constant 7 GB/s. Conversely, IOPs stay constant until the bandwidth reaches 7 GB/s, linearly decreasing afterward. Block sizes slightly less than 8 KiB show the highest IOPs and bandwidth.

The first benchmark—presented in Figure 6.1—aims to test the limits of the P2PDMA transfer system on our test system. In contrast to all other benchmarks in this thesis, this benchmark uses a fully trimmed partition to achieve the highest possible performance. The other three benchmarks—presented in Figures 6.2, 6.3, and 6.4—as well as all remaining benchmarks use an SSD partition fully filled with random data. This ensures that our results stay consistent across multiple benchmark runs and that they more closely reflect real-world performance.

Performance on a trimmed partition. We test random reads on a 428 GiB partition, spanning around half the available storage on the test SSD. We choose this size as a compromise between preserving another partition already present on the SSD and allocating enough storage with enough unique LBA addresses. Specifically, the 428 GiB partition has around 900 million sectors of 512 B. Accessing the partition with 8 KiB granularity leaves around 56 million unique LBA addresses for our benchmarks.

In contrast to other benchmarks in this thesis, we trimmed the partition beforehand. This ensures we get the best possible performance from of our SSD, such that we measure the limits of the P2PDMA transfer system on our test system.

The benchmark uses multiple blocks with up to 256 SIMD lanes, each filling one I/O queue with an effective queue depth of 369 for multiple iterations. In NVMe SSD benchmarks, queue depth defines how often the process waits for

the command to complete. For instance, a queue depth of 256 means the process waits on every 256th command completion, while a queue depth of one means the process waits on every command. The queue depth of 369 is a result of how we allocate and use the I/O queues. `libnvm` [40] allocates GPU memory in 64 KiB pages for DMA buffers, as GPUDirect RDMA [52] enforces 64 KiB aligned virtual addresses. A 64 KiB page can hold an I/O queue pair with exactly 738 entries. As we cannot submit 738 commands at once, we instead alternate between submitting 737 commands and a single command. As such, we call this an effective queue depth of 369.

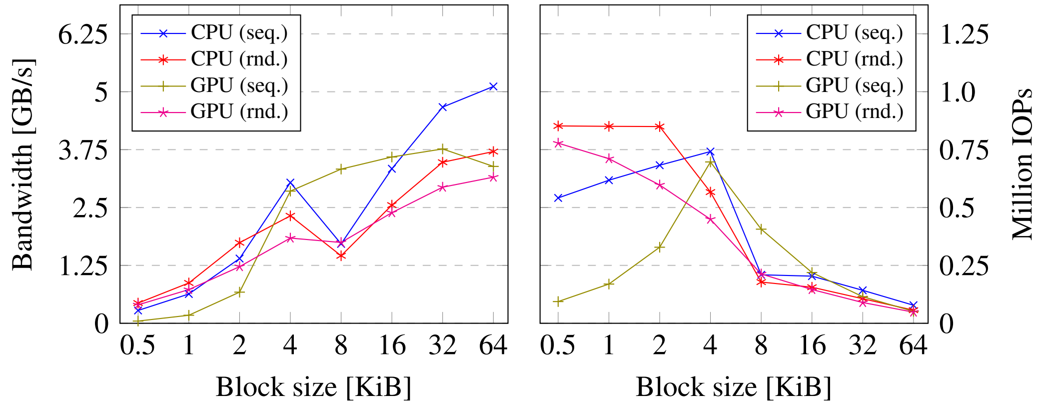
Each SIMD lane reads a fixed size block with every command at a pseudo-randomly generated LBA address. The SIMD lanes read the blocks into an output buffer allocated on the host system. We choose to use host memory for the output buffers, as future use of our NVMe driver in GPU4FS [42] will likely be optimized for direct transfer of user data to or from host memory where the data is needed. Our read and write benchmarks using only our queue management system also use buffers allocated on the host.

To generate the pseudo-random LBA addresses from which the SIMD lanes read, we initially choose a seed value for each SIMD lane individually on the host with `rand()` [43] seeded with the current time. On the GPU, we use a 63-bit linear-feedback shift register (LFSR) [68] to generate additional random numbers when needed. We use this random number generator in all benchmarks presented in this thesis.

We measure both bandwidth in bytes per second and I/O operations per second (IOPs) across varying block sizes from 512 B up to 64 KiB. During each benchmark run, the GPU only uses a fixed block size for every command.

Figure 6.1 shows that the GPU is capable of fully utilizing the advertised performance of the SSD. For smaller block sizes, the GPU fully saturates the advertised one million IOPs possible on our SSD until the bandwidth reaches the advertised 7 GB/s, after which IOPs steadily decrease while maintaining the maximum bandwidth. For block sizes slightly less than 8 KiB, the SSD performs most efficiently, providing both the highest bandwidth and IOPs.

Using 8 KiB blocks has another advantage over other block sizes on our test SSD. It is the largest size representable with only two PRP entries without using additional PRP lists. This is due to our test SSD using 4 KiB pages. Other SSDs may use a different page size or allow configuring the page size. The NVMe command data structure uses two PRP entries to describe the data region. The first always points to the first page of the region, and the second points either to the second page or to a PRP list if the command transfers more than two pages [63]. As we want to avoid allocating a PRP list for every SIMD lane operating on the SSD, we use 8 KiB blocks in future benchmarks evaluating the software cache.



(a) Measured bandwidth during read benchmarks.

(b) Measured IOPs during read benchmarks.

Figure 6.2: Random and sequential read benchmarks both on the GPU and CPU, measuring bandwidth and IOPs. Bandwidth generally increases with larger block sizes, though we see a noticeable drop in performance at 8 KiB blocks on the CPU and at 64 KiB blocks on the GPU during sequential reads. IOPs decrease with larger block sizes during random reads. For sequential reads, however, IOPs first increase until reaching 4 KiB blocks, decreasing afterward.

Read performance. We test both random and sequential reads on the same partition, similarly to the trimmed partition benchmark, measuring both bandwidth in bytes per second and IOPs across varying block sizes from 512 B up to 64 KiB.

For the CPU benchmarks, we configure `fio` to use 64 jobs, a queue depth of 256, and direct I/O to directly measure the performance of the I/O stack without using the filesystem cache on Linux. We use the `libaio` engine to enable asynchronous I/O for parallel execution of I/O commands. As we conducted the CPU benchmarks before developing the GPU benchmarks, we initially chose a queue depth of 256 instead of choosing the same queue depth as the GPU benchmark. Due to time constraints, we choose not to rerun the CPU benchmarks with a different queue depth.

On the GPU we use multiple thread blocks of up to 256 SIMD lanes with each thread block filling an I/O queue with 738 entries in the same way as in the read benchmarks. Each SIMD lane reads either a pseudo-randomly generated LBA address or a sequential LBA address calculated from the SIMD lane's index into the output buffer allocated on the host.

Figure 6.2 shows that the GPU generally performs between 5 % and 20 % worse than the CPU during random reads, with a few exceptions. During sequential reads, we see a larger spread, with the GPU showing between 17 % and 194 % of the CPU's performance. Additionally, both the GPU and CPU fail to

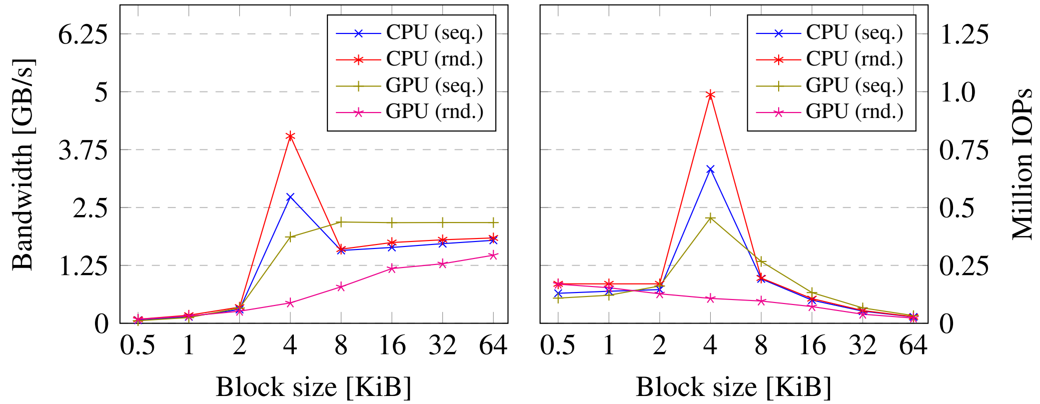
achieve the advertised 7 GB/s read bandwidth and one million IOPs. We assume the slightly worse performance of the GPU benchmark is due to inefficiencies in either our benchmark code or in the P2PDMA transfer mechanism. However, we have not investigated this discrepancy further, as these benchmarks only serve as a baseline for further benchmarks using our queue and cache subsystems. We see a steep drop in performance going from 4 KiB to 8 KiB blocks on the CPU, with the GPU showing higher performance than the CPU. This steep drop in performance may be caused by the host system and the SSD using 4 KiB memory pages. As such, using block sizes larger than 4 KiB incurs a performance penalty as transfers use two memory pages.

