Karlsruhe Institute of Technology

GPU4FS: A Graphics Processor-Accelerated File System

Master's Thesis
submitted by

Peter Maucher

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Jun.-Prof. Dr. Christian Wressnegger
Advisor:	Lukas Werling, M.Sc.

Friday 10th December, 2021 – Saturday 6th August, 2022

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, Saturday 6$^{\text{th}}$ August, 2022

Abstract

Modern file systems with fast storage media take up valuable CPU resources, especially if used with Intel Optane in write-heavy applications. Optane is written synchronously, and accessing CPU cores stall if Optanes relatively low write bandwidth is saturated. To combat that issue, we propose GPU4FS, a novel GPU-accelerated user-space file system mainly targeted at Intel Optane as the main representative of high-performance non-volatile memory. GPU4FS relieves the CPU from all file system management tasks, including the writes to the Optane DIMMs. The CPU only queues requests to the file system into a shared command bu er.

With our prototype, we achieve similar bandwidth figures to CPU-side Optane tasks using a fio benchmark and sequential writes to DAX-EXT4. During the benchmark, the CPU tasks saturate four cores. Our GPU4FS prototype only uses a measured 12 % of a single core, and o ers a 33x reduction in CPU usage. We also show that we o er twice or more performance improvement for a parallel, CPU-bound blender render benchmark when running in parallel to GPU4FS, as compared to the CPU-side file system tasks.

Based on our findings, we conclude that future research into file system accelerators poses a worthwhile topic.

Kurzfassung

Moderne Dateisysteme benötigen vermehrt Rechenleistung, auch weil die Performance der darunterliegenden Speichertechnologien gestiegen ist. Intel Optane zeigt dabei in schreiblastigen Anwendungsfällen eine besonders hohe CPU-Auslastung. Das Problem sind die synchronen Schreibzugri e und die relativ geringe Schreibbandbreite von Intel Optane. Sobald diese Bandbreite erreicht ist, fangen die Kerne an zu stallen. Als Lösung für dieses Problem schlagen wir GPU4FS vor, ein neuartiges GPU-beschleunigtes Dateisystem im Userspace. GPU4FS ist besonders für Intel Optane als der wichtigste Typ von nichtvolatilem Speicher gedacht. GPU4FS übernimmt alle Dateisystemverwaltungsaufgaben von der CPU, einschließlich der Schreiboperationen auf die Optane-DIMMs. Die CPU muss ihre Anfragen nur noch in einen geteilten Kommandopu er einfügen.

Unser Prototyp erreicht ähnliche Bandbreiten wie bei CPU-seitigen Zugri en auf Optane durch fio und durch sequentielles schreiben auf DAX-EXT4. Dabei lasten die CPU-seitigen Dateisystembenchmarks vier Kerne voll aus, wohingegen unser Prototyp nur 12 % eines Kernes nutzt, und damit die benötigte CPU-Zeit um den Faktor 33 senkt. Zusätzlich erreicht ein parallel zu unserem Prototypen

laufender Blenderrenderbenchmark um mindestens einen Faktor 2 höhere Ergebnisse im Vergleich zu den CPU-seitigen Dateisystemen.

Unsere Ergbnisse motivieren zukünftige Forschung an Dateisystembeschleunigern.

Contents

List of Listings

List of Figures

1 Introduction

In recent years, new storage technologies were developed that promised large gains in storage performance. In parallel, CPUs still gained performance, but slower than storage, especially in single threaded and lowly threaded applications. As a consequence, storage-related tasks slowly required more and more CPU resources. File systems are one important case for CPU management of storage devices. For example, the CPU has to walk the directory tree, read and write files, and has to manage the underlying device. These compute resources cannot be used for application work.

One new storage technology is non-volatile memory like Intel Optane [33]. Optane blurs the line between said storage media and main memory. Optane promises speeds similar to that of DRAM, and some flavors sit directly on the memory bus instead of being accessed via external busses like PCIe. This means that DDR4-Optane is accessed with normal load and store commands, and not using DMA. Effectively, Optane promises to be a new level in the memory hierarchy. At the same time, PCIe-accessed NVMe-SSDs have also gained in performance.

On the one hand, Intel Optane promises fast, non-volatile storage, which makes them an interesting candidate to build file systems on, especially for read-heavy applications. On the other hand, writing to Intel Optane Memory from the CPU poses bandwidth problems: A single Dual Inline Memory Module (DIMM) of Optane Memory can only sustain up to around $2\,\mathrm{GB\,s^{-1}}$, far below the bandwidth of Dynamic Random Access Memory (DRAM) [31]. This bandwidth can be achieved with one to two cores, which are fully occupied with the synchronous writes. If more cores are writing to the same DIMM, the involved cores are stalled and make little progress [38]. This means that the CPU is waiting for Optane, but the operating system (OS) recognizes the cores as busy and does not schedule different threads. As a consequence, a system featuring heavy writes to Optane can be severely slowed down by parallel writes, even for processes which do not want to use Optane memory.

As a solution, coprocessors promise to take over the file system management tasks from the CPU. Coprocessors are usually well suited to complete tasks asynchronously and in parallel. In a file system context, delayed asynchronous completions and out-of-process handling are already quite common. File system coprocessors promise to free up CPU resources and allow the OS to schedule out waiting processes until the coprocessor signals completion. The CPU is freed to work on applications instead of the file system.

An important family of accelerators are GPUs: Starting with the invention of CUDA and OpenCL, and later Vulkan [29], GPUs have become commodity high-performance compute accelerators. Modern APIs offer easy access to this compute potential, and since GPUs are sold by the millions every year, they are quite affordable. This makes GPUs ideal candidates for experiments with coprocessor acceleration.

In this thesis, we present GPU4FS, a novel GPU-based file system accelerator for fast modern storage. We propose a full design for a modern file system which promises to mitigate the write deficiency of Intel Optane. Additionally, GPU4FS is designed to accomodate another level of file system caches in the GPU's Video RAM (VRAM). With our design, the CPU only inserts file system commands into a command buffer shared with and executed by the GPU. This bypasses the CPU's write bottleneck and frees up resources commonly used for file system management tasks.

We implement a subset of our design to demonstrate its feasibility. In our evaluation, we show that the GPU can closely match the CPU's write performance in both sequential writes and in actual file system tasks, all while reducing CPU usage drastically. Even in the worst case we evaluated, five parallel file writes at maximum bandwidth only use a reported 12 % of a single core. In a traditional CPU task, this would have saturated all four cores we measured on. We verify our results by measuring the performance degradation in a parallel benchmark. There, we found that the performance impact of our GPU file system is rather negligible with only 8 %. The CPU file systems, however, show slowdowns of a factor of two or greater.

This thesis is divided into multiple parts: In Chapter 2, we introduce core concepts central to this thesis. In Chapter 3, we discuss prior work of relevance for this thesis. In Chapter 4, we explain the design of a fully-featured GPU4FS implementation. We present the actual implementation of the demonstrator evaluated for this thesis in Chapter 5. We evaluate the demonstrator in Chapter 6. We use the results found in the previous chapters to propose future work in Chapter 7 and conclude this thesis in Chapter 8.

2 Background

This section introduces the main components of the thesis. We introduce different storage devices that our file system could run on, different flavors of file systems that serve as a basis for our design, and GPU acceleration as a central part of GPU4FS.

2.1 Storage Devices

The memory of a computer is usually divided into two distinct types: Non-volatile block-addressable storage and volatile, byte-addressable main memory, though modern technology like Intel Optane, see Section 2.1.2, blurs the line between these types.

2.1.1 Block Devices

Block devices are random access devices with one key difference: Instead of being written in terms of bytes, they are accessed in whole blocks. Common sizes of these blocks are 512 B or 4 kB. Most block devices are also non-volatile, i.e., they do not lose their information when the system is powered down.

The block structure necessitates that to read a single byte, the whole block needs to be fetched from the storage device. Write amplification [8] is the resulting problem when writing: If a single byte is changed, the whole block is fetched first, then modified and then written back, even though most data did not change. Therefore, a few random writes of a few bytes can trigger much larger block rewrites, which quickly eats up both read and write bandwidth.

Solid State Drives

In recent years, flash-based Solid State Drives (SSDs) have progressively become both more affordable and larger in capacity.

The flash technology offers higher sequential read and write speeds compared to earlier Hard Disk Drives (HDDs), in the order of several gigabytes per second [19]. Additionally, no mechanical movement is needed for an access, which means that SSDs can serve hundreds of thousands random requests per second, far outperforming HDDs in this metric.

A recent improvement is the usage of the Non-Volatile Memory Express (NVMe) protocol, which is based on the Peripheral Component Interconnect Express (PCIe) bus [19]. compared to earlier protocols, NVMe offers both higher speeds and better random access performance. Additionally, NVMe is designed to cater to highly parallel applications by offering multiple independent queues which can be filled concurrently by independent

threads. NVMe SSDs have become the de-facto standard in high-performance storage systems.

2.1.2 Byte-Addressable Storage

In a byte-addressable device, each byte can be individually addressed, read, and altered. This avoids the problems of wasted read bandwidth and write amplification. The most common types of byte-addressable devices are flavors of volatile main memory. It comes in different flavors, like the Double Data Rate (DDR)standards for PCs and servers, currently in version 4, or Graphics DDR (GDDR) for GPUs, currently in version 6. GDDR is optimized for bandwidth, whereas DDR is optimized for latency [61].

Non-Volatile Memory A special type of main memory has been evolving in the last few years: Non-Volatile Memory (NVM) that is compatible to some DDR controllers. Arguably the only important representative is Intel's Optane NVM memory. It offers a line of DDR4 socket-compatible Dual Inline Memory Modules (DIMMs), which some Intel Server CPUs can access the same way as DDR4 memory. Optane does not lose its data on power loss, and also offers much higher read speed as compared to even the fastest NVMe SSDs. It also serves even more individual random accesses than NVMe SSDs, offering latencies in the order of 500 ns [37].

Optane has one major downside, though: Compared to sequential writes to an NVMe SSD, writes to Optane are relatively slow, offering at most about $2\,\mathrm{GB\,s^{-1}}$ per DIMM [31]. This low bandwidth poses a problem: Optane looks like main memory to the CPU, so accesses use load and store commands instead of Direct Memory Access (DMA). DMA enables the CPU to configure hardware units and do other work while these units execute the memory accesses. Using load and store commands necessitates waiting for the memory subsystem, which stops meaningful work in this hardware thread. NVM shares one characteristic with DRAM that amplifies this issue: to a certain extent, the write speed can be increased by using multiple threads. For Optane, more than one thread might be needed to sustain NVM write speeds, but each thread will be significantly slowed down by the accesses. Hence, a significant part of the CPU is blocked by waiting on the memory subsystem.

As a solution, prior work tried to use the Intel I/OAT subsystem. It offers DMA functionality, in particular to accelerate `memcpy()` and `memset()` calls using a hardware implementation [6]. The I/OAT system was initially designed to accelerate $10\,\mathrm{Gbit\,s^{-1}}$ ethernet. Its design shows this in its operating system focussed access mode: it uses physical instead of virtual main memory. It also shows unexpected behavior related to NVM: While it can sustain multiple $\mathrm{GB\,s^{-1}}$ of write bandwidth into main memory, it collapses to about $500\,\mathrm{MB\,s^{-1}}$ into NVM [38]. If accessed from the CPU via synchronous load and store instructions, the modules can sustain much higher speeds. We assume this behavior to either be caused when the Optane DIMMs are overwhelmed by parallel accesses, or that the slowdown is triggered by some kind of write reordering issue. This makes Intel I/OAT unusable for NVM applications.

2.1.3 Discussion

Out of the three storage media presented here, GPU acceleration only makes sense for two: Memory-mapped NVM and PCIe-accessed NVMe SSDs.

To justify both the occupation of the computational resources on the GPU, and the overhead of a GPU, the access to the storage system has to be fast and easy to achieve. For HDDs, CPUs are fast enough to handle even multiple devices. The protocol used by HDDs is also difficult to implement, especially on a GPU. For SATA-accessed SSDs, the same arguments hold: In this configuration, the limiting factor moves from the storage medium to the interconnect: SSDs easily reach the bandwidth limits of SATA.

For NVMe SSDs, the protocol is a lot simpler, using just two register writes and some memory-mapped buffers [19]. It can be considered to map those registers and the buffers to the GPU, thus allowing direct writes bypassing the CPU. NVMe is also designed for parallel writes, which caters to the parallel nature of a GPU.

In case of directly memory-mapped NVM storage, this is even more applicable. A read or write is literally that, no further protocol needs to be implemented. It is also fast enough to require considerable performance from the CPU.

2.2 File Systems

After identifying the types of storage media that GPU acceleration makes sense for, we now discuss file system (FS) concepts that can be useful for a GPU file system. A file system is a system that maps files and folders to flat address space. The FS is also responsible for storing meta information like time stamps and owners. A file system is usually specified in two parts:

- The interpretation of the bits and bytes on the storage medium: With this specification, one would be able to implement their own file system driver. For an introduction, see Section 2.2.1.

- The driver: This is the software component that handles user requests and issues read and write commands to the drive. For more information about drivers, especially the interfaces to users, see Section 2.2.2.

Until around 2005, commercially used file systems have been built with hard drives in mind [18]. Since then, flash technology has become both drastically more affordable while gaining in capacity and speed. Starting in the early 2010s, additional technologies like memristors and Intel Optane have appeared. These technologies prefer quite different access patterns and can offer additional features: Optane DIMMs for example offer byte addressiblity and atomic accesses, and both Optane and NVMe allow for both massively parallel and random accesses. This necessitates a shift in file system design.

Even though history offers various types of file systems, we focus on file systems that were developed with Unix in general and Linux in particular in mind.

2.2.1 General Structure

This section introduces the features that common file systems implementations use on drives, for example, as in EXT4 [58].

On-Disk Structures

Every major FS has on disk structures which can generally be sorted into three categories: inodes, directories, and superblocks. Additionally, all these file systems use some kind of block structure connected via pointers. Many file systems include additional features like Journaling and Log Structuring.

Inode An inode contains the meta information for a file. This information includes the owner, time stamps, permissions, and file type (i.e., regular file or directory), as well as pointers to the actual data. The file name is not stored in the inode, but in the directory, as one file can be referenced from multiple directories, possibly with different names [24].

Pointers, Extents, and Blocks File systems have originally been developed for block devices. As the block needed to be written completely anyway, the file systems used these blocks for other purposes, too. For example, files are rounded up to the size of blocks when they are stored. To allow for efficient extension of files on the drive, files are usually not stored contiguously. Instead, file contents are split into block-sized chunks, which are then referenced by a pointer structure. One example for such a pointer structure is indexed allocation: Blocks on the storage device are referenced by an index structure similar to a page table. Each of these index pointers themselves are in blocks and are referenced by the block ID [24].

For large files and small blocks, this adds up to large index structures. To remedy that problem, more modern file systems like EXT4 [58] use a feature called extents to reference a sequence of blocks by their start ID and a length, thus decreasing seeks and lookups.

Directories A directory in a file system can either contain more directories or files. All the important information about these files, except the name, is stored either in the inode or the file contents. A directory is usually implemented as a special file with the contents being pairs of names and inodes. To manage these pairs efficiently, an index structure is built. A linked-list based implementation is one simple example, but poses issues with fast lookup in directories. Modern file systems like the B-tree File System (BTRFS) [13] and EXT4 [58] have switched to balanced trees, which reduce lookup time while keeping directory updates fast [24].

Superblock A file system needs to store a lot of meta information. Simple examples include caching the number of blocks used for fast lookup, but also more interesting information like the features configured for this instance of the FS. It also commonly contains pointers to a journaling structure, see Section 2.2.1. A system that is mounting the file system to use it also has the difficulty of not knowing the offsets of the root

directory without a convention. All these issues are usually solved by adding a superblock at a fixed, defined offset and size on the disk. When mounting the file system, the first step is to parse that superblock and use the information to configure the file system driver [24].

Journaling and Log Structuring

One important issue for file systems is crash consistency [20]. As the data is written to disk both asynchronously and usually out of order, in case of a crash of either the driver or the whole system, the invariants of the file system might be violated. Journals were invented to solve this issue: In its core, a journal contains a list of all things the FS tries or tried to do. It is only cleaned up after the FS is sure that the data was written to disk completely. If there are open entries in the journal after a crash, the driver can parse the journal and redo or undo changes as recorded in the journal to the file system until it reestablished all the invariants. The journal itself is written block-wise, with the block device guaranteeing that blocks are either written completely or not at all [24].

These journals have the disadvantage of duplicating all writes, once to the journal and once to the actual FS. For that reason, usually only metadata like file creation or file renaming is recorded, but not changes in the file's content. This means that crash consistency is still also a responsibility of the application running on top of the FS [24].

An attempt to remedy this issue while also increasing write performance is log structuring [2]. After the data is committed to a journal, it needs to be written to disk, in an order not controlled by the FS. This leads to relatively many seeks for the write operations, reducing performance. To avoid this issue, all changes are written to new blocks consecutively. This means file data is not necessarily consecutive anymore. The new data is committed by updating index pointers, the old data is garbage collected afterwards. The original intention was to accelerate writes to HDDs as log structuring promised more sequential writes. Additionally, this reordering promises better crash safety, as the pointers are only updated when the write operation is complete. On hard drives with their seek deficiency, the gains are smaller than expected [3], as reads suffer from the additional seeks. However, on media with high seek performance like (NVMe) SSDs and Optane NVM, log-structured file systems show real advantages. The advantages get even more pronounced on DRAM-like NVM systems which are not crash-consistent on the block level, but rather on an 8 byte level. In this case, the crash consistency needs to be guaranteed with exactly one pointer or index update [21].

Advanced Features

In addition to these common features, modern file systems like BTRFS [13] and ZFS [5] offer advanced features, like RAID, checksumming and encryption.

The Redundant Array of Independent Disks (RAID) [1] offers a solution for storage errors on disk. Such errors might occur due to disks dying or because of bits flipping on the drive. To solve this issue, information is stored independently on several drives. This can either be achieved by having multiple disks storing the same information, called

RAID1, or storing parity information of the blocks on disk. There are two common implementations which differ in the number of drives which can fail, RAID5 and RAID6. The former can sustain one drive failure, the latter can sustain two.

Formerly, the RAID implementation was independent of the file system, and the file system just used a virtual block device. BTRFS and ZFS incorporate the redundancy information into the file system information, which allows for more fine-grained control of the amount of redundancy. For example, metadata might be stored more often than file contents depending on their importance or the file system might want to make sure that a whole file is sitting on different drives redundantly.

Another feature that helps with data recovery are checksums. If a RAID implementation detects a mismatch between different copies, it cannot necessarily decide which copy contains the correct or faulty data. As a solution, modern file systems store additional checksums. This way, the file system can detect mismatches more easily, as checksum computation does not necessitate requests to the different drives. In case of errors, the file system can choose a copy which matches one of the redundant checksums. Even if the correct data cannot be recovered, the application can be notified that an erorr has occured.

Another feature is encryption. Previously, encryption was implemented on the block level [59], similar to RAID. This leads to issues with flexibility, as different applications have different requirements for encryption. An example is the Android mobile operating system, where certain device features like alarm clocks should run even without the system being decrypted. This can be solved with a file system which supports partial encryption for certain files and folders, like ZFS or Android File-Based Encryption (FBE) [56].

2.2.2 Interfaces

Multiple more or less common interfaces to file systems exist. Two important examples are introduced here.

POSIX

The Portable Operating System Interface (POSIX) [25] is a specification for operating systems. It includes function specifications for many major interfaces of operating systems, including the access to the FS.

A POSIX-compliant file system driver has to implement a few common functions: There is the pair of `read()` and `write()` to access the data, as well as `open()` and `close()` to manage the connection between process and file system. There are also some functions to manage the file system itself: `creat()` and `unlink()` as well as `mkdir()` and `rmdir()` are used to create and remove files and folders, respectively. There are additional functions to manage metadata like timestamps.

