Karlsruhe Institute of Technology

RAID on a File System Level for GPU4FS

Bachelor's Thesis
submitted by

Gregor Lucka

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Peter Maucher, M.Sc.

18th of May 2023 – 18th of September 2023

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

Karlsruhe, September 18, 2023

Abstract

This thesis presents a flexible RAID system integrated into GPU4FS, a novel GPU accelerated file system. Software RAID systems provide flexibility over hardware RAID systems, yet tasks involving complex parity coding strain CPUs, especially in double and triple parity configurations. To address this, we propose a GPU based RAID system integrated into GPU4FS, leveraging the GPUs parallel processing for efficient parity calculation and data handling.

Our core design concept centers on a logical address space managed by GPU4FS, allowing dynamic allocation of files and pages with specific RAID configurations, tailoring performance and redundancy to individual files and processes. Evaluations across RAID levels consistently demonstrated higher write bandwidths and reduced CPU utilization compared to CPU based RAID systems on Optane. Furthermore, the evaluation showed that parity calculation adds minimal computational overhead, affirming the efficiency of our GPU centric approach.

By harnessing GPU computational power, we present an innovative alternative to CPU based RAID systems, offering competitive bandwidths while reducing the average CPU usage by a factor of x13 to x21 depending on the RAID level.

Kurzfassung In dieser Arbeit stellen wir ein flexibles RAID-System vor, welches wir in GPU4FS einem neuartigen GPU beschleunigtem Dateisystem integrieren. Software RAID Systeme bieten erhöhte Flexibilität gegenüber Hardware RAID Systemen, doch Aufgaben wie komplexe Paritätscodierung belasten die CPUs insbesondere bei Nutzung von doppelter und dreifacher Parität. Um dieses Problem zu lösen, stellen wir ein GPU basiertes RAID-System vor. Das System ist in GPU4FS integriert und nutzt die Parallelverarbeitung der GPU für eine effiziente Paritätsberechnung und Datenverarbeitung.

Unser zentrales Designkonzept basiert auf einem logischen Adressraum, der von GPU4FS verwaltet wird und die dynamische Allokation von Dateien und Seiten mit spezifischen RAID Konfigurationen ermöglicht, wodurch Leistung und Redundanz für einzelne Dateien und Prozesse optimiert werden können. Die Auswertung über alle RAID Level hinweg zeigte durchweg höhere Schreibbandbreiten und eine geringere CPU Auslastung im Vergleich zu CPU basierten RAID-System auf Optane. Außerdem zeigte die Evaulation, dass die Berechnung der Parität nur minimalen Rechenaufwand verursacht, was die Effizienz unseres GPU zentrierten Ansatzes bestätigt.

Durch die Nutzung der GPU-Rechenleistung stellen wir eine innovative Alternative zu CPU basierten RAID Systemen vor, die eine wettbewerbsfähige Bandbreite bietet und gleichzeitig die durchschnittliche CPU-Nutzung um einen Faktor von x13 bis x21 abhängig vom RAID Level reduziert.

Contents

1 Introduction

New storage media promises higher performance, but the single-core performance of CPUs has not kept pace with these advancements[23]. Consequently, an increasing demand for CPU resources arises for storage-related tasks. To address this challenge, Maucher [23] introduced GPU4FS, a file system designed to operate on GPUs. GPUs are suited for highly parallel computations, and allow asynchronous and parallel handling of requests in Maucher's file system. Notably, their work demonstrated GPU4FS's ability to compete in terms of write bandwidth while significantly reducing CPU utilization [23].

In historical context, RAID systems were often implemented in hardware to enhance performance [16]. As CPUs gained more processing power, entirely software-based RAID systems emerged, capable of competing in terms of performance. Software RAID systems offered increased flexibility and portability compared to proprietary hardware solutions [16]. However, tasks involving parity coding, crucial for introducing data redundancy, demanded substantial CPU resources for computation [16]. With single parity, the risk of a second disk failure during recovery remained high, necessitating the transition to double parity [17], with triple parity becoming increasingly popular in recent years [22]. The shift to double and triple parity imposed a heavier burden on CPUs for RAID management tasks. In response, prior research aimed to reduce stress on the CPU by offloading coding tasks to dedicated hardware [16, 17, 21, 28].

Several approaches were explored, with some implementing coding on FPGAs [17] and others leveraging GPUs [16, 21, 28]. Both approaches significantly improved throughput, but GPUs, being more widely available and flexible, are our preferred choice. Thus, we advocate for the integration of a GPU-based RAID system into GPU4FS, preserving the flexibility and portability of software RAID systems while offloading management tasks to the GPU, thereby relieving the CPU.

GPU4FS stands out as an ideal candidate for this integration, since the data required for the parity calculation is already available on the GPU for writing the data to disk [23]. This eliminates the need for additional data transfers, as seen in previous approaches that only utilized the GPU for coding [16, 21]. Integrating our RAID subsystem into a file system permits the creation of a versatile system supporting RAID levels on a file-by-file basis. Furthermore, it paves the way for the future integration of checksums into our RAID system, analog to the implementations in ZFS [19] and BTRFS [26].

This thesis spans several chapters, commencing with an exploration of the fundamentals required for the conception of a (GPU-accelerated) software RAID in Chapter 2. We discuss prior work proposing different approaches to RAID in Chapter 3. Chapter 4 introduces our proposed design, followed by Chapter 5, which presents the actual implementation, offering insights into our implementation process. In Chapter 6, we evaluate our implementation and compare it to CPU-based RAID systems. The thesis concludes with Chapter 7, where we outline future work, and Chapter 8, where we provide our concluding remarks.

2 Background

This section serves as an initial exploration into the required background for the conception of a (GPU-accelerated) software RAID. We describe the necessary fundamentals such as the different RAID configurations, file systems and GPUs. This understanding serves as the basis for our subsequent goal: enhancing an existing GPU-based file system (GPU4FS) with a GPU-based RAID subsystem.

2.1 RAID

In 1988, Patterson et al. introduced the concept of Redundant Array of Inexpensive Disks (RAID), a taxonomy encompassing five distinct configurations for disk arrays, which differ in terms of disk space utilization, reliability, and performance [24]. In this section, we explore both hardware and software implementations of RAID, while also discussing their benefits and drawbacks.

2.1.1 RAID Levels

RAID configurations are commonly referred to as levels. RAID level 2 and 3 are less commonly used and are therefore omitted from this section [7]. RAID level 6 was introduced later to provide reliable redundancy for multiple disk failures [7].

Stripes represent arbitrary contiguous byte units, usually chosen to be a divisor of the medium's capacity. It's worth noting that the stripe size is traditionally fixed and remains constant for a given RAID instance setup [10].

The stripe size significantly impacts the array's performance. A large stripe size leads to most individual files being primarily written to a single disk, enabling parallelism for multiple concurrent requests involving different files to achieve high throughput [7]. Conversely, using a small stripe size results in individual files being striped across multiple disks, enhancing the parallelism of reads and writes within a single file [7].

The evaluation of performance across various RAID levels is not addressed in this section, as it heavily relies on the specific workload and the corresponding choice of stripe size [7].

Raid Level 0: Striping RAID-0 is the simplest form of data organization, where data stripes are distributes across the disks. A full-stripe consists of the data stripes that are distributed in a round-robin fashion across the disks, covering one complete cycle through all the disks before starting from the first disk again [7].

An example illustrating the distribution of the data stripes $0, \ldots, 15$, which are written sequentially, across the disks, is provided in Table 2.1.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Table 2.1: RAID-0 on four disks: Data stripes are distributes across the four disks in a round-robin fashion. The data stripes, between the horizontal lines, make up one full-stripe.

RAID-0 provides the upper limit for available disk capacity. With N disks, each containing C stripes, RAID-0 (striping) yields a total capacity of $N \cdot C$ data stripes. However, RAID-0 offers no data redundancy. In case of a single disk failure or data corruption, data becomes irrecoverable [7].

RAID Level 1: Mirroring On RAID-1 disks are partitioned into disjunct, equally sized sets [7]. The disks within a set mirror their data to each other, and data stripes are distributed across the different sets [7]. Reading can be done from any of the set's disk, and writes need to be duplicated to all the set's disks [7].

Table 2.2 shows an example of how the data stripes $0, \ldots, 7$, which are written in sequence, are distributed to two sets each containing two mirrors.

Set 0		Set 1	
Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Table 2.2: RAID-1 on four disks: Data stripes are distributed across sets and mirrored within a set.

Mirroring the data stripes to m disks reduces the capacity to $C \cdot \frac{N}{m}$ available data stripes [7]. Theoretically, RAID-1 can suffer from $\frac{N}{m} \cdot (m-1)$ disk failures if exactly one disk per set survives [7]. However, this scenario depends on chance. We can guarantee data recovery if any $m-1$ disks fail because at least one mirror disk per set survives, which holds a copy of the data [7].

A Note on Consistency: We have to ensure that we update a RAID system atomically, otherwise power losses and system failures can leave an inconsistent state [7]. This is known as the **consistent update problem** [7]. For example, on RAID-1, one disk could already been updated with a new data stripe whereas the mirror disk is not yet up to date [7]. After the system reboots, we have an inconsistent state and don't know which mirror has the updated data stripe [7]. Note that the **consistent update problem** applies to all other RAID levels as well.

Raid Level 4: Using Parity Instead of having each data stripe duplicated, redundancy can be achieved by adding a parity stripe to each full-stripe [7]. The parity is usually a simple bit-wise XOR of each data stripe in the full-stripe [7]. In the case of RAID-4 the parity is collected on a single disk [7].

To compute the parity: Let $D_{ij}^{k} \in \{0,1\}$ be the j-th bit of the k-th data stripe of the i-th full-stripe, with $k \in \{0,\ldots,N-2\}$. Let P_{ij} be the j-th bit of the parity stripe of the i-th full-stripe. The parity bit P_{ij} is computed as $\mathrm{XOR}(D_{ij}^{0},\ldots,D_{ij}^{N-2})$ and is collected on the N-th disk. An example can be found in Table 2.3.

Disk 0 (D^0)	Disk 1 (D^1)	Disk 2 (D^2)	Disk 3 (P)
D_0^0	D_0^1	D_0^2	P_0
D_1^0	D_1^1	D_1^2	P_1
D_2^0	D_2^1	D_2^2	P_2
D_3^0	D_3^1	D_3^2	P_3

Table 2.3: RAID-4 on four disks: Data stripes are distributed across the data disks (Disk 0-2) and full-stripe's parity stripe is stored on Disk 3.

The computation of the parity stripe can be easily parallelized by distributing the work across multiple threads. To achieve this, each stripe is divided into equally sized blocks. Blocks at the same offset within different data stripes are XORed by a single thread. A visualization can be found in Figure 2.1. This approach ensures that the bitwise calculations for these corresponding blocks within the stripes can be performed independently without the need for inter-thread communication.

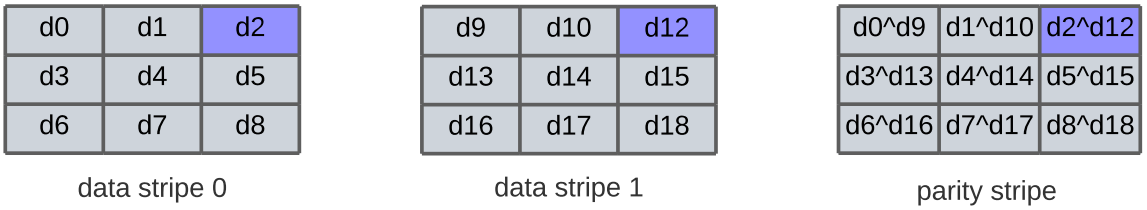


Figure 2.1: Parity stripes can be calculated in parallel. A single thread is responsible for XORing ($\wedge$) the blocks at the same offset on each data stripe. For example, the purple highlighted blocks, are processed by a single thread.

If a data word within the full-stripe is modified, a parity update needs to be calculated and applied to the parity. The parity update is just an XOR of the original data word and the modified data word. This update needs to be XORed with the parity word on the parity stripe and committed to the parity disk [7].

RAID-4's capacity is $(N-1)\cdot C$ data stripes, higher than RAID-1's capacity, but for the sacrifice of the performance required to compute the parity [7].

Any single disk can fail without data loss [7]. If the parity disk fails, no data needs to be reconstructed. The parity disk can be replaced, and the parity can be recomputed as stated

above [7]. If a data disk fails, the data D^x from the failed disk $x \in \{0, \ldots, N-2\}$ can be reconstructed:

1. Compute the XOR of the non-failed data disk:
   $\tilde{P}_{ij} = (D_{ij}^0, \ldots, D_{ij}^{x-1}, D_{ij}^{x+1}, \ldots, D_{ij}^{N-2})$
2. Reconstruct the lost data D^x by comparing $\tilde{P}$ with P.
   - If $P_{ij} = \tilde{P}_{ij}$ then $D_{ij}^x = 0$.
   - Otherwise $D_{ij}^x = 1$.

   Comparing the $\tilde{P}_{ij}$ with P_{ij} is equivalent to $D^x = \tilde{P}_{ij} \,\mathrm{XOR}\, P_{ij}$.

Raid Level 5: Striping Parity A problem with RAID-4 is that on each write, the parity information needs to be written to the parity disk [7]. This puts a lot of stress on the parity disk and bottlenecks parallel writes to different data disks [7]. This is known as the **small write problem** [7]. Therefore, RAID-5 rotates the parity information across the disks [7]. A visualization can be found in Table 2.4.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	P_0
3	4	P_1	5
6	P_2	7	8
P_3	9	10	11

Table 2.4: RAID-5 on four disks: Data stripes are distributed across the disk, with parity stripes being rotated across the disks.

Rotating the parity reduces the stress on a single disk and therefore increases the reliability of the disk array. The capacity is the same as on RAID-4 [7]. Because RAID-5 eliminates the small-write flaw of RAID-4 and otherwise behaves the same as RAID-4, it has replaced RAID-4 nearly entirely [7].

Raid Level 6: Recovery From Multiple Disk Failures In contrast to other RAID levels, RAID-6 disk arrays are able to recover from $m \geq 2$ disk failures reliably by having m parity stripes per full-stripe.

A commonly used code for RAID-6's parity stripes is the Reed-Solomon erasure code [25]. For simplicity's sake, we assume to have n data disks and m separate parity disks, whereas in reality the parity stripes are rotated around the disks analog to RAID-5.

For simplicity's sake, we assume that we only have one full-stripe with each disk only containing one stripe to have fewer indices in the equations, but the math can be applied to each full-stripe independently.

We can view our problem as the calculation of the parity stripe c_i by applying the function $\mathrm{F_i}$ to the n data stripes $d_1, \ldots, d_n$ with $i = 1, \ldots, m$ [25]:

$$c_i = \mathrm{F}_i(d_1, \ldots, d_n)$$

We also need a function $\mathrm{G_i}$ to update the parity c_i if one data stripe d_j is modified to d'_j [25]:

$$c'_i = \mathrm{G}_{i,j}(d_j, d'_j, c_i)$$

We want c_i to be a linear combination of the data stripes [25]:

$$c_i = \mathrm{F}_i(d_1, \ldots, d_n) = \sum_{j=1}^{n} d_j f_{i,j}$$

We can rewrite the data and parity stripes as the vectors D and C and $\mathrm{F_i}$ as the row of the matrix F [25]:

$$FD = C$$

We choose F to be the $m \times n$ Vandermonde matrix with: $f_{i,j} = j^{i-1}$ [25]:

$$\begin{pmatrix} f_{1,1} & f_{1,2} & \ldots & f_{1,n} \\ f_{2,1} & f_{2,2} & \ldots & f_{2,n} \\ \vdots & \vdots & & \vdots \\ f_{m,1} & f_{m,2} & \ldots & f_{m,n} \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix} = \begin{pmatrix} 1 & 1 & \ldots & 1 \\ 1 & 2 & \ldots & n \\ \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & \ldots & n^{m-1} \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

On data stripe changes from d_j to d'_j, we can subtract out the portion of the parity that corresponds to d_j and add the required amount for d'_j, because we chose F to be a linear function [25]:

$$c'_i = \mathrm{G}_{i,j}(d_j, d'_j, c_i) = c_i + f_{i,j}(d'_j - d_j)$$

To recover from disk failures, we define the matrix $A = \begin{pmatrix} I \\ F \end{pmatrix}$ (identity matrix stacked on top of F) and the vector $E = \begin{pmatrix} D \\ C \end{pmatrix}$ [25]. We get the following equation ($AD = E$) [25]:

$$\begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ 1 & 1 & 1 & \ldots & 1 \\ 1 & 2 & 3 & \ldots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \ldots & n^{m-1} \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

Every disk has a corresponding row in the matrix A and in the vector E [25]. For the data recovery, we delete the rows of each failed disk [25]. The result is a new matrix A' and a new vector E': $A'D = E'$ [25].

Because F is a Vandermonde matrix (and because the stacked matrix was the identity), every subset of n rows of the matrix A is linear independent [25]. A' is therefore non-singular,

and the values of D can be calculated by solving the linear equation system with Gaussian elimination [25]. If exactly m devices fail, A' is a $n \times n$ matrix, and we still have n equations to solve for D [25]. Once the values of D are recomputed, the missing parity values of the failed parity disks can be calculated [25].

The above linear algebra is guaranteed to be correct, but using regular arithmetic in a computer with finite precision may render the Gaussian elimination unsolvable in many cases [25]. Therefore, all arithmetic operations have to be performed over Galois Fields, which operate on a finite set of integers [25]. The details on how to perform arithmetic over Galois Fields is left out for the sake of brevity.

Since all parity operations are just basic linear algebra, they are perfectly suited to be parallelized on a GPU. Note that Reed-Solomon codes can be used to protect a RAID instance from $m \geq 2$ disks failures. For the common cases of $m = 2$ and $m = 3$ simpler coding schemes are available such as Row-Diagonal Parity [12] and EVENODD [8].

2.1.2 Hardware RAID

Hardware RAID systems manage the disks independently of the operating system by using an external hardware controller [7]. All management is done separately, and the disk array is presented to the operating system as a single disk. The controller is responsible for striping the data across the drives, and adding redundancy information. Usually, hardware RAID systems only use the parity for reconstruction and don't do any read verification [16]. Read verification ensures that the data on the disk is not corrupted, for example by employing checksums.

A selling point for hardware RAID systems is that they do not need any additional CPU time, for example for computing parity. To mitigate the **consistent update problem**, hardware RAID systems are equipped with batteries to ensure consistent updates and write back of their caches in case of a power loss [7].

2.1.3 Software RAID

Instead of having external hardware managing multiple disks, software RAID systems do all management tasks in software. This approach provides more flexibility and transportability, as the disks are presented to the host operating system individually [16]. This flexibility comes with the cost of having to spend CPU time on RAID management tasks like parity calculation. Additionally, software RAID systems cannot rely on batteries to leave a consistent state in case of a power loss, instead they have to rely on logging [7].

Historically, software RAID systems were implemented as isolated software components from file systems: The software RAID would abstract the individual disks into a single virtual disk on which a regular RAID-agnostic file system could be created. On the one hand, this approach hides the complexity of managing multiple disks from the file system. On the other hand, the file system has no way of verifying the integrity of the data served by the software RAID, as not all software RAID systems employ read verification. Another benefit of a software managed RAID is the flexibility of configuring RAID Level and performance characteristics like redundancy and disk usage for a specific use case [26].

2.1.4 Discussion

RAID 5 and 6, which heavily involve linear algebra computations, can benefit from GPU acceleration, as GPUs specialize in parallel linear algebra operations and have the potential to significantly speed up these calculations. This potential speed-up could notably enhance the data reconstruction and parity calculation processes.