The noise in the CPU benchmark is small, with a standard deviation of less than 0.1 % of the mean in both bandwidth and IOPs across all configurations. In contrast, the GPU benchmarks show more noise, with a standard deviation of around 1 % of the mean for block sizes less than 64 KiB. Interestingly, the GPU shows a lot of noise for 64 KiB blocks, with a standard deviation of around 12.8 % of the mean. We have no plausible explanation for this discrepancy other than our output buffers not being aligned to 64 KiB boundaries. However, this is unlikely the cause of this discrepancy as the output buffers lie in host memory and not in GPU memory where GPUDirect RDMA [52] enforces 64 KiB alignment.

Due to time constraints we have not investigated the distribution of bandwidth, IOPs, and latency. As such, all benchmarks in this thesis present noise as the standard deviation of the dataset. We encourage further analysis of the results of our benchmarks in future work.

Write performance. We test both random and sequential writes on the 428 GiB partition similarly to the read benchmarks, measuring both bandwidth in bytes per second and IOPs across varying block sizes from 512 B up to 64 KiB. On the CPU we use `fio` [29] with 64 jobs, a queue depth of 256, and direct I/O. On the GPU we use multiple thread blocks of up to 256 SIMD lanes with each thread block filling an I/O queue with 738 entries. The SIMD lanes on the GPU write data from an input buffer allocated on the host. To preserve the untrimmed state of the SSD partition, we filled the input buffer on the host with pseudo-random data generated using `rand()`.

Figure 6.3 shows that the GPU performs between 20 % and 40 % better than the CPU using block sizes larger than 4 KiB with sequential writes. In contrast, the GPU generally performs between 10 % and 30 % worse than the CPU with random writes. For 4 KiB blocks the CPU massively outperforms all other configurations, reaching the advertised one million IOPs. This spike in performance may be due to the SSD and the host system using 4 KiB pages, leading to optimized transfers between the host and SSD, though this requires further research.



(a) Measured bandwidth during write benchmarks.

(b) Measured IOPs during write benchmarks.

Figure 6.3: Random and sequential write benchmarks both on the GPU and CPU, measuring bandwidth and IOPs. Both benchmarks on the CPU and the sequential write GPU benchmark show a steep increase in both bandwidth and IOPs using 4 KiB blocks, with the CPU outperforming all other configurations at this block size. In contrast, the random write GPU benchmark shows a more gradual increase in bandwidth with increasing block sizes and a gradual decrease in IOPs. The GPU outperforms the CPU in sequential writes at block sizes above 4 KiB, while the CPU shows higher performance than random writes on the GPU.

Still, both the GPU and CPU fail to reach the advertised 5 GB/s write bandwidth. Noise in both bandwidth and IOPs is similar to the read benchmark on the CPU with a standard deviation of less than 0.1 % of the mean, while the GPU, again, shows more noise with a standard deviation of around 3 % of the mean. However, in contrast to the read benchmark, we do not see a spike in standard deviation for 64 KiB blocks, suggesting that this discrepancy only occurs in one transfer direction, going from the SSD to the host. Still, further research is required to fully explain this discrepancy.

Latency. We test random reads and writes on the same partition as the read and write benchmarks. For the CPU benchmark, we use `fio` [29] with a single job and a queue depth of one to measure the latency of single I/O commands. `fio` [29] measures latency as the time from the creation of an I/O command to the completion of the command. As `fio` uses `libaio` internally, the measured latency includes overhead introduced by `libaio` and the Linux I/O stack.

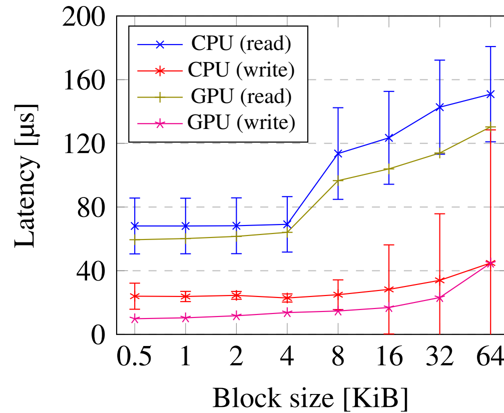


Figure 6.4: Random read and write benchmarks on the GPU and CPU, measuring latency per I/O command across different block sizes. The GPU and CPU follow the same curve for read and write latency, with the GPU showing lower latency than the CPU. Additionally, the CPU shows a lot of noise in latency, especially for larger block sizes during write commands. The error bars on the CPU benchmarks mark one standard deviation from the mean.

The GPU benchmark uses a single SIMD lane, repeatedly submitting a single command and waiting for its completion until it has filled the queue once. As such, the benchmark submits 738 commands in total in a run, which is the size of the I/O queue pair we allocate. We use GPU events with a resolution of 0.5 µs on the host to measure the total time spent executing the GPU kernel. As such, this time includes startup and teardown latency of the GPU kernel [50]. We approximate the latency of each command by dividing the total time by the number of commands executed during the benchmark. Amortizing the latency across multiple commands per run reduces the effects of startup and teardown latency and the low resolution of GPU events. Inevitably, this approach of measuring latency has the undesirable effect of hiding the variance between individual commands.

Figure 6.4 shows that the GPU has slightly lower latency than the CPU in both read and write benchmarks, both showing low latencies around 100 µs, which corresponds to the typical latency of SSDs [66]. We assume the higher latency of the CPU benchmark is due to the overhead introduced by `libaio`, which `fio` uses internally, and the kernel. Additionally, the CPU benchmark shows a lot of noise with standard deviation up to 26 % of the mean in the read benchmark and up to 186 % of the mean in the write benchmark at the largest block size. A possible source of this latency may be the Linux I/O scheduler reordering commands [82]. In contrast, the GPU benchmark has less noise with a standard deviation on the order of 0.1 % of the mean in the read benchmark and up to 6 % of the mean in the

write benchmark at the largest block size. However, the actual noise of the GPU benchmark is likely higher due to our method of measuring latency.

Our comparison between the CPU and GPU is mismatched in two regards. First, `fio` [29] has an advantage as it times individual commands and does not include startup and shutdown latency. In contrast, the GPU has an advantage as it directly uses the I/O queues, while I/O commands from `fio` go through `libaio` and the Linux I/O scheduler before reaching the I/O queues of the SSD. However, these unfair advantages do not detract from the purpose of this benchmark as a baseline for further comparison.

Discussion. The baseline performance benchmarks show that our approach is competitive with CPU-based approaches, with only a slight degradation in performance, possibly due to inefficiencies in our benchmark code. We achieve the maximum bandwidth and IOPs possible on our test SSD with a trimmed partition. BaM [38, 69, 70] achieves similar results, reaching maximum IOPs on their test SSDs, even with multiple SSDs in parallel. However, it is unclear whether Qureshi et al. trimmed the SSDs before their benchmarks.

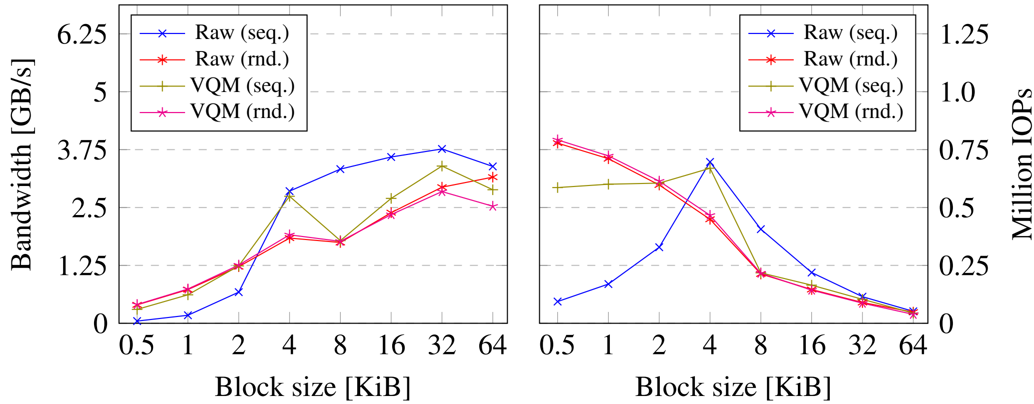
While BaM successfully saturates the available PCIe bandwidth of an `x16` PCIe 4.0 link, our demonstrator is bounded by the limits of our test SSD. Further research using multiple faster SSDs in parallel is necessary to test the limits of the P2PDMA transfer system.

6.2 Queue Management Performance

We evaluate our queue management system similarly to the baseline performance benchmarks to allow for comparison between our system and raw queue management. We focus on the added overhead our system introduces, especially regarding latency as the system uses locks and atomics on global memory. Additionally, we test latency under a synthetic workload using multiple SIMD lanes to see the effects of contention on locks, atomics, and global memory on latency.