These file systems have the benefit of being widely available [18]. This means benchmarks are widely available as well. On the other hand, the implementation of a complete and mostly POSIX-compliant file system takes a lot of effort. POSIX requires more interfaces and functions that need to be implemented, as compared to a special purpose file system.

A special-purpose file system might lack the capability for hard and soft links, special metadata like time stamps, or even complex folder hierarchies.

The Case of File Mapping The `mmap()` system call, specified by POSIX, is designed to map files into the address space of the requesting process. In user space file systems, the `MAP_SHARED` mode poses difficulties: It allows for one file to be mapped by multiple processes simultaneously, and allows inter-process communication using shared memory through this buffer. In a user space context, a process cannot modify the page tables of another process, and thus cannot share the buffer without kernel support. The POSIX specification assumes an operating system-managed file system buffer, which requires kernel cooperation with the user space file system [25].

The feature also conflicts with some optimizations, especially caching: A private write cache, possibly containing metadata, needs to be in sync with the changes done through direct memory writes. The metadata also needs to be protected from writes by other processes.

Therefore, some kind of external support is required to support `mmap()`, e.g., the kernel or a centralized and privileged user space management process [18].

Key/Value-Store File Systems

In case of an application-specific file system, only a subset of these functions and functionalities is needed. Often, they can be layered on top of existing file systems, themselves residing in a file. But, these ideas can also be applied to a file system that directly resides on hardware. One example is a key/value-store, which essentially offers the functions `put(key, value)`, `value = get(key)` and possibly `erase(key)` [18]. They need special benchmarks, but can offer significantly more performance per implementational effort.

2.2.3 User Space vs Kernel Space

The file system driver needs to be located somewhere. Commonly, this is done in the kernel, but user space file systems have been emerging.

Kernel Space File Systems

Originally, file systems have been implemented in the kernel. The OS is tasked with abstraction, and the file system is a major example for abstractions. User space communicates to the kernel via system calls, which mostly represent the high-level functionality of file systems. For example, in Linux there is a system call called `write` which implements the POSIX `write()` functionality [24]. Inside the kernel, the different file systems are abstracted in a virtual file system (VFS). The VFS abstracts the different file system drivers towards user space.

Mixed User/Kernel Space File Systems

Implementing parts of drivers in user space has several advantages: The security of the kernel can be guaranteed more easily, and the driver can be updated independently of the kernel. Additionally, the interface can be used to implement new services like sshfs [60], which mounts the file system of a different computer via the network.

In a microkernel system, implementing drivers in user space is the intuitive approach, but user-space FS drivers are also used in monolithic kernels like Linux. The IPC interface depends on the implementation, but is specified by the kernel or a related API. One such example is the Unix File System in User Space (FUSE) [57] implementation, which forwards system calls back into the user space file system driver implementation.

The main difference for this definition to a real user space file system is that a mixed user/kernel space file system still heavily relies on kernel APIs, kernel support, or kernel-related documentation and cannot easily be separated from the kernel they are developed for.

User Space File Systems

In recent years, user space file systems like Aerie [18] have emerged. The main idea is to allow user space direct read-only access to the storage medium. This means that most common operations like `open()` and `read()` need no syscall or inter-process communication (IPC).

The main difference between the papers is the way that they handle writing to the medium. Some papers, for example Aerie [18], Strata [23], or SplitFS [32] remap certain parts of the drive to be exclusive to a process. In this area, the process gains write permissions. A different common attempt, used, among others, by EvFS [34] and the paper "Towards High-Performance Application-Level Storage Management" [16], is to enforce write separation with hardware support in the storage medium. A third attempt is to offer the write data to a trusted party, like the kernel or an privileged user-space process. This is the way that Strata or Aerie, respectively, handle the metadata updates. To avoid communication bottlenecks, Aerie uses a command buffer for metadata changes. The requesting process queues commands in the shared memory buffer, which are worked by the trusted party.

Even though direct access to the medium is efficient, untrusted applications cannot be allowed complete access to the medium. This would allow user space to overwrite file system metadata. This can be used both to gain illegal access, or to destroy the invariants of the file system. Therefore, a trusted component needs to handle metadata changes, and, to some extent, write requests for file data. In classical file systems this role is filled by the kernel, and this is a valid implementation for a user level file system, too. However, the trusted component can also be implemented in a priviledged user space process, similar to the design of a microkernel operating system. This is the main idea of Aerie [18]. In both cases, kernel and user space, the trusted component has full read and write access to the drive. Its task is to validate the requests from other processes, and commit them to the drive.

2.2.4 Discussion

As presented, most modern file systems offer similar sets of basic features. This holds true for many parts of the implementation, too. Except for reading and writing, basic features of a file system do not profit from parallelization greatly, however. Additionally, reading and writing is handled using DMA outside the CPU, except in the NVM use case. GPUs excel in massively parallel compute, so using them in parallel tasks with large datasets increases their usage greatly. This suggests implementing a more fully-featured modern file system, including encryption, raid and checksumming. Each of these features profits from fast, parallel compute on the whole working set of the file system. The working set of a file system includes all files read and written at any given point, so it is pretty large at high FS usage. For this reason, the design of GPU4FS is fully prepared to accommodate these advanced features.

GPU4FS aims to be generally usable, so a POSIX-compliant interface makes sense. However, the additional code required for a fully POSIX-compliant file systems is too great to demonstrate the feasibility. Therefore, our demonstrator only supports a minimal interface, but the design is fully prepared for POSIX compliance.

GPU4FS selects to run in user space, and uses a privileged process to manage write permissions. To manage the inter-process communication, we use a shared command buffer that is also mapped to be GPU visible. With this design, we exploit the discussed benefits of user-space file systems while allow for efficient communication with the GPU.

2.3 GPUs

Since their introduction in the 1980s, Graphics Processing Units (GPUs) have gained more and more responsibilities. What started as a simple converter from a pixel buffer to a VGA output learned to draw first 2D and then 3D shapes and has evolved into a general purpose parallel computer. In parallel to its features, the graphics APIs also evolved: At first, OpenGL and DirectX only supplied pixel data, but learned to support graphics programs, so-called shaders, and general compute tasks [27].

2.3.1 Basic Structure

Modern GPUs are composed of multiple single instruction, multiple data (SIMD) processor cores. Each of these cores commonly has between 8 and 64 SIMD lanes, which execute the same instruction on different data. Each processor core also contains multiple process counters to run multiple hardware threads on the same set of execution units. This way, memory latency can be hidden by executing other threads in parallel [30].

Modern GPUs either access main CPU memory or have dedicated memory called video random access memory (VRAM). VRAM is optimized for throughput, not latency, and commonly accessed via wide memory busses. The memory is connected to the GPU cores via complex memory management units (MMUs), which can coalesce and reorder memory accesses. A dedicated memory GPU also has access to a part of main CPU memory, usually via the PCIe bus. The memory controller can transparently map main memory

into the GPU's own address space. This is implemented using a special address range in the GPU's physical address space, which is translated to main memory. The Graphics Address Remapping Table (GART) [11] technology was initially developed for Accelerated Graphics Port (AGP) [4] GPUs. AGP is a predecessor of PCIe. GART is still used on modern AMD GPUs. Notably, AMD uses the name Graphics Translation Table (GTT) synonymously.

2.3.2 Communication

To communicate with the GPU, the CPU usually has three possible options: Memory-mapped PCIe registers, shared memory, and, to some extent, interrupts. The register file is mapped into memory, and accesses are transferred to the GPU. There are a fixed number of those registers, which means that they can only be used for predetermined purposes. These include power management, configuration, and control of the other communication channels. Similar issues arise with interrupts: There are very few different interrupts both CPU and GPU can raise, and they are mostly used to communicate important events such as completions.

The main path of communication relies on shared-memory command buffers. These buffers are commonly implemented as ring buffers, which are indexed using control registers. The primary command buffer contains pointers into secondary buffers. These secondary buffers contain all the commands the GPU should execute, like GPU process execution or copying between main memory and VRAM [17].

2.3.3 Programming Model

GPUs are programmed using shaders, which are compiled on the CPU, usually at runtime. The name shader comes from the initial purpose, programmed shadows in games. For the programmer, each SIMD lane of a shader looks like its own thread, without much interdependence to other threads. Only the existence of synchronization primitives inside local working groups and some fast shared memory hint towards the existance of the SIMD nature of GPUs. For this reason, the complex MMUs described above need to be able to gather those independent requests from the SIMD lanes into fewer, wider requests for VRAM to improve latency and throughput [30].

Traditionally, GPUs were programmed using APIs like OpenGL [50] and DirectX up to version 11 [52], that hid most of the complexity of GPUs like command buffers and transfers from the user. With the invention of compute-focussed APIs for graphics cards, like CUDA [53] and OpenCL [49], GPUs became more versatile, and General Purpose Computation on GPUs (GPGPU) has become common. The APIs offered direct access to the GPU without having to go through the graphics pipeline. Additionally, they offered a more low-level access, especially regarding buffer management. This meant that they also offered some performance benefits.

In recent years, new APIs like Vulkan [29], DirectX 12 [52], and Metal [39] offer unified APIs for both graphics and compute. While DirectX 12 and Metal are limited to specific Microsoft and Apple products, respectively, Vulkan-capable hardware and drivers are

commonly available for all major platforms and operating systems. For Linux, these drivers are also open source for both Intel and AMD graphics cards, which allows for easy hacking in the drivers.

2.3.4 Linux GPU Driver Stack

GPU4FS is built on top of a modified Linux GPU driver. To understand these modifications, we introduce the driver stack here.

The graphics drivers under Linux are generally divided into two parts, a kernel space and a user space component. Each of these components is again separated into subcomponents.

Kernel Driver The kernel-space driver manages process separation and device setup, including tasks like power management and device hibernation. It also provides user space with a safe interface for further device configuration. Each GPU family has its own driver family, for example `AMDgpu` [44] for modern AMD GPUs, `nouveau` [42] for NVidia GPUs or `i915` [45] for Intel integrated GPUs. Each of these drivers uses a set of libraries that fulfill different tasks, like the Direct Rendering Manager (`DRM`), the Graphics Execution Manager (`GEM`), or the Translation Table Maps (`TTM`) subsystem. GEM and TTM are different implementations for graphics memory management.

The DRM subsystem offers a common interface to user space, offering the GPU to user space as a special file with different ioctls. Different DRM drivers offer similar but not identical feature sets, for example, to configure TTM vs. GEM-based GPUs.

On the user space side, another DRM library (`libdrm`) is needed that interfaces to the DRM subsystem in the kernel. This library is mostly concerned with translating more generic calls from the rest of the graphics stack to the exact DRM call needed by the kernel.

The actual graphics drivers are layered on top of the `libdrm`, and are usually part of the Mesa3D project. Many modern drivers, for example the driver we use in this thesis, are implemented using the Gallium3D library. Gallium3D offers implementations for common problems, which are used by different drivers. As shown in Figure 2.1, Gallium3D-based drivers use multiple layers: One layer is needed to translate the calls from APIs like OpenGL, Vulkan, and DirectX into a common format. The next level interprets this common format for each type of GPU. Te last level communicates the results to the kernel via the `libdrm`, and also talks to the window management system (WinSys). The kernel then actually manages the devices like CPU and GPU.

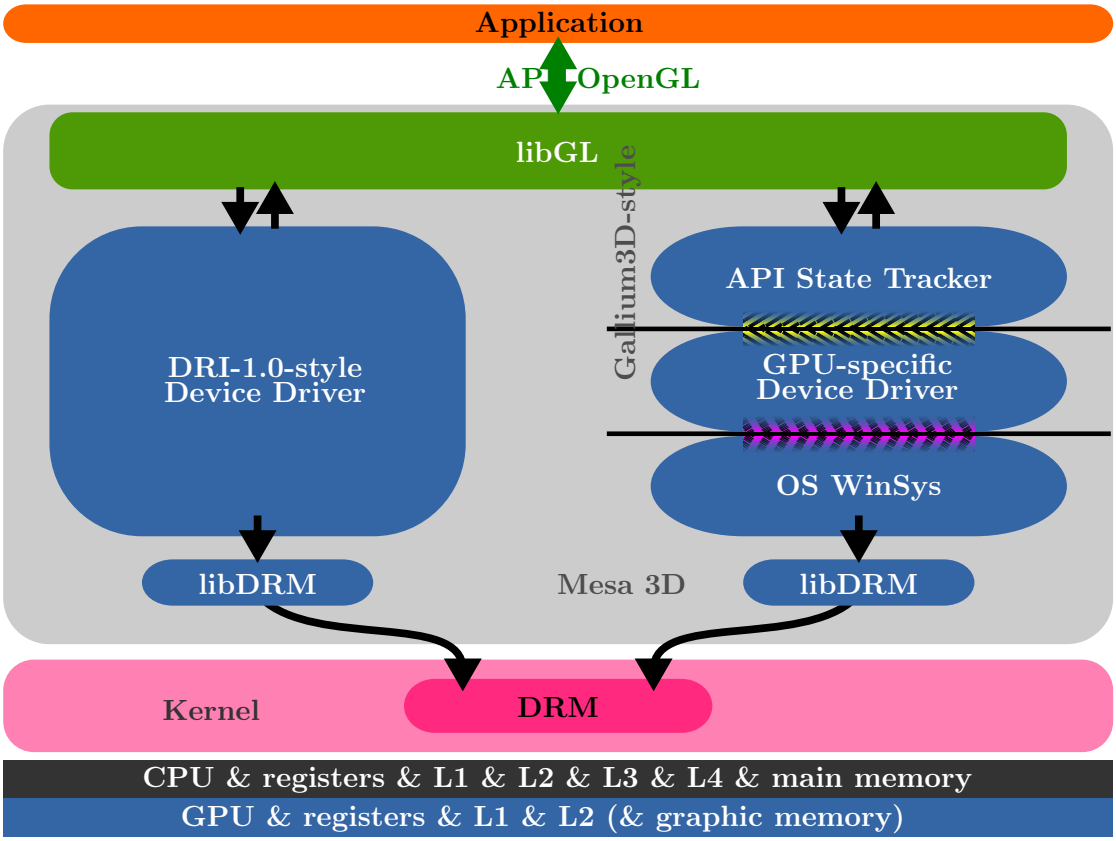


Figure 2.1: Layout of a Linux Mesa3D open-source GPU driver [14].

3 Related Work

In this chapter, we present prior work for file systems, GPGPU, and file system accelerators.

3.1 File Systems

The purpose of a file system is to map tree-like directory and file structures to flat address spaces on drives. Traditionally, file systems have been implemented as part of the kernel, but recently, user space file systems have emerged.

3.1.1 Kernel-Space File Systems

Implementing the file system in the kernel is the intuitive choice, as the kernel manages the access to the devices. Therefore, many different kernel-space file systems with different feature sets have been developed in the last decades.

EXT4

The EXTended file system in its fourth release, EXT4 [58], is a journaled file system that finds wide-spread usage in many Linux distributions due to its stability. EXT4 also serves as inspiration for the on-disk data structures of GPU4FS, especially in the inode. Additionally, EXT4 supports the Linux Direct Access (DAX) mode, which disables caches and writes directly to the Optane NVM.

ZFS

ZFS [5] is a modern data-centre oriented copy-on-write file system originally developed by SUN for the Solaris platform. In addition to traditional file system tasks, it also offers RAID functionality, volume management, and checksumming to avoid data loss, and encryption for data security. The integration of these features into the file system allows both for easier configuration and better performance. Another performance optimization is an aggressive multi-level caching system integrated into the file system.

This collection of features makes ZFS an interesting target for accelerators, as each of these features uses precious CPU time.

BTRFS

The B-Tree File system, BTRFS [13], is another attempt at a modern file system similar to ZFS, with one of the most relevant differences being the license. On the implementation

side, it uses trees for many purposes, not only for the directory entries, hence the name. Using trees can improve lookup times in many file systems.

NOVA

NOVA [21] is a kernel-space file system specially designed for non-volatile memory on the memory bus. NOVA is designed to be a fully featured POSIX-compliant file system, including strong guarantees for consistency and good parallel performance. The main design trait of NOVA is the log structuring approach: It incorporates the benefits in random access performance by structuring the log as a simple linked list, and parallelizes by offering one independent log per inode.

3.1.2 NVM User Space File Systems

There are a few file system papers that demonstrate the feasibility of user-space file systems. All of them work under the premise that context switches, to both kernel and other processes, should be avoided.

Aerie

In [18], the authors present Aerie, and demonstrate that user-space file systems are feasible. Their design moves the trusted component that manages the FS into a separate user space process instead of the kernel. The trusted component is augmented by a small kernel module which allows the user space driver to change mappings in the requesting processes. To allow for efficient file writes, Aerie remaps parts of the NVM as simultaneously readable and writable. To avoid stale references from the command buffer into the file system, the driver process hands out locks to parts of the file system. The requesting process then is the only one allowed to access this memory. The authors deliberately chose this design as they identified unique file accesses to be the norm.

Additionally, they show their design to be flexible enough that the same driver process can support both a POSIX-compliant system as well as a key-value file system.

Strata

In [23], the authors demonstrate Strata, another attempt at user space file systems. Their main idea is to build a layered system and use NVM as one of the storage layers, but also support other layers like NVMe SSDs. Again, file writeback is solved by remapping the respective area as read/write. Unlike Aerie, the trusted component moved to the kernel. Strata focuses on supporting different storage media, which usually need more kernel support than NVM.

EvFS

The main idea of EvFS as presented in [34] is to reimplement the whole file system stack, including a lot of the caching, in user space. Instead of involving a trusted component, this

design splits the NVM in sectors using a proposed hardware feature they call namespaces. Each process is offered full read/write access to its namespace, and the hardware enforces separation between the processes.

SplitFS

Kadekodi et al. present SplitFS [32], which tries to avoid some of the costs of going through the kernel by intercepting POSIX calls, and replacing them by directly `mmap()`-ing the underlying device. The requests are then handled using CPU loads and stores in userspace instead of using the underlying kernel file system.

Towards High-Performance Application-Level Storage Management

In [16], an even more flexible system is proposed: a unified hardware interface that allows hardware to expose extents. These extents are similar to the NVM namespace feature that EvFS uses. Each process can then build its own file system in the extent offered by the base system with full read and write access. This allows each process to have a file system catered to its unique characteristics, but does not offer a unified file system anymore.

3.1.3 Discussion

The prior work presented here demonstrates the benefits of user-space file systems compared to common kernel-based file systems. They achieve both better versatility and higher speeds, and are better suited to modern storage media.

There are some common issues with these results, though: The benchmark for these file systems are other, hard drive-optimized file systems. This means that they might also show the benefits of modern, flexible hardware and optimized operating systems, not just of the file system. Both the new and the old system suffer from the problem of slow write speeds to NVM from CPU. The NVM papers, in particular Aerie [18], Strata [23], and SplitFS [32], mostly use direct writes to limit CPU usage in the file system management. The major exception is EvFS [34], which includes multiple layers of main memory caches to improve latency.

Another issue that older papers face is that the performance characteristics that were promised by the manufacturers and simulated for these early results [18] are not reached by actual hardware [31]. For example, simulated hardware would not face the issues found with Intel I/OAT and NVM [38]. This might explain some of the design decisions.

3.2 GPU

There are many APIs for GPU programming which might be usable to implement a GPU file system. In [29], "Novel Methodologies for Predictable CPU-To-GPU Command Offloading", the authors show that the Vulkan graphics API is highly flexible and well

suited for GPGPU, even when the task is latency-bound, or when the GPU is only executing on small data sets. Vulkan can therefore be used to implement GPU4FS.