Considering the trade-off between software and hardware RAID implementations, the former exhibits distinct advantages in terms of flexibility and portability. Software RAID solutions offer the freedom to adapt configurations based on specific requirements and are not tied to proprietary hardware constraints. However, the utilization of software RAID places an increased computational burden on the CPU. The CPU can also be freed from the RAID management tasks by offloading expensive computing task like parity computing to the GPU or dedicated programmable hardware [16]. Offloading the computational expensive tasks could relieve the CPU and potentially lead to improved overall system performance.

2.2 File Systems

A file system is a software component that abstracts the linear, flat address space of a storage medium into a hierarchical structure composed of files and directories. Each file within a file system represents a linear array of bytes and is identified by a unique low-level name known as the inode number. A file system encompasses two primary components: the file system's data structures and the driver implementation, both of which work in tandem to organize user data and metadata to facilitate file system API requests. With an understanding of a file system's data structures and the intended API, one can implement their own driver [7].

2.2.1 On-Disk Data Structures

In this section, we summarize the basic building blocks which are adopted by the majority of contemporary file systems to structure their metadata and file data on disk.

Blocks Initially developed for block devices like hard drives, file systems rely on a concept known as blocks. Typically, a file system defines a block size along with additional special block sizes that are integer multiples of the minimum. The file system manages the disk's linear address space in segments corresponding to the minimum block size [7].

Block Pointer Block pointers are responsible for indicating the location of specific data structures within the file system. These pointers come in two primary flavors: direct and indirect. A direct block pointer references data on the disk using an offset. In contrast, an indirect block pointer points to a block containing a list of pointers, each of which points to a specific data block or additional indirect pointers. This hierarchy of pointers contributes to efficient data organization and retrieval [7].

Extents Another method for handling block pointers is through the use of extents. Extents combine block pointers with a length parameter, denoted as n, which specifies the number of contiguous blocks. In contrast to a single block reference, an extent points to a sequence of n continuous blocks [7]. Using extents eliminates the need for special block sizes [26].

Inode The inode is the data structure tasked with storing essential metadata about a file. Each inode possesses a unique number associated with the file system instance, which aids the file system driver in determining the inode's block address. This metadata-rich structure includes information such as timestamps, access control permissions, file size, and most crucially, a list of pointers pointing to the actual data contained within the file. The inode has limited space for pointers, necessitating the use of indirect pointers and/or extents to manage expansive files [7].

Directory Within the domain of file systems, a directory acts as a special file type. Instead of containing arbitrary bytes, a directory holds a collection of mappings that associate user-readable filenames with the corresponding file's inode number. By forming nested directory structures, a hierarchical directory tree emerges, with the root directory serving as the starting point for this organizational hierarchy [7].

Superblock The superblock has a critical role within the file system architecture, as it stores the metadata vital to the file system instance. Positioned at a fixed offset of the file system partition, the superblock houses essential information to set up the file system driver, including a pointer to the root inode and data structures dedicated to managing free space. Notably, the superblock usually commences with a distinct magic string, which serves as an identifier recognizable by the hosting operating system. This recognition enables the operating system to invoke the correct driver for mounting the specific file system instance [7].

2.2.2 Checksums

During our discussion on RAID, our focus was primarily on data recovery following a disk failure. However, an additional challenge arises from silent failures attributed to data corruption [7].

Data corruption can occur on disks due to unavoidable bit flips. To prevent serving bad data to the user, a mechanism is required to identify corrupt data blocks [26].

Upon detection of a corrupted block, a RAID system can seamlessly replace the corrupted block with an intact copy due to redundant information included by the RAID system [26].

For the detection of corrupted data blocks, checksums are used. A checksum involves a function that takes a data block and generates a concise summary of its contents. The idea is that if any bit within the block flips, the resulting checksum will differ [7].

On a read request to a data block, we recompute the block's checksum and compare it to the stored checksum, which was calculated when the data block was written. If the checksums match, the data block can be delivered to the user. Conversely, if the checksums do not match, it necessitates the recovery of the original content [7].

Various checksum functions are available, each summarizing the contents of a larger data block into a compact data word. While collisions between some data blocks are inevitable due to compression, a well-designed checksum function minimizes the likelihood of such occurrences, while remaining computationally efficient [7].

2.2.3 File Mapping

The POSIX-compliant UNIX system call `mmap()` facilitates the mapping of files or devices into the virtual address space of the requesting process, a concept referred to as memory-mapped I/O. The content is loaded on-demand, triggered by a page access. It is possible to map a file as shared, creating a shared buffer between processes [1].

2.2.4 Crash Consistency

The primary objective of a file system is to persist data. Challenge to this goal potentially arise from power losses or system (driver) failures, as they may render the file system inconsistent. For instance, if a file system operation involves modifying two data structures, A and B, a system crash after modifying A but before modifying B can result in an inconsistent on-disk structure [7].

Journaling One method to address this issue is termed write-ahead logging or journaling. The concept involves committing desired changes to a journal in an atomic manner prior to making the changes. In the event of a system failure during an operation, the journal logs provide a record of incomplete operations, enabling retries or rollbacks. Once an operation concludes, the corresponding log entry is marked as invalid and is subsequently freed. While this approach introduces a minor overhead to each operation, it reduces the recovery workload significantly [7].

Copy-On-Write Another strategy is the utilization of copy-on-write methodology. Using this technique, updates to data structures are never done in place, instead a copy is modified in RAM and then written to a previously unused location on disk. Following this, the pointer in the primary data structure, which references the unmodified version, is replaced atomically, effectively committing the update. Subsequently, the original location of the updated data structure can be freed [7].

2.2.5 Discussion

We have delved into the foundational elements that constitute contemporary file systems. This knowledge will motivate our subsequent rationale for extending GPU4FS with a file system managed RAID subsystem. Within this discourse, we have explored two distinct methods, journaling and Copy-On-Write, that serve to establish crash consistency.

The **consistent update problem**, underscores the necessity for RAID systems to adopt crash consistency. An approach could entail a hybrid strategy, wherein file updates transpire

through copy-on-write, but the act of committing the updates requires a departure from the copy-on-write paradigm. This is necessary as updates are performed across multiple disks, making atomic updates of pointers impossible. To address this synchronization demand, a journaling mechanism should be employed to confirm the synchronization of updates across all disks.

Additionally, we have explained the concept of checksums and their role in augmenting RAID systems. Employing checksums facilitates online verification during reads, thereby acting as a safeguard against the propagation of corrupted data to users.

Introducing the notion of file mapping, we have unveiled the ability to map a file's pages into a process's virtual address space or even map the contents of entire disks like Optane. Through APIs like Vulkan these pages can also be mapped to the GPU, affording the GPU direct access to disk resources to read and write RAID-related information [23].

2.3 GPU

GPUs, or Graphics Processing Units are coprocessors originally designed to handle graphics-related tasks, such as projecting 3D scenes into flat 2D images that can be displayed on monitors. Because of their origins in computer graphics, GPUs are specialized in accelerating linear algebra. In recent years, GPUs have also been used for general-purpose computing, typically on large data sets due to their parallel nature and specialization in linear algebra.

2.3.1 GLSL

GLSL is a programming language used to write programs for GPUs, known as shaders [3]. The GPU4FS demonstrator employs the Vulkan API, and its shader code is written in GLSL [23].

Since we are extending this demonstrator and due to the absence of universal terminology for GPU concepts, we will use GLSL terminology where available and revert to the terms used by Hennessy and Patterson [18] as needed. We first discuss the abstraction introduced by GLSL and then proceed to explain how these concepts correspond to GPU hardware.

Work is abstracted into a three-dimensional compute space [3]. The GPU programmer divides the compute space's work among several independent workgroups per dimension [3]. A workgroup comprises multiple shader invocations (workers). The amount of invocations per workgroup is defined by the local size, represented as a three-dimensional vector [3]. Individual invocations within a workgroup are executed in parallel and can communicate through shared memory [3]. Workgroups are executed independently and in arbitrary order, while coordination among them is facilitated by atomic memory operations over global memory [3].

2.3.2 Architecture

GPUs, are multiprocessors designed for parallel computing of vectors. They consist of single instruction multiple data (SIMD) processors capable of executing workgroups simultaneously. Each SIMD processor acts as an independent core, executing one workgroup at a time [18].

A hardware scheduler assigns workgroups to these processors until all tasks in the compute space are finished. Within each SIMD processor, multiple invocations grouped as threads of SIMD instructions are scheduled and executed in lockstep [18].

These processors have their own on-chip memory, which is shared among their execution units. Shared variables in GLSL are stored in this fast local memory [18].

Off-chip global memory, known as VRAM, is also available and shared by the entire GPU and all workgroups. VRAM serves as a resource for sharing data and synchronizing work among parallel-running workgroups [18].

In GPUs, vector loads are performed as gather operations, which involve a base address and an index vector which offsets the individual loads from the base address. This allows sparse vectors to be loaded into a dense vector within a register file. The counterpart operation for storing a vector is called scatter and follows the same base address and index vector approach [18].

To optimize memory access, the GPU's memory interface unit identifies sequential memory accesses within a SIMD processor by analyzing the index vector. It then combines these requests into a more efficient, larger sequential memory access. To ensure optimal performance, GPU programmers need to ensure that addresses used in loads and stores are contiguous [18].

In a SIMD processor, there's only one program counter, which means invocations can't execute different instruction streams simultaneously. Instead, they have to take turns executing sequentially [18].

For example, imagine eight invocations in a workgroup executing code with nested if-else statements. Initially, all eight invocations work together in parallel. However, as they encounter if-else branches, half follow the "then" path while the other take the "else" path. After each branch, the number of invocations running parallel halves. The reduced parallelism leads to lower efficiency and throughput [18]. After the first branch, only four invocations work in parallel, at 50 % efficiency. After the second branch, this drops to two invocations at 25 % efficiency. With more nested branches, only one invocation works per branch, severely limiting parallelism. A visualization of the active invocations per branch can be found in Figure 2.2.

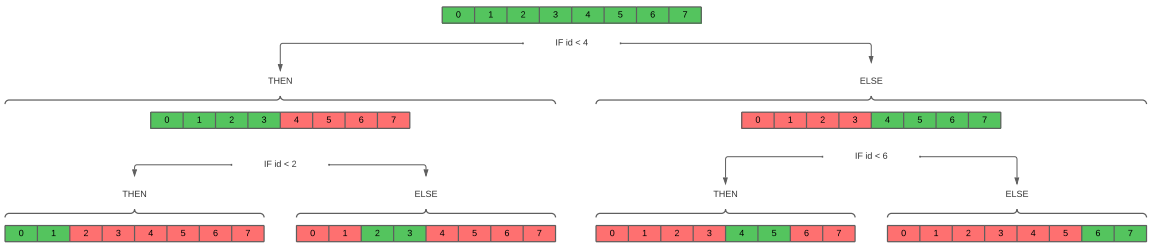


Figure 2.2: This figure illustrates the influence of branching on the scheduling of invocations within a GPU. In the visualization, green shading signifies active invocations, while red indicates inactive ones.

2.3.3 Discussion

We discussed the hardware architecture of GPUs to motivate what a GPU programmer has to keep in mind to effectively utilize the GPU hardware. The three main takeaways are: Maintaining sequential memory accesses whenever feasible to keep gathering and scattering operations effective. Furthermore, minimizing branching within a workgroup to have as many invocations as possible running in parallel. While the programming model suggests that each invocation is independent, the programmer needs to bear in mind that instructions of single invocations are grouped into SIMD instructions and executed in parallel as a thread of invocations.

3 Related Work

In this chapter, we commence our exploration of software RAID systems by presenting two distinct approaches. Our discussion starts with the conventional approach, where multiple disks are abstracted into a single virtual disk, hiding block-level RAID management from the file system through virtualization. Following this, we delve into an alternative paradigm where file systems take on direct disk management.

While examining these approaches, it's important to note that they are rooted in the concepts introduced in the "A case for redundant arrays of inexpensive disks (RAID)" paper by Patterson et al. [24], which first proposed the RAID taxonomy. These concepts have influenced various disk array techniques discussed in this chapter and our own disk array design. A summary of the RAID paper can be found in section 2.1.

We explore RAID hardware acceleration by studying one FPGA-based approach and prior research efforts that utilize GPUs to offload particular RAID management tasks. To conclude this chapter, we summarize GPU4FS, an innovative user-space file system on the GPU, and justify why it is the ideal candidate for expansion to a RAID-aware file system.

3.1 Block Software RAID Systems

For the traditional block approach to RAID, we will discuss Linux's Logical Volume Manager [29] and Linux's software RAID solution called Multiple Devices [29].

3.1.1 Logical Volume Manager (LVM)

Volume managers organize multiple devices and can abstract them into a single device in software [29]. This can be necessary as RAID-agnostic file systems and databases must be created on a single disk [29]. The solution on Linux is called Logical Volume Manager (LVM).

LVM is a suite of user level management tools and a mid-level block driver to map logical to physical blocks before sending them to the lower level host bus adapter drivers [29]. This is realized through queues to which block requests, abstracted as `buffer_heads`, can be posted [29]. They each contain the fields `rdev` which is the device identifier, and `rsector` which is the block number. In case of LVM the device identifier of the logical volume is used [29]. The mid-level LVM driver will use `rdev` and `rsector` to translate to a physical device and block. The driver posts the updated `buffer_head` to the request queue of the physical device [29].

LVM manages storage on multiple levels: The lowest level is the Physical Volume, which is a single device or a partition [29]. On each Physical Volume, a Volume Group Descriptor Area is allocated to contain the configuration information [29]. Multiple Physical Volumes are

merged into a Volume Group [29]. A Volume Group can be viewed as a large pool of storage from which Logical Volumes can be allocated (comparable to partitions) [29]. Logical Volumes are the virtual block devices on which file system or databases can be created [29].

A Volume Group splits each of its assigned Physical Volumes into small units called Physical Extents [29]. These Physical Extents can then be allocated to Logical Volumes [29]. Each Physical Extents can only be allocated to one Logical Volume [29]. This allows to move free Physical Extents between Logical Volumes of one Volume Group in order to shrink or expand them [29].

3.2 Multiple Devices

Multiple Devices (MD) is Linux's software RAID solution. Compared to LVM, much less administration is required as multiple disk are only abstracted as a single device [29]. MDs are configured in a RAID table, which declares [29]:

- `raiddev` the identifier for the logical device.
- `raid-level` 0, 1, 5 or 6 to specify the RAID Level of the logical device.
- `nr-raid-disks` defines the number of physical disks.
- `persistent-superblock` if set to 1 the config is stored on each device for auto mounting.
- `chunk-size` defines the stripe length.
- a list of all the physical devices (length must equal `nr-raid-disks`)

Once the RAID is set up, block requests are intercepted and translated analog to LVM's `buffer_head` translation [29]. The difference is the need to stripe and mirror the data or complement the write request with parity information according to the specified RAID Level. In case of RAID-1 the length of the requests queues can be used to select the least busy mirror for a read.

3.3 File System Managed RAID

In the following section, we'll examine two file systems that independently manage multiple disks, exploring their underlying motivations.

3.3.1 ZFS

ZFS was developed by SUN™ in 2001 [9]. Among other things, it tried to solve the following problems: The storage was divided into many sub-volumes, and an external logic volume manager (like LVM) was needed to build the file system on a single logical unit [26]. Sometimes

data on the disk became corrupted and since the file system had no way to detect this, the corrupted data was passed to the application.

Consider a file system implementing checksums and running on a Linux's MD RAID-1 instance, which itself has no support for checksums [26]. If the file system detects a checksum error while reading a block, it needs to retrieve a (hopefully) uncorrupted copy from a mirror [26]. Linux's MD hide from which disks the data came and does not allow to request data from a specific mirror [26]. The file system knows that the data is corrupted, but has no way to repair the data. Motivated by this issue, ZFS chooses to implement its own volume manager and checksums every data block. ZFS includes the checksum into its block pointers, allowing online read verification as the data is accessed [26].

ZFS manages disks as one large storage pool, no separate volume manager is used [9]. The file system is organized as a tree of data blocks [9]. Updates to the tree are performed with copy-on-write [9]. To commit the updates, the pointer to the tree's root is overwritten atomically [9].

ZFS uses a RAID algorithm based on RAID-4, called RAID-Z [19]. The main difference is that the stripe length varies between write requests. Like in RAID-4 the last chunk in a stripe is the parity chunk to provide fault tolerance to single disk faults [26]. The stripe sizes dependents on the IO request size [26]. Since the IO request size is varying, parity information will be spread among the disks somewhat evenly [26], being close to a RAID-5 configuration. Because of the varying stripe lengths, the parity information ends up being distributed among the disks, thus avoiding the **small write** problem.

To overcome the **consistent update problem**, RAID-Z uses copy-on-write and adaptive stripe size to ensure that updates are always full-stripe [26]. Updates are cumulated in DRAM and atomically committed to the tree as described earlier, therefore avoiding complex logging.

An example of the adaptive stripe length can be seen in Table 3.1 and Table 3.2.

IO size	data chunk	parity chunk	stripe size
4 kB	d_{11}	p_{12}	$1+1$
12 kB	d_{13}, d_{14}, d_{15}	p_{21}	$3+1$
8 kB	d_{22}, d_{23}	p_{24}	$2+1$
16 kB	$d_{25}, d_{31}, d_{32}, d_{33}$	p_{34}	$4+1$

Table 3.1: IO requests and their corresponding RAID-Z stripes. Example taken from [26].

disk	D_1	D_2	D_3	D_4	D_5
row 1	d_{11}	p_{12}	d_{13}	d_{14}	d_{15}
row 2	p_{21}	d_{22}	d_{23}	p_{24}	d_{25}
row 3	d_{31}	d_{32}	d_{33}	p_{34}	

Table 3.2: State of the disk array after performing the requests from Table 3.1. Example taken from [26].

RAID-Z supports RAID level 0, 1, 5 (RAIDZ) and 6 with double parity (RAIDZ2) or triple parity (RAIDZ3) [5]. The RAID level is set for the entire storage pool and cannot be managed on a per-file level [9].

3.3.2 BTRFS

BTRFS is a file system that organizes every data structure into B-trees, hence the name BTRFS. The BTRFS paper uses the term B-Trees [26], but actually uses B+-Trees [11]. Only in this section we adhere to BTRFS naming convention.

Metadata duplication and RAID capabilities are directly incorporated into BTRFS, with support for varying disk sizes within the BTRFS disk array. Instead of blocks, BTRFS uses extents to eliminate the need for special block sizes. To circumvent serving corrupted data, BTRFS chooses to employ checksums in pointers analog to ZFS [26].

BTRFS manages the file system as a forest of trees. For BTRFS' RAID implementation, the important trees are the chunk and the device tree. Each device is split into large chunks. The chunk tree contains the mapping from logical chunks to physical chunks. While the device tree contains the reverse mappings. The rest of the file system only sees and references logical blocks [26].

Using large chunks keeps the trees small and therefore enables in-memory caching of these trees to minimize the lookup overhead caused by the added indirection. The abstraction of logical chunks also enables physical chunks to be moved around to combat fragmentation and to rebalance disk usage when adding a new disk. This only requires an update of the mapping without having to backtrack and fix references [26].