The first three benchmarks—presented in Figures 6.5, 6.6, and 6.7—are similar to the baseline performance benchmarks. These benchmarks serve as a comparison between our queue management system and raw queue management. The last benchmark—presented in Figure 6.8—uses a synthetic workload to show how contention on locks and atomics affects latency.

Read performance. We evaluate read bandwidth and IOPs similarly to the baseline performance benchmarks on a 428 GiB partition. We allocate 24 virtual queues and a virtual queue map with 48 entries for this benchmark to allow dynamic remapping of virtual queues.



(a) Measured bandwidth during read benchmarks.

(b) Measured IOPs during read benchmarks.

Figure 6.5: Random and sequential read benchmarks using raw queues and our queue management system, measuring bandwidth and IOPs across different block sizes. The queue management system shows results more in line with the random read benchmarks using raw I/O queues with sequential read bandwidth dropping significantly with block sizes larger than 4 KiB. The measured IOPs follow a similar trend for our queue management system.

We only allocate 24 virtual queues due to a bug in `libnvm` [40] misconfiguring the Admin queue and causing Admin commands to fail unexpectedly. Even though we fixed the bug in `libnvm`, we choose not to revise the benchmarks to use more queues due to time constraints. The underlying physical queues have 738 entries each as we allocate one 64 KiB page per physical queue similarly to the baseline performance benchmarks.

Each SIMD lane uses our virtual queue map to execute multiple I/O commands in a loop. Queue depth does not apply here, as the virtual queue system processes commands and completions in parallel, with each SIMD lane only waiting for its own completion entry. The SIMD lanes read a fixed size block from the SSD either at a pseudo-randomly generated LBA for the random read benchmark or at an LBA calculated from the lane's index for the sequential read benchmark. The SIMD lanes read into an output buffer allocated on the host for similar reasons to the baseline performance benchmarks.

Figure 6.5 shows that our queue management system's performance is similar to the baseline performance, with the peak bandwidth dropping from 3.75 GiB/s to around 3.25 GiB/s. While random reads show results nearly identical to the baseline performance, sequential reads show a drop in performance for larger blocks and an increase in performance for smaller blocks, approaching the performance seen in the random read benchmarks. A possible explanation for this behavior

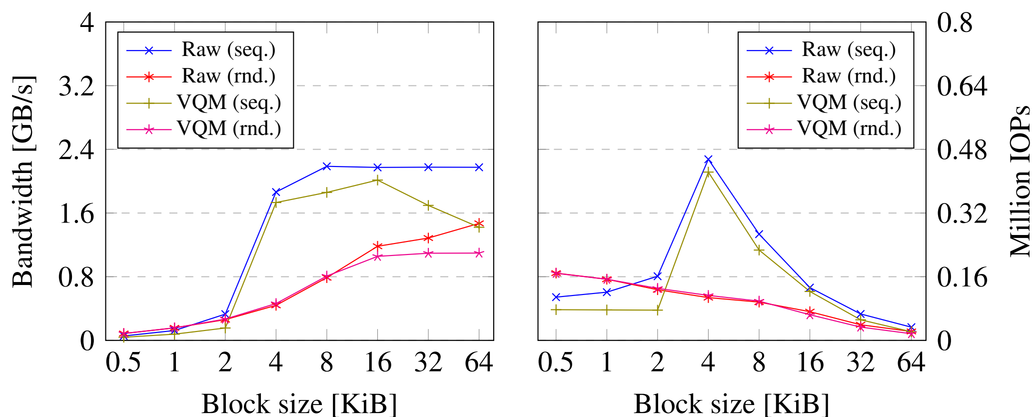
is, that our queue management system introduces noise in the order SIMD lanes commit commands. As our virtual queues use a global ticket counter accessed using an atomic fetch-and-add operation, GPU scheduling and contention on the atomic counter introduces randomness in the order of threads acquiring tickets. As such, the sequential read benchmark shows results similar to the random read benchmark.

Noise using our queue management system is generally lower than during the baseline performance benchmark. The queue management system shows a standard deviation of around 0.1 % of the mean in both bandwidth and IOPs for all block sizes except 64 KiB. For 64 KiB blocks, random reads using our queue management system show a standard deviation of around 18.8 % of the mean and sequential reads around 12.9 %. Similar discrepancies for our queue management system and raw queue management suggest the cause for these discrepancies lies in our hardware configuration and is likely not an effect of our benchmark code. However, further research is required to prove this theory.

Write performance. We evaluate write bandwidth and IOPs similarly to the read benchmarks with 24 virtual queues and 48 entries in the virtual queue map. The SIMD lanes write pseudo-random data from an input buffer on the host to pseudo-random or sequential LBA addresses on the SSD.

Figure 6.6 shows results similar to the read benchmarks. For sequential writes we see a drop in performance for larger blocks, with the peak bandwidth dropping from 2.2 GiB/s to 2 GiB/s, though the bandwidth drops less sharply than in the read benchmark. This suggests, that the introduced randomness of the virtual queues affects writes less than reads. However, it is still unclear why the randomness impacts writes less.

The queue management system shows similar noise to the baseline performance benchmarks for most configurations, with a standard deviation of around 3 % of the mean. In contrast to the baseline performance benchmarks, however, we see a spike in noise again at a large block size. In these benchmarks, the standard deviation spikes to around 17 % of the mean at 32 KiB blocks and drops back down to around 3 % at 64 KiB blocks. As these spikes in noise happen irregularly but always late into the benchmark at high block sizes, we suspect that an issue such as thermal throttling during longer repeated benchmark runs may be the cause of these spikes. Further analysis using longer benchmark runs with additional measurements of SSD and GPU temperature is necessary to confirm this theory.



(a) Measured bandwidth during write benchmarks.

(b) Measured IOPs during write benchmarks.

Figure 6.6: Random and sequential write benchmarks using raw queues and our queue management system, measuring bandwidth and IOPs across multiple block sizes. The queue management system shows a slight decrease in performance, especially during sequential writes. Still, the results generally follow the same curve as the raw I/O queue benchmarks.

Latency. We measure latency compared to the baseline latency benchmark on the same partition as the read and write benchmarks using randomized read and write commands. We use a single SIMD lane to submit 8192 read commands with a pseudo-randomly generated LBA on a single virtual queue. Afterward, we calculate an amortized latency per command by dividing the total runtime of the GPU kernel by the number of commands executed. Similarly to the baseline performance benchmarks, we use GPU events with a resolution of 0.5 µs to measure execution time.

Figure 6.7 shows a constant latency of 290 µs for both read and write commands using our queue management system. This is higher than we expect at around three to thirty times larger than the baseline latency benchmark. Instead, we expect a small increase in latency due to the added latency of atomics and global memory accesses, which both add up to 300 clock cycles or around 175 ns of latency each at our GPU's core frequency according to recent microbenchmarks [1, 30]. Still, the added latency of atomics and global memory does not explain why our system's latency is three times larger than the baseline latency. Note, this benchmark only uses a single SIMD lane and therefore does not introduce contention among atomics or on global memory.

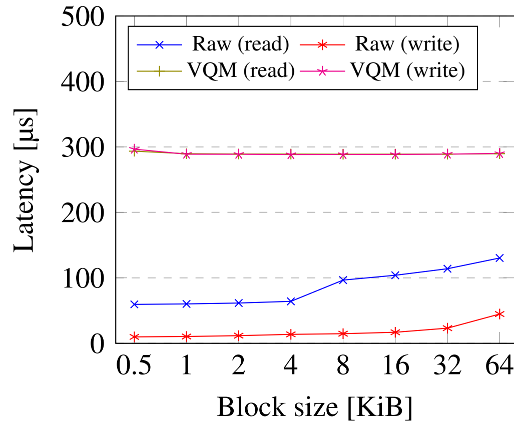


Figure 6.7: Random read and write benchmarks using our queue management system and raw I/O queues, measuring latency across varying block sizes. The queue management system shows a significant increase in latency going up to 290 µs of latency. Both read and write latency are nearly constant and identical across all block sizes.

Another reason for this discrepancy could be the use of the MMIO store instruction through PTX [58] inline assembly. This instruction and its memory consistency guarantees are not necessary in the baseline performance benchmarks, as only one thread writes to the doorbell registers and polls for command completions. However, BaM [38, 69, 70] achieves low latencies around 25 µs using an SSD with a latency around 10 µs using the same PTX instruction in their virtual queues. As such, this still does not fully explain the cause of this discrepancy.

Both read and write benchmarks show similar noise with a standard deviation of around 1 % of the mean for small block sizes, decreasing down to 0.1 % of the mean with growing block sizes.