3.2.1 GPU File Systems

In the literature, GPU file systems usually try to expose the CPU's file system to the GPU or similar accelerators. With that, graphics programs can access the file system, which can ease software development. Some examples are [36] or [15].

Both of these papers put their emphasis on buffer management between the host and the client memory, especially on data transfers. These buffers can then be modified from accelerator-side functions.

As presented above, there is literature on the topic of bringing file systems to the GPU. GPU4FS implements the inverse idea of bringing GPU acceleration into a CPU file system, which is a novel concept.

3.3 File System Accelerators

Some research has gone into accelerating file system tasks using special hardware.

3.3.1 Substep Accelerators

There are some papers that put some steps in the file system pipeline into accelerators. [26] for example uses an Field Programmable Gate Array (FPGA) accelerator for file system encryption tasks.

3.3.2 FSMAC

In [10], the authors present the File System Metadata ACcelerator. It uses NVM as the accelerator for normal disks. It splits the file system contents: metadata goes to NVM, while file contents are stored on the disks. This work blurs the line between a tiered file system like Strata and an accelerator. We name FSMAC here as it is implemented entirely in kernel space.

3.3.3 Moneta-D

The Moneta-D accelerator as presented in [9] moves security checks into hardware: Each process gets access to a certain read queue on the non-volatile device. The kernel only programs the access checks, and each process can read and write directly to its extent. The kernel still handles file system metadata, but reading and writing file data is done completely in user space, with the access checks done in hardware.

To achieve this, they construct an FPGA that offers userspace an easy interface for read and write. On file `open()`, the kernel programs the FPGA so it offers direct access to the file for a process, and the process then can communicate directly with the hardware to read and write. The implementation still uses a normal file system, only lifting the task of access checks to external hardware.

3.3.4 Discussion

A common pattern is that prior work tries to move individual tasks to the accelerator, not the file system as a whole. The benefits of lifting most if not all management tasks from the CPU are not explored.

4 The Design of GPU4FS

Current file systems for Optane show high CPU usage due to the synchronous writes to the medium, which stall the CPU cores. To solve this problem, we aim to offload as much of the file system as possible a GPU.

To customize our file system to the requirements of both the Optane memory and the GPU, we develop a new file system, GPU4FS, instead of adapting an existing one. This file system consists of both an on-disk specification and a driver on both CPU and GPU.

This chapter details a complete implementation, including all features. In Chapter 5, we shows what was actually implemented and evaluated for this thesis.

4.1 Two Minute Design Overview

GPU4FS is a GPU-accelerated 64 bit user space file system. Each of these qualifiers shapes the full-scale design, which can be seen in Figure 4.1.

GPU Acceleration A GPU cannot work on its own, but needs to be configured by the CPU. As discussed in Section 2.3, the GPU management should be implemented by a process running in user space. As described in the next paragraph, this process assumes another responsibility.

User Space File System The goal for GPU4FS is to reduce CPU time. Going to the kernel by executing syscalls takes up resources that should be spent in the application. In a user-space file system, depending on the read/write path chosen, after the initial setup no further kernel interaction is necessary. In addition, communication to the GPU does not use the normal system call interface, and GPU configuration is traditionally easier from user space than from kernel space. Thus, we decided to implement a user space file system.

64 Bit File System To be compatible with modern systems and be usable into the future, while also being easily implementable, all offsets and sizes and nearly all counters are 64 bit in size. The one exception are the number of hardlinks per file, which is 32 bit wide[1]. Intel Optane is also 8 B-crash consistent and much more performant if accesses are 8 B-aligned, so nearly all data structures are stored that way. The one exception works on strings and 1 B counters. Therefore, the accesses to NVM are made in an aligned fashion, and are reorganized in registers on the executing device.

[1] 4.2 billion subdirectories ought to be enough for everyone...

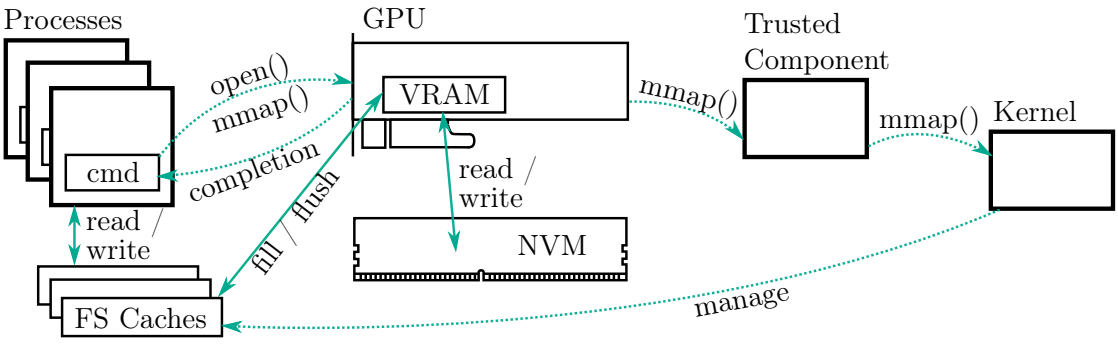


Figure 4.1: GPU4FS with caching and the trusted component. Requests are queued into the command buffer and request data is stored in the caches. The GPU parses the command, and fetches the data from the FS caches into the VRAM cache. It then writes to NVM using the data in VRAM. When loading, the data is fetched from NVM into VRAM, and then stored to the FS caches.
In case of a command that needs OS support, like `mmap()`, the GPU also inserts the command into the trusted component's command buffer. The trusted component then issues system calls to the kernel, which can execute management tasks with kernel privileges. The completion of the syscall in the trusted component is then signaled back to the GPU, which forwards the completion to the requesting process.

GPU Communication The communication between CPU and GPU is done via shared memory. The CPU maps a buffer to be GPU-accessible, both in read and write. Inside this buffer, the CPU sets up data structures, which are in turn parsed by the GPU. Similar to the way graphics drivers are working, a set of command ring buffers contain the requests of a certain process to the GPU. These commands reference data which is also stored inside that memory region. After the GPU executed all these buffers, it signals completion by setting a flag inside the shared memory region. The other alternative, interrupts signaling completion, would have to be handled by the kernel, which we try to avoid.

File System Connection A process wishing to use a GPU4FS-formatted partition needs to establish communications to the GPU. For this reason, it signals the CPU-side GPU management process that it wants to use the file system. A shared memory region is created by the management process and mapped to the GPU. The requesting process also maps the shared memory. From this moment on, the requester can communicate with the GPU directly. No further interaction from either the kernel or the management process is needed until either cleanup is required or a shared file mapping is used.

Caching One important performance optimization is the use of file system caches. To allow the runtime sharing of caches for a shared file mapping, the file system needs to be

able to map pages inside of another process at runtime. To avoid all unneeded system calls and to guarantee consistency which is tracked on the GPU, the request gets passed to the GPU as a normal command, but the GPU forwards it to the management process. In turn, the management process calls into the kernel and requests the remapping. For this purpose, the management process is trusted by the kernel to execute valid remapping requests. While the kernel communication is ongoing, the GPU already loads the information from disk to VRAM so the caches can be filled directly after the mapping is completed.

4.2 On-Disk Data Structures

GPU4FS is an inode-based filesystem, with its data stored in aligned blocks. Directories are special files whose data is interpreted by the file system driver. Special information like the root directory and allocator metadata is collected in the superblock.

4.2.1 Blocks and Block Pointers

The pointer size is a major consideration for every file system. EXT4 [58] has been extended to support larger pointers in the past, and other file systems, e.g., BTRFS [13] and NOVA [21], have been developed for 64 bit pointers directly. In the NOVA paper, the authors also argue that this size is well-suited to Intel Optane memory given Optanes crash consistency and atomicity guarantees. We decide to also implement a file system using 64 bit pointers since it fits the target storage medium well.

Another configuration point is the internal block size of GPU4FS. Other than common block-addressable storage media, Optane DIMMs are byte-addressable. Common file systems like EXT4 [58] are built with the storage medium's block size in mind, but for Optane NVM, such sizes are less obvious. Luckily, the fact that Optane DIMMs behave like normal DRAM comes to help here:

On x86-64 systems, the MMU organizes DRAM into 4 kB, 2 MB and 1 GB pages. DRAM and Optane both can only be mapped in page granularity. To enforce process and visibility separation, the blocks a file consists of have page size and are aligned to page boundaries. Therefore, the drive is split into aligned 1 GB blocks, each of them can be further subdivided into 2 MB and then 4 kB blocks, recursively.

An inode, see Section 4.2.2, does not need a complete 4 kB page, so the recursive block splitting idea is extended. Depending on the build configuration, the size of an inode is either 128 B or 256 B. The smallest page size, 4 kB, is an integer multiple of both 128 B and 256 B.

This leaves us with four possible block sizes, one for inodes and three for page sizes. By design, each block on drive is aligned to at least 128 B. Therefore, an offset into the disk that needs to address a block has its 7 least significant bits (LSB) not set, which can be used for tagging. As can be seen in Figure 4.2, of these 7 bits, 4 are currently used:

- 2 bits = 4 possible values tag the size of the block pointed to,
- 1 bit flags whether the block pointer is valid,

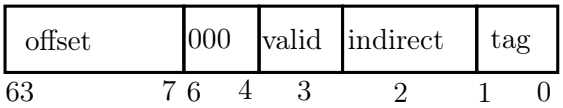


Figure 4.2: Bit usage in the tagged block pointer. 57 bits are used for the offset and three are unused. The remaining four signal whether the pointer is valid, indirectly pointing to more block pointers, and the size of the data referenced by the pointer.

- and 1 bit flags whether the offset pointed to is indirect.

An indirect block does not point to data directly, but instead points to another block which contains block pointers. This feature is recursive, so that infinitly large files could be accessed in theory. In practice, both file and drive size are limited to 2^{64}B.

4.2.2 Inode

The GPU4FS inode, as seen in Figure 4.3, is modeled with the POSIX [25] requirements for file systems in mind. Aerie [18] shows that a generally usable, POSIX-compliant file system in user space is possible, and GPU4FS aims to be generally usable. Therefore, we incorporate the POSIX requirements directly into the design.

We borrow all necessary flags from a preeexisting POSIX-compliant file system, in particular EXT4 [58]. There are some notable differences to EXT4 though, which are explained here. A major one is that GPU4FS has been designed from the ground up to be 64-bit, so all fields are aligned and grouped. EXT4 has been extended multiple times, and different parts of the same number are spread all over the inode. Another major difference is the inode number: It is simply the physical offset of the inode on disk, without any complicated translation. In EXT4, the file system preallocates space for inodes on different parts of the disc, and then assignes numbers to each inode in each of these areas. This makes additional lookups necessary.

Time For future-proofing reasons, GPU4FS uses nanosecond-precision timestamps. Given the general alignment rules that we established, each time stamp should be stored in a 64 bit number. Additionally, the year 2038 problem [7] should be avoided as far as possible. This problem exists as POSIX [25] initially specified the time to be presented in a signed 32 bit number. This number represents the time in seconds since the first of January, 1970, and overflows 68 years and a few days after that day.

In a GPU4FS timestamp, the nanoseconds are stored in the least significant 30 bits, while the seconds since epoch are stored unsigned in the most significant 34 bits. This means that 2^{34}s or about 544 years can be stored in one time stamp. GPU4FS therefore has a year 2514 problem[2]. Larger time stamps can be enabled for future GPU4FS users using the built-in extension feature, that we will describe in the following paragraph.

[2] more precisely: a 30th of May 2514, 01:53:04 am UTC problem

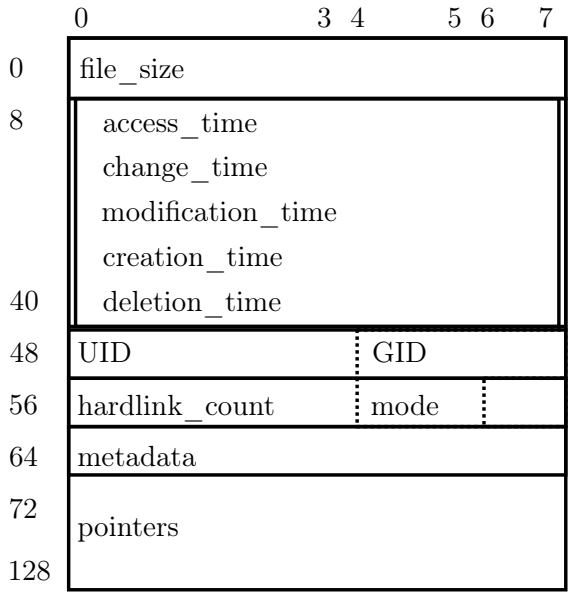


Figure 4.3: The GPU4FS inode. It contains the file size, all required time stamps, user and group IDs, a 32 bit hardlink count, 16 bit mode flags, a pointer to meta data, and the actual file pointers. Notably, two bytes with offset 62 and 63 are currently unused, as `hardlink_count` and `mode` only need six bytes, and metadata should be aligned to ensure performant and consistent NVM accesses.

Extensibility As evidenced by the repeated extension of EXT4 [58] and the resulting spread of bit patterns over the inode, extensibility is important. For that purpose, the GPU4FS inode offers space for one, possibly indirect, block pointer. This bears some similarity to the different data streams in the NTFS file system, with each stream containing independent data [51]. The storage pointed to can be used for existing meta information like access control lists (ACLs) or SELinux tags. The extra space can also be used for more uncommon features, like storing which application should open the file.

Due to alignment, there are also two free bytes which can be used for later extensions if the space is sufficient. These two bytes are the result of packing the access permissions and the file type into 16 bits, which together with the 32 bit inode count only fills 48 of 64 bits. This issue is illustrated in Figure 4.3.

4.2.3 Directories

Directories in GPU4FS are implemented as a file, with a special tag in the inode. The directory entries are stored as a linked list. This is similar to early file system implementations [24].

In our implementation, each directory entry has four fields: the block pointer of the inode, the offset into the directories' file of the next directory entry, the length of the name and the name string itself. As discussed before, we model GPU4FS to be POSIX-compliant. In POSIX, a file name length is limited to 255 characters. This length can be stored in one byte. Therefore, the length of the string is stored the next byte at offset 16, and the filename starts directly afterwards at offset 17 of the directory entry. To access the NVM performantly, the length and the string are loaded and stored together, and only separated in the CPUs' or GPUs' registers. In Figure 4.4, we show an example for a directory containing two files, `I_like_trains.mp4` and `Whooosh.mp4`. As an optimization, unneeded elements of the name can be overwritten by another entry, as long as the 8 B-alignment is guaranteed. To avoid having to walk the linked list for lookups and insertions, it is desirable to implement a tree structure, similar to what BTRFS [13] uses, in a later iteration.

4.2.4 Superblock

To manage the main file system, a simple superblock is used. Its main task is to persist the allocation metadata, see Section 4.3.5, between mount operations. Storing the allocation data in the superblock is also found in other file systems like NOVA [21].

The superblock also references the root inode, and contains fields to signal which features this instance of the file system uses. Examples for such features include the inode size or the usage of extended attributes, like larger time stamps. Again, GPU4FS borrows from prior file systems: EXT4 [58] uses the superblock in a similar fashion, i.e., to signal which features are active for this instance of file system.

	0	1	2	3	4	5	6	7
0	ptr_to_inode							
8	next_file_offset=40							
16	l=17	I	_	l	i	k	e	_
24	T	r	a	i	n	s	.	m
32	p	4						
40	ptr_to_inode							
48	next_file_offset=0							
56	l=11	W	h	o	o	o	s	h
46	.	m	p	4				

Figure 4.4: Example content of a GPU4FS directory, containing two files. The second file follows directly after the name of the first file, reclaiming the unneeded space in the filename as far as the alignment allows. The filename is not 8 B-aligned, as the length l shifts the string by one byte.

4.3 Runtime

Most of the features of the file system on disk are borrowed from prior file systems. Most novel features are found in the runtime management of the file system, especially how communication is handled from processes to GPU and back. Given that the latency between operation queuing and completion can be quite long and involved, some thought is given to crash consistency, particularly at runtime.

4.3.1 User Space File System

The goal of GPU4FS is to free the CPU as much as possible. Mainly, this is achieved by moving the file system itself to the GPU, but other optimizations are used, too. An important one is the decision to implement a user space file system, as guided by prior research: Aerie [18] demonstrates a POSIX-compliant file system in user space, which is generally usable and only needs minor kernel support. Instead, a user space process manages the storage medium. In EvFS [34], Yoshimura, Chiba, and Horii additionally show that extended file system caches in user space are possible and offer performance improvements. Strata [23] on the other hand moves part of the management into the kernel.

To decide on the location of the trusted component that manages the file system, we look onto the other requirement: managing the GPU. In Linux, the GPU configuration are commonly done in a user space process. This process can also be used to run the file system management, as presented by Volos et al. in Aerie [18]. Therefore, our setup combines these two functionalities, Aerie-style user-space file system management, and GPU management, into one single GPU4FS process.

4.3.2 Command Buffer and Inter-Process Communication

With the layout of a GPU and the way that GPU4FS is set up, the only feasible way for inter-process and inter-device communication is shared memory. GPU drivers use ring buffers as command buffers, with a insertion and a completion pointer. Each entry of this primary ring buffers contains both a command and possibly a reference to more data [17]. The GPU4FS buffer is setup in a similar way, but also contains individual completions for each entry as the operations can vary massively in both time and urgency. Also, if commands are handled in an out of order fashion, the requesting process might be able to continue with some work out of order, too. The exact layout of the GPU4FS command buffer as implemented in the demonstrator can be found in Section 5.5. During the implementation we found the exact contents of the commands to change with the implementation, but the basic idea stayed the same.

The main goal of GPU4FS is to reduce CPU time as much as possible. Therefore, the required CPU effort for command buffer prepation should be minimized. For this reason, the commands are quite high-level and as much work as possible is pushed to the GPU. For example, instead of letting the calling process work out the inode of a file, the path is pushed to GPU directly. This also has a latency benefit, because the GPU can directly read from storage and follow the directory structure. The CPU would have to request each directory from the GPU individually.

For GPU4FS, we expect to use around one GPU command per POSIX call. The requesting process can `read()` and `write()` files from and to the file system cache, create and remove files and folders and so on. Since we are not limited to the POSIX specification, some optimization is possible: For example, a command that creates a file with a given content can additionally be supported. If the process opens a file on the GPU4FS drive for the first time, however, a connection to the GPU4FS management process and to the GPU needs to be established. In this case, we expect to use more than one GPU command in addition to the CPU work.

4.3.3 Writing to Disk

In a file system, two types of data can be written: Metadata like folders, file block pointers and file names, and actual opaque content in files. GPU4FS does not interpret the content and returns it as written, metadata, however, is processed by the file system and will not be interpreted by the users. Also, metadata is usually small, while file content data can be quite large. Therefore, the paths for a metadata update and a content write differ slightly, as can be seen in Figure 4.5.

1. The process queues a write request. The request contains all necessary information, including the open file descriptor or the path depending on the interface used. It also contains a pointer to a buffer inside the shared memory region. This buffer contains the actual information to be written.

   In case of a metadata update, no file content data is needed.