RAID level 0, 1, 5 and 6 can be configured on a logical chunk level. The abstraction to logical chunks allows granular control of the RAID level and stripe length to optimize reliability or adjust the bandwidth of subvolumes for different workload requirements. The physical chunks are grouped according to the RAID level of the corresponding logical chunk [26]. For example, to enable mirroring, the physical chunks are grouped into pairs [26] (cf. Table 3.4). For striping, n physical chunks from different disks are grouped to a logic chunk [26] (cf. Table 3.3). For level 5/6 additional physical chunks are reserved for parity information [26].

logical chunk	disk 1	disk 2	disk 3
L_1	C_{11}	C_{21}	
L_2		C_{22}	C_{31}
L_3	C_{12}	C_{23}	
L_4		C_{24}	C_{32}

Table 3.3: RAID-1 with one large disk (disk 2) and to smaller disks (disk 1, 3). Example taken from [26].

logical chunk	disk 1	disk 2	disk 3	disk 4
L_1	C_{11}	C_{21}	C_{31}	C_{41}
L_2	C_{12}	C_{22}	C_{32}	C_{42}
L_3	C_{13}	C_{23}	C_{33}	C_{43}

Table 3.4: BTRFS: RAID-0 striping to four disks with a stripe size of four. Example taken from [26].

3.4 Discussion

Modern file systems, like ZFS and BTRFS, are aware of their deployment across multiple disks and actively manage these underlying disks. The significance of checksums, essential for preserving data integrity, is emphasized by both works. This highlights the deficiencies in conventional systems, such as Linux's MD RAID, which lack comparable mechanisms.

BTRFS takes physical disks and RAID management and abstracts it into logical chunks. This abstraction allows for dynamic adjustments in several ways. These chunks can have different RAID levels, either increasing or decreasing reliability. They can also have varying stripe lengths to fine-tune inner file or inter file parallelism and bandwidth.

This flexibility accommodates the specific needs of different subvolumes. Some subvolumes might require higher reliability, while others may prioritize performance even if it means sacrificing some reliability.

Furthermore, the file system and the user can determine the performance and reliability characteristics for these logical chunks, allowing for tailored tuning of subvolumes to suit their intended use cases.

3.5 Previous RAID Acceleration

In this section, we present prior publications that offload some of the RAID management workload to external hardware.

3.5.1 FPGA Accelerated RAID-6

In 2006, hardware RAID systems were performing better than software RAID systems on the CPU [17]. RAID-5 was widely used, and it was common for a second disk to fail during the rebuild process, which highlights the need for RAID-6 [17]. However, the adoption of RAID-6 was limited because only costly and complex proprietary hardware solutions were available [17].

As an alternative, Gilroy and Ivrine [17] proposed an FPGA-based RAID-6 accelerator. FPGAs offer a practical way to develop and implement accelerated RAID algorithms. Their primary goals were to achieve better performance in terms of data throughput and rebuild speed compared to software-based solutions, and to relieve the CPU from coding tasks [17].

The FPGA's adaptability allows for easy system design modifications in the future. The authors implemented Reed-Solomon Coding on the FPGA, resulting in faster rebuild speeds

and higher data throughput. Notably, when facing single disk erasures, CPU utilization dropped by one-third compared to software-only solutions. This reduction increased to half of the CPU consumption for double disk erasures [17]. This work underscores the potential of using hardware accelerators to improve RAID performance and system efficiency.

3.5.2 A Lightweight, GPU-Based Software RAID System

Curry et al. [16] developed a software RAID that offloads the computation of error-correcting codes used in RAID-6 to the GPU. In their previous works [13, 14, 15], they demonstrated the GPU's superiority over the CPU in coding-related tasks. In this study, they integrated their GPU-based coding into an existing software RAID named Gibraltar [16].

Their objectives were to keep the flexibility of a software RAID, while achieving higher performance than a fully CPU-bound RAID by offloading coding to the GPU. This was important since hardware RAID systems at that time promised superior performance. To achieve this, they replaced their software RAID's erasure correction engine with a GPU shader. Additionally, they extended their software RAID's capabilities beyond typical hardware RAID systems. They introduced read verification and implemented Reed-Solomon coding support for a variable number of parity disks ($m \geq 2$) [16].

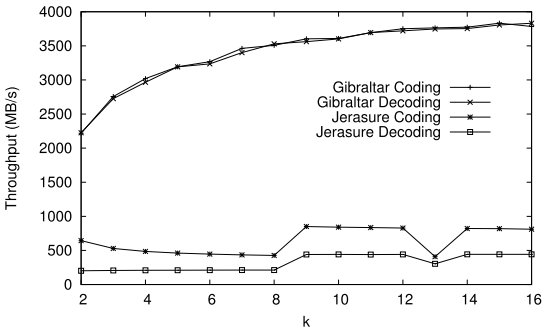
The implementation of their software RAID is entirely situated in user space. However, they need to determine which files were present in the file system cache. This is necessary as they only perform read verification on data retrieved from the disks, avoiding slowing down reads to already verified data from the cache. Consequently, they bypass Linux's buffers and cache the file's blocks in user space [16].

Owing to CUDA's issues with mapping files opened with the O_DIRECT flag to bypass Linux's caches, they were unable to directly map the disk or the user space buffer to the GPU. Instead, they opted to copy the entire stripe to VRAM. Despite this limitation, they were still able to maintain speed improvements over a CPU-based software RAID [16].

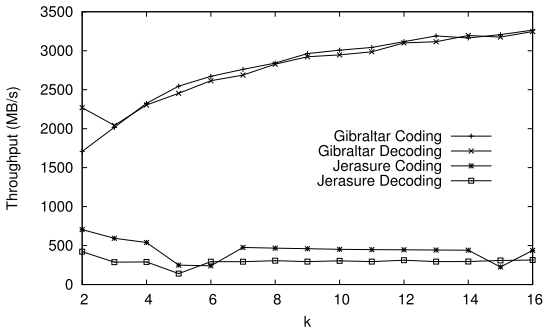
All user reads and writes are directed to their user space stripe cache. Once a stripe is completely written to the cache, the computation of the erasure codes occurs asynchronously on the GPU. A "victimizer" process is responsible to asynchronously write back stripes, complemented with coding information, to the disks using Linux's asynchronous I/O [16].

Their GPU-based coding, introduced in 2012, achieved a bandwidth of 4 GB/s. A plot of their achieved throughput can be found in Figure 3.1. The GPU shader, coded in CUDA, accepted k data buffers and returned m parity buffers. The throughput of the error-coding shader can be found in Figure 3.1.

A noteworthy outcome of their study is the observation that parity verification did not notably slow down reads; the slowdown primarily resulted from the extra bandwidth required to read the parity chunks from disk. Curry et al.'s [16] work not only establishes the feasibility of offloading parity computation in software RAID to the GPU but also underscores its potential to deliver significant performance gains.



(a) Throughput (de-)coding $m = 2$ parity blocks from k data blocks



(b) Throughput (de-)coding $m = 3$ parity blocks from k data blocks.

Figure 3.1: Comparison of throughput for Reed-Solomon Coding on CPU (Jerasure) vs. GPU (Gibraltar). Here, k represents the number of data blocks, each with a size of 2MB. Both plots are taken from [16].

3.5.3 On the Use of GPUs in Realizing Cost-Effective Distributed RAID

This section summarizes a paper by Khasymski et al. [21], presenting a software RAID solution that utilizes GPU-based parity computation. This solution is tailored for parallel file systems within high-performance computing environments. The authors deploy cost-effective GPUs on both client and server nodes to accelerate the process of calculating parity[21].

The main focus of their work is to ensure end-to-end data integrity by conducting encoding and decoding at the compute node. This strategy aligns with the point of data generation and consumption. The authors adopt a distributed strategy for parity computation on a per-file basis within compute nodes. This approach contrasts with a centralized backend, which commonly handles all parity computations [21].

By following a file-oriented methodology, their system accommodates RAID-1 for achieving redundancy for small files, seamlessly transitioning to RAID-6 as the file size increase. The framework also enables concurrent rebuilds. In this client-centric rebuilding approach, tasks for reconstruction of individual files are handled by individual clients [21].

The authors' investigation into a CPU coding library uncovers that a substantial portion of the runtime (95 %) is consumed by XOR operations. This outcome is not unexpected due to the nature of RAID-6 parity computation on Galois Fields, where addition and multiplication translate to bitwise XOR and AND operations. In response, the authors opt to transfer the bitwise operations to the GPU, leveraging its inherent capacity for Single Instruction Multiple Data (SIMD) parallelism[21].

The implementation on the GPU, realized using CUDA, demonstrates a coding throughput of 3 GB/s on each client. This rate adequately saturates their available network bandwidth, accommodating the transmission of both data and parity blocks[21].

3.5.4 Discussion

All three studies underscore the superior computational capabilities of hardware accelerators over the CPU in parity coding. Due to the general availability of GPUs in computer systems, it makes sense to design a RAID system accelerated by the GPU for higher adoption rates instead of relying on FPGAs. This distinction demonstrates the potential of a GPU-accelerated RAID system.

In both GPU focused publications, the GPU is exclusively employed for computing coding information, while disk read and write operations are managed by the CPU. This configuration implies that a fully GPU-based system could potentially surpass the performance of these approaches. This advantage stems from the fact that data is already accessible to the GPU without necessitating additional data transfers between the CPU and GPU.

The paper "On the Use of GPUs in Realizing Cost-Effective Distributed RAID" highlights the advantages of adopting a file-level approach to RAID implementation. This approach enhances flexibility and load balancing in distributed environments.

3.6 GPU4FS

Modern file systems running on fast storage media consume valuable CPU resources to fully utilize the underlying storage device, especially when Intel Optane is employed due to its relatively low write bandwidth [23]. These accesses tie up the CPU and DMA is not an option due to Optane's synchronous memory accesses [23]. Therefore, it is reasonable to employ a coprocessor capable of handling file system tasks asynchronously and in parallel. GPUs, being affordable high-performance accelerators, serve as suitable candidates for offloading file system tasks, as modern APIs like CUDA, OpenCL, and Vulkan provide convenient access to the computational potential of GPUs. Maucher proposed an innovative solution known as GPU4FS, a GPU-accelerated user-space file system designed for high-performance non-volatile memory [23].

As the GPU cannot function independently and requires configuration from the CPU in user space, a user space management process is needed. During the setup phase, the management process maps the NVM disk to the GPU, which requires help from the kernel. When a process intends to use a GPU4FS instance, the user space management process creates a shared memory region between the requesting process and the GPU. The requesting process can directly interact with the GPU via the shared memory region, eliminating the need for further kernel involvement. Special permissions are required for the management process to remap pages in different processes when dealing with shared pages [23]. A visualization can be found in Figure 3.2.

The GPU takes over all file system management tasks from the CPU. Requests are queued into a shared command buffer by the CPU and subsequently executed by the GPU. Completion is indicated by a flag within the shared memory region. The commands are designed to offload as much work as possible onto the GPU [23].

Considering that Optane behaves similarly to normal DRAM and is organized into 4kB, 2MB, and 1GB pages by the MMU on x86-64 systems, both DRAM and Optane can only be

mapped in page granularity. To ensure process separation and visibility, GPU4FS aligns file blocks with page boundaries and blocks are page-sized units. These blocks can be recursively divided, with the smallest block size being 128 bytes, the size of an inode [23].

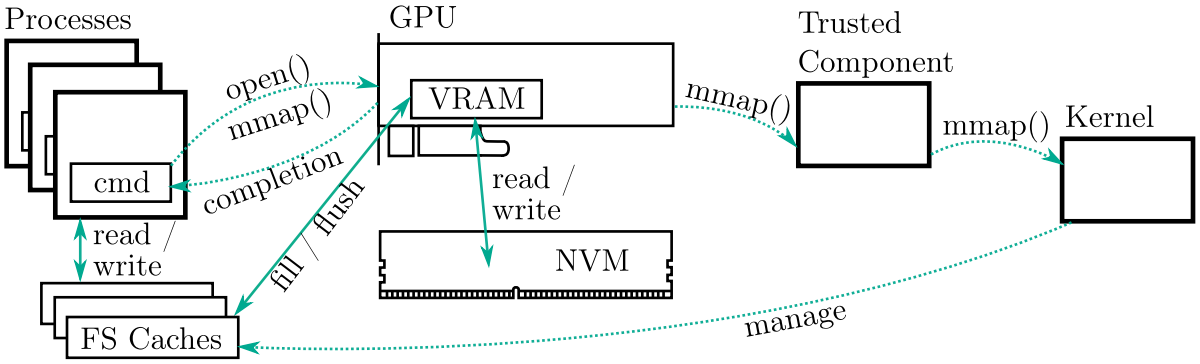


Figure 3.2: GPU4FS employs caching alongside a trusted component. Requests are enqueued within the command buffer. Upon receiving a command, the GPU interprets it and retrieves corresponding data from the FS caches, transferring it into the VRAM cache. Subsequently, the GPU utilizes this data to facilitate writing operations to NVM. In situations involving data retrieval, information is pulled from NVM to the VRAM cache before being stored back into the FS caches. For commands necessitating OS intervention, such as mmap(), the GPU takes an additional step by introducing the command into the command buffer of a trusted component. This trusted module subsequently initiates system calls to the kernel, allowing for the execution of administrative tasks with elevated kernel privileges. This figure is taken from [23].

3.6.1 On-Disk Data Structures

In this section, we provide an overview of some fundamental data structures in GPU4FS that require extension in our RAID design. For in depth information about GPU4FS's data structures, we defer to [23].

Block Pointer Given the possible sizes of 128B, 4kB, 2MB, and 1GB, a two-bit tag is necessary to encode the size of the referenced block within a pointer. Furthermore, a block pointer contains a 1-bit flag indicating its validity and another 1-bit flag to indicate whether the pointer is indirect or not, leaving three bits unused. The remaining 57 bits indicate the offset on the physical drive where the referenced page start [23]. A visualization of the block pointer can be found in Figure 3.3.

Inode Since our extension of GPU4FS with a RAID subsystem necessitates no changes to the inode, we provide a concise section here and refer readers to [23]. In its pursuit of POSIX compliance, GPU4FS adopts all necessary flags from EXT4. The fields are all 64-bit aligned

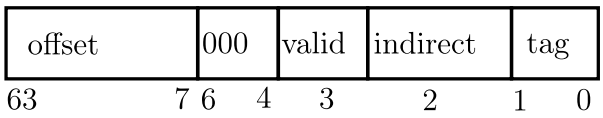


Figure 3.3: Bit usage of GPU4FS's block pointer. Three bits remain unused, 57 bits are used for the offset of the referenced page on the disk. The remaining bits are used to signal whether the pointer is valid and indirect and to differentiate between the size of the page (tag). Figure taken from [23].

and grouped. A notable distinction from other file systems is the inode number, which directly corresponds to the physical offset of the inode on the disk, without complex translation [23].

Directories To write to a directory, an exclusive lock needs to be acquired in the directory's inode on-disk. GPU4FS's demonstrator currently only has one directory, the root directory [23].

3.6.2 Runtime

During runtime, commands are queued by the requesting process into the shared command buffers and executed by the GPU. This section lists the most important commands and how they are processed [23].

Commands As mentioned before, request are queued to a command buffer. Each command comprises 128 bytes divided into 16 64-bit words. The commands form a linked list, commencing with a metadata command and concluding with a termination command. Workgroups are dispatched to traverse the list, aiming to atomically acquire each command descriptor and execute the command if successful. Once the work is completed and the completion flag is set, the workgroup proceeds to follow the list until the termination command is reached [23]. A visualization of the command basics can be found in Figure 3.4

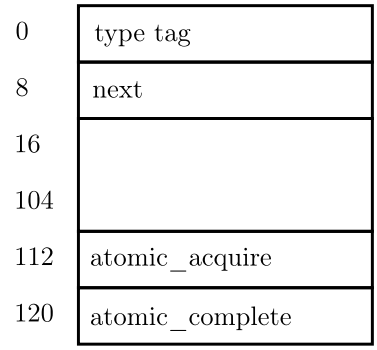


Figure 3.4: A basic command descriptor. The payload between offset 16 and 112 depends on the command implementation. Figure taken from [23].

The **metadata command descriptor** holds information to configure the shader invocations: The amount of shader invocations to be dispatched and a separate_execution flag. The flag

indicates whether workgroups should work together on a single command or work on different commands in parallel [23]. A visualization of the metadata command can be found in Figure 3.5

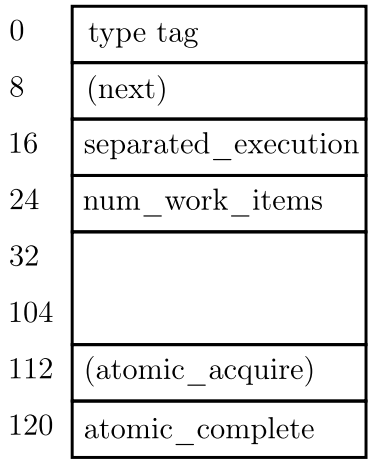


Figure 3.5: GPU4FS metadata command. Figure taken from [23].

The **file write command descriptor** is employed to establish a new file within GPU4FS and transfer initial contents to NVM. The command descriptor includes the offset in the command buffer to the file content, along with metadata like file size, length of the file name, offset in the command buffer to the file name, position of the directory for adding the file, and the offset of the CPU-prepared inode in the command buffer [23]. The exact 64 bit aligned command descriptor can be found in Figure 3.6

A file write command involves the following steps [23]:

1. The process enqueues a write request, including information like the open file descriptor or path and a pointer to the shared buffer containing the file content to be written.
2. The GPU reads the request, copies it to VRAM, and verifies it there.
3. The GPU executes the request:
   a) The GPU allocates pages on NVM for the file's content and the inode.
   b) Subsequently, the file data and inode are sequentially written to the NVM disk, ensuring uniform gathering and scattering to optimize GPU performance.
   c) Following this, block pointers to the allocated pages are written to the copied inode.
   d) In the final step, the file name and inode pointer are written to the specified directory. Before a directory update, its inode must be atomically locked to secure exclusive write privileges.
4. The GPU concludes the command by setting the completion flag in the command buffer.

A visualization of these steps can be found in Figure 3.7.

For reading, data is copied to VRAM and then forwarded to the DRAM space of the requesting process. In the case of shared files, data is sent to the shared file system caches.

0	type tag
8	next
16	file_size
24	num_SIMD_lanes
32	file_data_offset
40	filename_length
48	filename_offset
56	directory_position
64	inode_offset
72	inode_position
80	file_position
88	
104	
112	atomic_acquire
120	atomic_complete

Figure 3.6: GPU4FS's file write command descriptor. Figure taken from [23].

VRAM serves as a second-level cache, potentially holding data previously written by a process or retrieved from NVM, thus reducing the need to access NVM [23]. A visualization of the read process can be found in Figure 3.8

Evaluation The initial exploration focused on determining the number of shader invocations required to best utilize Optane's 2 GB/s write bandwidth from the GPU using a GPU-based parallel memcopy. The results indicated that a peak of 1.9 GB/s could be achieved using 320 shader invocations on Maucher's test configuration. Optimal performance was found between 256 and 512 shader invocations. Thus, the next benchmark uses the file write command to write multiple files in parallel. It runs on two workgroups, each consisting of 256 shader invocations, allowing parallel handling of two files [23].

With 512 invocations, the memcopy benchmark measured 1.5 GB/s. Writing multiple files using 512 invocations resulted in a bandwidth of 1.4 GB/s, suggesting that the Optane DIMM get overwhelmed. For single-file writes with only 256 shader invocations, a performance of 1.9 GB/s was measured [23].

In summary, GPU4FS achieved over 80 % of the maximum Optane write bandwidth using a GPU instead of a CPU. This achievement occurred while maintaining CPU usage below 5 % of a single core [23].