Latency under load. To further demonstrate the noise introduced by our virtual queues, we evaluate latency under a heavy synthetic workload. We use 8912 SIMD lanes executing either random read or random write commands in a loop to simulate a large I/O workload. Similarly to the read and write benchmarks, we again use 24 virtual queues and a virtual queue map with 48 entries.

Instead of calculating an amortized latency using GPU events, we use an internal GPU register `%globaltimer` to measure the time for each command and SIMD lane individually. The `%globaltimer` register returns a time value in nanoseconds global to the GPU. While the PTX ISA [58] discourages the use of this register, the CCCL [54] libraries use the register as a system clock to implement `std::chrono::system_clock::now` [26]. As such, we choose

to use the internal register similarly to measure the execution time of individual I/O commands. We calculate execution time by taking the difference between the time retrieved from `%globaltimer` before setting up the I/O command and after returning from our queue management system.

This method of measuring latency on the GPU is preferable as it does not include startup and shutdown latency and does not hide noise as when averaging the latency of multiple commands. Due to time constraints, we did not rework previous latency benchmarks to use this method of measuring latency over the method using GPU events.

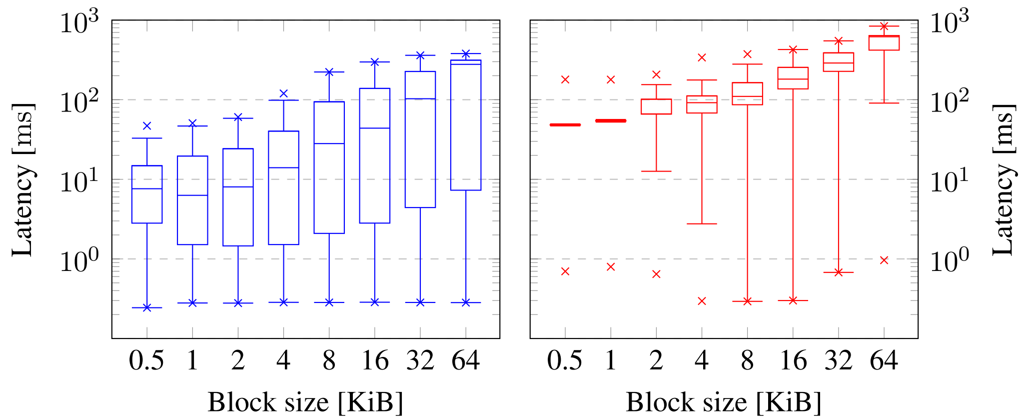
Figures 6.8 and 6.9 show boxplots of the collected datasets. The edges of the boxes denote the upper and lower quartile of the dataset. The whiskers denote the highest or lowest value still inside $1.5 * IQR$ above or below of the relevant quartile, where IQR is the interquartile range, i.e., the interval between the upper and lower quartile. The marks above and below denote the minimum and maximum of the dataset. The line inside the boxes denotes the median of the dataset.

Figure 6.8 shows read latency on the order of 1 ms to 100 ms and write latency on the order of 100 ms to 1 s for the respective interquartile ranges. The read benchmark shows a lot of noise with the interquartile range spanning multiple orders of magnitude. In contrast, the write benchmark shows less noise with the interquartile range contained within one order of magnitude. This likely has the same cause as the effects on bandwidth and IOPs we see between read and write benchmarks, with write benchmarks showing results more close to the baseline performance. Both benchmarks generally show low minimum latency less than 1 ms around the average latency of 290 µs measured in the latency benchmark, with a few exceptions for the write benchmark.

We expect a larger spread for the latency of individual commands under load due to the effects of GPU scheduling and contention on the atomics and global memory. However, the measured spread, median and maximum latencies are larger than anticipated, suggesting that the SIMD lanes experience a form of thread starvation in our implementation. Poor use of atomics and an inefficient memory layout may be the cause for the apparent thread starvation.

6.3 Software Cache Performance

Our evaluation of our software cache focuses on the latency this abstraction adds for different kinds of cache hits and misses, as well as the ratio between cache hits and misses for different cache sizes and under different loads. To this end, we conducted two types of benchmarks.



(a) Latency during read benchmark.

(b) Latency during write benchmark.

Figure 6.8: Random read and write benchmarks using multiple SIMD lanes and our queue management system, measuring latency per I/O operation. Both benchmarks show increasing latency with larger block sizes and a minimum latency of around 200 µs. The read benchmark shows a lot of noise with the boxes spanning multiple orders of magnitude, while the write benchmark shows less noise.

For the first benchmark—presented in Figure 6.9—we measured the latency of the different cache operations using a synthetic workload. This allows us to evaluate the overhead introduced by our software cache in comparison to using only the queue management system under load, as we use the same configuration of SIMD lanes for both benchmarks. Additionally, we can compare the latency of different types of cache hits and misses.

There are two types of cache hits for our software cache. First, a normal cache hit occurs when a SIMD lane successfully acquires a valid cache line already present in the cache. In contrast, a locked cache hit occurs when a SIMD lane finds and acquires a valid cache line only after locking the cache set for eviction. This differs from a normal cache hit in terms of latency, as the SIMD lane generally has to wait for the lock to be released in this case.

We can categorize cache misses into three categories for our software cache. A normal cache miss happens when a SIMD lane evicts a valid and clean cache line after locking the cache set. This differs from a write-back cache miss, for which the evicted cache line has its dirty bit set and the SIMD lane has to write the data to the SSD. Lastly, a cold cache miss happens when the cache line chosen for eviction does not have its Valid-bit set. This is only the case, when the cache line is cold, i.e., the cache line has never had data stored in it before. This type differs from the other types in latency, as this case does not require multiple rounds of our clock replacement algorithm.

The second benchmark—presented in Figures 6.10 and 6.11—uses an idea adapted from Traeger et al. [90] using traces of NVMe commands captured using `ftrace` [76] on Linux. We stripped the traces of commands not yet supported by our software cache, such as the flush command, which flushes the volatile write cache present on many SSDs to non-volatile memory. By replaying these traces on the GPU using our software cache, we can evaluate the cache hit rates across different cache configurations.

Latency evaluation. We measure latency under load using 8192 SIMD lanes repeatedly acquiring and releasing pseudo-random cache lines. Each SIMD lane decides whether to dirty the cache line based on a pseudo-random value. To measure the latency of each individual acquire operation, we use the internal `%globaltimer` register and calculate the difference in its value captured before and after the acquire operation. Then, we store the measured time value based on the type of cache hit or miss reported by our cache API.

We use an 8 GiB large cache split into 4096 sets of 256 cache lines with a cache line size of 8 KiB. We use 8 KiB cache lines as it is the largest size on our test SSD that does not require additional PRP lists. Furthermore, we choose an 8 GiB large cache as the largest cache size that still fits in GPU memory alongside the latency measurements. Then, we decided to split this cache into sets of 256 cache lines as a multiple of the SIMD thread size of 32 SIMD lanes.

Figure 6.9 shows that the normal cache hit latency is multiple orders of magnitude lower than the latency of locked hits or cache misses, with an average latency of around 1 ms and going as low as 2 µs. For all types of misses, the interquartile range lies between 10 ms and 100 ms of latency, with minimum latencies around 1 ms. Cold misses incur the lowest latency, as they do not include the latency of the clock replacement algorithm and the second I/O command for write-back misses. Normal misses and write-back misses show similar latency, with write-back latency being slightly higher as it necessitates a second I/O command.

Interestingly, locked hits incur the highest latency of all types with a median latency above 100 ms and a minimum latency of around 5 ms, though this type of cache hit only happens in an edge case. If a SIMD lane finds the correct valid cache line only after locking the cache set for eviction, one of the previous SIMD lanes that held the lock must have evicted a cache line from the set and loaded the correct block in the time the SIMD lane was waiting for the lock to be released. As such, this type of cache hit contains both the latency from the cache miss that happened before and the latency from waiting for the lock to be released. For all type of cache hits and misses, the maximum latency is high, going up to 10 s for locked hits and misses.

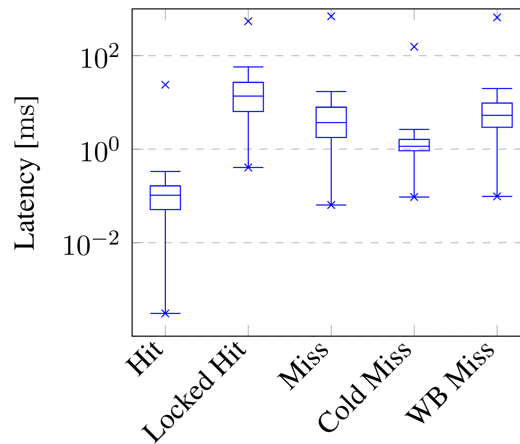


Figure 6.9: Cache latency benchmark using multiple SIMD lanes during different types of cache hits and misses. A normal hit shows the lowest latency around 1 ms with a minimum of 2 µs. The different types of misses all fall around 10 ms to 100 ms and a locked hit shows the largest latency of around 100 ms to 500 ms. All types show large maximum values up to 10 s of latency.