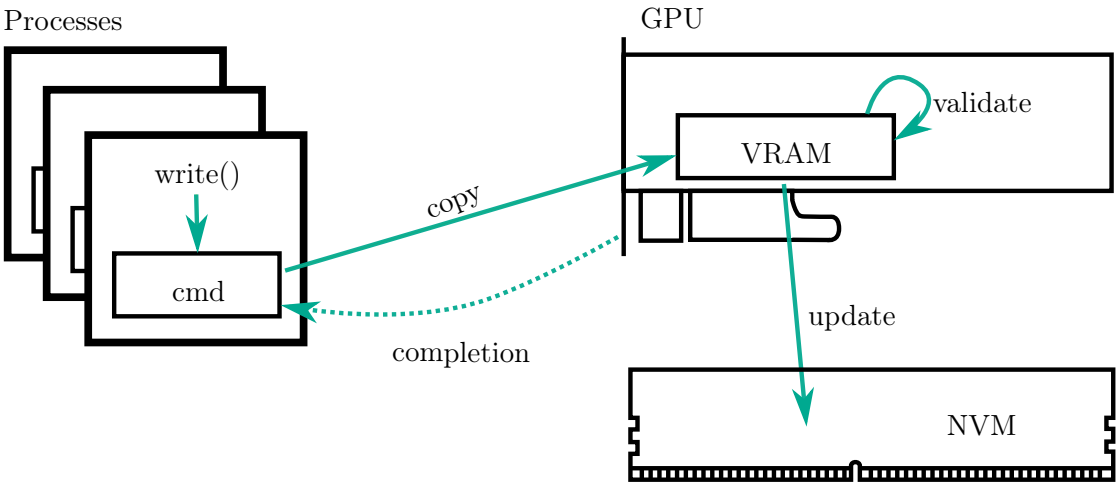


Figure 4.5: The write path in GPU4FS. The CPU queues the command and the data in its command buffer. The GPU copies it to VRAM, validates it, and updates the NVM accordingly. After this process is finished, the GPU signals the completion in the command buffer.

2. The GPU reads the request, copies it to VRAM and verifies it there. The copy operation is needed as otherwise a malicious process could modify data that has been marked sane before.

3. The GPU handles the request. It loads the file system data either from NVM or cached data from VRAM, allocates pages as needed and updates inodes and directories. It also adds the operations to the journal.

4. When the GPU finishes the command, it sets the completion flag in the command buffer. The process now knows that all data is successfully written to disk.

4.3.4 Reading from Disk

In other NVM file systems like Aerie [18] or SplitFS [32], the CPU reads directly from disk. Indeed, prior work [31] shows that the read performance of Intel Optane NVM memory is a lot better than NVMe SSDs. It is still lacking compared to DRAM, however. For this reason, caching data in DRAM can be a performance optimization if data is read multiple times.

In GPU4FS, we use the GPU to load data from storage, with the CPU only queuing a read command. The GPU can fetch data to VRAM and then forward it to the DRAM space of the requesting process. This can either be done directly into the shared memory buffer, or into shared file system caches as described in Section 4.3.8. If the data is read back privately to the shared memory buffer of the process, no mapping operation involving the kernel needs to started. For private mappings, this setup therefore is beneficial as compared to shared file system caches. This process can be seen in Figure 4.6.

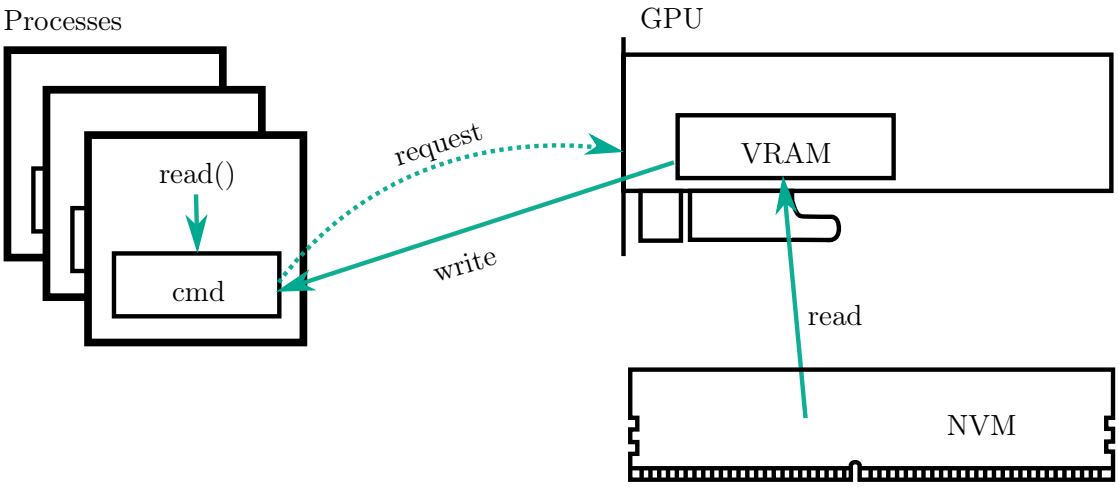


Figure 4.6: The private read path in GPU4FS. The CPU queues the command, which the GPU parses and executes afterwards. The GPU loads the data from NVM into its VRAM buffer, and writes it into a reserved space inside the shared command buffer.

With this design, the GPU has a central position in which it can synchronize, order and arbitrate read and write requests, even before data is written to disk. With this, VRAM can also act as a file system cache: the GPU can serve read requests from data it has previously loaded from NVM or that has been written by a process, without going to NVM.

4.3.5 Memory Allocation

When new data, like an inode or a file, is written, blocks need to be individually assigned to store that data in. To avoid overwriting data, a centralized block memory allocator for the storage medium needs to be used. Every file system has to solve this problem using some kind of block allocator [24]. In case of GPU4FS, this memory allocator needs to run on the GPU, which is an inherently parallel device.

Memory Allocation is an inherently serialized task, however: each request accesses the same large pool and each block can only be allocated once. Luckily, with the recursive block splitting strategy outlined in Section 4.2.1, as soon as a block is split up the allocator offers multiple pages that can be handed out without much serialization. The arbitration can be implemented easily and scalably using an atomic index. Serialization using locks is only needed when the preallocated pages run out. In this case, one larger page needs to be split into several smaller ones. To avoid fragmentation, as few pages as possible should be split.

Parallel Allocation To manage the simple case without overflows, only the number of elements (`size`), the atomic `index`, and a pointer to the list of block pointers (`data`)

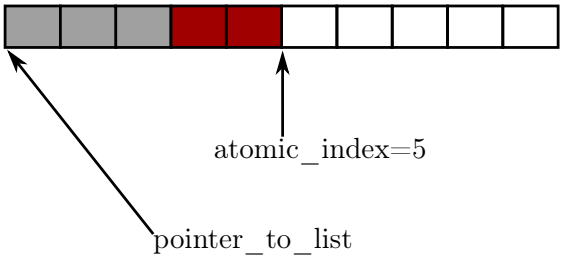


Figure 4.7: Atomic allocation of two block pointers (red). After the increase, the new index equals five, and block pointers with index three and four are allocated. The grey blocks were already allocated before, and the white blocks can be used by a another allocation request.

is needed, as illustrated in Figure 4.7. To add further lists, we need a lock for the serialization, and a pointer to the next such list. Sadly, we cannot clean up the memory when we allocate the new list as other processes might still be using the data. This means that as long as other processes are running, we can only allocate more memory to put these data structures in, but we can never free them. Therefore, some kind of stop-the-world garbage collector [12] is needed.

Garbage Collection To offer memory to the allocator, and implement the garbage collection, two equally sized, large buffers are used, the allocator and the cleanup buffer. The first offers memory which is allocated with the same parallel atomic allocation strategy. Newly created lists are written into this memory and referenced using pointers from the allocator data structure. When the allocator buffer inevitebly overflows, a cancellation is signaled to each process. At this point, allocating processes might still hold some pages but have not finished their requests. The block pointers to these pages are written back into the cleanup buffer. The number of allocated pages in flight is unknown, but the size of the allocation buffer is an upper limit. Therefore, a cleanup buffer with at least the size of the allocation buffer is definitely sufficient.

All processes signal completion of their writeback operation, and the last process to do so with an atomic decrease to zero is selected to reconstruct the allocator: Block pointers are type-tagged, so they can be copied from the cleanup buffer into new allocation lists in the allocation buffers. The process then creates the new allocation list headers, and finally resets the indices of both the allocation and the cleanup buffer.

4.3.6 Locking and Synchronization

Previous user space file systems took an easy approach into file and folder synchronization [18], [23]: A process wanting to operate on them got a unique lock, and these locks serialized the access to the file. In a POSIX environment, this is uncommon: Files can be opened and modified by multiple processes, with the kernel implementing synchronization.

In GPU4FS, the ordering is implemented on the GPU: Each process can put operations into its own command buffer in parallel and independently of other processes. When the

GPU parses these operations, it will order them using internal buffers and commit them to the drive. With this serialization, several behaviors can be considered, depending on implementation and user space requests: This means for example that with two processes both queueing the creation of file `foo` in folder `bar` both might succeed and return the same inode, or one of them might succeed and the other return an error. This also means that the GPU will arbitrate two write calls using the same file descriptor and decide on an order.

4.3.7 Kernel Communication

For a user space file system, kernel interaction is usually neither needed nor wanted. It is still sometimes necessary, for example at connection startup and shutdown, but crucially also at runtime when using caching, see below, and when modifying buffers. The latter can be needed if a process wishes to extend its shared command buffer size for better communicaton. In all cases, the GPU needs to be notified of such changes, so using the preexisting command buffer structure makes sense. This only solves the problem of getting the mapping information to the GPU, though.

To communicate to the kernel, the mapping information takes two more steps: The GPU is managed and programmed by a user space process, which can get the information by using a command buffer, and use system calls to talk to the kernel. This command buffer is special in that it is filled by the GPU and parsed by the managed process, and the management process is also responsible for flagging the completion.

The other issue is one of priviledge: The management process is asking the kernel to remap pages in a different process, something that should not be possible without further communication. The management process therefore needs special permission to enable this behaviour. Prior work, Aerie as described in Section 3.1.2, used a special Linux kernel module for this purpose.

In Figure 4.1, we present the whole process for an `mmap()` call.

4.3.8 File System Caches

Why Caches are Desirable The POSIX `mmap()` function allows multiple processes to map the same file region and work on them in a shared memory fashion. In a normal file system, this is achieved by mapping the file system caches into the processes so that they can directly access that memory. In a normal file system for block storage, these caches are necessary to mediate between the byte-addressable data written from processes, and the blocks in which storage need to be accessed.

File system caches can also increase performance. Even though reading from Optane is much faster than writing, DRAM is still faster [31]. If the same memory is read multiple times, caching can be beneficial. A GPU offers VRAM for an opportunity to additionally cache data that is evicted from the CPU. This allows for smaller buffers in main memory, freeing up memory for actual tasks.

Caches in GPU4FS In GPU4FS, some of these caches are managed by the priviledged process and the GPU in parallel, as shown in Figure 4.1. After a request comes from a process, the GPU sends a mapping request to the kernel as described in Section 4.3.7. While the kernel is processing, the GPU can preload the contents of the cache into its VRAM buffer. As soon as the mapping completion is signaled, the GPU can copy the data from VRAM into the file system caches in RAM. The initial copy to VRAM is an optimization in multiple regards: Firstly, having data in VRAM avoids the penalty of slower NVM as VRAM bandwidth is usually orders of magnitude larger [31] [61]. Secondly, the GPU can reorder the data from NVM to VRAM, so it is ordered in VRAM sequentially. This avoids walking the file system tree when eventually copying to DRAM.

4.3.9 Journaling and Consistency

A GPU in the write path leads to some latency between command submission and completion on disk. Journaling [24] and log structuring [2] both rely on few pointer updates, which is badly matched to a GPU. To add to this problem, the VRAM adds another level of cache which might not get flushed before a crash occures. Therefore, crash consistency is an important topic for a GPU-accelerated file system.

To cater to a GPU, we envision a parallel log structure, in cooperation with the memory allocator. A page is only marked as used if it is completely filled and committed to the file system. If a crash occures before the commit, the memory block will be garbage collected and returned into the pool of free pages. GPU4FS can then achieve crash consistency by copying the whole tree and only commit very late into this process. Even though some unneeded writing might occur, we expect the parallelization to be beneficial overall.

4.4 Discussion

We designed GPU4FS to be well-matched to the problem, namely to reduce the CPU usage for general-purpose Intel Optane file systems using a GPU. The different features of the file system can be followed out of the problem statement.

GPU4FS is tailored towards Intel Optane memory. Therefore, it uses 64 bit pointers, and 8 B-aligned data structures. Additionally, the block sizes of 4 kB, 2 MB, 1 GB, and either 128 B or 256 B follow directly from the MMU design and the memory allocation strategy.

To be generally usable, GPU4FS adheres to the POSIX specification in the inode. It also offers space for extensions to accomodate future use cases.

To reduce the CPU utilization, GPU4FS uses a GPU for most tasks. Additionally, we decide to implement a user space file system to further reduce the stress on the CPU.

The design of the inter-process communication interface using shared memory follows from the decision to use a GPU. The GPU also necessitates the parallel memory allocator, the kernel communication strategy using an intermediate process, and the parallel journal.

5 Implementation

This chapter details the implementation of GPU4FS and its supporting software for the evaluation in this thesis. We first implement a small CPU file system, then patched the AMDGPU [44] and RADV [43] driver stack to accomodate our work. After that, we used the patched driver for bandwidth tests and a small GPU-side file system write accelerator.

The current implementation is targeted for x86-64 Linux systems with modern AMD GPUs, and for Vulkan as the graphics API. GPU4FS uses the RADV Linux Vulkan driver with slight modifications, as described in Section 5.2. To implement the shaders, we use GLSL and Googles `glslc` GLSL to SPIR-V compiler. The CPU-side implementation uses C++-20 and is compiled using `g++`.

The GPU4FS implementation is tested on three systems

- a Lenovo T14 AMD Gen2 laptop using an AMD R7 4750G processor with an integrated Vega 7 GPU,
- a desktop PC with an AMD R9 5900X processor and a dedicated AMD RX 6800XT GPU,
- and a server with two Intel Xeon Silver 4215 CPUs, a dedicated AMD RX 6600XT GPU, and four DIMMs with a combined 512 GB of Intel Optane Memory. This is also the machine which is used for the benchmarks in our evaluation in Chapter 6.

5.1 GPU4FS on CPU

The CPU implementation of GPU4FS follows the design outlined in Chapter 4. To distinguish from the GPU-based implementation, the CPU implementation is called CPU4FS. It supports the major features, including unit tests for those.

- Blocks and block pointers: CPU4FS supports all types of block pointers, for inodes, small, large, and huge pages. It can decode the bit patterns and supports indirection using the respective flag in the pointer. CPU4FS fully supports walking a file as referenced from an inode. It can also skip invalid pointers.
- Inodes: CPU4FS supports full storage of the inode, customizable at compile time. It can encode and decode the times to both microsecond and nanosecond precision.
- Extensibility: The extensibility feature uses the same code path as the normal block pointers. The groundwork is therefore done, but no additional features are implemented.

- Directories: Directory listings and insertion into directories are implemented for linked list directory entries.
- Superblock: The current CPU4FS implementation does not look for a superblock, instead expecting a root directory inode at offset zero.

Additionally, CPU4FS contains a mostly complete implementation for the GPU memory allocation strategy, which is used for GPU debugging purposes. Because it is supposed to be easily portable to the GPU, the implementation does not use high-level serialization primitives, and instead relies on direct atomics. The prototype demonstrates the difficulties in this programming style, and is of limited reliability. The demonstrators both for CPU4FS and GPU4FS therefore use hardcoded offsets or basic sequential allocators.

5.2 GPU NVM Passthrough

With the CPU-side file system running, the next step is to write to the target storage system. For this thesis, the main target is NVM memory, but for debugging reasons normal on-disk files can also be used as the underlying device. In both cases, NVM and files, the procedure is the same:

1. Make the memory visible in the process context and accessible via virtual memory.
2. Ask the kernel to map the matching physical pages to the GPU via the GART.

Making the Memory Visible For both NVM and normal files, the procedure is to use the mmap system call. `mmap()` maps a file into main memory. For normal files, the data is copied into the file system cache, and this cache is mapped into the address space of the user process. Copying into a cache can be avoided for NVM memory as it supports being mapped directly.

Using the GART to Make the Memory GPU-Visible On AMD GPUs as used for this implementation, the GPUs can already access main memory via the Graphics Address Remapping Table (GART). The GART acts as a page table in the GPU physical device memory address space. An access into the GART address range is translated into a host memory address. The GPU then requests this address from main memory, usually via PCIe. The IOMMU, if enabled, then acts as another page table and translates the host device memory address into a host physical memory address. This physical memory address can also be mapped into the CPUs process's address space via the CPU's MMU. The different address spaces can be seen in Figure 5.1.

Some kind of mapping is required for normal API execution, as data needs to get to the GPU in some way. This can either be done by mapping CPU memory to the GPU, GPU memory to the CPU, or, using some non-memory-based protocol. For performance and programming reasons, all major modern Graphics and Compute APIs require some amount of shared memory. In Vulkan for example, there is the triple of functions `vkCreateBuffer()`, `vkAllocateMemory()`, and `vkMapMemory()` [48]. These functions

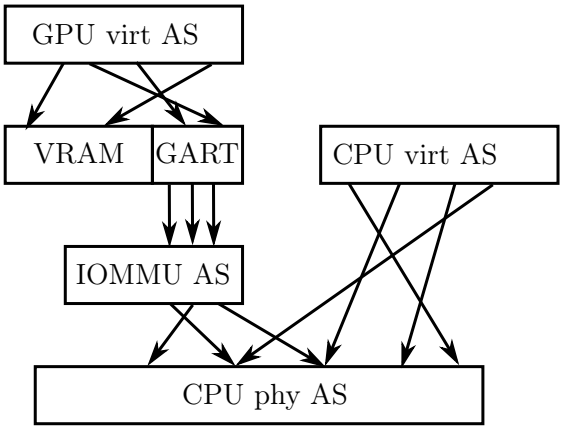


Figure 5.1: GPU shared memory: The GPU has its virtual address space (virt AS), which maps to VRAM and to GART. GART addresses are forwarded to the IOMMU, which translates it to the CPU's physical address space (phy AS). Similarly, the CPU can translate virtual to physical addresses. If the mappings overlap in CPU physical memory, the memory is shared.

first tell the API to create a buffer with a given size and usage in `vkCreateBuffer()` and tell the API about its type and residency in `vkAllocateMemory()`. If the buffer resides on the CPU, the API configures it to be mapped to be GPU-visible. Also, if the buffer resides on the CPU, `vkMapMemory()` offers a CPU pointer to the underlying memory so it can be filled and later used by the GPU.

The Vulkan API also offers an extension called `VK_EXT_external_memory_host`, which is supposed to map memory from other sources to the GPU. The RADV driver does support it [47], but with some limitations: Crucially, it can not map non-anonymous memory, for example from files or NVM, to the GPU without patching. The problem is buried in the call stack of the `vkAllocateMemory()` function. The call is as follows:

1. The function call gets forwarded into the GPU-agnostic libvulkan.

2. libvulkan detects that the selected device is an AMD one and defers the call into the RADV Vulkan driver for AMD GPUs.

3. RADV maps all Vulkan objects to objects used by the common Mesa and Gallium implementations. The call does not need heavy driver support like a shader compile, but only modifies buffer objects (BOs). Therefore, RADV calls into the AMDGPU DRM driver to make kernel-level modifications.

4. Inside DRM, the `amdgpu_create_bo_from_user_mem()` function is called. This function is essentially a wrapper for the `amdgpu_bo_create` kernel-level DRM call. This kernel-level call creates a buffer object for a given memory range, which is the functionality we need. Crucially, in an unpatched user space DRM driver, multiple flags are passed to the kernel, one of them being:

`args.flags = AMDGPU_GEM_USERPTR_ANONONLY | ...` [1] [46].
This flag requires the mapped memory to be anonymous, so we removed the flag. This allows the usage of the `VK_EXT_external_memory_host` extension with shared memory. We do not know why this flag was set in the first place, but assume the reason to be some detail of the extension's specification.

5. The unmodified underlying linux kernel maps makes this memory visible in the GPUs GART.

This functionality is demonstrated using a simple Vulkan program, which draws a triangle from a buffer that resides inside a file. A separate process can open the file and modify data, which alters the triangle drawn to the screen, without any other communication to the first process.