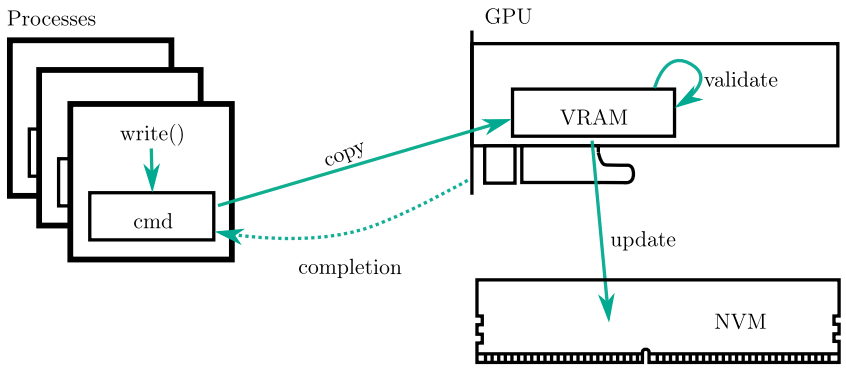


Figure 3.7: Visualization of a CPU process requesting a write command. Figure taken from [23].

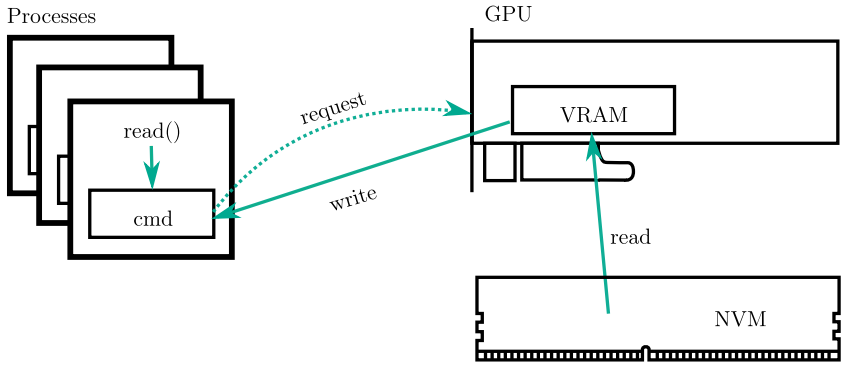


Figure 3.8: Visualization of a CPU process requesting to read a file. Figure taken from [23].

3.6.3 Discussion

GPU4FS has demonstrated the viability of a GPU-based file system, showcasing its potential to compete effectively in terms of bandwidth while freeing CPU resources.

Earlier studies [21, 16, 28] have underscored the acceleration potential that GPUs offer for parity coding. Capitalizing on this, GPU4FS takes advantage of its direct access to both data queued for disk writing and the disk itself from the GPU. This strategic positioning empowers GPU4FS to efficiently manage RAID tasks, enhancing system reliability and overall bandwidth by capitalizing on the capabilities of multiple disks right from the GPU.

4 Design

The main idea for GPU4FS's RAID subsystem is that pages of RAID files live in a logical (virtual) address space managed by the file system. A logical chunk is a continuous memory area of arbitrary size within this logical address space. Each logical chunk comes with metadata attached, which defines the properties of the chunk, for example its RAID level 4.1.4. Logical block pointers reference pages in logical chunks within the logical address space 4.1.2. The RAID level can therefore be set on a file or even on a page level.

The advantages of this approach are flexible and dynamic allocation of files and pages of a specific RAID level without having to predefine subvolumes. Another benefit is that performance and redundancy requirements for a specific file can be tuned for the access patterns of a process interacting with these files.

When GPU4FS operates on multiple disks, a distinction must be made between the physical address space of each disk and the logical address space. When a logical chunk is created, chunks of physical memory of multiple disks are allocated and allotted to the logical chunk. These logical data chunks are mapped into the logical address space of size 2^{64} byte. A logical block pointer references a page at an offset in this logical address space.

If a write to a block address in this logical address space occurs, the RAID subsystem transparently resolves the chunk metadata and ensures that the data is written to the correct physical disk(s) and supplement with the necessary information like parity. For example, in the case of a RAID-1 chunk, copies must be distributed to all mirror disks.

To make the RAID system performant, we leverage the parallel nature of the GPU in our design to efficiently handle write and read request to these logical chunks. The goal is to design a flexible software RAID located on the GPU, which can compete in write bandwidth with CPU based RAID systems while relieving the CPU from expensive management tasks.

4.1 On-Disk Data Structures

To incorporate a RAID subsystem into GPU4FS, certain modifications are necessary to its on-disk data structures. First, supplementary information must be introduced to the superblock. This information is required to bootstrap the RAID subsystem during mounting.

GPU4FS's block pointer needs to be modified to enable the distinction between block pointers referencing the local physical address space of a specific disk or the global logical address space.

No modifications are made to the inode structure, given that the inode just stores our updated block pointers without necessitating any supplementary metadata.

The on-disk structure of directories remains unchanged, although a few considerations regarding updates and locking of directories need to be made in the context of a RAID system.

We conclude this section by introducing the additional on-disk data structures logical chunk tree and logical chunk descriptor necessary for the organization of our chunk based RAID system.

4.1.1 Superblock

To be able to (re)mount a RAID instance of GPU4FS, a few additions to GPU4FS's superblock are necessary. Every disk in the disk array maintains its own superblock. Each of these superblocks mirrors common information about the file system instance, therefore acting as a RAID-1 chunk outside the logical address space. The superblock also accommodates disk specific information, which is not mirrored. In case of a disk rebuild, the common information can be copied from another disk with an uncorrupted superblock and the disk specific information is reconstructed during rebuild 4.2.5. Each of the disks is identifiable by a unique UUID.

Common information includes the total physical size and the total remaining physical space of the disk array, the allocation information for the logical address space and an ordered list of all UUIDs of the array's disks.

Disk specific information includes the total physical size and remaining physical space on this disk, a unique UUID to identify the disk and allocation information for the physical address space of the disk. Additionally, physical block pointers to the root node of the logical chunk tree and to the head of the chunk list (the chained leaf nodes of the chunk tree) 4.1.5 are included in the device specific information of the superblock.

Since all disks of the array contain the list of all the disk's UUIDs in their superblock, any disk can be used to start the mounting process. The user space managing process uses the list to verify that all the array's disks are present. If any disk is missing, regular mounting cannot proceed and the user is asked to run a rebuild attempt with replacement disks 4.2.5. In the case that all disks are present, the user space managing process maps the disks in the order of the list to GPU buffers. The order in which the disks are mapped to the buffer is important, as only the list/buffer indices are used in structures like the logical chunk descriptor 4.1.4.

4.1.2 Block Pointer

In order to differentiate between the global logical address space and the physical address space of each disk, modifications to the block pointer are required. By explicitly indicating whether a block pointer refers to a physical location directly on one of the disks or a virtual location in the logical address space, a number of advantages are realized:

RAID can be retroactively enabled on an existing GPU4FS instance without necessitating any modifications to non-RAID files.

Requested pages cannot be mapped directly from disk to the address space of the requesting process for the purpose of safeguarding the RAID system invariants. The requested page's content first has to be copied to a buffer before it can be mapped. Changes in the buffer must be written back to the disks, taking into account the invariants of the RAID system. Supporting physical block pointers allows the inclusion of files bypassing the RAID subsystem. Allowing these page aligned files to be mapped directly from one physical disk to a process.

Given the necessity of employing physical offsets for referencing metadata on the same disk, explicit encoding of a pointer's physical or logical nature simplifies the debugging process.

By supporting both physical and logical block pointers, the system gains increased flexibility at the expense of a single bit.

First, we introduce a new flag called `physical` to indicate whether a block pointer references a physical page (`physical` $= 1$) or a page within a logical chunk (`physical` $= 0$).

For the physical pointer, we need $m = \lceil \log_2 n \rceil$ bits to distinguish between the physical address space of n disks. The disk/buffer index d is encoded within the m uppermost bits of the block pointer, resulting in a maximum addressable physical address space of 2^{64-m} bytes per disk. The disk index d references the disk mapped to the d-th buffer on the GPU. The list of UUIDs in the superblock specifies the order in which the disks are mapped to the GPU buffers, as explained in section 4.1.1. A bit-by-bit breakdown of the physical block pointer can be found in Table 4.1.

disk index		physical offset		00		physical = 1	valid	indirect	tag	
63	$63-m+1$	$63-m$	7	6	5	4	3	2	1	0

Table 4.1: Bit usage of a physical block pointer. Two bits remain unused, m bits are used to encode the disk id and $57 - m$ bits are used for the offset to the referenced page on the specified disk. For physical block pointers, the physical flag is always set. The remaining bits are used to signal whether the pointer is valid and indirect and to differentiate between the size of the page (tag).

For the logical pointer, the pages offset in the global logical address space is included in the pointer. A bit-by-bit breakdown of the logical block pointer can be found in Table 4.2.

logical offset		00		physical = 0	valid	indirect	tag	
63	7	6	5	4	3	2	1	0

Table 4.2: Bit usage of a logical block pointer. Two bits remain unused, 57 bits are used for the offset of the referenced page in the logical address space. For logical block pointers, the physical flag is not set. The remaining bits are used to signal whether the pointer is valid and indirect and to differentiate between the size of the page (tag).

4.1.3 Directories

GPU4FS's directory consist of pairs of file names and inode numbers. In the unmodified version of GPU4FS, the inode number corresponds directly to the physical on-disk offset of the respective inode [23]. In essence, the 64 bit long inode number is a physical block pointer, as the inode itself is page sized by design and aligned to page boundaries.

Rather than using the physical offset of the inode as the inode number, we use a direct block pointer that points to the inode. Therefore, directories contain our updated block pointers as inode numbers. During directory traversal, certain entries might reference inodes situated

within the physical address space of a designated disk, while others reference inodes within the logical address space. Using a block pointer as the inode number, allows us to include both physical and logical files in the same directory.

Directories are special files and can thus be included in a logical chunk. GPU4FS gains exclusive write access to a directory by atomically locking its inode directly on-disk. For a RAID backed directory, this on-disk locking is not optimal, as every lock and unlock requires a parity or mirror update to avoid violating the RAID invariants. As a solution, we propose an atomically updatable list in VRAM of all locked directories instead of relying on on-disk locking.

4.1.4 Logical Chunk Descriptor

The logical chunk descriptor contains metadata describing a logical chunk, primarily its RAID level and size. This metadata must be redundant to enable chunk recovery. If we want to recover from m disk failures, we need to mirror the logical chunk descriptor to $m + 1$ disks. However, mirroring does not necessarily occur at the same offset on all disks. We always allocate each logical chunk descriptors on the same $m + 1$ disks. The space required for these logical chunk descriptors is negligible, as they are only 128 Bytes in size. To achieve high performance, a RAID instance should keep the amount of logical chunks (and therefore the amount of logical chunk descriptors) small to keep address lookups in the chunk tree fast 4.2.3.

A byte-wise breakdown of the metadata of a logical chunk can be found in Table 4.3.

	0	1	2	3	4	5	6	7
0	size							
8	raid_level	redundancy	*unused*		stripe_length			
16	linear_allocator							
24	num_stripes				*unused*			
32	physical_chunk_size							
40	list of physical chunks							
⋮								
120								

Table 4.3: Byte wise composition of the metadata describing a logical chunk.

All logical chunks possess their own block allocator, responsible for managing allocation requests within the logical chunk 4.2.2. The logical chunk descriptor includes a list of physical chunks distributed across different disks, collectively forming the logical chunk. This list solely comprises physical block pointers, which already encapsulate the disk ID and physical offset 4.1.2. The block pointer's tag is disregarded, as the physical chunk size may not align with any of GPU4FS's page sizes. Hence, the physical chunk size is explicitly included within the logical chunk descriptor (`physical_chunk_size`). Each physical chunk has the identical size.

The stripe length field denotes the number of bytes written sequentially to a single disk before writing the next stripe length bytes to the next physical chunk in the cyclic list. The stripe length therefore specifies after how many bytes the data should be striped.

A notable field in the chunk descriptor is the redundancy value. For RAID-1, it encodes the number of mirrors per set, while for RAID-6, it contains the amount of parity disks. Consequently, for RAID-1, the physical chunk list encompasses `redundancy` $\cdot$ `num_stripes` entries, organized into `num_stripes` sets.

By default, the logical chunk is sized to match an inode, allowing it to accommodate $(128 - 40)/8 = 11$ block pointers pointing to distinct physical chunks. If more physical chunks are needed due to the availability of more than 11 disks, the data structure can be conveniently extended to a multiple of inode sized pages. The space available due to the extension can then be used to accommodate $128/8 = 16$ additional pointers per inode sized page.

4.1.5 Logical Chunk Tree

The logical chunk tree serves as the central data structure organizing the allocated logical chunks within the RAID subsystem. Structured as an ordered, self-balancing B+-tree, the chunk tree facilitates the mapping of logical addresses to their corresponding logical chunk descriptors. Fast logical address translation, as outlined in section 4.2.3, is enabled through the tree structure. Furthermore, the logical chunk tree acts as a list of all allocated chunks, offering a means to iterate through all existing logical chunks in the system.

To ensure the ability to recover from m disk failures, we require $m + 1$ replicas of the chunk tree on different disks. Additionally, on each of the $m + 1$ disks, there is a local copy for every logical chunk descriptors positioned at distinct physical offsets, as detailed in section 4.1.4. It's important to note that each chunk tree contains physical offsets pointing to their corresponding local replicas of the logical chunk descriptors on the same disk. This configuration ensures redundancy and enables efficient recovery.

B+-Trees adhere to specific invariants defined by the minimum degree, denoted as t. All non-root nodes are required to possess at least $t - 1$ keys. The root node, following its initial insertion, may hold a minimum of one key. Each node can hold a maximum of $2 \cdot t - 1$ keys, all arranged in ascending order. The child situated between two keys, k_i and k_{i+1}, encompasses all keys within the range $[k_i, k_{i+1})$. Hence, we have a pointer between two adjacent keys, therefore we need two additional pointers, one for the smallest key and one for the largest key. The pointer associated with the smallest key directs to the child that holds all keys smaller than the node's minimum. Similarly, the pointer corresponding to the largest key directs to the child containing keys equal to or greater than the node's maximum. Consequently, the amount of child nodes equates to the key count plus one. A visualization of an inner node's keys and corresponding child pointers can be found in Figure 4.1. Importantly, B+-Trees maintain uniform leaf levels and exclusively insert values into leaves [6].

The base addresses (offsets) of the logical chunks are used as the sorting keys in the B+-tree. The inner nodes exclusively store keys along with their corresponding $<$ and $\geq$ child references. This configuration establishes an index over the base addresses, allowing rapid address lookups with a time complexity of $\mathcal{O}(\log n)$, where n signifies the amount of inserted logical chunks [11]. Inserting a chunk into the tree also has a time complexity of $\mathcal{O}(\log n)$ [11].

Leaf nodes include the base address (key), size (limit) and offset to the logical chunk descriptor of the contained logical chunks. The offset is physical, as the replica of the logical chunk descriptor is stored on the same disk as the chunk tree. The Leaf nodes also includes a

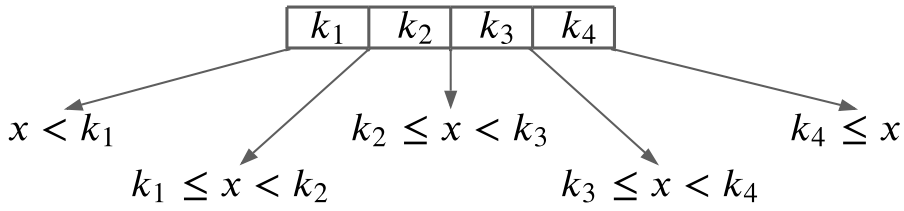


Figure 4.1: Visualization of an inner node with four keys and the five corresponding child pointers.

subset of the metadata from the chunk descriptor for each contained chunk. These leaf nodes are chained into a sorted list using a next pointer. To distinguish between inner nodes and leaf nodes, a leaf flag is incorporated within every node.

The design ensures the absence of duplicates within the chunk tree. The logical chunk allocator ensures that the address ranges of the chunks never overlap, thereby guaranteeing an unambiguous mapping from logical address to logical chunk.

In lower level implementations of B+-Trees, nodes possess a size equivalent to that of a page. Keeping nodes page sized allows efficient caching of frequently accessed nodes in memory and enables infrequently accessed nodes to be swapped out effectively. The choice of the minimum degree t depends on the selected page size, as the number of keys a node can accommodate is constrained by this size [11].

To minimize the overhead linked to the logical chunk abstraction, a pragmatic approach involves utilizing rather large sized chunks. Using large chunks results in a limited amount of logical chunks within the system, keeping address translation fast and making debugging more manageable during development. Since we don't have many chunks in the tree, we choose the smallest page size (inode size of 128B) for internal nodes.

Within an internal node, 16 bytes are allocated for metadata, such as the amount of keys within the node and a leaf flag to differentiate internal nodes from leaves. Consequently, $128 - 16 = 112$ bytes remain for keys and child pointers. Considering the need for one more child pointer than keys, this leaves room for $(112/8 - 1)/2 = 6.5$ keys. Solving the equation $2 * t - 1 = 6.5$ yields $t = 3.75$, rounded down to $t = 3$. Thus, each node can accommodate $2 * t - 1 = 2 * 3 - 1 = 5$ keys. For a precise description of the byte-aligned internal node data structure, refer to Table 4.4.

	0 7
0	num_keys
8	metadata
16	list of 5 base addresses (keys)
48	
56	list of 6 child pointers
96	
104	unused
120	

Table 4.4: The byte aligned inode sized inner node.

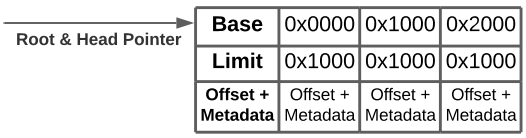
Concerning leaf nodes, 16 bytes are allocated for metadata, encompassing the amount of keys and the leaf flag that designates a leaf node. An additional 8 bytes are allocated for the physical offset of the subsequent leaf node, facilitating the construction of an ordered linked list of all leaf nodes. As leaf nodes lack children, no child pointers are stored within them. Each key (base address) requires storage of the limit (chunk size) and the physical offset to the corresponding logical chunk descriptor for address translation. Additionally, a subset of metadata from the logical chunk descriptor is retained, including information like the RAID level.

Each of these data components occupies 8 bytes, and with storage required for 5 keys, the total space consumption becomes $(2+1+5*4)*8 = 23*8 = 184$ bytes. To ensure alignment with page boundaries, the size of a leaf node is extended to two inode-sized pages (256 bytes). The precise byte-aligned structure of a leaf node is presented in Table 4.5.

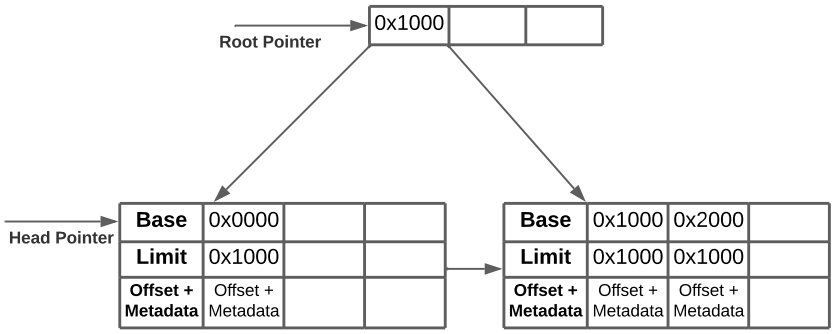
	0 7
0	num_keys
8	metadata
16	next_offset
24	list of 5 base addresses (keys)
56	
64	list of 5 limits
96	
104	list of 5 offsets
96	
144	list of 5 chunk_metadata
176	
184	unused
248	

Table 4.5: The byte aligned two inode sized leaf node.