The software cache amplifies the effects of GPU scheduling, lock contention, and contention on atomics seen in Figure 6.8. We see maximum latencies of more than one second for acquiring a single cache line even though the NVMe command itself only takes around 100 µs. We suggest implementing some form of realtime constraints in further iterations of our NVMe driver and software cache to mitigate these effects and reduce the maximum latency under load.

Replaying captured I/O traces. We measure cache hit ratios using traces of NVMe commands captured beforehand. Similarly to previous benchmarks, we use 8192 SIMD lanes to replay the commands on our demonstrator. We use a global atomic counter to step through the captured traces, with each thread of SIMD lanes atomically fetching and increasing the counter to acquire a unique command from the trace. This sacrifices some accuracy of the replay for increased throughput, as SIMD threads may execute commands from the trace in parallel and out of order. Then, each SIMD thread acquires the relevant cache lines and simulates work on them by calculating the XOR between the data and a pseudo-random key. If the replayed command is a write command, the SIMD thread marks the cache line as dirty when acquiring the cache line. We use atomic counters for the different kinds of cache hits and misses to accurately calculate the ratio between cache hits and misses. We evaluate different cache configurations with cache sizes ranging from 512 MiB up to 8 GiB using 4 KiB and 8 KiB cache line sizes.

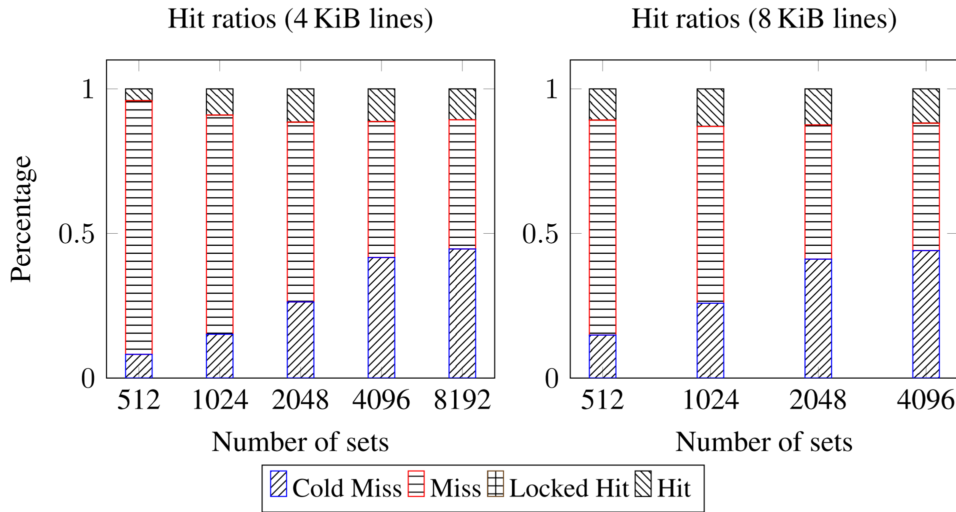


Figure 6.10: Cache hit and miss ratios across varying cache sizes replaying a trace of compiling a freestanding `gcc` [22] C/C++ toolchain. The ratio of cache hits increases up to 2048 or 1024 cache sets at which point it stays constant. Cold cache misses increase with larger caches and locked hits are rare making up less than 1 % of cache operations.

We captured two traces using `ftrace` [76] on Linux. First, we captured a trace while compiling a freestanding `gcc` [22] C/C++ toolchain on a newly formatted Ext4 [79] partition on our test system. This trace emulates a workload mostly composed of write commands. Second, we captured a trace while running a system update of an Arch Linux [31] installation on a laptop used for personal use formatted with a Btrfs [78] partition. This trace captures a more balanced workload comprised of both read and write commands.

For the `gcc` compilation trace, Figure 6.10 shows larger hit rates with larger cache sizes up to 2 GiB, afterward hit rates stay constant. This is expected, as the trace accesses only 2153 LBA addresses of the 48865 unique addresses multiple times. Cache hit and miss rates show similar values for the same cache sizes, independent of cache line size.

For the system update trace, Figure 6.11 shows larger hit rates with larger cache sizes, similarly to Figure 6.10. At the largest cache size, normal cache hits make up 33 % of all cache operations. As opposed to the `gcc` compilation trace, cache hits do not stay constant at some cache size and instead keep increasing. Still, we see diminishing returns when doubling the cache size from 4 GiB to 8 GiB only yields a 5 % increase in cache hits. This is expected with only a limited number of commands in the trace because any working set has a maximum cache hit rate, as each accessed cache line incurs at least one cache miss on a cold cache.

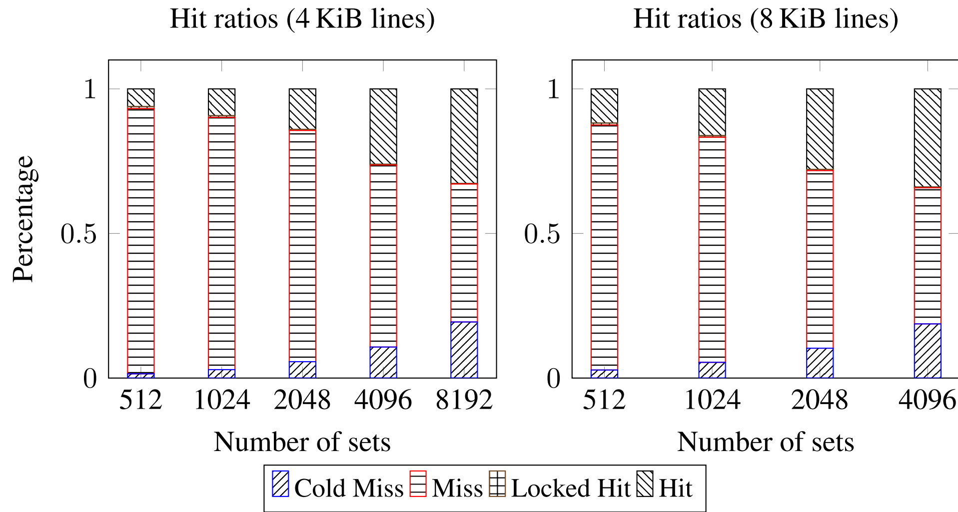


Figure 6.11: Cache hit and miss ratio across varying cache sizes replaying a trace of a system update on a laptop used for personal use. The cache hit ratio and cold cache miss ratio increases with larger block sizes, while locked hits make up less than 1 % of cache operations.

Specifically, this trace contains 336789 unique starting LBA addresses of which 69038 appear multiple times and a total of around 503022 commands. As such, the theoretical maximum cache hit rate is also approximately 33 %. This is only an approximation, as the calculation does not include the number of blocks read per command and aligning the starting LBA to a cache line. Both benchmarks show that locked hits make up less than 1 % of the total number of cache operations, suggesting that this edge case is uncommon even under heavy load.

While the benchmarks give some insight into the efficiency of our software cache, the traces used are too short for the chosen cache sizes. In both benchmarks, we hit the theoretical maximum cache hit rate, suggesting the software caches evaluated are too large to sufficiently evaluate the cache hit ratio. Additionally, we captured the traces on CPU-based filesystems such as Ext4 [79] and Btrfs [78]. We suggest reevaluating our software cache using longer traces captured on a GPU4FS [42] partition after integrating our NVMe driver into GPU4FS.

We cannot compare our results directly to BaM's [38, 69, 70] software cache, as Qureshi et al. use different workloads for their evaluation. Depending on the workload, BaM achieves cache hit ratios above 90 % with a 64 GiB large software cache. However, we see that our results roughly follow the same patterns, with the cache hit ratio increasing more steeply at lower cache sizes and flattening out towards some limiting cache hit ratio for large cache sizes.

6.4 Discussion

This section discusses our results in a broader context, providing more detailed explanations of the results and details on theoretical limits, specifically for our software cache. We mainly compare our results to BaM [38, 69, 70] where applicable, as BaM serves as a basis for our NVMe driver design.

Our queue management system shows an unexpectedly high increase in latency, up to 290 µs. It is possible our demonstrator uses atomics poorly, which introduces a lot of contention on the atomics and global memory. The most likely culprit is not using padding to pad our atomics to 128 B boundaries. As NVIDIA GPUs process memory in 32 B, 64 B, and 128 B transactions, having multiple atomics tightly packed amplifies contention on global memory [49]. Accessing multiple atomic simultaneously in a 128 B memory region likely generates multiple transactions to global memory for each group of successfully executed atomic instructions. High contention on the atomics amplifies this effect, as SIMD lanes might have to retry the atomic instruction multiple times. This explains the apparent thread starvation we see using a synthetic workload, as unlucky SIMD lanes may take a long time to successfully execute an atomic instruction under contention. However, this is only a guess and further research into this possibility is required.