5.3 GPU Command Buffer Structure

To implement a flexible and configurable GPU shader, a command buffer, as introduced in Section 4.3.2, is implemented. This section details the general command buffer, the commands for the tasks themselves are introduced in Section 5.4 and Section 5.5.

5.3.1 Command Buffer Layout

The command buffer manages the execution of different tasks on the GPU. The buffer is divided into three parts: metainfo, commands, and additional command data. Each command descriptor consists of a block of 16 numbers, each of them being 8 B in size, which results in a 128 B command. The command descriptors form a linked list, started by general execution metadata applicable to all command descriptors and terminated by a termination command descriptor. The workgroups parse the command buffer by walking the list, trying to atomically acquire each command descriptor, and executing the command if successful. Afterwards, the workgroups continue to follow the list until the termination is reached.

The command descriptors have a common prefix and postfix, and contain a command-specific payload. The prefix consists of two numbers, and is responsible for the queue management:

1. A type tag: Each different command is identified by a tag, which the shader uses to decide which code path to execute. Currently, five different types are specified:
    0. `MEMSET`: Sets the memory in the *to* buffer to a 8 B pattern p, with a byte length of l.
    1. `MEMCOPY`: Copies l bytes from the *from* buffer to the *to* buffer.
    2. `FILE_PATH`: Creates a file. This command has multiple parameters which are explained in Section 5.5.

[1] Commit `7c28f528309d15163678ac1a49e161e3b1692b50`, file `amdgpu/amdgpu_bo.c:581`

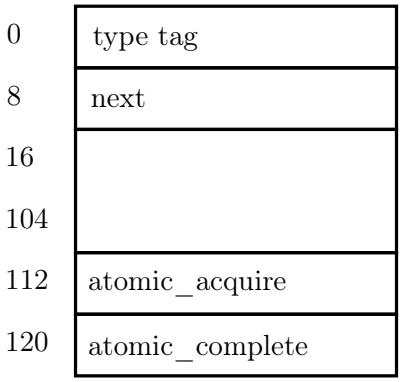


Figure 5.2: An empty command descriptor. The payload between offset 16 and 112 depends on the specific command type.

3. `META_INFO`: Contains metainfo applicable to all different commands. See below for more information.
4. `TERMINATE`: Signals the termination of the command queue.

2. A next offset: This contains the offset of the next command descriptor, from the beginning of the command buffer. This allows each individual workgroup to walk along the queue of commands until it reaches the termination.

The postfix also consists of two atomic numbers, and is responsible for the management of the atomics inside the command descriptor:

15. The acquisition counter: Each workgroup atomically increases this field by one when it reaches the command descriptor, and checks the return. If the returned value is zero, the workgroup acquired this command descriptor. Otherwise, the command descriptor is skipped and the next one in the list is tried.

16. The completion counter: After they successfully finished the task, each SIMD lane in the workgroup increases atomically increases this field by one. The command is considered complete as soon as this value equals the workgroup size.

An empty command buffer can be seen in Figure 5.2.

5.3.2 The Metadata Command

The command buffer starts with a set of meta information that is shared for all executions. The metadata block itself shares the layout of a command descriptor, including type, next, acquisition and completion fields. Of these four, the acquisition and next fields are unused. The metadata does not need to be executed, so an acquisition makes no sence, and the next field is unused as the execution starts by trying to acquire the successing block. The type tag is set by the CPU side to avoid mistakes. Lastly, the completion field

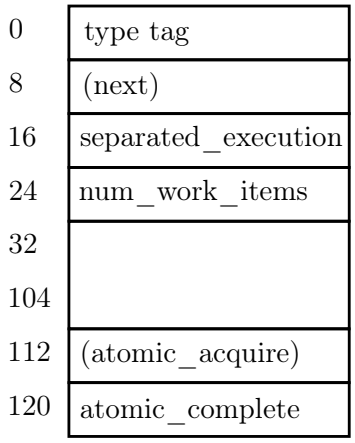


Figure 5.3: The meta data command descriptor. It contains one boolean flag signaling separated execution, and the number of SIMD lanes.

in metadata is increased by every SIMD lane that is exiting the shader, thus allowing to signal overall process termination.

The metadata contains two numbers as a payload: Firstly, the number of SIMD lanes that should be executed. All other SIMD lanes, and thus workgroups just terminate instantly, without signaling completion. This allows for an easy programming model inside both the shader and the CPU-side code as the number of SIMD lanes needs to be communicated to the shader at runtime. The other solution would be to recompile the shader each time, which is not feasible. Therefore, the CPU can just launch the maximum number of workgroups and rely on them to terminate immediately.

Secondly, the payload contains a boolean flag, which signals separated execution. For certain tasks, it makes sense to have all work groups work together, which is called combined execution. In this case, all workgroups that have not terminated in the step before execute the first command and then terminate, this time signaling completion. Separated execution means that each workgroup processes a different command. The combined mode is used by both the memcopy and memset commands, whereas the file creation uses separated execution. The whole meta data command descriptor can be seen in Figure 5.3.

5.3.3 Parsing the Command Buffer

In GPU programs, the programming model dictates that each SIMD lane is programmed individually, and the compiler bundles them together in the background. We found it difficult to synchronize the different unbundled SIMD lanes, and to enable communication between them. In a workgroup, synchronization is easy and fast communication is enabled via special shared memory. To avoid any race conditions inside one workgroup, a single SIMD lane is picked which parses the command buffer. The other SIMD lanes are disabled during the selection, or waiting for the executing SIMD lane inside a barrier. The code for the work group synchronization can be seen in Listing 1.

```
413 void dispatch_independent(int start_block) {
414     uint local = gl_LocalInvocationID.x;
415
416     int block = start_block;
417     while (true) {
418         int block_offset = block * block_size;
419         if (local == 0) {
420             do { block = acquire_new_block(block); } while (block != 0);
421         }
422
423         barrier();
424         block = new_block;
425         if (terminate != 0) return;
426
427         dispatch(new_block);
428     }
429 }
```

Listing 1: The GLSL code to synchronize the selection of the next command descriptor inside the workgroup. `new_block` and `terminate` are both workgroup-local variables to communicate the results inside the workgroup. `barrier()` is a barrier that releases only when all SIMD lanes in the workgroup have reached it. The main part is the `do while`-loop that follows the linked list until a command descriptor is found. The outer loop is used to work on command descriptors until the termination is reached. `dispatch()` selects the correct code to actually execute the command.

Each workgroup selects its representative using the local workgroup id, and each representative calls into `acquire_new_block()`, as seen in Listing 2. This function is responsible to acquire the block or to figure out the next block that can be tried. The first step is to check whether the block signals a termination by checking the command descriptor type of the new block. If it is not a termination, the function tries to acquire the block by using an `atomicAdd()` of one onto the acquire field of the command descriptor. If the return value is zero, the block is considered acquired and zero is returned to signal success. In both cases, termination and successful block acquisition, a shared variable is set to communicate the result to the rest of the workgroup.

If neither the acquisition nor the termination returned, the function has the job of figuring out the next command descriptor. There is a possible race here, though: Command descriptors might be written in parallel to GPU execution, so a next element in the list might not yet be available, which is signaled through a value of zero in the next field. Zero can be used to signal "not a valid block," as this block offset is reserved for overall execution metadata and can never contain a valid command descriptor. For

this reason, the next field is repeatedly polled in an atomic fashion, until a value is stored on the CPU. The CPU process atomically stores a new offset, which will reach memory eventually and will terminate the loop. In the future, a different GPU process could also store command descriptors into this work queue. This would allow for communication between GPU processes.

A notable observation are the various casts to `int`: GLSL offers only quite limited support for 64 bit numbers, in particular, GLSL cannot use such numbers as indices into arrays. The shader is therefore filled with such explicit casts.

5.4 Efficiently Writing NVM with the GPU

The next step is to test the write performance of NVM to disk. After demonstrating that the GPU can successfully read both files and NVM, we then benchmark the write performance to NVM. For this, we implement a Vulkan compute shader that copies file buffers. On the CPU-side, we use a program with three tasks: Firstly, the program establishes a Vulkan context for the compute shader used. Afterwards, it maps the source and target file buffers, and makes them available to the GPU. As the last step, the program configures the command buffer and the respective memory region.

In addition to the general structure described before, the command buffer contains two more elements: the number of bytes to copy, and the number of SIMD lanes. The `memcopy` command is using combined execution as described in Section 5.3.2. To be prepared to only use a subset of all workgroups, the number of SIMD lanes to use for the copy operation is stored in the command descriptor, too. The `memcopy` command buffer can be seen in Figure 5.4

In the GPU shader, copying data from one buffer offset to another is quite common, so we implement a `copy_helper()` that is also used for file creation. The offsets are not given in bytes, but in factors of eight byte increments as GLSL does not support reinterpret casts of buffer pointers, and using eight byte accesses makes the code both easier and faster. For that reason, a lot of divisions by the constant `sizeof_int64_t` happen, as the CPU communicates in byte offsets. This conversation is done on the GPU as the POSIX CPU interfaces communicate sizes in Byte [25], and this division can be moved to the GPU.

To share the copy functionality with the file system implementation, a helper function called `copy_helper()` is used. It takes offsets into the from and to buffers as well as a size. To be more easily usable, it also takes a buffer id parameter for both the from and the to buffer. The exact parameter list is shown in Listing 3.

In addition than memcopy, memset is also implemented. A memset command descriptor has the same parameters as memcopy, but adds a 64 bit pattern that is used to fill the target. The memset command works similar to memcopy, with the exception that a user-provided pattern is written instead of data loaded from memory.

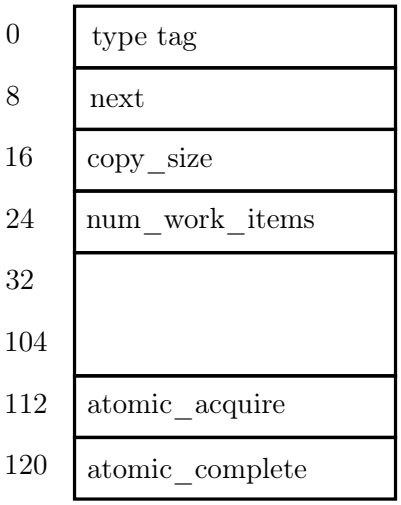


Figure 5.4: The `memcopy` command descriptor. It, too, contains the number of work items, in addition to the value set in the metadata command descriptor. This is used to test separated execution as described in Section 5.3.2. Additionally, the `copy_size` represents the number of bytes to copy.

5.5 GPU File System

As stated before, we use the same shader for memcopy, memset, and file write, and the behavior is selected by using a command descriptor in the command buffer. The entire implementation is tested by queueing a set of file writes, timing the completion and then checking that all files were written correctly. This implementation uses CPU4FS to interpret the pattern on disk.

File Write Command Descriptor The command for the writing files contains three types of numbers in addition to a normal command descriptor: source offsets into the command buffer, target positions on the drive, and meta information. The whole command descriptor can be seen in Figure 5.5. In detail, each of the fields in the descriptor has the following purpose:

2. The file size, to determine how much needs to be copied.
3. Reserved for the number of SIMD lanes for this file, but currently unused, see below.
4. The offset of the file data in the command buffer, to know where the data is coming from.
5. The length of the filename, to allow for compressed directory storage.
6. The offset of the file name in the command buffer.
7. The position of the directory to be written to on the drive.

8. The offset of the inode in the command buffer. The information in the inode is copied to the drive directly, the GPU only updates the respective pointers.
9. The target position of the new inode on the drive.
10. The target position of the new file on the drive. The file is written sequentially.

With a progressing implementation, two fields can be removed: the target positions of file and inode. Currently, the block allocator, see Section 4.3.5, is not working reliably. When it is ready, the GPU task can allocate these blocks itself, and no preallocation is be needed. Fixing the memory allocator is future work.

A third parameter is currently unused, but reserved for future use: the parameter for the number of SIMD lanes. It allows that large or urgent files can allocate more SIMD lanes. Currently, the number of SIMD lanes per file is hardcoded, and the total number of SIMD lanes is programmed in the metadata block.

Main Functionality To write a file to disk, the GPU has to fulfill four steps:

1. Writing the file data to disk,
2. writing the inode to disk,
3. setting the block pointers in the inode, and
4. updating the directory.

These steps can be executed in any order and independent of each other, although it is easier if the inode is written first and the block pointers afterwards. This is because the inode can be copied as a whole without overwriting the pointers that would have been set already. Writing the inode and the file data each requires a simple call to the `copy_helper()` function, as shown in Listing 4 as exemplarily shown for the inode. Given that most of the time is spent in the file data copy operation, we do not exploit further parallelism here and handle one file with one workgroup sequentially.

The remaining tasks, i.e., directory update and file pointer write, are both more involved and are separated into their own functions.

Directory Updates Updating the directory takes a few steps:

Firstly, the whole inode is locked, as can be seen in Listing 5. The unused extension pointer in the directory's inode is used for the locking. To avoid a race condition, only one SIMD lane is picked for each workgroup. This lane then `atomicAdd()`s that area by one, trying to aquire the spinlock. As soon as zero is returned, the lock is considered held. At the end of the critical section, as soon as all SIMD lanes in the workgroup have finished the update, the lock is freed by writing zero to that disk area.

After that, the directory inode is read and the storage area for the directory contents is found, with the implementation shown in Listing 6. The code currently assumes that the whole directory content fits into one page, so only the first inode is loaded. To get the

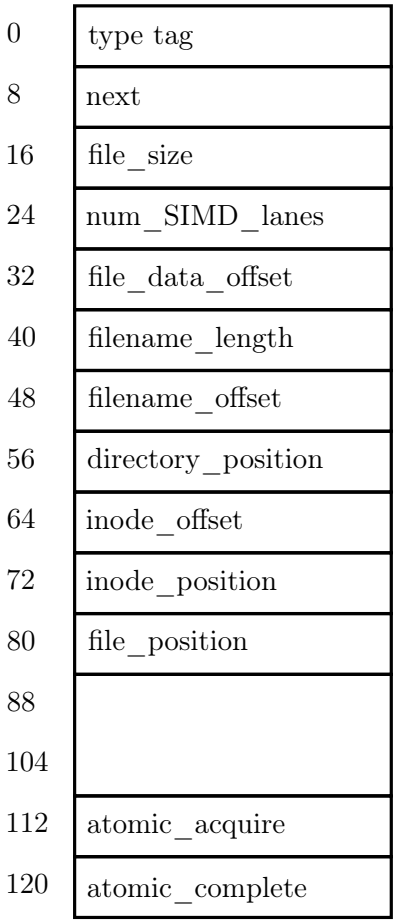


Figure 5.5: The file writing command descriptor. In addition to an empty command descriptor, several offsets into the command buffer and positions on drive are transmitted to configure which data is copied from DRAM to NVM. The file size and file name length control the amount of data to be copied. `num_SIMD_lanes` is currently unused, but added as a preparation for a future feature, as described in Section 5.5.

correct offset, the bit flags are removed, and the file queue is parsed. By convention, the first directory entry starts at offset zero. The shader then walks the next pointers until one of them contains zero. The previously found offset therefore is the last element in the directory entry list. All SIMD lanes in the workgroup walk the directory independently. This is not an issue as most SIMD lanes will work in lockstep either way, further, we can rely on the caches of the GPU to accelerate the accesses. Also, this avoids communication between the SIMD lanes which makes the code easier to maintain. At this point, a GLSL barrier is inserted to make sure that all SIMD lanes found the end before we start to modify the contents.

Currently, GPU4FS does not support file removals from directories. Therefore, we can assume that the last entry we found is also the last entry on disk and the next entry is inserted directly after it. For this reason, the offset that was just found can be extended by the size of the last directory entry: Eight bytes for the inode block pointer, eight bytes for the next pointer, and string length plus one byte for the string, rounded up to an alignment of eight bytes. After the new offset is found, three values are written: The block pointer for the new inode, the next pointer in the current directory entry is set to 0, and the next pointer in the last entry in the list is set to the correct new offset.

The last step before releasing the spinlock is to write the new file name and its length. The length is loaded from the command buffer, but with the packed storage, see Section 4.2.3, the stores become misaligned. The GLSL programming language does not allow for reinterpreting casts or misaligned stores, so instead the data is aligned by loading the aligned string from the command buffer and shifting appropriately. At this point, the system's endianness, little endian, needs to be taken into consideration. To insert the string's length at the first byte, it has to be stored in the least significant eight bits. The remaining bits are filled by shifting the loaded numbers, which contain the actual strings, appropiately. The code for this can be seen in Listing 7.

Block Pointer Writes The last step is to write the block pointers in the inode. Each inode has space for seven block pointers, each pointing to either a 4 kB, 2 MB, or 1 GB block. This provides for a reach of 28 kB, 14 MB, or 7 GB. A different combination of blocks is also possible. Alternatively, it could access seven indirect blocks. Currently, the block memory allocator is unreliable, so each inode sits in an otherwise unused, pre-allocated 4 kB page. This means that further inode-sized blocks can be allocated there, e.g., for indirect blocks. We use this space to allocate up to seven inode-sized indirect blocks for a total of 896 B or 112 direct pointers.

To write these pointers, the first step is to figure out the required amount of blocks. For memory mapping reasons, inode-sized pages make no sense here, so the size is rounded up to be 4 kB-aligned. The number is then split up to get the number of each page size needed. The shader loops through the number of pages needed, and then writes them to disk, together with their respective bit flags.

For the sake of implementation simplicity, all files are currently written using indirect pointers, even if the number of pointers would fit into the inode. To build a valid block, all unneeded pointers in a referenced block need to be marked invalid. Therefore, the

shader sets the pointers to zero. As a last step, the indirection pointers are set in the inode: The offset is computed, and the pointers are set as valid and indirect, together with the type.

5.6 Lessons Learned

During the implementation, several hurdles were found that hindered the progress. Mostly these hurdles can be found in the GPU programming tooling and description.

The most time-consuming part was the mapping the NVM to the GPU. Even though it is completely supported by the kernel and works with minor modifications in the user space drivers, only very little documentation is available that hints to this feature being easily available. To find the patch described in Section 5.2, nearly a month of stepping through instrumented driver code was necessary. The drivers also contain dynamically linked shared objects, the type of which can be decided at runtime. Also, to avoid issues with dynamic loading, the drivers aggressively strip symbol information. Both features hamper the usability of CPU-side debuggers.

On the GPU, debugging is even harder, and the setup of toolchains like RenderDoc is complicated. For debugging and printing, we mapped a file with the established code path and wrote all information we want from the GPU to the file. Given that we did not use no high-level primitives like `printf()`, the results had to be interpreted using a hexeditor. To use printf, we would have needed external libraries, which we deemed to be more effort than hexeditor-debugging. Indeed, this style of debugging felt natural relatively quickly.