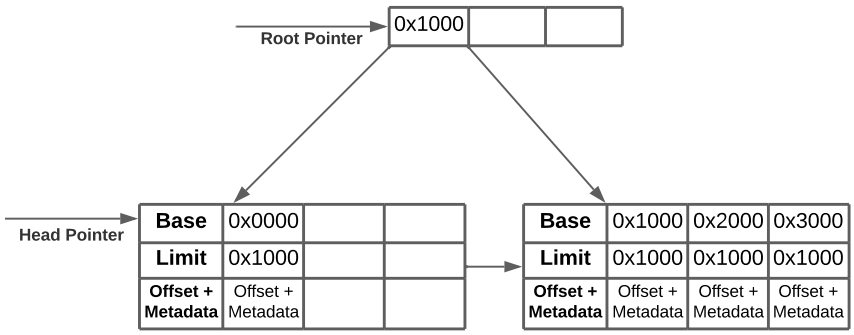
To understand the insertion of a chunk into the tree, consider the following example. We choose $t = 2$ so each node can hold up to $2 * t - 1 = 2 * 2 - 1 = 4 - 1 = 3$ keys. We want to insert chunks of size 0x1000 into the tree. In the beginning, we have an empty leaf node as the root. Both the head pointer of the list and the root node pointer reference the root. We insert the chunks with the base address 0x0000, 0x1000 and 0x2000 into the root node.



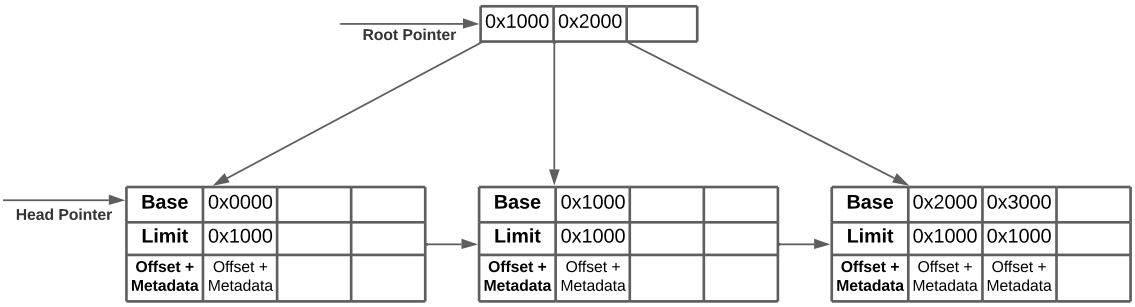
When we try to insert the next chunk with the base address 0x3000, we discover that the root node is full and needs to be split. We need a new root node, but this time the root is an inner node. We promote the t-th (second) key to the new root, and copy all the keys k and their corresponding metadata with $k_t \leq k$ to a new leaf node. We set the root pointer to the new root and set the next pointer of the split leaf node to the new leaf node. We add the child pointers to the leaf nodes to the new root.



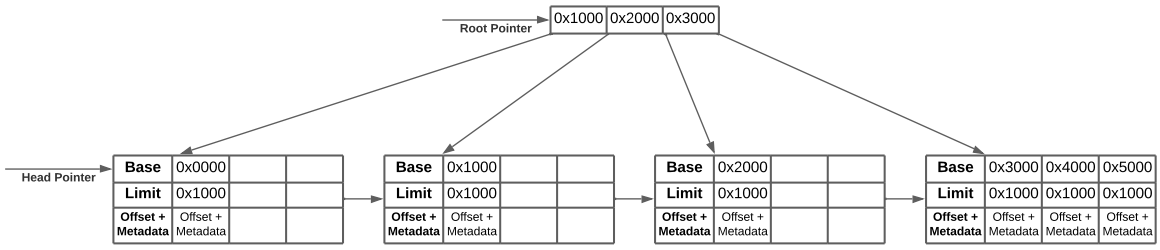
Now we can proceed to insert the 0x3000 chunk.



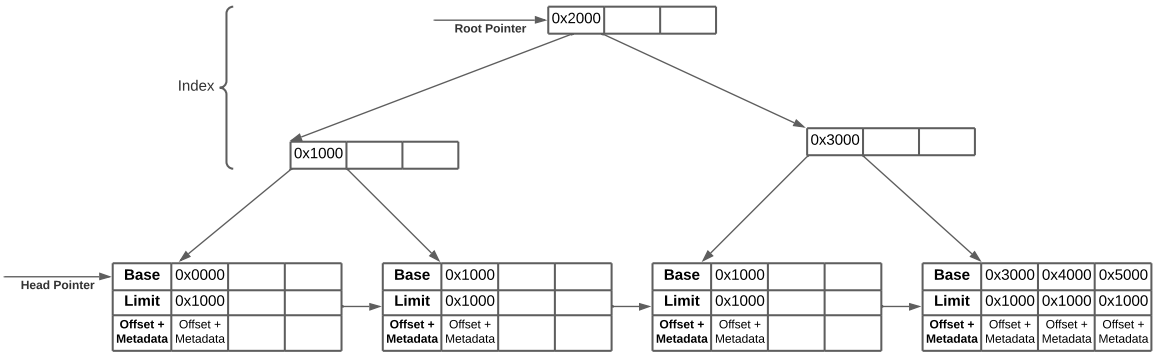
When we attempt to insert the 0x4000 chunk, we discover that the target leaf node is full. We split the full leaf into two analog to the first split, but this time we insert the t-th (second) key as an index key into the existing root.



Inserting 0x4000 results in a full leaf node, when inserting 0x5000 the full leaf node is split similar as before and 0x3000 is promoted to the root as an index key.



When inserting 0x6000, we discover that the root node is full and preemptively split it, promoting the index key 0x2000 to a new root.



Afterwards, we can continue to insert 0x6000, which will split the full leaf node and promote 0x4000 as an index key to its parent node.

To summarize the insertion process, we follow the index formed by the inner nodes. If we encounter a full inner node, we split it, and promote the key k_t at position t to the parent node. If the root is full, a new root needs to be allocated and the root pointer needs to be updated. When a full leaf node is split, we need to copy the t biggest keys and metadata to the new leaf node and update the next pointer of the split leaf node to the new leaf node. The key k_t is also inserted into the parent inner node as an index key.

Splitting full inner nodes while traversing down is called preemptive splitting. For example, while we traverse down to the target leaf node, we encounter two full inner nodes, we discover that the target leaf node is full as well and requires splitting. We promote its key to the parent, but the parent is full as well and also requires splitting, but its parent is also full and requires splitting.

Preemptive splitting avoids splits propagating up the tree, allowing us to implement B+-Tree without needing back pointers to a node's parent or recursion. The importance of the previous point stems from the fact that back pointers require additional space, and recursion is simply not possible in GLSL.

4.2 Runtime

In this section, we cover how the introduced data structures are used to facilitate read and writes to the RAID subsystem. First, we explain how a logical chunk can be allocated and how pages

can be claimed from a logical chunk. We describe how address translation is used to resolve an accessed page from a logical chunk. We continue by explaining how reads and writes are orchestrated and conclude by explaining how the system can recover from disk failures.

4.2.1 Logical Chunk Allocation

A logical chunk is allocated when a new file is created and no existing chunk of the requested RAID level offers sufficient space 4.2.2. Alternatively, a blank chunk might be explicitly requested. The size of a logical chunk can either adopt the default system value, initially set through the metadata command 5.3.1, or assume the maximum of the chunk size and the initial file size specified in the file write command.

Allowing the requesting process to specify the RAID level and chunk size at runtime brings several benefits. The application can offer insights into application-specific sizes that align with GPU4FS's block sizes. This permits flexible tuning of a chunk's performance configuration by the file system or the user based on benchmark results. Moreover, the file system driver could intelligently derive the block sizes accessed by the application, enabling it to optimize chunk size and characteristics like stripe length for improved application-specific performance.

Permitting partial writes to full-stripes adds complexity to parity calculations and introduces challenges in RAID management. To address this, we round up the chunk's size to encompass only complete full-stripes, while zeroing out any remaining bytes in partially used full-stripes. Once the logical chunk's size is determined, the global (logical) allocator is used to claim a portion of the logical address space. The global allocator returns the base address of the logical chunk. It is essential to update the global allocator's state in each superblock 4.1.1.

Following this, we need to allocate the physical chunks constituting the logical chunk. Calculating the size of each physical chunk requires consideration of the following factors. For RAID-0, the logical chunk size is divided by the number of stripes, as the data gets equally distributed among the stripes. RAID-1 necessitates allocation of physical space equal to the logical chunk size times the number of mirrors (redundancy). The result is divided by the number of disks (stripes), For RAID-5, the logical chunk size is divided by the number of stripes minus one, as a complete physical chunk's size is needed for parity. The same applies to RAID-6, except here, the logical chunk size is divided by the number of stripes minus the number of parity disks (redundancy). The calculated physical chunk size is rounded up to align with page boundaries.

It is worth noting that rounding up each physical chunk to align with page boundaries results in a surplus of unused physical bytes. The resulting physical chunk size should be taken into consideration when requesting a chunk size for a specific RAID level and number of stripes configuration.

With the physical chunk size determined, the next step is to allocate physical chunks from a set of distinct disks. Several strategies can be employed to select the disks for allocation. The simplest approach involves allocating from all available disks in a round-robin manner. While effective when all disks share the same characteristics, such as bandwidth and size, this approach might not be suitable when dealing with disks of varying sizes. In this scenario, selecting disks with the lowest space utilization helps to maintain balance. If significant bandwidth variations exist among the disks, users could specify the preferred performance

group for a chunk, aiding in the selection of appropriate disks. Alternatively, monitoring the access frequency of each disk and allocating from the least frequently accessed disk could achieve balanced access across the disk array. This latter approach can also be used online to migrate highly accessed chunks to separate disks for better load distribution.

After physical chunks are allocated and their physical block pointers buffered, logical chunk descriptors are created on $m + 1$ disks 4.1.4. Initially, the logical chunk's block allocator is initialized, reserving the bytes for the file write request that triggered the creation of the chunk. The physical chunk size is recorded in the logical chunk descriptor for tasks such as recovery and defragmentation. In addition, the chunk's size, RAID level, number of physical chunks, the stripe length, and the redundancy for RAID levels 1 and 6 is copied to the logical chunk descriptor 4.1.4.

The physical block pointer to the local logical chunk descriptor is inserted into the local logical chunk tree 4.1.5. Once the logical chunk is incorporated into every tree, the original file write request can proceed 4.2.4. Subsequent requests can allocate pages from the allocated logical chunk if sufficient space is available 4.2.2.

To free a chunk, all pages within in the chunk, must have already been freed. The logical chunk is then removed from the chunk tree. Afterwards, the physical chunks are freed and finally the logical chunk descriptors are freed. Removing a chunk from the chunk tree, may result in a node to contain fewer than $t - 1$ keys. In that case, nodes need to be merged, potentially up to the root, to sustain the invariants of the B+-Tree [11].

4.2.2 Page Allocation From Chunks

To allocate pages of a specific RAID level, we first check for sufficient space in the existing logical chunks. For each RAID level, we maintain a free list of chunks with available space. The elements in the free list contain the base address of the logical chunk, an offset to its descriptor, and the available space. If a chunk in the free list with available space is found, we use its page allocator to claim the requested pages from the chunk 4.1.4.

If the allocation succeeds, the mirror logical chunk descriptors need to be updates as well. We can resolve their offsets, by searching for the chunk's base address in their respective local logical chunk tree.

For every allocated page, we return a logical block pointer. The logical offset is the sum of the chunk's base address and the offset to the page within the chunk, returned by its page allocator.

To free a page, we simply return the page to the chunk's allocator. Afterwards, we insert or update the chunk's available space in the appropriate free list.

4.2.3 Address Translation

We have to translate read/write requests to a page via a block pointer to the RAID level dependent disk operations. We need different translation strategies for physical and logical block pointers. First, we check if the physical flag is set to tell physical and logical block pointers apart 4.1.2.

For physical block pointers, we take the m uppermost bit encoding the disk index to determine which buffer we need to access 4.1.1. To access the requested page, we take the physical offset included in the pointer to access the earlier identified buffer to facilitate reads and writes to that page.

For logical block pointers, we take the logical offset to search the chunk tree. As mentioned in 4.1.5 we only have the base addresses as keys in the chunk tree, but we store the corresponding limits in leaf nodes. Using the logical offset instead of the base address still directs us to the correct leaf, as we always choose the child node between the base addresses $k_1 \leq$ address $< k_2$ until we reach a leaf node. The chunks do not overlap by design 4.1.5, guaranteeing that the logical offset always satisfies the inequality $k_1 \leq$ address $< k_2$ for the sorted keys (base addresses) within an inner node.

Once we reached a leaf node, we linearly follow the sorted base addresses and check if the logical offset is between the current base address and the corresponding limit plus base. If no match is found within the target leaf node, the address is invalid and an error is raised. If a match is found, we get the offset to the logical chunk's descriptor.

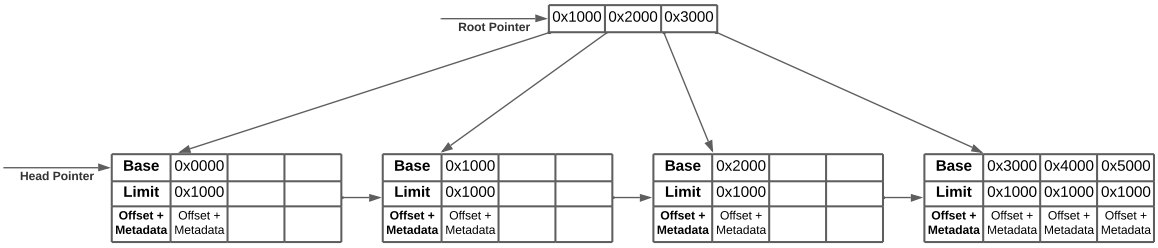


Figure 4.2: A sample logical chunk tree containing five 0x1000 bytes sized logical chunks.

Consider the example chunk tree in Figure 4.2. If we want to resolve the chunk for the address 0x4080, we sequentially walk the index keys in the root node until we find a bigger or equal key than 0x4080. As there is no bigger or equal key, we take the last child pointer and find a leaf node. In the leaf node, we sequentially check if our search value is between the current base and limit (base $\leq$ value $<$ base + limit). This is the case for the second key 0x4000, we follow the offset to the corresponding logical chunk descriptor.

The inner logical chunk offset is equal to the logical offset from the block pointer minus the chunk's base address.

The next step is to determine on which physical chunk and what inner physical offset the page begins, This depends on the RAID level configuration and the stripe length.

To determine the starting position of a page address within a RAID system, we follow different procedures based on the RAID level.

For RAID-0 we need to complete the following steps:

1. First, we find the number of stripes that precede our target stripe. This is achieved by dividing the inner logical chunk offset by the stripe length, using integer division.

2. To pinpoint the offset within the current stripe, we calculate the modulo of the same operands.

3. Next, we determine how many full-stripes come before our target. This is achieved by performing an integer division of the number of stripes (as calculated in step 1) by the total number of data stripes.

4. The modulo operation on the same operands identifies the current stripe within the current full-stripe.

These calculations provide us with the precise starting location of the page: both the full-stripe and the stripe within that full-stripe, including the offset within the stripe.

For RAID-1, the steps are identical. For RAID-5 and RAID-6 the calculations are similar, but we adjust the stripe count by adding the number of parity stripes found within the full-stripes (step 3.) preceding our target stripe and within the target full-stripe. After we updated the stripe count, we need to repeat step 3 and 4. This accounts for the additional parity stripes introduced by these RAID levels.

We have now determined the starting point from where we can write or read the page according to RAID level, as explained in the next section.

4.2.4 Reading and Writing

For reading, we simply translate the block pointer to the beginning of the referenced page as explained is section 4.2.3 and copy the page's content from the data stripes to VRAM. For a RAID-1 read, we can simply choose a mirror chunk to read from and for RAID levels 5 and 6, we just skip the parity stripes while copying the data. From VRAM, the page can be mapped to the requesting process.

For writing, we need to touch multiple disks to ensure the RAID level invariants. For RAID-0 we simply translate the page's address and copy data words until we reach the end of the stripe. We resume copying on the next physical chunk until all content has been copied. For RAID-1 we duplicate a write to all mirrors, while ensure that a SIMD thread always writes to the same mirror at a time, to keep accesses sequential.

For RAID level 5 and 6, we first consider how complete full-stripes are written to disk, before discussing how a stripe can be updated. Each stripe contains an amount of data words. This amount is a multiple of the workgroup size. When writing a stripe's data words to disk, we distribute the writes to the invocations of the workgroup. To visualize this, we organize the stripe's data words into k rows, where each row contains workgroup size data words. Each invocation is responsible for copying the data words corresponding to its column to disk and compute the parity for each stripe corresponding column. This visualization can be found in Figure 4.3.

Distributing the data words to invocations in this way, results in SIMD threads only accessing memory sequential, which increases bandwidth and allows for independent parity calculation without synchronization overhead.

For RAID-5 we maintain a parity cache of a stripe's capacity. The parity stripe is calculated independently on a word by word basis. Due to the independence between parity words, we can distribute their calculation to the invocations, eliminating the need for communication between them. While copying the data stripes to one disk at the time, the invocations update

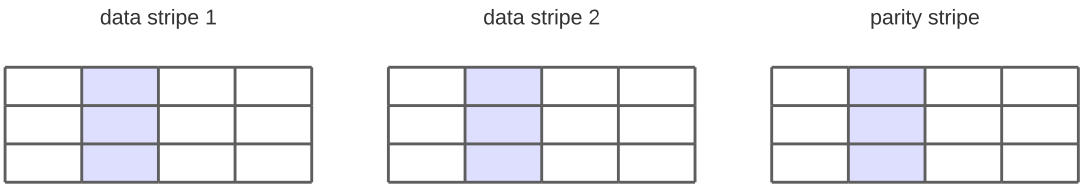
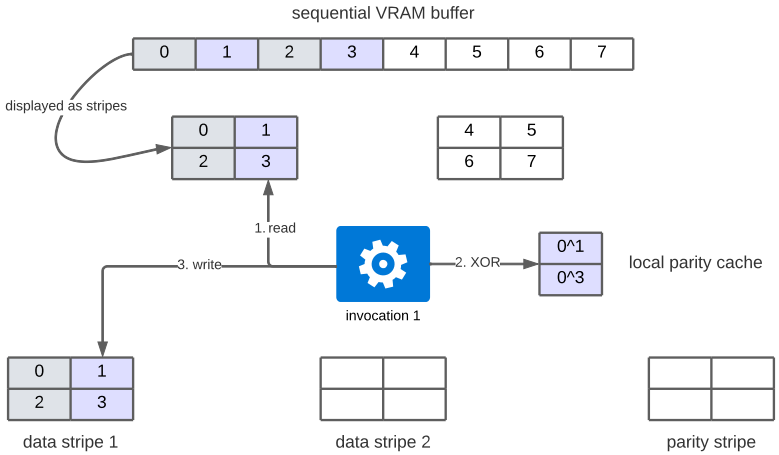


Figure 4.3: A full-stripe, containing two data stripes and one parity stripe. A stripe contains 12 data words, organized into three rows. Each column is designed for one invocation (4 invocations in total). For example, the second invocation is responsible for writing each stripe's second column and calculating the parity for data words in the second column.

their portion of the parity cache in their local memory by XORing the read data word with their cache's content. After a full data stripe is written, the invocations copy their portion of the parity cache to the disk and the cache is zeroed. This allows us to only touch the data buffer once and use the fast private local memory, without any synchronization overhead. As a stripe contains k times the workgroup size data words, one invocation calculates and caches k parity words.