The software cache seems to amplify these effects, showing higher latency under load than with only our queue management system. It adds another layer of atomics and locks, which likely increases the latency for the same reasons as with the queue management system. A notable difference, however, is the presumably high contention on the locks for each cache set, higher than the contention on the locks used in our queue management system. While our queue management system uses locks for small resources such as single command IDs or single slots in the physical queue, the software cache uses a lock for a complete cache set and for a large section of the evict-and-replace algorithm. Our demonstrator currently uses a simple but likely inefficient spinlock implementation for all locks in our software cache and queue management system. These spinlocks likely cause thread starvation, especially in our software cache implementation. Replacing these locks with fair locks where possible acts as a form of soft realtime constraint, as the latency of these locks is bounded by the number of SIMD lanes.

Our software cache design differs greatly from BaM's software cache, especially in its memory footprint. BaM [38, 69, 70] incurs a constant overhead for the amount of storage accessible as its software cache uses a fixed size array of cache line metadata preallocated for the entirety of the accessible storage media. For instance, a 1 TiB SSD split into 4 KiB cache lines would require 2 GiB of memory for the cache line metadata. In contrast, our approach only stores cache line metadata for cache lines present in the cache, thereby reducing the memory

footprint used for an 8 GiB cache with 4 KiB cache lines to around 32 MiB. Still, this change comes at the cost of performance as our approach now requires a linear search to find the correct cache line, while BaM achieves constant time lookups. We mitigate this issue by using a set-associative cache instead of a fully-associative cache, reducing the linear search to a constant factor k. BaM's software cache would introduce a large memory overhead with large capacity enterprise-grade SSDs and software RAID, rendering it unusable for our usecase. As such, we see the tradeoff in lookup performance for a smaller memory footprint as beneficial.

Even though our software cache is not fully-associative, we achieve the theoretical maximum cache hit rates of around 33 % replaying our traces. In theory, any set-associative cache can simulate a smaller fully-associative cache. Take for instance an 8 GiB cache split into 4 KiB cache lines with 8192 cache sets of 256 cache lines. The cache can store a total of 2^{21} or around two million cache lines. From the theorem presented by Adas et al. [2] it follows that the cache can store around 1.5 million cache lines with a probability of at least 50 % without requiring to evict a cache line. Additionally, the cache can store one million cache lines with a probability greater than 99 %. However, the theorem assumes cache lines stored in the cache are fully independent, which is unrealistic in real workloads.

In conclusion, our demonstrator successfully proves the feasibility of a general-purpose NVMe driver fully running on the GPU, providing both bandwidth and IOPs comparable to CPU-based approaches. However, the demonstrator has issues, especially regarding latency, which require further research and development. In its current form, we deem the demonstrator unfit for usage under heavy load, as unlucky applications may have to wait multiple seconds for individual filesystem commands to complete due to the thread starvation issues outlined above.

Chapter 7

Future Work

The results of this thesis show different areas where our approach needs improvement and leaves some open questions for further research:

Integrate our driver into GPU4FS. The goal of this thesis is to design and develop an NVMe driver and software cache suitable for use in GPU4FS [42]. Integrating our demonstrator into GPU4FS is the next step in this process and enables more sophisticated filesystem benchmarks for our NVMe driver. GPU4FS already provides a layer of abstraction under which we can integrate our demonstrator. This abstraction layer includes wrapper functions to transfer data to and from a storage medium using an opaque storage base pointer. We can point this storage base pointer to a wrapper around our NVMe driver to translate the read and write operations into cache operations.

Port our driver to AMD GPUs. In its current form, our demonstrator only supports NVIDIA GPUs. Most notably, it is necessary to port our use of PTX [58] inline assembly to either Graphics Core Next (GCN) [3] inline assembly for AMD or generic atomic intrinsics usable on both platforms.

Additionally, our demonstrator currently uses a special MMIO store instruction using PTX [58] inline assembly to write to the doorbell registers. It is necessary to port this instruction to AMD using either a comparable GCN [3] instruction or a combination of atomic intrinsics and memory fences if it is possible to achieve similar semantics in this way.

Finally, `libnvm` [40] currently only supports NVIDIA GPUs through GPUDirect RDMA [52] to map GPU memory for P2PDMA. We suggest testing out the undocumented API of the `include/drm/amd_rdma.h` header in the AMDgpu [12] kernel driver we discussed in Section 2.3.5 to see if it provides similar capabilities to GPUDirect RDMA. If this fails, the P2PDMA API [83] on Linux could

work as an alternative, but it would require access to the `pci_dev` structure associated with the AMD GPU, which may require modifying the ADMgpu kernel driver.

Investigate the high latency of our approach. Our evaluation shows that our queue management and software cache adds a large amount of latency to I/O operations, especially if the system is under heavy load. Further research is necessary to find and mitigate the source of the added latency.

We suspect our use of atomics producing inefficient memory transactions on global memory to be a major source of latency for our demonstrator. Optimizing the memory layout of atomics by adding padding as it is used in BaM [38, 69, 70] likely improves the latency, as it ensures operations on separate atomics use separate memory transactions.

Reducing the latency of single I/O operations with a more optimized memory layout likely also reduces the average latency under load. Still, we suggest extending or redesigning parts of our NVMe driver and software cache to implement soft realtime constraints on latency as to reduce the likelihood of I/O commands taking a large amount of time to complete.

The locks our demonstrator uses are an area in which we can introduce realtime constraints. In its current form, our demonstrator only uses simple spinlocks using an atomic fetch-and-or operation. Spinlocks do not provide fairness, as any SIMD lane may have to wait arbitrarily long to acquire a lock. Using fair locks instead, such as a ticket-based lock, would guarantee that SIMD lanes acquire a lock in the order they arrive at the lock operation. This serves as an upper bound for the latency added by a lock, proportional to the number of SIMD lanes.

Find the bandwidth limit using this approach. The SSD we use for benchmarks limits the bandwidth and IOPs we can achieve. As the read benchmark on the trimmed partition in Figure 6.1 shows, we are capable of reaching the advertised bandwidth and IOPs with our approach.

This limit is less than the maximum theoretical bandwidth achievable on a x4 PCIe 4.0 link and less than the results Qureshi et al. [38, 69, 70] present for BaM. Further research could evaluate the theoretical limits of our approach on modern GPUs using more modern SSDs with support for PCIe 5.0 and multiple SSDs simultaneously until the SSDs are no longer the bottleneck.

Additionally, research on the bandwidth of our software cache design is necessary, especially for different types of cache hits and misses and under different workloads. Our benchmarks only evaluate latency and cache hit rates for our software cache. We suggest conducting additional microbenchmarks on latency and bandwidth, as well as more long term benchmarks on cache hit rates.

Extend our NVMe driver. Currently, our demonstrator only makes use of basic NVMe features mandatory for all SSDs. Supporting and using SGLs enables larger block sizes for our software cache without additional PRP lists allocated per SIMD lane. However, as SGLs are intended for the message-based transport model, SGL support is rare for the PCIe-based transport model, with only a few manufacturers advertising the feature for their datacenter SSDs [63, 89, 91]. Next, we can use the weighted round-robin arbitration mechanism to prioritize metadata and journal updates, for example. Furthermore, our demonstrator only uses basic NVMe read and write commands. We suggest using more specialized commands for better performance where applicable. Most notably, the trim command can be used on file deletion to efficiently free up the blocks used by the file.

Furthermore, our demonstrator requires better memory management. Currently, our demonstrator wastes memory when allocating buffers for the physical I/O queues, as we allocate memory for each queue individually in 64 KiB chunks. Preallocating all memory necessary for the I/O queues in one call allows us to more tightly pack the I/O queues in memory.

Extend our software cache. The current design of our software cache only supports a single NVMe namespace and only GPU memory. This prevents us from implementing more advanced filesystem features such as software RAID and `mmap`. Extending our software cache design with more metadata in the cache line tag or integrating our cache into a multi-level cache hierarchy is necessary to support these features. For instance, including the NVMe device identifier and the namespace identifier in the cache line tag allows us to store blocks from multiple devices in the same cache. Furthermore, including residency information on where the data is stored in memory allows the cache to store data in both host memory and GPU memory depending on its usage.

Chapter 8

Conclusion

In this thesis, we aimed to test the feasibility of an NVMe driver and software cache fully implemented on the GPU, with the goal of integrating our design into GPU4FS [42] in future work. Currently, GPU4FS only supports byte-addressable NVM storage with Intel Optane [25]. Integrating our NVMe driver and software cache into GPU4FS enables the use of common block-addressable storage devices using the NVMe protocol.

In our work, we designed an NVMe driver and software cache with the parallelism of modern GPUs in mind, taking BaM's [38, 69, 70] design as the basis for our design. We implemented a demonstrator based on our design and evaluated our NVMe driver using microbenchmarks, measuring bandwidth, IOPs, and latency. We evaluated our software cache with synthetic workloads and traces of NVMe commands captured during real-world workloads.