Also, the manual locking implemented in the shader and large worksets combined with slow NVM memory can both trigger kernel timeouts for the shader execution. This leads to a reset of the GPU, which also means losing all debugging information that might still be available in the GPU. As a workaround, during debugging we stored as much information as possible to the device early, so it was still available even if the GPU crashed.

```
380 /*!
381  * \brief try to acquire the new block
382  * \param block: where to start in the chain
383  * \return 0 if successful, next element to check otherwise
384  */
385 int acquire_new_block(int block) {
386     const uint next_command_offset    = 1;
387     const uint already_acquired_offset = 14;
388 
389     int block_offset = block * block_size;
390 
391     // check termination first
392     if (int(config.data[block_offset]) == 4) {
393         terminate = 1;
394         return 0;
395     }
396 
397     // try to acquire
398     int64_t result = atomicAdd(config.data[block_offset + already_acquired_offset], 1);
399     if (result == 0) {
400         new_block = block;
401         return 0;
402     }
403     // follow next chain
404     int64_t next = 0;
405     do {
406         // fancy way to load the variable, I guess :()
407         next = atomicAdd(config.data[block_offset + next_command_offset], 0) /
            ↪  sizeof_command;
408     } while (next == 0);
409 
410     return int(next);
411 }
```

Listing 2: The GLSL code to acquire a new block number `block`. First, the code will check whether it is a termination by comparing the command descriptor type to 4, set the termination flag and return if it is a termination. Otherwise, it will try to acquire the block by atomically increasing the acquire counter in the command descriptor. If the return is zero, the acquisition was successful and again a flag is set and returned. If this is also not successful, we spin on the `next` field until we get a non-zero block offset. As soon as that happens, we return the new, untested block.

```
125 void copy_helper(int from, int from_buffer, int to, int to_buffer, int size, int
    ↪  work_group_size, int global_offset) {
```

Listing 3: The copy helper takes seven parameters. The `size` contains the number of blocks that should be copied. `from` and `to` are the offsets into the buffers that are copied from and to, respectively. As with all indices and sizes, `size`, `from`, and `to` are in terms of 8 B indices, and need to be of type `int` or `uint` to work in GLSL. `from_buffer` and `to_buffer` contain the id of the buffer to be selected. A value of 0 represents the `from` file buffer, 1 the `to` file buffer, and 2 the command buffer. The `work_group_size` is the number of SIMD lanes that execute this request. The `global_offset` is the number of the lowest-numbered SIMD lane, and allows to use the function with either all workgroups or only one workgroup.

```
348     const int inode_target_offset = int(config.data[block_offset +
        ↪  inode_position_offset] / sizeof_int64_t);
349     const int inode_source_offset = int(config.data[block_offset +
        ↪  file_inode_offset_offset]) / sizeof_int64_t;
350     copy_helper(inode_source_offset, 2, inode_target_offset, 1, int(INODE_SIZE /
        ↪  sizeof_int64_t),
351         int(work_group_size), int(base));
```

Listing 4: In the current implementation, the inode is simply copied from the command buffer to the disk. All offsets are loaded from the command descriptor in the buffer, casted to `int`, and copied to disk.

```
266     if (local == 0) {
267         int64_t atomic_acquire = 0;
268         do {
269             atomic_acquire = atomicAdd(nvm.to[dir_inode_position +
                ↪  dir_inode_lock_offset], 1);
270         } while (atomic_acquire != 0);
271     }
272
273     barrier();
```

```
317     barrier(); // end of modification, now unlock critical section
318     if (local == 0) nvm.to[dir_inode_position + dir_inode_lock_offset] = 0;
```

Listing 5: The first step is to lock the directory for an update, using a spinlock in the directory inode. One SIMD lane repeatedly tries to lock the spinlock, the others wait for completion in the barrier. After all SIMD lanes finished the directory update, the lock is released. This is done by a singular SIMD lane to avoid multiple unlocks.

```
278     int current_offset                      = 0;
279     while (bool(nvm.to[directory_target_offset + current_offset +
        ↪  directory_next_pointer_offset])) {
280         current_offset =
281             int(nvm.to[directory_target_offset + current_offset +
                ↪  directory_next_pointer_offset] / sizeof_int64_t);
282     }
283     barrier(); // please all find the same thing
```

Listing 6: Loop through the directory, each time testing the offset of the next directory entry. If the offset of the next element is zero, the last directory entry is found. To avoid that some SIMD lanes modify the directory before others are finished, a barrier is added at the end.

```
306     const int file_name_elements = (file_name_length + sizeof_int64_t) / sizeof_int64_t;
307     int64_t first_file_name_word = config.data[file_name_offset] << 8 |
        ↪  file_name_length;
308     nvm.to[next_directory_entry_drive_offset + directory_length_offset] =
        ↪  first_file_name_word;
309
310     if (local < file_name_elements && local != 0) {
311         int64_t first    = config.data[file_name_offset + local - 1];
312         int64_t second   = config.data[file_name_offset + local];
313         int64_t combined = (first >> (8 * (sizeof_int64_t - 1))) | (second << 8);
314         nvm.to[next_directory_entry_drive_offset + directory_length_offset + local] =
            ↪  combined;
315     }
```

Listing 7: The file name is written to disk. This part requires a lot of shifting, as the file name is stored in an aligned fashion in the command buffer, but the length byte is added at the start on disk. The first element is written by all SIMD lanes, so some overwriting is possible, but not harmful. For performance reasons, the rest is copied in parallel by different SIMD lanes. No loop is needed as a workgroup will always have at least 32 elements for hardware performance reasons. All 256 bytes (255 characters and one size byte) can be written in one go.

6 Evaluation

The aim of this thesis is to show that file system acceleration on a GPU reduces stress on the CPU while offering competitive file system performance. We evaluate our demonstrator in two parts: First, we discuss the results of simply writing to NVM, and whether we achieve competitive write performance. In the second part, we examine the file system regarding both write performance as well as CPU stress. Some of the early results also influenced the design and implementation of later features.

6.1 Test System

We evaluate the approach on our GPU Intel Optane system. It is equipped with

- Dual Intel(R) Xeon(R) Silver 4215 CPUs, operating at 2.5 GHz. It supports Intel I/OAT copy offloading.
- 128 GB of DDR4 at 2400 MT/s, distributed into 64 GB per CPU and eight 16 GB DIMMs, respectively.
- 512 GB of DDR4-socket-compatible Intel Optane memory at 2400 MT/s, distributed into 256 GB per CPU and four 128 GB DIMMs, respectively.
- a Samsung NVMe SSD,
- 3 Micron NVMe SSDs,
- and an AMD RX 6600XT GPU, from AMD's Navi 2 GPU generation, together with 8 GB of dedicated VRAM. The GPU is equipped with eight PCIe Gen4 lanes, but the CPUs only support PCIe Gen3. Therefore, the communication uses eight PCIe Gen3 lanes.

With its two CPUs, the system uses a NUMA configuration. All benchmarks shown avoid inter-processor communication bottlenecks by using the Optane DIMMs attached to the same CPU as the GPU. We tested an asymmetric configuration writing with the GPU to one CPU and the Optane DIMMs connected to the other. This showed slight degredations in performance, but with a high variance. We presume this to be caused by latency when writing across the inter-processor link, but did not study that effect further.

6.2 Memcopy

A file system needs to store the data from the processes to disk. This means that copying from main memory to NVM is an important part of each NVM file system and needs to provide high bandwidth as well as low latency. For one Intel Optane DIMM, the maximum throughput is at about $2\,\mathrm{Gbit\,s^{-1}}$. An interesting observation is that, under certain circumstances, more than one CPU thread is needed to satisfy the DIMM, but the bandwidth also stalls or even degrades when more than two threads are used [31].

For the GPU, the same characteristics are expected, with a peak bandwidth at a certain number of SIMD lanes, then a plateau and later a slight degredation as the number of SIMD lanes increases. The actual bandwidth can be seen in Figure 6.1 for different workgroup sizes of the shader presented in Section 5.4. We show bandwidth with a 32 SIMD lane workgroup as it matches the hardware [27], and 256 SIMD lane workgroup as it is used for the file system operation. As shown in the graphs, the workgroup size does not make a big difference. Smaller workgroups allow for more independent scheduling, and larger workgroups can provide local communication for more SIMD lanes. As a comparison, we also measure the copy bandwidth from DRAM to DRAM via the GPU. This plot shows four interesting results:

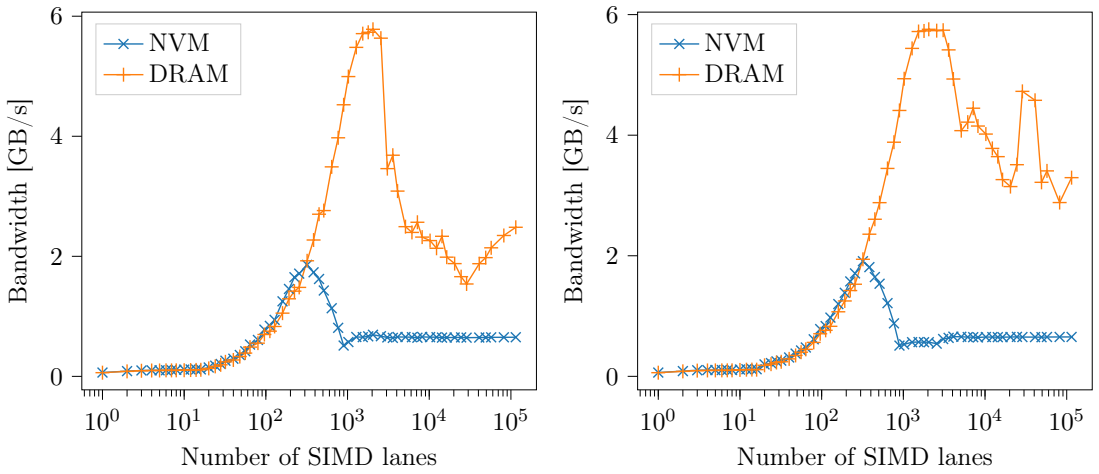
1. The GPU bandwidth to DRAM and NVM is very close until the maximum NVM bandwidth is reached. This suggests that up to peak NVM bandwidth, the shader is bottlenecked elsewhere, possibly waiting for read requests from DRAM.

2. the CPU bandwidth to NVM per number of threads is mostly flat, and the GPU bandwidth shows clear spikes. The peak performance with above $1.9\,\mathrm{GB\,s^{-1}}$ is reached at 320 SIMD lanes, after which the bandwidth quickly falls, to a minimum of about $0.5\,\mathrm{GB\,s^{-1}}$ at 896 SIMD lanes.

3. the GPU bandwidth to NVM reaches a plateau with more SIMD lanes at about a third of the maximum bandwidth ($0.65\,\mathrm{Gbit\,s^{-1}}$). This plateau is reached at 4096 SIMD lanes.

4. the bandwidth is mostly independent of the workgroup size, which allows for free selection of a workgroup that fits the task, as we will explain in the following paragraph.

To be practical, the GPU's copy bandwidth needs to be similar to the maximum that the CPU achieves. This is the case between 256 and 512 SIMD lanes, where the bandwidth is above 75% of the maximum bandwidth of CPUs to NVM. As described in Section 5.5, the different requests should be handled as parallel as possible. The bandwidth above 512 SIMD lanes shows a sharp decline, and 256 is the minimum for acceptable performance. The workgroup size can not be changed at runtime, therefore a size of 256 SIMD lanes per workgroup makes sense. Otherwise, either the bandwidth for single file writes or multiple parallel file writes would suffer. This also means that only two files can be handled in parallel with reasonable bandwidth for the general shader.



(a) workgroup size of 32 (b) workgroup size of 256

Figure 6.1: GPU write bandwidth to Intel Optane and DRAM, per number of SIMD lanes, for different workgroup sizes. The Optane bandwidth closely follows the DRAM bandwith up to the peak, where it breaks down quickly. DRAM shows similar behavior for higher number of SIMD lanes.

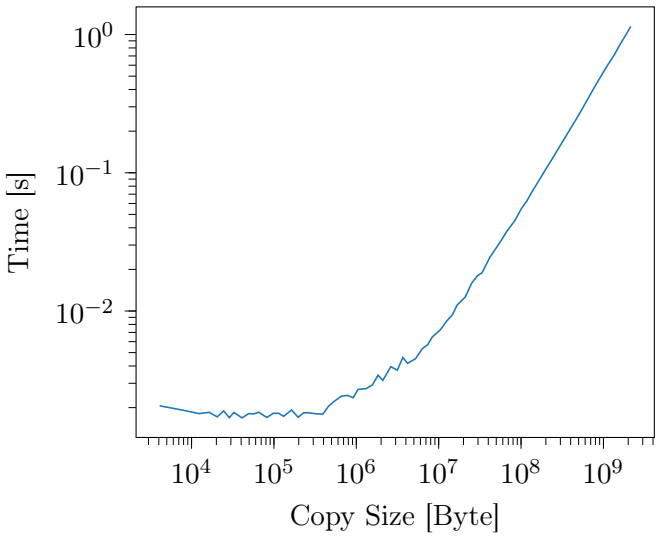


Figure 6.2: GPU write time to Intel Optane and DRAM, per data size. Uses 320 SIMD lanes with 32 work items per workgroups for maximum performance. The crossover point between startup latency and write latency is at about 2 MB.

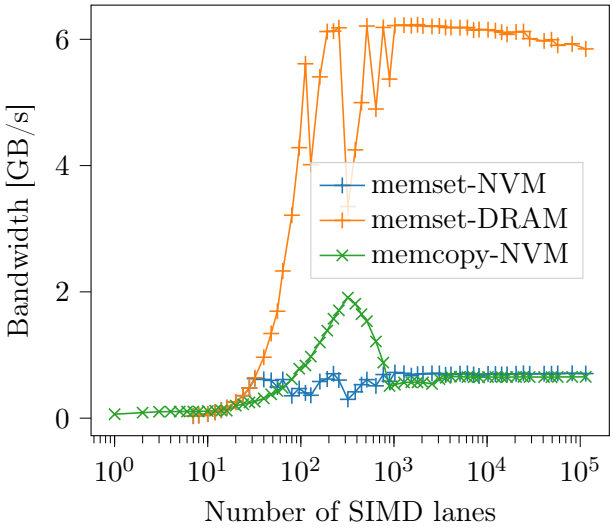


Figure 6.3: GPU memset bandwidth to Intel Optane and DRAM, per number of SIMD lanes, for different workgroup sizes. In addition, copy performance to NVM is shown. The main observations are that memset bandwidth is noisier, that the benchmark crashes when using only few SIMD lanes, and that it never reaches peak Optane performance.

The other relevant metric for a file system is latency. Programming the GPU, and transferring the data through different busses takes time, and the processes should not wait. Knowing the latency is also important in order to determine the limiting factors for the bandwidth. As can be seen in Figure 6.2, the crossover point is at about 2 MiB. The startup and teardown time can be estimated to about 1.5 ms, as compared to the store latency of CPU writes to NVM of about 100 ns [31].

6.3 Memset

We also implemented memset as a shader command. This command is used to write eight byte patterns to NVM or DRAM. Figure 6.3 visualizes the NVM write bandwidth of memset operations with different amounts of SIMD lanes. We expected a similar bandwidth curve as with memcopy, however, the peak bandwidth should be reached with slightly less SIMD lanes since memset operations only need to pass the PCIe bus once, as opposed to memcopy, which requires additional PCIe operations. Instead, the bandwidth never exceeds 0.75 Gbit s^{-1}. The bandwidth is also much less consistent around the area where memcopy shows the highest bandwidth, and it shows a slightly higher plateau bandwidth. Another interesting observation is that the shader hard-timeouted when using less than seven SIMD lanes, and is considerably slower than memcopy for less than 20 SIMD lanes.

6.4 File Write

To evaluate the file system behavior, a third command is implemented. This command can insert files into an existing folder. We measure write bandwidth, latency and CPU usage. We also compare the bandwidth result while while a multi-threaded benchmark is running on the CPU to validate the reported CPU usage.

6.4.1 Bandwidth and Latency

As described in Section 5.5, the demonstrator for GPU4FS currently only supports up to 112 inode pointers, which makes measurements in certain file size ranges impossible. Therefore, the measurements shown have gaps between the kilobyte domain (4 kB to 448 kB), the megabyte domain (2 MB to 224 MB), and the large domain (around 1 GB). To compensate, we show plots writing both a single file and multiple files in Figure 6.4. The bandwidth for all tests with more than one file line up well, so we can use this data to fill up the empty spaces. The single file write shows a quite different behavior. This can be attributed to the fact that only 256 instead of 512 SIMD lanes are used, which means that the expected bandwidth is slightly higher. This also explains why for a single file, the DRAM plot follows the NVM plot closely: both are limited by the GPU, not by the memory technology.

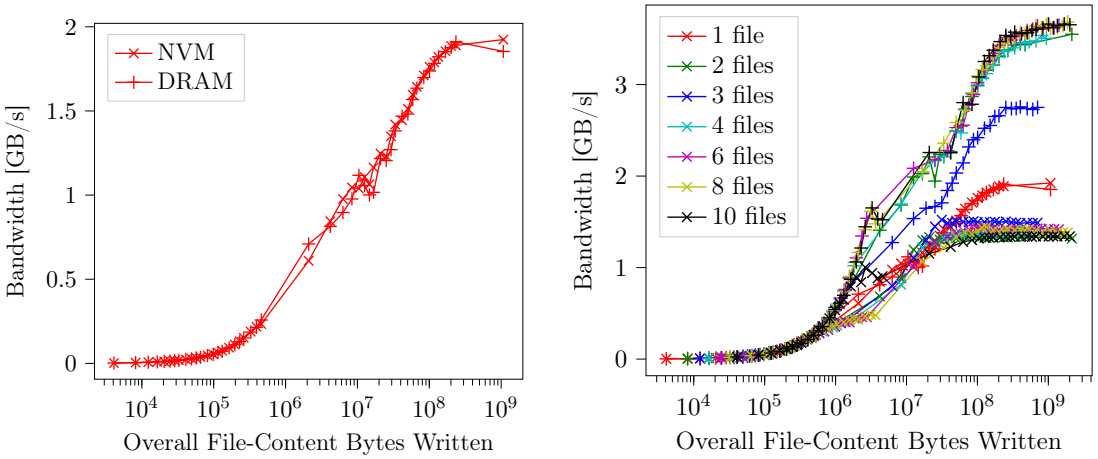
In the case of multi-file writes, a slower bandwidth of around $1.4\,\mathrm{GB\,s^{-1}}$ can be seen which seems to be caused by overhead in the NVM DIMM. With the memcopy command, a bandwidth of around $1.5\,\mathrm{GB\,s^{-1}}$ for 512 SIMD Lanes can be achieved, so the file system loses some but not much performance compared to the best case here. Interestingly, the write bandwidth for the single file case does not suffer from this slowdown, achieving a plateau of about $1.9\,\mathrm{GB\,s^{-1}}$. In comparison, the maximum memcopy bandwidth we measured is also around this figure, albeit at slightly higher SIMD lane counts. We expect the lock-read-modify-write-unlock pattern in the directory to be the cause for the main performance difference between memcopy and file writing.

One interesting case are the blue plots for three files: The bandwidth at larger file sizes for DRAM is consistently lower than for other multi-file writes, including two files, and the DRAM bandwidth is consistently higher: This can be explained by the way the GPU handles the files: firstly, it processes two files, achieving high bandwidth to DRAM and lower bandwidth to NVM. The third file is then handled by a single workgroup, which achieves lower bandwidth to DRAM and higher bandwidth to NVM.

In general, the performance in both the DRAM as well as the NVM case match the behavior shown for the memcopy case, which means that the file system overhead does not impact the bandwidth greatly.

The startup latency of a file system task for the GPU is similar to the latency of a memcopy operation, as can be seen in Figure 6.5. The main bottleneck seems to be the startup of the GPU, not the file system work. We expect smaller latency if the GPU environment, including the shaders, is already set up.

Similar to the memcopy results, we measure a crossover point for being bandwidth-limited versus being startup latency-limited of 2 MB of overall file content written. This



(a) Writing a Single File to NVM and DRAM (b) Writing 1, 2, 3, 4, 6, 8, and 10 files in parallel to NVM (x) and DRAM (+)

Figure 6.4: GPU file write bandwidth to Intel Optane (x) and DRAM (+), per file size and number of files. The bandwidth for one file is identical for Optane and DRAM. This implies that for a single file, the GPU is the bottleneck, not Optane.

is independent of single file size, even the test where we write 20 individual files shows that point at the mark of 100 kB, see Figure 6.5. We also measure very little difference between the latency of writing 1 up to 23 small, 4 kB files. We measure differences in the completion time being of about 5 %, but the results do not show a clear upward trajectory with more files. We assume the differences to be mostly noise.