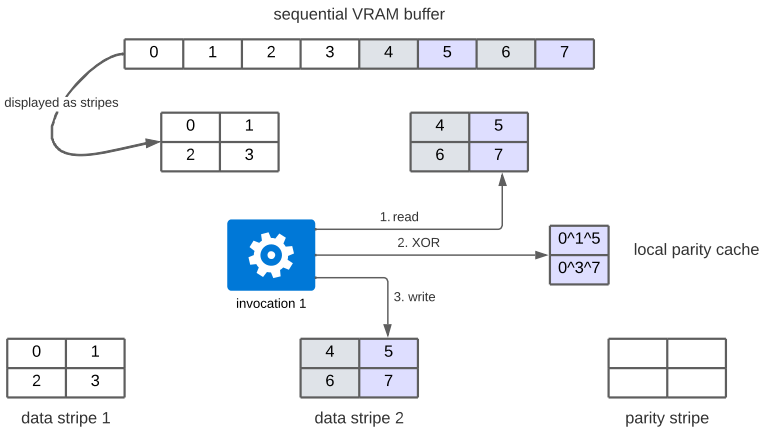
Consider an example where a sequential VRAM buffer contains eight data words. We want to copy these data words into a RAID-5 chunk containing three stripes, with a stripe length of four data words. We have a workgroup consisting out of two invocations. Invocation 0 and 1 work in parallel on copying the buffer. Since the stripe length is four, and we have two invocations, we organize the stripe into $k = 4/2 = 2$ rows. The workgroups copy the data stripe by stripe, and write stripes row by row.

For the visualization, we focus on invocation 1 (purple), but keep in mind that the following steps happen for invocation 0 (grey) in lockstep for its column.

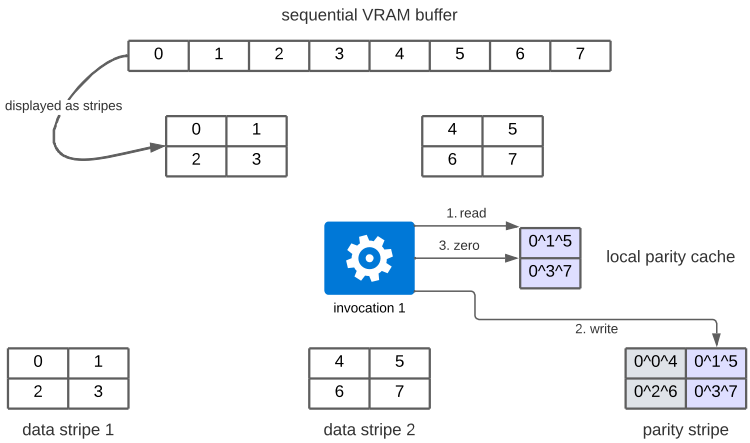


Note that below the buffer in visualization we displayed the buffer's data organized in stripes, this is just for visualization purposes the data words contained in the sequential VRAM buffer.

Invocation 1 is responsible to copy each row of the second column and compute the parity for the second column. First, invocation 1 reads data word 1 from the buffer, XORs it with its zero initialized parity cache and writes the data word to the first data stripe. Since invocation 0 does the same for data word 0, the GPUs MMU can coalesce their buffer reads and disk writes into sequential accesses. The parity cache is in private local memory, allowing invocations fast access without any synchronization. The invocations 0 and 1 repeat the steps for the second row (data words 2 and 3). Now the first stripe is completely written, and the invocations begin to copy to the next stripe.



The steps are repeated for the next stripe. Note, that the local parity cache includes the XOR result of the previous stripe's column. After the last data stripe is completely written, the local parity cache holds the computed parity for the second column.



In the last step, the invocations copy the parity row by row to the parity stripe, resulting in sequential writes. After each row is written to the parity cache, the invocations zero out their local parity cache for the next full-stripe.

For RAID-6 the parity is also calculated on a word by word basis. We also employ the distributed parity cache, but it has to hold m parity stripes and m is known at runtime. As mentioned in 2.1.1, the Reed-Solomon coding works on a word by word basis, which enables the invocations to calculate the parity independently without needing to communicate analog to RAID-5.

As the computation is independent, we focus on one invocation for now, but keep in mind the calculations happen in parallel on the other invocations. Again, we distribute the cache to the invocations private local memory. To understand the computation of the parity disks, look at the following example: We have three data words d_i and two parity words c_j we want to compute from the data words. $f_{i,j}$ denotes the entries of the Vandermonde matrix. The parity words c_1 and c_2 are calculated like this:

$$\begin{pmatrix} f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,1} & f_{2,2} & f_{2,3} \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} f_{1,1} \cdot d_1 + f_{1,2} \cdot d_2 + f_{1,3} \cdot d_3 \\ f_{2,1} \cdot d_1 + f_{2,2} \cdot d_2 + f_{2,3} \cdot d_3 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

Instead of first copying the data and then doing the parity calculation, we do both simultaneously, computing the parity step by step. In the first step, the invocation copies the first data word d_1 to the first stripe. The invocation uses the scalar d_1 to scale the first column of the Vandermonde matrix, which has m rows, and write the resulting vector of length m to the invocation's local cache. Next, the invocation copies the data word d_2 to the second stripe. The invocation use the scalar d_2 to scale the second column of the Vandermonde and add the result to the value in the cache. The invocation repeats these steps until all data words are copied. Afterwards, the invocation has the m parity words in its local cache and copies one at a time to the parity stripe of the full-stripe. As a reminder, the add and multiply operations are performed over Galois Fields 2.1.1. As a stripe contains k rows, one invocation calculates and caches $k \cdot m$ parity words.

Note that for RAID-5 and RAID-6 additional zeroed data words may need to be copied to make writes full-stripe. Otherwise, the parity won't include the data of the complete full-stripe and a subsequent rebuild will fail.

For partial updates to a full-stripe, we first need to read in the data words which are about to be overwritten. With the new and old data word, we can calculate the parity update in the local cache as explained in 2.1.1 for RAID level 5 and 6. We can proceed to write the updated data words and then update the corresponding parity words with the parity update. Note that for partial row updates, some invocations need to idle.

If the complete full-stripe is overwritten, we can just consider this as a new write as explained earlier.

4.2.5 Rebuild

For rebuilding, let's assume m disks failed, and we know which of the disks failed and replace them with spare disks. While mounting the disk array, the CPU detects the replacement disks and after confirmation from the user, the CPU continues to dispatch the shader code to the GPU in rebuilding mode. The GPU copies a new superblock containing the common information from an intact disk's superblock to the replacement disks.

As m disks failed, we at least have one working copy of the chunk tree and of every logical chunk descriptor. We use the list formed by the tree's leaf nodes to verify and attempt to restore the integrity of every chunk. For rebuilding, the existing chunks are split among the workgroups. Each workgroup is responsible for restoring the allotted chunks.

For a chunk, a workgroup first has to check whether the replaced disks were used in the chunk. If not, the chunk is still functional. Otherwise, the corresponding physical chunks need to be restored by the workgroup.

A RAID-0 chunk offers no redundancy and can therefore not be restored. The user can opt to free the resources associated to the logical chunk like the chunk descriptor and the physical chunks or choose to insert zeroed replacement physical chunks instead. This might be useful if the user wants to attempt data recovery, requiring application specific knowledge about the chunk's data. If more than m disks fail, and we still have a copy of the chunk tree, the user can consider the same approach for the other RAID levels. If no copy of the chunk tree persists, no recovery is possible.

In any case, we first need to allocate the new physical chunks on the replacement disks and update each logical chunk descriptor, with the new physical chunk offset. The size for the physical chunk is taken from the logical chunk descriptor 4.1.4.

For RAID-1 the invocations copy the contents of intact physical mirrors to the newly allocated physical chunks. Again, if all the mirror disk of one set fail, we cannot rebuild the chunk.

For RAID-5 a chunk can only be repaired if exactly one disk failed $m = 1$. In that case, the invocations XOR the intact stripes to restore the content of the replaced physical chunk.

For RAID-6 we can recover from $m \geq 2$ disk failures. The intact data of a full-stripe needs to be read to memory to solve the linear equation system as outlined in 2.1.1. As the linear equation system can become quite big if multiple disks are used and to avoid branching during the solving process, a SIMD thread's invocations need to work together on solving one equation system at a time. Afterwards, the invocations write back the restored data and repeat the steps until the entire chunk has been restored.

After all chunks have been restored, the RAID subsystem has to ensure that $m + 1$ copies of the chunk tree and of each logical chunk descriptor exist. Afterwards, the system resumes normal operation and workgroups begin to work on processes' requests in the command buffer.

5 Implementation

This chapter outlines the integration of our RAID subsystem into the GPU4FS demonstrator. The proposed design was realized with certain simplifications necessitated by time constraints and limitations imposed by GLSL.

We opted not to implement RAID-6 due to the time constraints it posed. The reconstruction process involves efficiently solving linear equation systems over Galois Fields, which would have constituted a significant undertaking deserving a separate thesis.

The chapter commences by an explanation of the limitations imposed by GLSL and the existing GPU4FS demonstrator. Subsequently, we explore how these limitations influenced the implementation of the chunk tree design. We then detail the process of integrating our RAID system into the GPU4FS demonstrator.

We discuss our initial approach that we abandoned due to its poor performance, and explain how the insights from this approach motivate the current design. We proceed to present the implementation of this refined design, providing explanations for selected code snippets.

Furthermore, we describe the role of the CPU in our development process and benchmarks, illustrating how it verifies the GPU's work.

To conclude the chapter, we outline the implementation of the disk array rebuild.

5.1 Limitations

A major limitation of GLSL is the lack of dynamic memory allocation. Shared and local memory can only be statically allocated, and VRAM can only be used in predefined buffers bound by the CPU before the shader is dispatched. These limitations restrict us to statically limit the size of some data structures, such as the local parity cache, and limits us to only update data structures in place on disk.

Another challenge is the lack of a working block allocator, ensuring page aligned allocations on the disks. To make allocations, a simple linear allocator is used, which reserves the requested bytes via atomic adds to a counter and returns the previous counter value as base offset. The caller must ensure to round up the request to a multiple of the inode size, otherwise the block pointers do not work. To synchronize the counter of a linear allocator across multiple disks, requests are performed exclusively on a single, predefined disk and the copies of the counter are updated via an atomic maximum operation.

5.2 Logical Chunk Tree Implementation

In a previous iteration of the design, a B-Tree was employed instead of a B+-Tree. The key distinction lies in the fact that B-Trees permit data storage within their inner nodes, as opposed to B+-Trees, where the inner nodes solely function as an index. B+-Trees exclusively store data within their leaf nodes, enabling these leaf nodes to be linked together into a sorted list.

To address the need of allocating pages from pre-existing chunks and for reconstruction purposes, we required a list of existing chunks. For the former, we initially proposed free lists in our design. However, due to the lack of dynamic memory allocation and deallocation capabilities, we chose not to implement the free list. This decision stems from the dynamic nature of free lists, demanding unpredictable allocation and deallocation calls. Nevertheless, in order to maintain a list of existing chunks, we converted our previously implemented B-Tree into a B+-Tree.

The implementation of the B+-Tree in GLSL posed a challenge. GLSL provides support for structs but lacks support for pointers [3], making it impossible to cast anonymous buffers to a struct pointer. Interacting with an on-disk data structure requires adding up various offset to access an eight-byte data word, followed by bitwise operations to extract the required sub-bits from the data word. During development, the most prevalent issues stemmed from miscalculated offsets and forgetting to convert between byte offsets and buffer indices. Due to the unavailability of shader code debugging, we had to analyze the contents of each disk using a hexadecimal editor after the shader's completion. Although one adapts to this form of debugging, it significantly slows down development, particularly when dealing with dynamic, non-sequential data structures like the B+-Tree.

To ensure synchronized access to any tree copy across workgroups, a workgroup must obtain a global read-write lock, situated in VRAM. Updates, including insertions, need to be applied to all instances of the chunk tree. For each update operation, the write lock is acquired just once, and all tree copies are updated before the lock is subsequently released.

An additional challenge arises from using the non-exclusive read lock before a workgroup scans the existing chunks for available space. Initially, when no chunks are available and the lock is non-exclusive, all workgroups reach the same conclusion that no chunks exist and proceed to allocate their own chunks. If the number of files exceeds the number of workgroups, subsequent files are allocated to the existing chunks. If all files should be written to a single chunk, an exclusive lock is imperative, necessitating sequential processing for all chunk allocations. A potential workaround involves allocating an initial chunk before workgroups commence working on the command queue. Consequently, despite having implemented the capability to locate and allocate pages from pre-existing chunks, we ultimately decided not to use it, opting to allocate a separate chunk for each file. We decided on this to avoid the situation for RAID-5 where workgroups have to synchronize parity calculations for adjacent files on a full-stripe. This situation is explained in detail in a later section 5.5.

5.3 Commands

In this section, we explain how we integrated our RAID subsystem into the GPU4FS demonstrator. We used the file write command as our entry point. However, before processing the file write commands, some configurations have to be done, therefore we had to extend the metadata command as well.

5.3.1 Metadata

We expanded the functionality of GPU4FS's metadata command to establish default values for our RAID subsystem. We augmented it by incorporating information about the number of disks connected to the GPU. In our specific implementation, we defined four buffers in the shader. Each disk is mapped to a single of these buffers. Within the GLSL framework, all defined buffers have to be mapped. Consequently, even if we only use two disks, we are still required to map the two remaining buffers. In our particular scenario, we opted to map empty files to allow the shader to execute. We then pass the maximum number of available disks to the shader via the metadata command to instruct the shader to ignore the unneeded buffers.

Since the mounting process must be initiated on the CPU, the CPU needs to communicate the IDs of the failed/missing disks to the shader for a rebuild before regular operations can resume. This communication is facilitated through the `disk_to_repair` field within the metadata command. However, it's worth noting that in our current demonstrator, we've only implemented the capability to recover from a single disk failure. Consequently, we only include one ID in the metadata command.

Additionally, it is possible to specify the default size for a logical chunk via the command buffer. As the command buffer possessed ample unused space, we employed an inefficient eight-byte data word for each parameter, avoiding the need for bit shifting. The byte alignment of extended metadata command can be found in Table 5.1.

	0 7
0	type tag
8	(next)
16	separated_execution
24	num_work_items
32	num_disks
40	disk_to_repair
48	chunk_size
56	
104	
112	(atomic_acquire)
120	atomic_complete

Table 5.1: The extended metadata command.

5.3.2 File Write

To integrate RAID into GPU4FS, additional parameters had to be incorporated into the file write command. These parameters encompass the RAID level, the number of disks designated for chunk allocation (`num_stripes`), the stripe length, and redundancy. In case of a physical write (`raid_level` $= -1$), the target disk can be specified.

Once more, we chose the most convenient approach, avoiding bit shifting in favor of a less space-efficient method for data alignment. The byte alignment of extended file write command can be found in Table 5.2.

	0 3	4 7
0	type tag	
8	next	
16	file_size	
24	num_SIMD_lanes	
32	file_data_offset	
40	filename_length	
48	directory_position	
56	inode_offset	
64	inode_position	
72	file_position	
80	raid_level	
88	num_stripes	disk
96	stripe_length	redundancy
104		
112	atomic_acquire	
120	atomic_complete	

Table 5.2: The extended file write command.

5.4 Caching

In our design, only one workgroup works on writing an individual file. All the workgroup's invocations need to know the information about the target chunk contained in its logical chunk allocator. To avoid that each of the workgroups needs to read in the logical chunk descriptor from disk, it is copied to faster shared memory. One invocation is responsible to read logical chunk descriptor into shared memory. The invocation first resolves the target address to the offset to the logical chunk descriptor by walking the chunk tree as explain in section 4.2.3. After retrieving the offset, the invocation initializes the shared logical chunk descriptor cache. Allowing the workgroup's other invocations to quickly access the logical chunk descriptor without having to walk the chunk tree and retrieving it from disk themselves.

5.5 Writing

GPU4FS's demonstrator currently only writes files initially, but does not support file updates. Therefore, we have excluded updates to full-stripes from our implementation. Updates require

the calculation of a parity update for RAID 5 and 6, since GPU4FS does not yet support file updates we have avoided the additional complexity in the implementation.

In our first implementation, chunks could be used for multiple files, now every file gets their own chunk. This was changed because when multiple files are written to one full-stripe in RAID-5, inter workgroup communication is required to correctly calculate the parity for that full-stripe. In this approach, the parity is calculated inplace on the disk's parity stripe with atomic XOR operations. Early benchmark results showed that the performance suffered strongly from the use of the atomic operations. We explain this approach in depth in the next section.

Alternatively, one workgroup has to write its file first and then the other workgroups writes its file afterwards and calculates a parity update as detailed in 2.1.1. However, this approach would mean a sequentialization of the file writes to one chunk and thus destroy the advantage of the GPU's parallel architecture.

Certainly, it is possible to align pages of different files in a chunk in a way that they always start in a new full-stripe, but for the demonstrator which only writes a few files, it is reasonable to create a separate chunk per file.

To integrate our RAID subsystem, we first merged our chunk allocation into GPU4FS's `allocate_blocks` method. This method is responsible for allocating the target pages for the file's content. It reserves sequential space from a linear allocator and provides a block pointer for each page. The code snippet can be found in Listing 1.

Listing 1 `allocate_blocks` allocates the requested pages from the block allocator and writes the resulting block pointers to a buffer. We integrated our chunk allocation and page allocation from existing chunks logic into the method.

```
uint64_t inode_offset;
int disk;
if (raid_level == -1) {
    // if file should be written physically allocate from specified disk
    // if no disk is specified, use disk with the most space
    inode_offset = (disk == -1) ? alloc_from_any_disk(amount_bytes, disk)
                                : alloc_from_disk(disk, amount_bytes);
} else {
    // try to allocate amount_bytes from exisiting chunk of raid_level
    if (!tryAllocFromExistingChunk(raid_level, amount_bytes, inode_offset))
    {
        // no chunk has sufficient space, allocate new chunk
        inode_offset = alloc_chunk(chunkSize, amount_bytes, raid_level,
                                   numStripes, redundancy, stripeLength);
    }
}

// add page offsets to inode_offset and format as block pointers
```

RAID level -1 indicates that the file should bypass the RAID subsystem, requiring the specification of the target disk. When specifying the target disk as -1, GPU4FS will automatically select the disk with the most available space.

The function `alloc_from_any_disk` returns the physical offset for the allocated blocks and returns the chosen disk. Meanwhile, `alloc_from_disk` is responsible for allocating `amount_bytes` from the specified disk and providing the offset.

If the file should be written to a RAID chunk, we proceed to verify the availability of a chunk with adequate space and the requested RAID level. Here, the function `tryAllocFromExistingChunk` traverses the chunk list formed by the chunk tree, evaluating whether the referenced logical chunk has sufficient space and the required RAID level. If the required space is found, it claims that space from the chunk's linear allocator in an atomic operation. Subsequently, the linear allocators in the replicas of the logical chunk descriptor must be updated via an atomic maximum operation. Upon successful allocation, the function returns true, and the target address is written to `inode_offset`. The target address is calculated as the chunk's base address plus the offset to the allocated block.

In cases where no chunk possesses adequate space, the function returns false, prompting the allocation of a new chunk using the parameters from the file write command. Following this, the block pointers for the requested pages are formatted and written to a buffer.

After the successful page allocation from a chunk, one workgroup resolves the target logical chunk descriptor from the first returned block pointer by walking the chunk tree. The workgroup initializes the shared logical chunk allocator cache as explain in section 5.4. The information is then accessible from the cache to all invocation for the facilitation of the actual file write.