Our results show that our demonstrator is competitive with CPU-based NVMe drivers. We achieve between 70 % and 90 % of the bandwidth of the Linux I/O stack using random reads and between 60 % and 90 % of the bandwidth using random writes for most configurations. However, we see that the CPU-based approach massively outperforms our approach using 4 KiB random writes, with our approach only achieving 11.4 % of the bandwidth. Furthermore, our approach introduces an unusually large amount of latency of up to 290 µs in microbenchmarks and between 1 s and 10 s using a synthetic workload.

We conclude that our demonstrator is competitive with CPU-based approaches regarding bandwidth and IOPs, yet requires further optimization to reduce the latency of individual commands. The abnormally high latency makes our demonstrator unusable for high performance applications, but it still successfully demonstrates that GPU-based NVMe drivers are a viable alternative to traditional CPU-based approaches.

Our work presents possibilities for future work into GPU-based NVMe drivers. We propose further research into lowering the latency of our approach, as well as

porting our demonstrator to AMD GPUs. Additionally, we suggest extending our demonstrator with support for more advanced NVMe features and with support for both GPU and host memory in our software cache. Lastly, we recommend running additional benchmarks on our demonstrator with multiple faster SSDs to evaluate the limits of our approach.

Bibliography

[1] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed A. Badawy. Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, Sep. 2022.

[2] Dolev Adas, Gil Einziger, and Roy Friedman. Limited associativity makes concurrent software caches a breeze. In *Proceedings of the 23rd International Conference on Distributed Computing and Networking*, ICDCN '22, page 87–96, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3491003.3491013.

[3] Advanced Micro Devices, Inc. Graphics Core Next Architecture, Generation 3, August 2016. https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/gcn3-instruction-set-architecture.pdf. (retrieved on 08/18/2024).

[4] Advanced Micro Devices, Inc. AMD EPYC™ 9004 SERIES PROCESSORS, 2022-2023. https://www.amd.com/system/files/documents/epyc-9004-series-processors-data-sheet.pdf. (retrieved on 06/16/2024).

[5] Advanced Micro Devices, Inc. Hipcc documentation, 2024. https://rocm.docs.amd.com/projects/HIPCC/en/latest/. (visited on 07/14/2024).

[6] Advanced Micro Devices, Inc. Non-AMD specific back-end implementation for HIP — v6.1.2, 2024. https://github.com/ROCm/hipother/tree/rocm-6.1.2. (visited on 07/16/2024).

[7] Advanced Micro Devices, Inc. Radeon™ RX 6950 XT Desktop Graphics Card, 2024. https://www.amd.com/en/products/graphics/

desktops/radeon/6000-series/amd-radeon-rx-6950-xt.html. (visited on 06/16/2024).

[8] Akber Kazmi. PCI Express™ Basics & Applications in Communication Systems, 2004. https://www.cs.uml.edu/~bill/cs520/slides_15C_PCI_Express.pdf. (retrieved on 04/06/2024).

[9] AMD. Introducing RDNA Architecture, 2019. https://www.amd.com/system/files/documents/rdna-whitepaper.pdf. (retrieved on 03/21/2024).

[10] AMD. AMD I/O Virtualization Technology (IOMMU) Specification Revision 3.09-PUB (48882), October 2023. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/specifications/48882_IOMMU.pdf. (retrieved on 04/04/2024).

[11] AMD. AMD Radeon™ RX 7900 XTX, 2024. https://www.amd.com/en/products/graphics/amd-radeon-rx-7900xtx. (visited on 04/03/2024).

[12] AMD. drm/amdgpu AMDgpu driver, 2024. https://docs.kernel.org/gpu/amdgpu/index.html. (visited on 04/03/2024).

[13] AMD. HIP Documentation, 2024. https://rocm.docs.amd.com/projects/HIP/en/latest/index.html. (visited on 04/02/2024).

[14] AMD. ROCK-Kernel-Driver version 6.0.x source code, 2024. https://github.com/ROCm/ROCK-Kernel-Driver/tree/roc-6.0.x. (visited on 04/03/2024).

[15] AMD. Tech Brief: AMD FirePro™ SDI-Link and AMD DirectGMA Technology, (n.d.). https://www.amd.com/system/files/documents/sdi-tech-brief.pdf. (retrieved on 04/03/2024).

[16] Apple Inc. Metal overview, 2024. https://developer.apple.com/metal/. (visited on 05/03/2024).

[17] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. *ACM Trans. Comput. Syst.*, 36(2), April 2019. https://doi.org/10.1145/3309987.

[18] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 83–98, New York, NY, USA, 2016. Association for Computing Machinery. `https://doi.org/10.1145/2872362.2872406`.

[19] Fernando J Corbato. *A Paging Experiment with the Multics System*. Massachusetts Institute of Technology, 1968.

[20] Docker Inc. Docker: Accelerated Container Application Development, 2024. `https://www.docker.com/`. (visited on 07/14/2024).

[21] Frank Shu, Nathan Obr. Data Set Management Commands Proposal for ATA8-ACS2, December 12 2007. `https://web.archive.org/web/20100613085651if_/http://t13.org:80/Documents/UploadedDocuments/docs2008/e07154r6-Data_Set_Management_Proposal_for_ATA-ACS2.doc`. (retrieved on 07/17/2024).

[22] Free Software Foundation, Inc. GCC, the GNU Compiler Collection, 2024. `https://gcc.gnu.org/`. (visited on 08/16/2024).

[23] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[24] IEEE and The Open Group. The Open Group Base Specifications Issue 7, 2018 edition, 2018. `https://pubs.opengroup.org/onlinepubs/9699919799/`. (visited on 04/05/2024).

[25] Intel Corporation. Intel® Optane™ Memory — Responsive Memory, Accelerated Performance, 2024. `https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html`. (visited on 08/30/2024).

[26] International Organization for Standardization. Programming Languages — C++, March 31 2020. `https://isocpp.org/files/papers/N4860.pdf`. (retrieved on 07/14/2024).

[27] iXsystems, Inc. ZFS Deduplication, 2024. `https://www.truenas.com/docs/references/zfsdeduplication/`. (visited on 04/06/2024).

[28] Jean-Philippe Brucker. Shared Virtual Addressing for the IOMMU, 2024. https://lwn.net/Articles/747230. (visited on 07/26/2024).

[29] Jens Axboe. fio — Flexible IO Tester, 2024. https://git.kernel.dk/cgit/fio/. (visited on 06/16/2024).

[30] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking, 2018. https://arxiv.org/abs/1804.06826.

[31] Judd Vinet, Aaron Griffin and Levente Polyák. Arch Linux, 2024. https://archlinux.org/. (visited on 08/18/2024).

[32] Kingston Technology Europe Co LLP and Kingston Digital Europe Co LLP. DDR5 4800MT/s ECC Registered DIMM, 2024. https://www.kingston.com/en/memory/server-premier/ddr5-4800mts-ecc-registered-dimm. (visited on 06/20/2024).

[33] Lennard Kittner. Directories for GPU4FS, April 04 2023. https://os.itec.kit.edu/deutsch/97_3912.php.

[34] Steve Lantz. GPU Characteristics, May 2023. https://cvw.cac.cornell.edu/gpu-architecture/gpu-characteristics/index. (visited on 03/21/2024).

[35] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[36] LLVM Team. Clang: a C language family frontend for LLVM, 2024. https://clang.llvm.org/. (visited on 07/14/2024).

[37] Gregor Lucka. RAID on a File System Level for GPU4FS, September 18 2023. https://os.itec.kit.edu/deutsch/97_3931.php.

[38] Vikram Sharma Mailthody. *Application Support And Adaptation For High-throughput Accelerator Orchestrated Fine-grain Storage Access*. PhD thesis, University of Illinois Urbana-Champaign, 2022.

[39] Kevin Marks. An NVM Express Tutorial, 2013. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130812_PreConfD_Marks.pdf. (retrieved on 03/22/2024).

[40] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. SmartIO: Zero-Overhead Device Sharing through PCIe Networking. *ACM Transactions on Computer Systems*, 38(1–2), July 2021. https://doi.org/10.1145/3462545.

[41] Peter Maucher. GPU4FS: A Graphics Processor-Accelerated File System, August 06 2022. https://os.itec.kit.edu/deutsch/97_3851.php.

[42] Peter Maucher, Lennard Kittner, Nico Rath, Gregor Lucka, Lukas Werling, Yussuf Khalil, Thorsten Gröninger, and Frank Bellosa. Full-Scale File System Acceleration on GPU. Tagungsband des FG-BS Frühjahrstreffens 2024, 2024.

[43] Michael Kerrisk. rand(3) — Linux manual page, 2024. https://www.man7.org/linux/man-pages/man3/rand.3.html. (visited on 08/21/2024).

[44] Michael Kerrisk. sysfs — a filesystem for exporting kernel objects, 2024. https://www.man7.org/linux/man-pages/man5/sysfs.5.html. (visited on 07/14/2024).