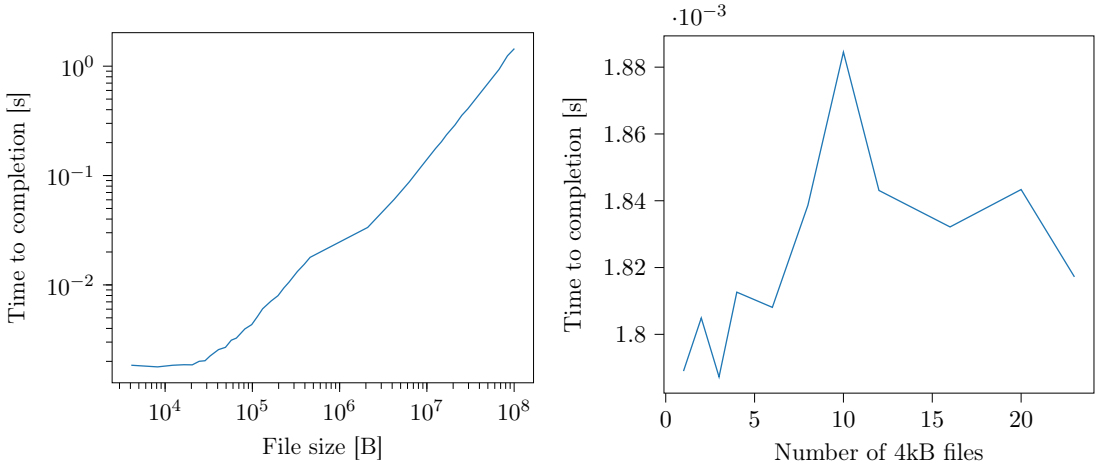
In conclusion, these observations closely match the results from the memcopy experiments. Given that a file write is mostly copying data around, this was expected, but the additional work in managing the file system does not show important overhead.

6.4.2 CPU Usage

The main goal of GPU4FS is to allow for high-performance parallel file-system implementations while relieving the CPU. In this section, we evaluate the stress GPU4FS puts on the CPU while running.

To evaluate, we run different file system tasks, either using GPU4FS or CPU-side file system tasks like `fio` [22]. We observe the reported CPU usage, and we run a parallel CPU benchmark to measure the slowdowns. In each case, we use `taskset 0xf` to limit both the benchmark process as well as the process using the file system to the first four cores. To avoid effects of the inter-processor interconnect, we make sure to use an Optane DIMM that is directly connected to the CPU that houses the first four cores and is connected to the GPU.

During execution of the GPU4FS file creation benchmark, the reported CPU usage varies between 5 % and 40 % of one core, and averages at 12 %. To compare, the reported



(a) Time to finish a 20 file write, per file size

(b) Latency of writing n files of 4 kB each.

Figure 6.5: GPU file write latency. Like in the memcopy case, the crossover between startup and bandwidth is at around 2 MB written as 20 times 100 kB files. The startup latency also dwarves the write time for multiple files.

CPU usage for memcopy is at only 2 % to 5 % of one core, and for memset it is even lower at 1 % to 2 %. We assume the usage figures to be caused by CPU side management requirements like formatting the drive with a clean file system, verifying the results and managing the GPU connection. The CPU overhead in the file system benchmark is larger than for memcopy and memset, which explains the difference in CPU usage. Plots for the CPU usage for memset, memcopy and 23 file writing can be seen in Figure 6.6. Each of the processes ran 50 iterations and touched about 1 GB of NVM. We also reconfigured the writing shader to use 5 workgroups with 64 elements each, which achieved over 1.8 GB s^{-1} in bandwidth, and therefore took less time. Additionally, the different runtimes caused by the large bandwidth differences can clearly be seen. The most extreme cases are memset with 211 s as the longest benchmark versus 29 s for memcopy.

As the parallel benchmark, we use a `blender` [40] render benchmark. This benchmark occupies all cores, and the results scale very well with the available compute resources. This blender render offers three subtests, called `monster`, `junkshop`, and `classroom`. Each of the benchmarks displays a result in the unit *samples per minute*. We benchmark blender version 3.2.1 using the blender benchmark tool `./benchmark-launcher-cli` [41] for Linux.

We compare six results, a reference run without anything running on the CPU, and five file system processes running in parallel to the blender benchmark:

- A run with GPU memcopy,
- a run with GPU4FS,
- a run with multi-file optimized GPU4FS as described above,

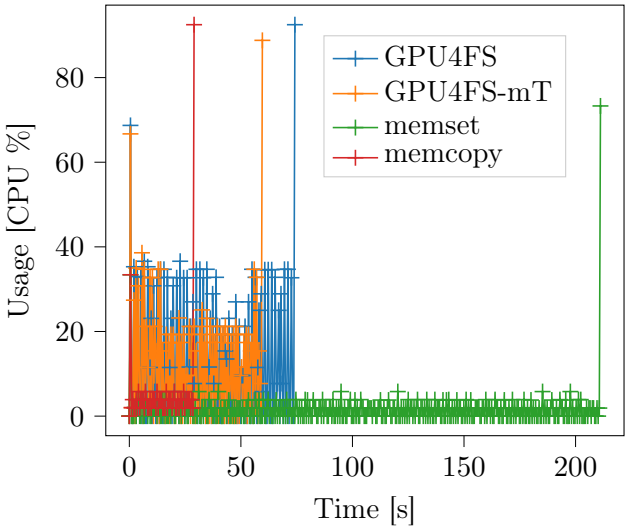


Figure 6.6: Reported usage of CPU cores usage for memcopy, memset, and file write. Additionally, a multiprocessing-optimized file write (GPU4FS-mT) is shown. The usage is low, but file writing has significantly more overhead than memset and memcopy. The peaks at beginning and end are caused by setup and teardown.

- a run with the filesystem benchmark `fio`,
- and a run with four parallel `dd` [35] command line tools writing one file each to a DAX-EXT4 file system directly on the NVM.

The benchmark results can be seen in Figure 6.7. The results show very competitive CPU performance for the GPU-accelerated file systems. The worst case is a slowdown of 7.5 % in the junkshop benchmark for the multi-file-optimized version of GPU4FS. In comparison, the best case for GPU-side tasks, memcopy, only shows slowdowns of 0.42 % to 4.7 %. The average overhead due to GPU management is around 4 %.

The CPU-side file system tasks on the other hand show slowdowns around a factor of two. The best case, `fio` running in parallel to the `classroom` benchmark gets a slowdown of 99.89 %, the worst case for the CPU-side tasks is `dd` in parallel to the `junkshop` benchmark at a more than 113 % reduction.

During the benchmarks, the bandwidth to the NVM is identical to the bandwidth without a benchmark running. For the GPU-accelerated tests, this is expected, as the CPU is not used for writing. For the CPU side, we expected some slowdowns, but the resulting equivalent of about two cores of CPU performance seems to be enough to saturate the NVM dimms.

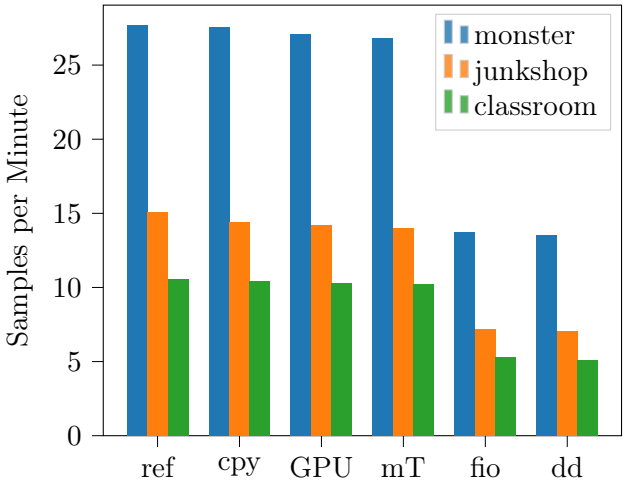


Figure 6.7: Blender benchmark results with different parallel file system tasks running on four cores. We show the reference results (ref), benchmark in parallel to memcopy (cpy), with GPU4FS (GPU) and GPU4FS in its multi-file version with more smaller threads (mT), and with fio and dd running in parallel on the CPU.

6.5 Discussion

After having presented these results, we now want to discuss their significance towards solving the problem, and the broader context of the results.

We identify the main problem that motivates this thesis to be the high CPU utilization when writing synchronously to Optane, which restricts parallel applications. We can verify that this problem exists with our own measurements, e.g., as shown with the benchmark numbers of `fio` [22] and `dd` [35] in Figure 6.7.

In contrast to the state of the art, GPU4FS demonstrates an at least 2x increase in performance of a CPU-bound application running in parallel to file system tasks, as can be seen in Figure 6.7. As shown in Figure 6.4, our solution also achieves competitive bandwidth, especially if writing a single file at a time. We also measure that, under certain circumstances, writing multiple files at full GPU bandwidth is possible. We consider searching for different ways to exploit more parallelism while keeping single file operation fast an interesting endeavour in the future.

A key factor to the GPU bandwidth to NVM is careful tuning into the bandwidth sweetspot. A new GPU in the system therefore could move the system out of the sweetspot if main performance characteristics change. A commercial-grade implementation therefore needs to offer a way to retest the bandwidth, and offer a way to reconfigure the grade of GPU parallelism. This might even be possible at runtime, if the bandwidth is measured during execution.

Another observation is that the file write can not quite sustain the same speeds as the memcopy operations for the same amount of SIMD lanes. We find two possible reasons

for this behavior: either the Optane DIMM suffers from some kind of read deficiency when it is overloaded by parallel write accesses, or the critical section in the shared inode explains the performance.

Currently, the implementation can either support fast writing of one to two files or writing a few files in parallel, not both. Writing multiple files in parallel has the advantage that small files can be written quickly in the name of latency without being blocked by prior writes of large files.

The demonstrator also does not support fine-grained latency measurements. Low latency operation is important when forced to handle large amounts of relatively small files, e.g., when compiling large projects. Currently, the only statements we can make are about the time between startup of the GPU task and the GPU signaling completion through the completion flag in the command buffer, as explained in Section 4.3.3. Given the measurements with 23 files processed by a single GPU command, as depicted in Figure 6.5, we assume that the time for setting up the shader dwarves the time that a single file system operation takes. However, without extended tests, we cannot verify this claim.

Additionally, the demonstrator is quite limited in its feature set. It can only append new files to a pre-existing directory, and cannot modify the file system, e.g., by deleting files or by creating directories. Our evaluation therefore focusses on file content writes. This means that the results we present are dominated by the relatively high-performance copying of data from DRAM to NVM. This assumption is backed by the observation that our file system measurements closely match the numbers observed in the memcopy tests, see Section 6.2.

In modern file systems, features more tailored towards the GPU have appeared, in particular RAID, checksumming, and encryption. These tasks are all easily and massively parallelizable. However, our demonstrator also does not implement these features, therefore we cannot yet evaluate whether this assumption holds in a file system context.

Alleviating these issues, namely parallelization, latency measurement capabilities, and feature set deficiencies, seems to be a worthwile task in the future. If the results match our expectations, using GPU acceleration for all kinds of file systems would offer another tool to speed up this ever-more CPU-intensive task.

A decision we made early into the project was selecting a GPU as the accelerator, as GPUs are commodity hardware and GPU driver modifications are relatively simple. GPUs, however, are not tailored well to file system tasks. As presented in Section 5.5, a major portion of the GPU code for the file system management is single-threaded. This means that during the management tasks, major parts of the GPU are unused. Therefore, now that the basic idea has proven to be worthwile, we expect specialized hardware like an FPGA to be an interesting area for further research. Specialized hardware could contain parts optimized for sequential management tasks, and other parts optimized for the highly parallel copy operations. Combined with the support for all file system features, one could construct small, efficient, full-scale accelerators for modern file systems like ZFS that can be inserted between the CPU and the storage media.

In conclusion, we show that the aim of the thesis, namely meaningfully reducing CPU utilization for NVM-based file systems using a GPU, can be attained under certain

circumstances. Our evaluation also shows valuable areas for future research: We consider latency testing and testing of meta data changes to be simple next steps. In the longer term, we want to suggest scaling up the implementation to the feature sets of modern file systems, or even changing the type of the accelerator to specialized hardware.

Observations about Optane Prior work [31] has shown that Optane suffers from reduced bandwidth if accessed by too many parallel CPU threads. Therefore, we expected our GPU-based solution to eventually hit a similar ceiling in bandwidth, and then to slowly degrade. Instead, as Figure 6.1 shows, the bandwidth reaches a clear peak after which it breaks down quickly to about a third of the performance. Additionally, the bandwidth limitation shown with the GPU is much more pronounced as with the CPU. Interestingly, DRAM suffers from a similar slowdown, just at a higher bandwidth and higher SIMD lane count. Whether this is an effect of DRAM or PCIe is not clear, though, as the bandwidth limit of eight lanes of PCIe Gen3 at $8\,\mathrm{GB\,s^{-1}}$ per direction is similar to the peak performance of DRAM at over $6\,\mathrm{GB\,s^{-1}}$.

We assume that Optane's bandwidth breakdown is caused by overflowing write queues either in the memory controllers leading to the Optane DIMM or in the DIMM itself. A high number of SIMD lanes would fill the queues quickly and then stall the memory controller. The stall would be communicated to the GPU, which waits for a continuation signal. Because of high parallelism, the queues would fill rapidly again, so the limit shows the bandwidth including repeated stalls. The bandwidth plateau of memset is higher than the limit of memcopy. We assume the reason to be that another minor latency penalty is taken when the GPU also has to load data from main memory, while other SIMD lanes fill up the write queues again.

Our observations might also explain the poor performance of Intel I/OAT when writing to Optane [38]. At peak DRAM performance, our measurements show a bandwidth of about $500\,\mathrm{MB\,s^{-1}}$, which is very close to the bandwidth Intel I/OAT shows. Similarily, the memset bandwidth also hovers around this figure, and shows unexpectedly low bandwidth for the number of SIMD lanes where Optane exhibits the highest numbers for memcopy, see Figure 6.3. We assume that Intel tuned the grade of parallelism of their I/OAT subsystem for peak DRAM performance, which happens to be unfit for Optane DIMMs. To the best of our knowledge, the grade of parallelism in I/OAT is not configurable in software. With modifications in hardware, Intel should be able to improve the I/OAT and Optane bandwidth greatly.

The last major observation we make is about the bandwidth plateaus that memset and memcopy reach with a high number of SIMD lanes. This plateau does not seem to be caused by either the GPU or the interconnect, as the DRAM numbers show significant changes. This holds in both the memcopy case, see Figure 6.1, and in the memset case, see Figure 6.3. Therefore, we assume the plateau to be an intrinsic characteristic of the Optane DIMM.

Further studies into these performance characteristics can be an interesting area for future research.

GPU4FS Outside of Optane Memory We evaluate GPU4FS against a system equipped with Intel Optane DIMMs as the storage medium. Other storage media, such as NVMe SSDs, have also gained high usage in the market, and accelerating file systems on such media could be interesting, too. Additionally, Intel seems to have discontinued their line of Optane Memory [55], which means that large-scale adoption of new Optane-and-GPU-equipped systems is highly unlikely. Whether GPU acceleration works for file systems on different storage media therefore is crucial to ensure the usability of the idea.

For NVMe SSDs, prior work discussed using the PCIe-P2PDMA feature [28], which gives the GPU direct access to NVMe drives while bypassing DRAM completely. Similar to our current design, the GPU would load data from storage into VRAM, process the data there and forward it to DRAM. Using P2PDMA to accelerate current file systems faces issues with regards to the file system caches in DRAM. These caches are flushes asynchronously, which can lead to file system consistency issues [28]. Given that, in our design, the GPU manages the caches, cache consistency can be guaranteed on the GPU directly.

Alternatively, the new CXL standard offers shared-memory semantics between different devices, not only the CPU [54]. For example, Samsung recently announced a CXL SSD which offers memory semantics [54]. These devices could allow for similar code as used before, so Optane promises to be usable as a development platform for future hardware.

In conclusion, GPU acceleration for file system tasks seems to be an interesting opportunity for future optimizations, even outside of the NVM context in which we evaluate it.

7 Future Work

The results of this thesis open new topics for interesting research:

Extending the Demonstrator In its current form, our demonstrator is not very versatile. An evaluation of an extended implementation could prove the concept to a much larger degree and with much higher confidence. This implementation should contain, among others: meta data changes like file renames and file movements, a GPU-side block allocator, and possibly even extended features like RAID.

Port to More Energy-Efficient Hardware In this thesis, we use a GPU because it is easily programmable and configurable. This comes with the downside of higher energy usage and relatively high purchase price, and leaves a lot of the GPU's die unused. The setup is also limited in that it passes its information through the PCIe bus twice, first to the GPU for processing and then back to either the NVDIMM or DRAM. On the other hand, more tailor-made hardware can alleviate these problems: It can be positioned between the CPU and the NVDIMM, thus using bus resources only once in the transaction. Custom hardware also needs only those resources that are actually required for the task. Lastly, file system tasks contain major sequential code paths which are not well-suited for GPUs, but can be improved with custom hardware, e.g., containing task-optimized CPU cores.

All in all, custom hardware, e.g., as implemented in an FPGA or ASIC, promises a more efficient, faster implementation that also uses less bus resources.

Exploiting more GPU Parallelism The parallel nature of the GPU requires that the number of SIMD lanes per workgroup needs to be preconfigured. Together with the slim window for efficient NVM writes from GPU, this poses problems for either few or many parallel files. A solution could be to implement an API workaround, which allows for dynamic workgroup sizing.

Exploring Intel I/OAT vs Memset vs Memcopy Bandwidth The different technologies built for highly parallel memory operations, like Intel I/OAT and the memset and memcopy behaviors all show a very similar performance for highly parallel requests, which hints at a general issue with the Optane DIMMs. Verifying this assumption and in turn, finding ways to limit the parallelism and to thereby increasing performance can be an interesting endeavour for future work.

8 Conclusion

To conclude, in this work we presented GPU4FS, a novel GPU-accelerated NVM file system built to relieve the CPU. File systems in modern storage applications use significant compute resources which could be used for other applications. GPU4FS aims to relieve this congestion by offloading the file system management to external general-purpose hardware, thus freeing up the compute resources for actual compute tasks.

In our work, we described the design decisions that guide us to our drafted full-scale GPU4FS implementation. We used our design to implement a demonstrator for the concept, and evaluated it in both high and low CPU utilization scenarios.

Our evaluation showed that a GPU-accelerated file system is indeed feasible: We achieved more than 80 % of maximum Optane write bandwidth while using a GPU instead of a CPU, all while keeping the reported CPU usage below 5 % of a single core. We measured a startup latency of the GPU at about 1.8 ms. At 2 MB or above, the startup latency was dwarfed by the write bandwidth to Optane.

With this result, we implemented the GPU-side file system, and evaluated its feasibility. GPU4FS still achieved the same bandwidth figures as our write tests before, all while keeping reported CPU utilization low, hovering at an average of about 12 %. The startup latency also stayed similar to the one measured with the write tests. A small file operation was very fast, and the GPU could handle multiple small files as soon as it is set up.

We verified these results against a parallel blender rendering process. The CPU-side file systems slowed down the render by a factor of two or more, all while being only marginally faster than the GPU4FS demonstrator. On the other hand, the largest slowdown we measured with a full-speed GPU4FS running in parallel is only at about 7.5 %.

Our work motivates future work into file system accelerators. A fully-featured modern file system could be implemented on the GPU while focussing on the parallel tasks in such a system. Alternatively, we suggest porting the file system accelerator to specialized hardware.

Bibliography

[1] David A. Patterson, Garth Gibson, and Randy H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". In: *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. SIGMOD '88. Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 109–116. ISBN: 0897912683. DOI: 10.1145/50202.50214. URL: https://doi.org/10.1145/50202.50214.