Now, the `file_create` method in GPU4FS invokes a copy operation, copying both the inode and file content to the pages previously allocated by `allocate_blocks`. Into this copy process, we integrate our RAID logic to distribute the data words across the stripes and compute the parity.

5.5.1 First Approach: Copying Word by Word

The initial approach to writing data to a chunk allowed for a flexible stripe length and involved determining where to write on a word by word basis. When selecting a stripe length that is not a multiple of the workgroup size, it resulted in a performance bottleneck. This occurred due to the non-uniform scatter operations of the invocations, because some invocations were already writing data to a different disk. Another challenge was posed by multiple invocations being responsible for computing the same parity word. Consequently, this required the caching of parity in slow global memory and the use of atomic XOR operations for synchronization. The atomic XORs proved notably slower compared to non-atomic XOR operations on local memory.

Additionally, for each data word the complete target offset needs to be recalculated, without retaining any context regarding the location of the current full-stripe and stripe itself. Since the GPU can perform these offset calculations quickly, performance is not noticeably affected. However, the code of `write_to_address` was unnecessarily complicated, which made debugging difficult. A code snippet of this approach can be found in Listing 2.

Listing 2 Copying from a VRAM buffer to a logical chunk on a word by word basis. For every data word, the exact target location is determined independently.

```
void copy_to_address(int source_offset, uint64_t target_address,
                     int size, int work_group_size) {
    uint worker = gl_LocalInvocationID.x;

    uint outer_reps = rounded_up_division(size, work_group_size);
    for (int i = 0; i < outer_reps; ++i) {
        uint offset = i * work_group_size + worker;
        if (offset >= size) return;
        // load data word from VRAM
        uint64_t loaded = config.data[uint(source_offset + offset)];
        // write data word to logical chunk
        write_to_address(target_address, int(offset), loaded);
    }
}
```

5.5.2 Second Approach: Copying Stripe by Stripe

As stripe lengths that are not multiples of the workgroup size lead to poor performance, it is reasonable to constrain the stripe length to a multiple of the workgroup size. Consequently, we limit the stripe length to $8 \cdot k$ times the number of invocations per workgroup, measured in bytes. We use the factor eight, because eight bytes is the word length of our shader. This approach ensures sequential writes by a workgroup's invocations to enhance write performance. As an added benefit, it enables invocations to independently calculate parity without synchronization over shared memory, as explained in section 4.2.4.

Initially, we determine the number of data stripes (`stripes`) required to accommodate the entire bytes of the sequential VRAM buffer (`size`) containing both the inode and the file content.

Subsequently, we determine the amount of full-stripes. It is important to note that, in the context of RAID-5, one stripe per full-stripe is reserved for parity. Following this, we proceed to initialize variables, such as the number of data rows per stripe. The file and stripe offsets are set to the local ID of the invocation. This setup is designed to force sequential reading and writing when considering all invocations within a workgroup simultaneously.

Next, we address each full-stripe independently. After writing each full-stripe, we update the stripe offset to indicate the starting point of the next stripe, shifting it by the stripe length. Additionally, we adjust the offset within the VRAM buffer by the cumulative size of the data stripes written in the preceding full-stripe. For RAID-5, we also update the location of the parity chunk for the next stripe in a round-robin fashion as it cycles through the stripes. The code snippet responsible for the setup, updating the offsets and parity stripe after a full-stripe write can be found in Listing 3.

Next, we explain the process of writing a complete stripe. In the context of RAID-5, for each full-stripe, we initialize the local parity cache to zero. We then proceed to iterate through the data stripes within the full-stripe, copying them one by one. In RAID-5, we defer copying the full-stripe's parity stripe until we have computed the parity, as previously mentioned.

In our design, we divide each stripe into `rows` rows. We perform a row-wise transfer of the data words from the VRAM buffer to the physical stripe on the disk, adjusting the offset based on the full-stripes copied earlier. After copying each data stripe, the parity cache stores the parity information for each row in every column for RAID-5. Subsequently, the parity is written row by row to the parity stripe. The code snippet for writing a full-stripe can be found in Listing 4.

Next, we explain the process of writing the rows within a stripe. During this operation, an invocation retrieves a data word from the buffer at the specified offset. If the entire file has been copied but the full RAID-5 stripe is not yet complete, it is necessary to zero out the remaining portion of the RAID-5 full-stripe. However, for other RAID levels, we can simply return after copying the file's data.

In the case of RAID-5, we update the parity cache by performing an XOR operation with its current content and the read data word. For RAID-1, the data word must be written to the mirrors. For all other RAID levels, the data is written to a single physical chunk. Regardless of the RAID level, we translate the stripe number to a physical disk and the physical chunk offset from the chunk's logical chunk allocator. Subsequently, we write the data to the identified disk at the physical chunk offset plus the stripe offset. We completed processing one row, and now we need to repeat the same steps for the remaining rows within the stripe. To achieve this, after each iteration, we update both the file offset and the stripe offset by the number of data words per row (`local_size`). This adjustment ensures that we move to the next row in the sequential VRAM buffer and on the target stripe.

The code snippet for writing a data stripe and updating the parity cache can be found in Listing 5.

5.5.3 Directories

Following the successful copying of the inode and file content onto the disk array, the GPU4FS `file_create` method proceeds to insert the filename and inode number into the root directory. As mentioned earlier, we did not implement updates to stripes, resulting in the GPU4FS root directory residing outside logical chunks.

Given that every path within the file system originates from the root directory, ensuring replicas of its content is essential to safeguarding the system against potential disk failures. To maintain resilience against up to m disk failures, we mirror both the root directory and its inode onto $m + 1$ disks, all situated at the same physical offset.

The root directory and its associated inode are positioned at the same fixed offset on $m = 2$ disks. Any write operation to the memory region of the root directory and its inode is duplicated to the mirror. Analog to the behavior of the superblock, the root directory acts as an RAID-1 chunk that exists outside the logical address space.

5.6 CPU Verification

During development and benchmarking, we need to confirm that the GPU has performed its tasks accurately. To achieve this, we employ verification on the CPU. It's important to note that this CPU verification would not be utilized in a production setting.

After the GPU completes its tasks, the CPU takes over to verify the consistency of on-disk structures and invariants. We have to extend GPU4FS's pre-existing CPU-side verification mechanisms to verify our RAID invariants.

GPU4FS's existing verification process involves a linear read of the root directory's files to ensure that each file has been correctly added to the root directory exactly once. The inode is resolved from its corresponding inode number within the directory, and the inode's metadata is subjected to a verification process. Subsequently, the file's content is read into a buffer and verified for its intended content.

In this context, we introduce our verification code, since we must resolve the inode and file data from multiple disks based on the RAID level in use. When the inode pointer is physical, we can utilize the existing verification procedures, but we are required to read from the disk encoded in the block inode number.

In scenarios where the inode pointer is not physical, the CPU relies on the chunk tree to resolve the corresponding logical chunk descriptor. While traversing this tree, the CPU ensures its integrity. Following successful resolution of the logical chunk descriptor, the CPU proceeds to verify the RAID level's associated invariants. For RAID-1, the CPU checks if the mirrors contain identical content, while for RAID-5, the parity for each full-stripe is recomputed and compared to ensure consistency.

After this verification process, the data containing the inode and the file's content is copied stripe by stripe into a buffer. This buffer is then passed to GPU4FS's inode and file verification routines.

The CPU's verification process is important during development, as it provides detailed error reporting, specifying the nature of the error and its exact on-disk offset. This information facilitates further investigation using tools like a hex editor, enabling a thorough examination and pinpointing of any issues and potential root causes back to the shader code.

5.7 Rebuilding

As mentioned earlier, we did not implement RAID-6. Consequently, we only implemented the capability to reconstruct a single disk. In this section, we describe our implementation to recover from a single disk failure.

To simulate a disk failure in our development and benchmark setting, the CPU first zeros out the target disk. This zeroing is just used during development and would not be done in a production environment. Following the disk zeroing, the CPU proceeds to write the default superblock.

The linear logical chunk allocator is then copied from another disk to the replacement disk. Subsequently, all workgroups collaborate to copy both the root directory and its associated inode to the replacement disk.

As detailed in Section 4.2.5, workgroups independently traverse the chunk tree's chained list to claim chunks for the rebuilding process. A chunk tree replica from an intact disk is used for the traversal. Each workgroup selects the chunks whose list index modulo the total number of workgroups is congruent to their workgroup ID.

For each claimed chunk, the workgroup allocates space for a replacement physical chunk and a new logical chunk descriptor. The size of the physical chunk is taken from an intact replica of the logical chunk descriptor. The offset to the logical chunk descriptor is inserted into the chunk tree of the replacement disk. Subsequently, the invocations within the workgroup collaborate to copy the contents of the replica logical chunk descriptor to the previously allocated location. Following this, the offset of the replacement physical chunk is synchronized across all replicas of the logical chunk descriptor.

To facilitate rebuilding, we employ the aforementioned caching mechanism for logical chunk descriptors. Here, one invocation per workgroup copies the logical chunk descriptor to the workgroup's shared cache, eliminating the need for further disk accesses to the logical chunk descriptor.

The invocations then initiate the RAID-level-dependent recovery process. For RAID-1, the invocations collaborate to copy data from a mirror to the replacement stripe. In the case of RAID-5, data from all intact stripes must be read in and XORed, with the resulting data subsequently being written to the replacement stripe.

Afterwards, the CPU verification takes over, confirming the rebuild's success. Again, this verification is only employed during development and benchmarking.

Listing 3 This code initializes the values necessary for to copying the buffer's data in full-stripe to the disks. After a full-stripe is written, the offsets are updated accordingly and the location of the next parity stripe is updated.

```
void copy_to_address(int source_offset, uint64_t target_address,
    int size, int work_group_size) {
    if(isPhysicalAddress(target_address)) {
        // use GPU4FS's old copy method but use disk encoded in target address
        return;
    }
    // we assume that we begin at the beginning of the stripe
    // and copy a files complete data to the chunk

    uint worker = gl_LocalInvocationID.x;

    // get value from already initialized shared cache
    int raidLevel, stripeLength, numStripes, redundancy;


    int stripes = rounded_up_division(size, stripeLength);
    int numDataStripes = numStripes;
    if (raidLevel == 5) {
        numDataStripes--;
    }
    uint fullStripes = rounded_up_division(stripes, numDataStripes);

    int rows = stripeLength / local_size;

    // invocations read/write their own column
    int offset = source_offset + int(worker);
    int stripeOffset = int(worker);

    int parityStripe = 0;

    for (int fullstripe = 0; fullstripe < fullStripes; ++fullstripe) {
        copy_full_stripe(raidLevel, redundancy, numStripes, rows,
             parityStripe, size, offset, stripeOffset);
        // update stripe offset by a stripe
        stripeOffset += stripeLength;
        // update offset by a full data stripe
        offset += numDataStripes * stripeLength;
        // move parity stripe in a round-robin fashion
        parityStripe = (parityStripe + 1) % numStripes;
    }
}
```

Listing 4 Code for writing a full-stripe to the disk array. The parity cache is zeroed, then all data stripes are copied while the parity is computed simultaneously. Afterwards, the computed parity is written to the parity chunk.

```
void copy_full_stripe(int raidLevel, int redundancy, int numStripes,
    int rows, int parityStripe, int size, int offset, int stripeOffset) {

    uint64_t parityCache[maxRows];

    if (raidLevel == 5) {
        for (int row = 0; row < rows; ++row) {
            // zero cache for next full-stripe
            parityCache[row] = 0;
        }
    }
    for (int stripe = 0; stripe < numStripes; ++stripe) {
        if (raidLevel == 5 && stripe == parityStripe) {
            // skip parity chunk
            continue;
        }
        copy_rows(raidLevel, redundancy, rows, size,
            offset, stripe, stripeOffset, parityCache);
        // update offset by a stripe
        offset += rows * local_size;
    }

    if (raidLevel == 5) {
        copy_parity(rows, parityStripe, stripeOffset, parityCache);
    }
}
```

Listing 5 The stripe is written row by row, where each invocation copies one data word per row. For RAID-5 the parity cache is updated and for the last full-stripe the remaining bytes are zeroed out. For RAID-1 writes are duplicated to the mirror disks.

```
void copy_rows(int raidLevel, int redundancy, int rows, int size,
    int offset, int stripe, int stripeOffset,
    inout uint64_t parityCache[maxRows]) {

    for (int row = 0; row < rows; ++row) {
        uint64_t data = 0;
        if (offset < size) {
            // load data word from VRAM buffer
            data = config.data[offset];
            if (raidLevel == 5) {
                // compute parity
                parityCache[row] ^= data;
            }
        } else if (raidLevel != 5) {
            return;
        }

        if (raidLevel == 1) {
            copy_to_mirrors(redundancy, stripe, stripeOffset, data);
        } else {
            copy_to_stripe(stripe, stripeOffset, data);
        }

        // update offsets by a row
        offset += local_size;
        stripeOffset += local_size;
    }
}
```

6 Evaluation

The goal of this thesis is to integrate RAID functionalities into GPU4FS. Our objective is to create a RAID subsystem capable of competing with CPU-based software RAID systems in terms of write bandwidth while reducing stress on the CPU. We begin this chapter by presenting our test system and setup. Next, we explain the different benchmarks we ran. We begin the actual evaluation by exploring some parameters that are independent of the RAID level. Subsequently, we present results for both write operations and data rebuilding for each RAID level to assess the effectiveness of our implementation. We conduct a comparative analysis, examining our system's bandwidth and CPU utilization in relation to CPU-based software RAID systems for each RAID level. Finally, we conclude this chapter by analyzing the latency of our system and end with a discussion of our findings.

6.1 Test System

We evaluate our implementation on the following test system:

- Dual Intel(R) Xeon(R) Silver 4215 CPUs, running at 2.5 GHz.
- 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.
- AMD Radeon RX 6800 GPU, from AMD's Navi 2 GPU generation, together with 16 GB of dedicated VRAM. The GPU is equipped with sixteen PCIe Gen4 lanes, but communication falls back to PCIe Gen3 as the CPU only supports Gen3.

We attempted to utilize both the two local and two remote Optane DIMMs. However, write performance was inconsistent from the GPU, primarily due to unknown Optane behavior when writing through PCIe and the processor interconnect. This variability made it challenging to maintain consistency across different benchmark runs and complicated the interpretation of the results. Thus, we only used the Optane and DRAM DIMMs local to the CPU to which the GPU is attached. To still be able to evaluate our disk array on four disks, we divided each Optane DIMM into two DAX devices (later referred to as partitions). The CPU maps the total of four DAX devices to the GPU.

For every test, we pin the benchmark to the CPU to which the GPU is attached using the `taskset` command to avoid going through the processor interconnect.

In our demonstrator, we arrange the block pointers to the physical chunk of a logical chunk in the logical chunk descriptor 4.1.4 in a way that every two stripes written in succession end up on two different Optane DIMMs, not just on different partitions. In addition, two consecutively allocated logical chunks each start on two different DIMMs. This way, we can distribute the accesses to the disk array as evenly as possible between the two local Optane DIMMs, even if the different workgroups do not operate in lockstep. That's why, we always choose two workgroups for the following benchmarks so that ideally, only one workgroup writes to a single Optane at a time. For some benchmarks, we alternatively present results for one workgroup per partition, which totals to four workgroups.

6.2 Benchmarks

In this section, we introduce the benchmarks used throughout the evaluation. We explain which parameters of a benchmark can be configured and what results are returned.

6.2.1 Memcopy

GPU4FS's memcopy command uses all dispatched invocations to copy a sequential VRAM buffer to a shader buffer [23]. We did not modify the original version of the memcopy command.

In our benchmarks, the target buffer is backed by either Optane or DRAM. For a single run, the number of invocations, the amount of bytes to copy and the target buffer can be specified. The benchmark returns the runtime of a single run. For each input, we record the runtime for 26 independent runs. To calculate the write bandwidth, we divide the amount of copied bytes by the average runtime over these 26 runs.

6.2.2 File Write

To evaluate the write bandwidth to our disk array, we write x files using the file write command 5.3.2. The command writes x files, each of the same specified file size, to logical chunks of the specified RAID level. Additionally, the stripe length needs to be configured.

Currently, GPU4FS's inode supports up to 112 block pointers. That's why we choose file sizes that are multiples ranging from 1 to 112 of 4KB, 2MB, or 1GB pages for the following benchmarks. We write x files with these file sizes. Unless otherwise specified we always use 1, 2, 3, 4, 6, 8 and 10 for the value of x.

The benchmark returns the runtime to write the x files. For each input, we record the runtime for 26 independent runs. To calculate the write bandwidth, we divide the total written bytes (x times the file size) by the average runtime over these 26 runs. We discuss both bandwidth and runtime in our evaluation.

6.2.3 Rebuild

The rebuild benchmark is used to evaluate the required time to rebuild the disk array. The benchmark first writes x files of a specified size and RAID level. Afterwards, one disk is

zeroed and the RAID subsystems rebuilds the zeroed disk. Only the time to rebuild the disk array is recorded. The inputs, amount of files and file size are the same as for the file write benchmark 6.2.2.

For each input, we record the rebuild time for 26 independent runs and calculate the average rebuild time.

6.3 Parameter Exploration

Before we can begin to evaluate our RAID subsystem, we first need to determine some parameters required for the following benchmark and to properly contextualize the subsequent results.

6.3.1 Maximum Bandwidth

First, want to determine the GPU's maximum bandwidth when transferring data from VRAM to Optane and DRAM. This information is crucial to accurately interpret the results of the following RAID benchmarks.

We run the memcopy benchmark 6.2.1 to copy 512MB of data. We evaluated the bandwidth for different amounts of shader invocations working together to copy the data. A plot of the bandwidth per amount of shader invocations both on one Optane DIMM and DRAM can be found in Figure 6.1.

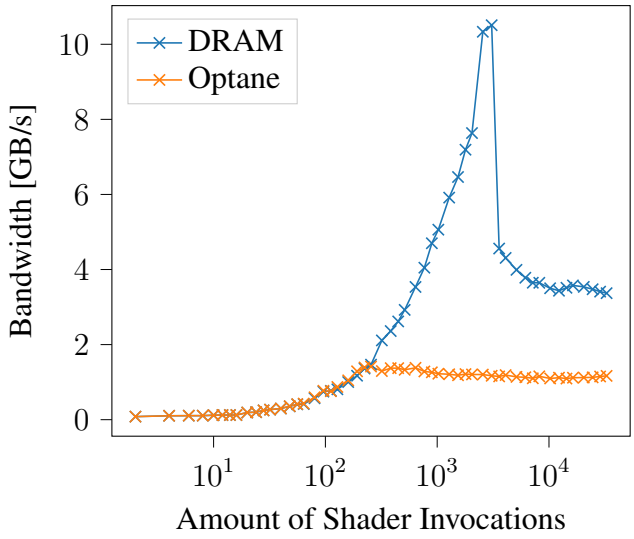


Figure 6.1: Write bandwidth when x invocations copy 512 MB from a VRAM buffer to one Optane DIMM or to DRAM.

For Optane there is no clear bandwidth peak, but a plateau begins at 256 shader invocations with a bandwidth of 1.4 GB/s, slowly decreasing with an increased invocation count. This is interesting as in GPU4FS's evaluation, Maucher achieved a clear peak of 1.9 GB/s at 320 invocation on his configuration [23], close to Optane's maximum write bandwidth of

2 GB/s. Another difference is that in Maucher's benchmark, Optane's bandwidth rapidly falls to 0.65 GB/s after the peak [23], but for us the bandwidth slowly decreases with an average of 1.2 GB/s for more than 256 invocations. We believe that the deviating behavior is due to the combination of a different GPU and other PCIe specs, leading to bundled write requests that overwhelm the Optane DIMM in our configuration.