[45] Microsoft. DirectX graphics and gaming, September 22 2022. https://learn.microsoft.com/en-us/windows/win32/directx. (visited on 04/02/2024).

[46] Microsoft. DirectX Shader Compiler, 2024. https://github.com/microsoft/DirectXShaderCompiler. (visited on 07/26/2024).

[47] Microsoft. High-Level-Shaderprogrammiersprache (HLSL), 2024. https://learn.microsoft.com/de-de/windows/win32/direct3dhlsl/dx-graphics-hlsl. (visited on 07/26/2024).

[48] NVIDIA. NVIDIA Tesla P100, 2016. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. (retrieved on 03/21/2024).

[49] NVIDIA. CUDA Programming Guide, March 07 2024. https://docs.nvidia.com/cuda/cuda-c-programming-guide. (visited on 04/02/2024).

[50] NVIDIA. CUDA Runtime API — API Reference Manual, April 2024. https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf. (retrieved on 04/04/2024).

[51] NVIDIA. CUDA zone, 2024. https://developer.nvidia.com/cuda-zone. (visited on 04/02/2024).

[52] NVIDIA. Developing a Linux Kernel Module using GPUDirect RDMA, March 2024. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html. (visited on 04/03/2024).

[53] NVIDIA. NVIDIA GPUDirect, 2024. https://developer.nvidia.com/gpudirect. (visited on 04/03/2024).

[54] NVIDIA Corporation. CUDA C++ Core Libraries (CCCL) — v2.5.0, 2024. https://github.com/NVIDIA/cccl/tree/v2.5.0. (visited on 07/14/2024).

[55] NVIDIA Corporation. NVIDIA CUDA Compiler Driver, June 20 2024. https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf. (retrieved on 07/14/2024).

[56] NVIDIA Corporation. NVIDIA Linux Open GPU Kernel Module Source — Version 555.42.02 , 2024. https://github.com/NVIDIA/open-gpu-kernel-modules/tree/555.42.02. (visited on 08/29/2024).

[57] NVIDIA Corporation. Overview — NVIDIA Container Toolkit 1.15.0 documentation, April 26 2024. https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/index.html. (visited on 07/14/2024).

[58] NVIDIA Corporation. PTX ISA — Release 8.5, June 20 2024. https://docs.nvidia.com/cuda/pdf/ptx_isa_8.5.pdf. (retrieved on 07/14/2024).

[59] NVIDIA Corporation and Affiliates. NVIDIA RTX A4500, 2022. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/rtx/nvidia-rtx-a4500-datasheet.pdf. (retrieved on 06/16/2024).

[60] NVM Express, Inc. NVM Express® NVM Command Set Specification, Revision 1.0d, December 28 2023. https://nvmexpress.org/wp-content/uploads/

NVM-Express-NVM-Command-Set-Specification-1.0d-2023.12.28-Ratified.pdf. (retrieved on 03/19/2024).

[61] NVM Express, Inc. NVM Express® RDMA Transport Specification, Revision 1.0c, December 27 2023. https://nvmexpress.org/wp-content/uploads/NVM-Express-RDMA-Transport-Specification-1.0c-2023.12.27-Ratified.pdf. (retrieved on 03/19/2024).

[62] NVM Express, Inc. NVMe® over PCIe® Transport Specification, Revision 1.0d, December 27 2023. file:///home/nick/Downloads/NVM-Express-PCIe-Transport-Specification-1.0d-2023.12.27-Ratified.pdf. (retrieved on 03/19/2024).

[63] NVM Express, Inc. NVM Express® Base Specification, Revision 2.0d, January 11 2024. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0d-2024.01.11-Ratified.pdf. (retrieved on 03/19/2024).

[64] Terzo Olivier, Djemame Karim, Scionti Alberto, and Pezuela Clara. *Heterogeneous Computing Architectures : Challenges and Vision.* CRC Press, 2019. http://www.redi-bw.de/db/ebsco.php/search.ebscohost.com/login.aspx%3fdirect%3dtrue%26db%3dnlebk%26AN%3d2141414%26site%3dehost-live.

[65] OpenGL Wiki contributors. Core Language (GLSL), 2024. https://www.khronos.org/opengl/wiki/Core_Language_(GLSL). (visited on 07/26/2024).

[66] Ismail Oukid and Lucas Lersch. On the Diversity of Memory and Storage Technologies. *Datenbank-Spektrum*, 18(2):121–127, July 01 2018. https://doi.org/10.1007/s13222-018-0287-8.

[67] PCI-SIG®. PCI Express® Base Specification Revision 4.0 Version 0.3, February 19 2014. https://astralvx.com/storage/2020/11/PCI_Express_Base_4.0_Rev0.3_February19-2014.pdf. (retrieved on 03/23/2024).

[68] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press, September 2007.

[69] Zaid Qureshi. *Infrastructure to Enable and Exploit GPU Orchestrated High-Throughput Storage Access on GPUs*. PhD thesis, University of Illinois Urbana-Champaign, 2022.

[70] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 325–339, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3575693.3575748.

[71] Nico Rath. Extending GPU4FS with Advanced File System Functionalities, August 29 2023. https://os.itec.kit.edu/deutsch/97_3929.php.

[72] Samsung Electronics Co., Ltd. Samsung V-NAND SSD 980, 2021. https://download.semiconductor.samsung.com/resources/data-sheet/Samsung_NVMe_SSD_980_Data_Sheet_Rev.1.1.pdf. (retrieved on 06/16/2024).

[73] Samsung Electronics Co., Ltd. Samsung V-NAND SSD 980 PRO with Heatsink, 2021. https://download.semiconductor.samsung.com/resources/data-sheet/Samsung_NVMe_SSD_980_PRO_with_Heatsink_Datasheet_211101.pdf. (retrieved on 06/16/2024).

[74] Debendra Das Sharma. PCI Express® 6.0 Specification at 64.0 GT/s with PAM-4 signaling: a low latency, high bandwidth, high reliability and cost-effective interconnect. In *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 1–8, 2020.

[75] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’13, page 485–498, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2451116.2451169.

[76] Steven Rostedt. ftrace - Function Tracer, 2024. https://www.kernel.org/doc/html/latest/trace/ftrace.html. (visited on 08/16/2024).

[77] Super Micro Computer, Inc. H13SSL-N, 2024. https://www.supermicro.com/en/products/motherboard/h13ssl-n. (visited on 06/16/2024).

[78] SUSE, Facebook, Western Digital, Oracle, Fujitsu, Fusion-IO, Intel, Linux Foundation, Red Hat, and STRATO AG. Welcome to BTRFS documentation!, 2024. https://btrfs.readthedocs.io/en/latest/. (visited on 08/18/2024).

[79] The kernel development community. ext4 Data Structures and Algorithms, 2024. https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html. (visited on 04/05/2024).

[80] The kernel development community. Linux source code, 2024. https://elixir.bootlin.com/linux/v6.10.1. (visited on 05/08/2024).

[81] The kernel development community. Linux source code, 2024. https://elixir.bootlin.com/linux/v3.19.8. (visited on 05/08/2024).

[82] The kernel development community. Multi-Queue Block IO Queueing Mechanism (blk-mq), 2024. https://www.kernel.org/doc/html/latest/block/blk-mq.html. (visited on 07/16/2024).

[83] The kernel development community. PCI Peer-to-Peer DMA Support, 2024. https://www.kernel.org/doc/html/latest/driver-api/pci/p2pdma.html. (visited on 04/03/2024).

[84] The Khronos® Group. OpenGL Wiki, 2018. https://www.khronos.org/opengl/wiki/. (visited on 04/02/2024).

[85] The Khronos® Group. Khronos SPIR-V Registry, 2022. https://registry.khronos.org/SPIR-V/. (visited on 04/02/2024).

[86] The Khronos® Group. Open Standard for Parallel Programming of Heterogeneous Systems, 2024. https://www.khronos.org/opencl/. (visited on 04/02/2024).

[87] The Khronos® Group. OpenGL, 2024. https://www.opengl.org/. (visited on 04/02/2024).

[88] The Khronos® Group Inc. Vulkan | Cross platform 3D Graphics, 2024. https://www.vulkan.org/. (visited on 05/03/2024).

[89] Toshiba Electronic Devices & Storage Corporation. Toshiba Memory Corporation Introduces World's First Enterprise-Class SSDs with 64-Layer 3D Flash Memory, August 7 2017. https://toshiba.semicon-storage.com/eu/company/news/2017/08/storage-20170807-1.html. (visited on 09/02/2024).

[90] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *ACM Trans. Storage*, 4(2), May 2008. https://doi.org/10.1145/1367829.1367831.

[91] Western Digital Corporation. Ultrastar DC SN840 — 1.6TB, 2024. https://www.westerndigital.com/products/internal-drives/data-center-drives/ultrastar-dc-sn840-nvme-ssd?sku=0TS1874. (visited on 09/02/2024).