[2] Mendel Rosenblum and John K. Ousterhout. "The Design and Implementation of a Log-Structured File System". In: *ACM Trans. Comput. Syst.* 10.1 (Feb. 1992), pp. 26–52. ISSN: 0734-2071. DOI: 10.1145/146941.146943. URL: https://doi.org/10.1145/146941.146943.

[3] Margo I Seltzer et al. "File System Logging versus Clustering: A Performance Comparison." In: *USENIX*. 1995, pp. 249–264.

[4] Intel Corporation. *Accelerated Graphics Port Interface Specification*. 1998. URL: http://esd.cs.ucr.edu/webres/agp20.pdf (visited on 06/22/2022).

[5] Jeff Bonwick et al. *The Zettabyte File System*. 2002. URL: https://www.cs.hmc.edu/~rhodes/courses/cs134/fa20/readings/The%20Zettabyte%20File%20System.pdf (visited on 06/22/2022).

[6] Karthikeyan Vaidyanathan and Dhabaleswar K. Panda. "Benefits of I/O Acceleration Technology (I/OAT) in Clusters". In: *2007 IEEE International Symposium on Performance Analysis of Systems Software*. 2007, pp. 220–229. DOI: 10.1109/ISPASS.2007.363752.

[7] M Taniyama et al. "Analysis of the Y2K problem from the viewpoint of risk communication". In: *WIT Transactions on Information and Communication Technologies*. Vol. 39. WIT Press, 2008, pp. 225–239.

[8] Xiao-Yu Hu et al. "Write Amplification Analysis in Flash-Based Solid State Drives". In: *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. SYSTOR '09. Haifa, Israel: Association for Computing Machinery, 2009. ISBN: 9781605586236. DOI: 10.1145/1534530.1534544. URL: https://doi.org/10.1145/1534530.1534544.

[9] Adrian M. Caulfield et al. "Providing Safe, User Space Access to Fast, Solid State Disks". In: *SIGPLAN Not.* 47.4 (Mar. 2012), pp. 387–400. ISSN: 0362-1340. DOI: 10.1145/2248487.2151017. URL: https://doi.org/10.1145/2248487.2151017.

[10] Jianxi Chen et al. "FSMAC: A file system metadata accelerator with non-volatile memory". In: *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. 2013, pp. 1–11. DOI: 10.1109/MSST.2013.6558440.

[11] DRI Developers. *Graphics Address Re-Mapping Table (GART)*. 2013. URL: https://dri.freedesktop.org/wiki/GART/ (visited on 06/22/2022).

[12] Lokesh Gidra et al. "A Study of the Scalability of Stop-the-World Garbage Collectors on Multicores". In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 229–240. ISSN: 0362-1340. DOI: 10.1145/2499368.2451142. URL: https://doi.org/10.1145/2499368.2451142.

[13] Ohad Rodeh, Josef Bacik, and Chris Mason. "BTRFS: The Linux B-Tree Filesystem". In: *ACM Trans. Storage* 9.3 (Aug. 2013). ISSN: 1553-3077. DOI: 10.1145/2501620.2501623. URL: https://doi.org/10.1145/2501620.2501623.

[14] ScotXW. *Illustrates the differences between the Gallium3D and the Direct Rendering Infrastructure graphics driver models*
By ScotXW - Own Work, CC BY-SA 3.0. 2013. URL: https://commons.wikimedia.org/w/index.php?curid=27894555 (visited on 08/03/2022).

[15] Mark Silberstein et al. "GPUfs: Integrating a File System with GPUs". In: *SIGARCH Comput. Archit. News* 41.1 (Mar. 2013), pp. 485–498. ISSN: 0163-5964. DOI: 10.1145/2490301.2451169. URL: https://doi.org/10.1145/2490301.2451169.

[16] Simon Peter et al. "Towards High-Performance Application-Level Storage Management". In: *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*. HotStorage'14. Philadelphia, PA: USENIX Association, 2014, p. 7. URL: https://dl.acm.org/doi/10.5555/2696578.2696585.

[17] Kun Tian, Yaozu Dong, and David Cowperthwaite. "A Full GPU Virtualization Solution with Mediated Pass-Through". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 121–132. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/tian.

[18] Haris Volos et al. "Aerie: Flexible File-System Interfaces to Storage-Class Memory". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. ISBN: 9781450327046. DOI: 10.1145/2592798.2592810. URL: https://doi.org/10.1145/2592798.2592810.

[19] Qiumin Xu et al. "Performance Analysis of NVMe SSDs and Their Implication on Real World Databases". In: *Proceedings of the 8th ACM International Systems and Storage Conference*. SYSTOR '15. Haifa, Israel: Association for Computing Machinery, 2015. ISBN: 9781450336079. DOI: 10.1145/2757667.2757684. URL: https://doi.org/10.1145/2757667.2757684.

[20] James Bornholt et al. "Specifying and Checking File System Crash-Consistency Models". In: *SIGPLAN Not.* 51.4 (Mar. 2016), pp. 83–98. ISSN: 0362-1340. DOI: 10.1145/2954679.2872406. URL: https://doi.org/10.1145/2954679.2872406.

[21] Jian Xu and Steven Swanson. "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories". In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 323–338. ISBN: 978-1-931971-28-7. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

[22] Jens Axboe. *FIO Flexible I/O tester*. 2017. URL: https://fio.readthedocs.io/en/latest/fio_doc.html (visited on 02/04/2022).

[23] Youngjin Kwon et al. "Strata: A Cross Media File System". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 460–477. ISBN: 9781450350853. DOI: 10.1145/3132747.3132770. URL: https://doi.org/10.1145/3132747.3132770.

[24] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC Boston, 2018.

[25] PASC. *The Open Group Base Specifications Issue 7, 2018 edition*. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/ (visited on 08/05/2022).

[26] Arkan Alkamil and Darshika G. Perera. "Efficient FPGA-Based Reconfigurable Accelerators for SIMON Cryptographic Algorithm on Embedded Platforms". In: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2019, pp. 1–8. DOI: 10.1109/ReConFig48160.2019.8994803. URL: https://www.inesc-id.pt/ficheiros/publicacoes/8197.pdf.

[27] AMD. *https://www.amd.com/system/files/documents/rdna-whitepaper.pdf*. 2019. URL: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf (visited on 02/05/2022).

[28] Shai Bergman et al. "SPIN: Seamless operating system integration of peer-to-peer DMA between SSDs and GPUs". In: *ACM Transactions on Computer Systems (TOCS)* 36.2 (2019), pp. 1–26.

[29] Roberto Cavicchioli et al. "Novel Methodologies for Predictable CPU-To-GPU Command Offloading". In: 2019. URL: https://drops.dagstuhl.de/opus/volltexte/2019/10759/pdf/LIPIcs-ECRTS-2019-22.pdf.

[30] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2019.

[31] Joseph Izraelevitz et al. *Basic Performance Measurements of the Intel Optane DC Persistent Memory Module*. 2019. DOI: 10.48550/ARXIV.1903.05714. URL: https://arxiv.org/abs/1903.05714.

[32] Rohan Kadekodi et al. "SplitFS: Reducing Software Overhead in File Systems for Persistent Memory". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 494–508. ISBN: 9781450368735. DOI: 10.1145/3341301.3359631. URL: https://doi.org/10.1145/3341301.3359631.

[33] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. "System Evaluation of the Intel Optane Byte-Addressable NVM". In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS '19. Washington, District of Columbia, USA: Association for Computing Machinery, 2019, pp. 304–315. ISBN: 9781450372060. DOI: 10.1145/3357526.3357568. URL: https://doi.org/10.1145/3357526.3357568.

[34] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. "EvFS: User-Level, Event-Driven File System for Non-Volatile Memory". In: *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*. HotStorage'19. Renton, WA, USA: USENIX Association, 2019, p. 16. URL: https://dl.acm.org/doi/10.5555/3357062.3357083.

[35] GNU Developers. *dd - convert and copy a file*. 2020. URL: https://man7.org/linux/man-pages/man1/dd.1.html (visited on 08/06/2022).

[36] Sebastian Reimers. "Extension of an accelerator-friendly in-memory file system for persistent storage". In: (2020). URL: http://os.inf.tu-dresden.de/papers_ps/reimers_bachelor.pdf.

[37] Jian Yang et al. "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory". In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/yang.

[38] Lukas Werling, Christian Schwarz, and Frank Bellosa. *Towards Less CPU-Intensive PMEM File Systems*. Talk presented at Fachgruppentreffen Betriebssysteme 2021, Trondheim. 2021. URL: https://www.betriebssysteme.org/wp-content/uploads/2021/09/FGBS_Herbst2021_Folien_Werling.pdf.

[39] Apple. *Metal - Accelerating graphics and much more*. 2022. URL: https://developer.apple.com/metal/ (visited on 08/05/2022).

[40] Blender Developers. *blender*. 2022. URL: https://www.blender.org/ (visited on 08/05/2022).

[41] Blender Developers. *Blender Benchmark*. 2022. URL: https://opendata.blender.org/ (visited on 08/05/2022).

[42] Freedesktop Developers. *Nouveau: Accelerated Open Source driver for nVidia cards*. 2022. URL: https://nouveau.freedesktop.org/ (visited on 06/27/2022).

[43] Freedesktop Developers. *RADV*
RADV is a Vulkan driver for AMD GCN/RDNA GPUs. 2022. URL: https://docs.mesa3d.org/drivers/radv.html (visited on 08/02/2022).

[44] Linux Developers. *drm/amdgpu AMDgpu driver*. 2022. URL: https://www.kernel.org/doc/html/latest/gpu/amdgpu/index.html (visited on 06/27/2022).

[45] Linux Developers. *drm/i915 Intel GFX Driver*. 2022. URL: https://www.kernel.org/doc/html/latest/gpu/i915.html (visited on 06/27/2022).

[46] Mesa Developers. *libdrm Direct Rendering Manager library and headers*. 2022. URL: https://gitlab.freedesktop.org/mesa/drm (visited on 08/05/2022).

[47] Romain 'Creak' Failliot, Tobias Droste, and Robin McCorkell. *mesamatrix*. 2022. URL: https://mesamatrix.net/ (visited on 08/05/2022).

[48] Khronos® Group. *Khronos Vulkan Registry*. 2022. URL: https://registry.khronos.org/vulkan/ (visited on 08/05/2022).

[49] The Khronos® Group. *OpenCL Overview*. 2022. URL: https://www.khronos.org/opencl/ (visited on 08/05/2022).

[50] The Khronos® Group. *OpenGL Overview*. 2022. URL: https://www.khronos.org/opengl/ (visited on 08/05/2022).

[51] LSoft Technologies Inc. *NTFS Multiple Data Streams*. 2022. URL: http://ntfs.com/ntfs-multiple.htm (visited on 08/04/2022).

[52] Microsoft. *DirectX graphics and gaming*. 2022. URL: https://docs.microsoft.com/en-us/windows/win32/directx (visited on 08/05/2022).

[53] NVIDIA. *CUDA Zone*. 2022. URL: https://developer.nvidia.com/cuda-zone (visited on 08/05/2022).

[54] Anton Shilov. *Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift*. 2022. URL: https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift (visited on 08/05/2022).

[55] Ryan Smith. *Intel To Wind Down Optane Memory Business - 3D XPoint Storage Tech Reaches Its End*. 2022. URL: https://www.anandtech.com/show/17515/intel-to-wind-down-optane-memory-business (visited on 08/05/2022).

[56] AOSP developers. *Android File Based Encryption*. June 06, 2022. URL: https://source.android.com/security/encryption/file-based (visited on 06/17/2022).

[57] FUSE developers. *Filesystem in Userspace*. May 06, 2022. URL: https://github.com/libfuse/libfuse (visited on 06/17/2022).

[58] Linux kernel developers. *EXT4 Linux kernel wiki*. September 20, 2016. URL: https://ext4.wiki.kernel.org/index.php/Main_Page (visited on 06/17/2022).

[59] LUKS developers. *Linux Unified Key Setup Website*. April 16, 2022. URL: https://gitlab.com/cryptsetup/cryptsetup/blob/master/README.md (visited on 06/17/2022).

[60] SSHFS developers. *SSHFS*. May 26, 2022. URL: https://github.com/libfuse/sshfs (visited on 06/17/2022).

[61] Areej Syed. *DDR4 vs GDDR6 Memory: Which One is Faster?* January 23, 2022. URL: https://www.hardwaretimes.com/ddr4-vs-gddr6-memory-which-one-is-faster/ (visited on 06/14/2022).

Glossary

ACL Access Control List, used to specify detailed access rights. 36

AGP Accelerated Graphics Port, a GPU interconnection bus and predecessor of PCIe. 22

AMD Major semiconductor company and competitor of Intel. 45, 87

AMDGPU Driver for modern AMD GPUs, both in user and kernel space. 45, 88

Android a mobile Linux distribution. 18, 86

API Application Programming Interface, description to program against. 11, 85–89

Apple a major OS vendor that also ships other software like Metal. 22, 87

ASIC Application Specific Integrated Circuit, a type of custom silicon. 75

BO Buffer Object, wraps a buffer that the GPU can use. 47

BTRFS B-TRee File System, a modern file system that is license-compatible with Linux. 16, 17, 25

C++ now a modern bare metal CPU programming language. 45, 86, 87

CPU Central Processing Unit, the brain of the computer. Used both for computation and configuration. 11, 85, 87

CPU4FS CPU fo(u)r FS, a CPU side implementation of GPU4FS for testing and debugging purposes. 45

CUDA previously Compute Unified Device Architecture, an NVidia developed-API to use their GPUs for compute purposes. 11, 22

CXL Compute Express Link, an open interconnect standard built on top of PCIe.. 74

DAX Direct Access, file system mode where file system caches are bypassed. 25

DDR Double Data Rate, modern protocol for DRAM communication. Optimized for both latency and throughput. 14, 86

DIMM Dual Inline Memory Module, a single mechanical memory component that can be put into a socket. 11

DirectX a Microsoft-exclusive graphics API. 21, 87

DMA Direct Memory Access, allows devices direct access to main memory without having to go through the CPU. 11, 14, 87, 88

DRAM Dynamic RAM, usually used as main memory for most devices. 11, 85

DRM Direct Rendering Manager, allows more direct access to GPU resources, including user-space configuration. 23

EXT4 The fourth installment of the extended file system, a family of file systems used in Linux. 16, 25, 34

extent A contiguous sequence of blocks, used for better sequential reads and writes while keeping overhead low. 16

FBE File-Based Encryption, an Android feature for fine-grained encryption. 18

FPGA Field Programmable Gate Array, hardware which can be reprogrammed after it has been shipped. 28, 75, 87

FS File System, maps linear data on storage to files data, folders, and other metadata. 15, 85, 87, 89

FUSE File system in USEr space, a Linux subsystem that forwards VFS calls back into user space. 20

g++ a C++ compiler. 45

GART Graphics Address Remapping Table, translates physical GPU addresses to physical CPU addresses, or to IO addresses if an IOMMU is present. 22, 46, 87

GDDR Graphics DDR, modern standard of GPU memory. Optimized mostly for throughput. 14

GEM Graphics Execution Manager, one of Linux' graphic memory managers. 23, 89

GLSL OpenGL Shader Language, can be used to program not only OpenGL shaders, but also Vulkan. 45, 51, 86

glslc compiler for GLSL to SPIR-V, built by Google. 45

Google an internet startup that ships a somewhat relevant mobile OS called Android. 45, 86

GPGPU General Purpose Computation on GPU, using the parallel computational resources for non-graphics applications. 22

GPU Graphics Processing Unit, used to compute graphics and other parallel applications. Usually needs a CPU for configuration and management. 11, 21, 85–89

GPU4FS GPU fo(u)r FS, a novel GPU accelerated NVM file system written in C++ and Vulkan. 12, 31, 85

GTT Graphics Translation Table, see GART. 22

HDD Hard Drive Disk, uses rotating magnetic metal plates to store information. 13, 88

I/OAT Intel Input/Output Acceleration Technology, DMA controllers initially designed to accelerate network communication.. 14

inode stores all information that is relevant for one file, except the file name.. 16, 34

Intel Major semiconductor company and competitor of AMD. 11, 14, 45, 85, 87, 88

IOMMU Input/Output MMU, translates what IO devices think are physical CPU addresses to actual physical CPU addresses. 46, 86

IPC Inter Processor Communication. Not to be confused with IPC. 87

IPC Inter Process Communication. Not to be confused with IPC. 20, 87

Lenovo a laptop manufacturer. 45

Linux A somewhat relevant OS. 19, 85, 86, 88, 89

Metal an Apple-only modern graphics API. 22, 85

Microsoft a major OS vendor that also ships other software like DirectX. 22

mmap Memory MAP, a system call that can map files and anonymous memory into the virtual address space. 46

MMU Memory Management Unit, translates address spaces, for example virtual to physical memory. 21, 46, 87

NOVA NVM-optimized log-structured file system. 26

NTFS New Technology File System, commonly used in the Windows operating system. 36

NVM Non-Volatile Memory. 14, 87, 88

NVMe NVM Express, a PCIe based storage protocol for fast SSDs. 13, 88, 89

OpenCL Open Compute Language, API to program compute accelerators with. Commonly used for GPUs or FPGAs. 11, 22

OpenGL Open Graphics Library, an old vendor-independent graphics API that has mostly kept up to date. 21, 86

Optane A RAM-like Non-Volatile Memory (NVM) storage technology developed by Intel. 11, 14

OS Operating system, the software that helps run useful code. Linux is a common representative. 11, 85–88

P2PDMA Peer-to-Peer DMA, PCIe feature that allows different non-CPU devices to communicate with each other. 74

PC Personal Computer, what you, the reader, are probably sitting in front of right now. 45

PCIe The Peripheral Component Interconnect Express is a common extension bus inside of PCs, used to connect different components like NVMe SSDs or GPUs. 11, 13, 74, 85, 87, 88

POSIX Portable Operating System Interface, a specification for OS interfaces. 18

RADV RADeon Vulkan, Vulkan driver for GPUs running with AMDGPU. 45

RAID Redundant Array of Independent Disks, a technology that stores redundant information to sustain a disk failure. 17

RAM Random Access Memory, volatile storage for all kinds of runtime data in a computer. 86, 89

Samsung Large Semiconductor company, producing, amongst others things, storage media. 74

SATA Serial "AT Attachment", an older protocol to connect storage like SSDs and HDDs. 89

SELinux Security-Enhanced Linux, hardened against some common attacks. 36

shader A program or process running on the GPU. 22

SIMD Single Instruction, Multiple Data: Process the same instruction stream on multiple parallel registers. Each of these streams is called a SIMD lane. 21, 88

SIMD lane A single register lane inside a wide SIMD register. 21, 49, 88, 89

Solaris Operating System developed by SUN. 25

SPIR-V Standard Portable Intermediate Representation, an intermediate representation of GPU program code that is GPU-independent. 45, 86

SSD Solid State Drive, flash-based storage medium. Commonly connected via NVMe or SATA. 11, 13, 87, 88

SSH Secure SHell, an encrypted network protocol. 89

sshfs SSH file system, a type of network file system. 20

SUN Defunct semiconductor and software company. 25, 88

TTM Translation Table Maps, another of Linux' graphics memory managers, more versatile than GEM. 23

VFS Virtual File System, abstracts different kernel drivers from the user/kernel interface. 19, 86

VGA Video Graphics Array, a connector and also protocol to connect a monitor to a computer. 21

VRAM Video RAM, dedicated memory for graphics applications, optimized for bandwidth. 12, 21

Vulkan Modern, close to the hardware API for both graphics and compute, mostly focused on GPUs. 11, 22, 45, 86–88

workgroup A set of SIMD lanes that are bundled together and can communicate and synchronize efficiently. 49

ZFS Zettabyte File System, complex FS developed for the server and datacenter market. 17, 25