For DRAM, a clear peak is reached at 3072 invocations at around 10.5 GB/s, which is about double of the maximum bandwidth Maucher achieved on eight PCIe Gen3 lanes [23]. After the peak, the bandwidth quickly falls down to an average of 3.7 GB/s for more than 3072 invocations.

For the next benchmarks, we assume that, on our test system, the maximum achievable write bandwidth to one Optane DIMM from the GPU is 1.4 GB/s. The results indicate that when writing with more than 256 invocations at a time, we can still sustain a bandwidth close to the GPU's maximum bandwidth, at approximately 1.2 GB/s per Optane DIMM.

6.3.2 Stripe Length

Before we can benchmark the write bandwidth for different RAID levels, we first need to examine the impact of the stripe length factor on the bandwidth.

As a reminder, we choose the stripe length to be k times the workgroup size times the word length in bytes. We named the involved factor k the stripe length factor. Choosing the stripe length to be a multiple of workgroup size times the word length ensures that all invocations write to a single disk sequentially before moving on to the next disk. This mechanism helps us balance the accesses between the two Optane DIMMs because individual workgroups only write to one DIMM at a time.

As GLSL only supports static allocations and a SIMD processor's local memory is quite small, we limited the parity cache to ten rows in the demonstrator. For this reason, the maximum stripe factor is $k = 10$

As the stripe length depends on the invocations per workgroup and the stripe length factor, we perform a two-dimensional grid search over both parameters. We dispatch two workgroups, one for each Optane DIMM. For the number of invocations per workgroup, we choose a lower bound of 128 invocations per workgroup, based on the previous memcopy result to start the grid-search before the peak of 1.4 GB/s per Optane DIMM. We choose 128 invocation per workgroup to still achieve acceptable bandwidth in the worst case that both workgroups write to the same Optane DIMM at the same time. For the upper bound, we choose 1024 as this is the maximum invocation count per workgroup supported by GLSL on our hardware. For the stripe length factor, k we search between 1 and 10.

We investigate the impact of the stripe length factor only for RAID-0, because the two-dimensional grid search takes significant time to complete. However, we assume the results to be applicable to the other RAID levels, since the stripe length factor is only responsible for how many bytes are written sequentially to a disk before the workgroup moves to the next disk. Hence, we conclude that the stripe length factor only affects the write bandwidth per Optane DIMM, independent of the RAID level.

In order to have enough data to copy, we write four 512MB files to RAID-0 chunks.

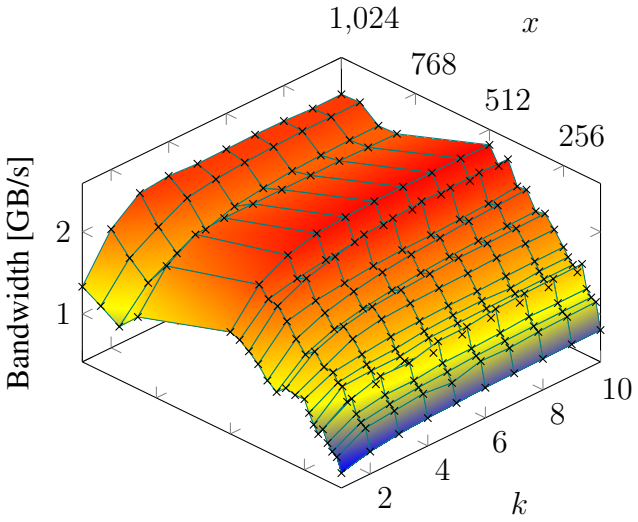


Figure 6.2: Bandwidth when writing four 512 MB RAID-0 files with two workgroups with x invocations per workgroup and a stripe factor k to two Optane DIMMs partitioned into four DAX devices.

The three-dimensional plot with the grid search results can be found in Figure 6.2. We plot the stripe length factor and the invocations per workgroup against the achieved write bandwidth. It seems that the stripe factor k has a logarithmic impact on the bandwidth. We can observe this effect independent of the number of invocations per workgroup. This means for our limited search grid, the higher the stripe length factor k, the higher the bandwidth. However, for most innovation counts, the impact of the stripe length factor becomes quite small for $k > 5$. Nevertheless, regardless of the invocation count chosen $k = 10$ yields the highest bandwidth. Hence, we choose a stripe length factor of $k = 10$ for all following benchmarks.

6.4 RAID-0

We begin the evaluation of our RAID-0 implementation by optimizing the number of workgroups and invocations per workgroup that best utilize the write bandwidth of the two Optane DIMMs in a RAID-0 configuration. We perform this optimization again because we cannot be certain that the memcopy results can be transferred directly because of potential overhead caused by the file system and the RAID subsystem. Subsequently, we evaluate our write bandwidth using the established parameters for both Optane and DRAM. To conclude this section, we compare our findings to other software RAID systems in terms of bandwidth and CPU utilization.

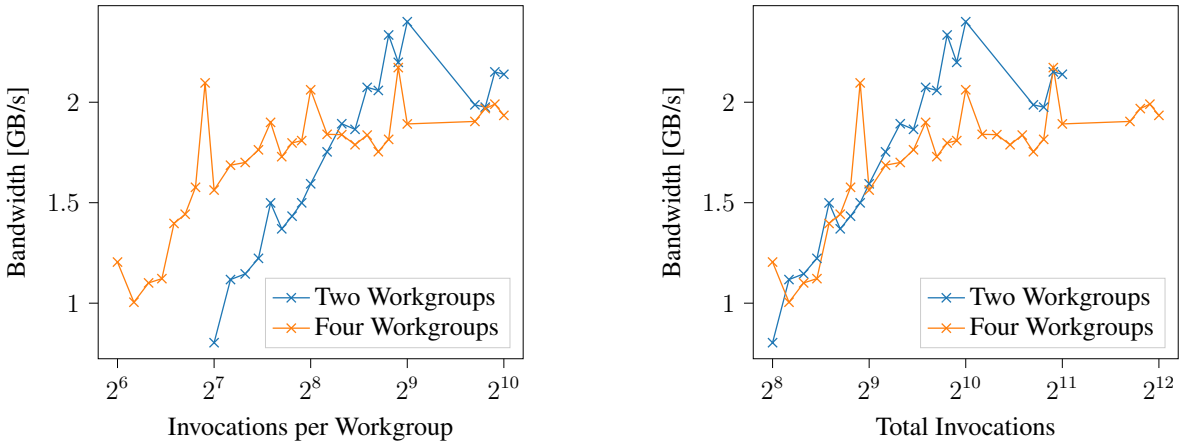
6.4.1 Parameter Exploration

First, we need to find the ideal combination out of amount of workgroups and invocations per workgroup that best utilize the bandwidth of both Optane DIMMs in a RAID-0 configuration.

A new optimization is necessary because the memcopy results cannot be transferred directly due to potential overhead of the file system and RAID subsystem. Additionally, if the accesses from different workgroups are not evenly distributed across the disks, it may also be necessary to reduce the number of invocations. For the above reasons, we again explore the appropriate number of invocations.

As argued in 6.1, ideally only one workgroup writes to a single DIMM at a time, therefore we dispatch two workgroups. Alternatively, we dispatch one workgroup per partition, totaling to two workgroups per Optane DIMM, to compare the bandwidth between both approaches.

We start the search at a total count of 256 invocations, as this resulted in the highest bandwidth of 1.4 GB/s per DIMM in the memcopy benchmark 6.1. We choose this number to still achieve acceptable bandwidth in the worst case that all invocations write to the same Optane DIMM at the same time. Again, we choose 1024 as the highest amount of invocation per workgroup, as this is the limit imposed by GLSL. To find the highest bandwidth, we run the file benchmark 6.2.2 to write four 512 MB files to RAID-0 chunks for different amounts of invocations per workgroup for two and four workgroups.



(a) Bandwidth for x invocations dispatched per workgroup.

(b) Bandwidth for x total invocations dispatched.

Figure 6.3: Bandwidth when writing four 512 MB RAID-0 files to two Optane DIMMs partitioned as four DAX devices. Two and four workgroups were dispatched.

The achieved bandwidth for both two and four workgroups and different number of invocations per workgroup can be found in Figure 6.3.

We can see that between a total of 256 and 512 invocations, two and four workgroups achieve close to the same bandwidth (Figure 6.3b). After 512 total invocation, two workgroups achieve higher bandwidths. When we compare the total amount of invocations, we can see that the amount of workgroups also has an impact on the performance. We observe that by using two workgroups, we can achieve higher bandwidths compared to using four workgroups, even though the same total number of invocations are dispatched. This could be due to the chunk allocation enforcing sequenciality through an exclusive write-lock or the writes are more balanced between the two DIMMs for two workgroups.

When using two workgroups, the bandwidth peaks at 2.4 GB/s for 512 invocation per workgroup, resulting in a total of 1024 invocations. The peak of 2.4 GB/s on two Optane DIMMs in a RAID-0 configuration aligns perfectly with the 1.2 GB/s per DIMM that we measured in the memcopy benchmark 6.1 when using more than 256 invocations per Optane DIMM. For more than 512 invocations per workgroup for the RAID-0 file write, the bandwidth decreases. The peak bandwidth of 1.4 GB/s per DIMM from the memcopy benchmark 6.1 is never reached, since the accesses of the workgroups are probably not equally distributed between the two DIMMs.

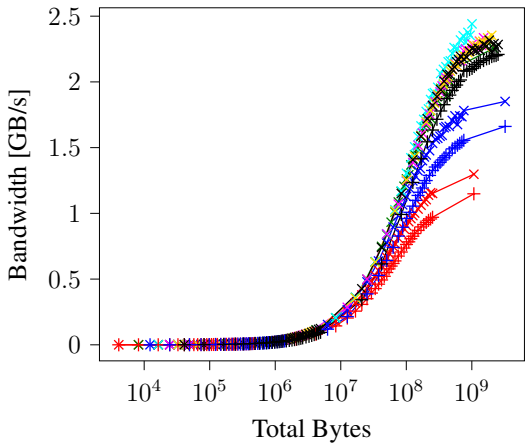
When using four workgroups to write the four files, we get the highest peak of 2.18 GB/s for 480 invocations per workgroups. We also see peaks for 256 (2.06 GB/s) and 128 (2.1 GB/s) invocations per workgroup. We can see that the bandwidth decreases after 512 invocation per workgroup, suggesting that over a total of 2048 invocations the bandwidth starts to suffer.

Another observation is that the shader always timed out for invocations counts between 512 and 832 per workgroup for both two and four workgroups. We did not investigate this further.

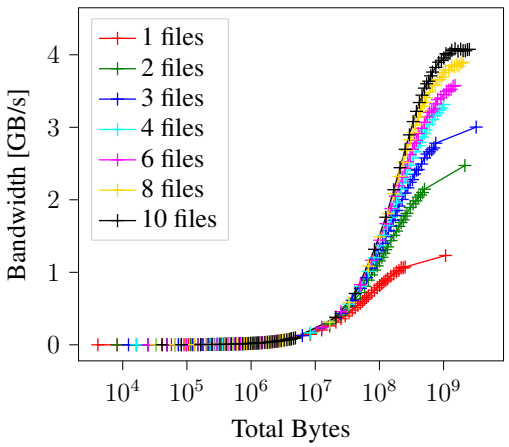
To conclude, we achieve the highest bandwidth on Optane on RAID-0 when dispatching 512 innovations per workgroup and two workgroups.

6.4.2 File Write

Now that we determined that 512 innovations per workgroup and two workgroups yield the highest bandwidth on two Optane DIMMs in a RAID-0 configuration, we can evaluate our RAID-0 implementation for different number of files and file sizes. We use the file write benchmark 6.2.2 to record the resulting bandwidth. We plot the total file bytes written against the bandwidth. We compare the achieved bandwidth between Optane and DRAM for the same input parameters (Figure 6.4a). Afterwards we compare the results to DRAM with optimized input parameters (Figure 6.4b).



(a) Two workgroups with 512 invocations each, writing to Optane (x) and DRAM (+).



(b) Ten workgroups with 1024 invocations each, writing to DRAM.

Figure 6.4: Bandwidth when writing x file bytes contained in k files, to four partitions in a RAID-0 configuration.

We first discuss the results on Optane (Figure 6.4a). For file counts including multiples of the amount of workgroups (2), we can achieve a bandwidth of around 2.4 GB/s. For one file we get a maximum bandwidth of 1.3 GB/s as only one workgroup is working while the other sits idle. We have the same effect for three files with 1.85 GB/s, as two files can be written in parallel and for the remaining file only one workgroup works while the other sits idle. The reason for the lower performance is that earlier we optimized performance for writing four files, which implies writing multiple files in parallel.

When the number of files is a multiple of the number of workgroups, we observe that the bandwidth primarily relies on the total number of bytes written, and the number of files written doesn't have a significant impact. The peak bandwidth of 2.45 GB/s is achieved when copying with four files. This nearly aligns with the average write bandwidth of 1.2 GB/s measured in the memcopy benchmark 6.2.1 for invocation counts higher than 256 on a single Optane DIMM during memcopy operation.

When writing with the same input parameters and workgroup configuration to DRAM (Figure 6.4a), we nearly get the same trend and behavior as on Optane, but the bandwidth on Optane is on average around 90 MB/s higher. We don't have an explanation for the deviating behavior.

To prove that neither the GPU nor our shader code bottlenecks the bandwidth, we dispatch 10 workgroups, each of the maximum workgroup size of 1024 invocations, to write to DRAM (Figure 6.4b). We achieve a maximum bandwidth of 4.1 GB/s which would be enough to saturate both Optane DIMMs. When writing less than four files, it becomes evident that the 1024 invocations per workgroup copying a single file are not enough to exploit the available bandwidth. Thus, a system which dynamically allows multiple workgroups to work on writing a single file would make sense, as already proposed by Maucher [23].

For RAID-0, we can conclude that we achieved almost twice the bandwidth as compared to the bandwidth of the memcopy benchmark 6.1 by using two Optane DIMMs in a RAID-0 configuration. The comparison to DRAM proves that the shader code is efficient enough to fully utilize both Optane DIMMs, if the 2GB/s per Optane DIMM were available from the GPU.

6.4.3 Comparison to CPU RAID Implementations

Next, we compare GPU4FS's achieved bandwidth and CPU usage to BTRFS, the EXT4 file system on MD RAID and ZFS also running the same two Optane DIMMs in RAID-0 configuration. For each of the software RAID systems we choose the default configuration and did not optimize any parameters ourselves.

Bandwidth To evaluate the bandwidth, we use the file write benchmark 6.2.2 to write one to ten files to the disk array. To write the files to the software RAID systems, we deploy one thread per file. After writing the file, the thread calls `fsync` to ensure that the file is written to the disk array. We write the same file sizes as for GPU4FS, as explained in section 6.2.2. We measure the time between dispatching the first thread and until all threads have joined the main thread. Again, we perform 26 independent runs and use the average runtime to calculate

the bandwidth. For GPU4FS, we use the previous results from the file write benchmark on Optane 6.4a. We present a plot for the achieved bandwidth for one file and ten files (Figure 6.5). The highest achieved bandwidth for each file system can be found in Table 6.1.

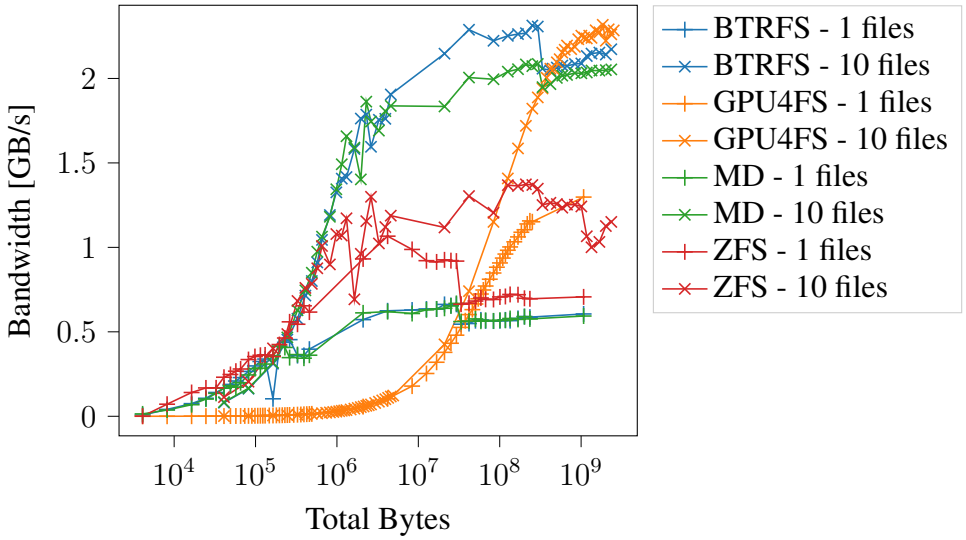


Figure 6.5: Bandwidth for writing 1 and 10 files to a RAID-0 disk array for different software RAID systems.

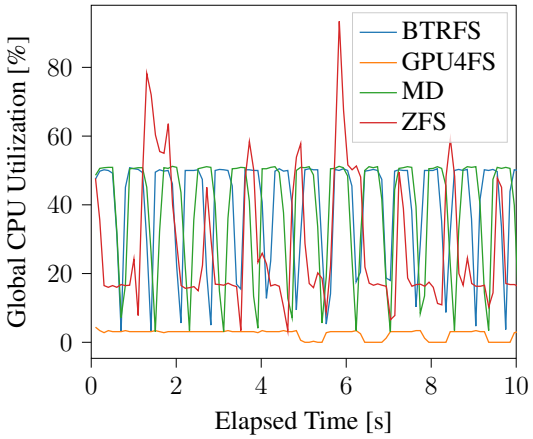
First, we can see that for all other RAID systems the bandwidth for small files is significantly higher than for GPU4FS. This is due to the high start up latency that we will discuss later.

ZFS performs best for one file but performs the worst for ten files. BTRFS and MD RAID seem to mostly follow the same trend. GPU4FS achieves the highest bandwidth of around 2.45 GB/s closely followed by BTRFS with 2.21 GB/s. This is interesting as already one thread per Optane DIMM is enough to reach the max bandwidth of 2 GB/s. So theoretically, a CPU based RAID-0 could achieve a bandwidth of up to 4 GB/s. A reason for the low performance could but that these software RAID systems were designed for block devices, and running them on byte addressable Optane in `FSDAX` mode results in suboptimal performance. Nevertheless, by this comparison and by reaching just over 4 GB/s on DRAM, we can conclude that GPU4FS is competitive in terms of write bandwidth and even outperforms its competitors on Optane in a RAID-0 configuration.

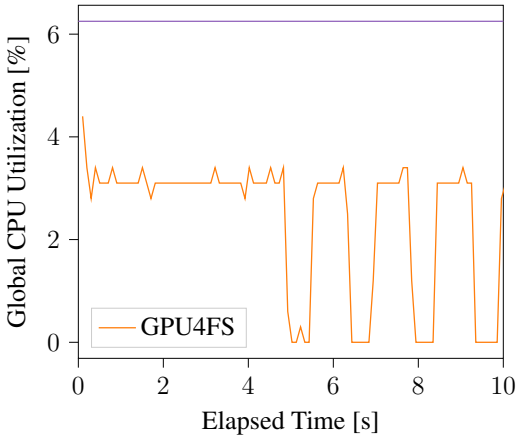
	BTRFS	GPU4FS	MD RAID + EXT4	ZFS
Max Bandwidth (GB/s)	2.31339	**2.44166**	2.08605	1.37827
Files	10	4	10	4
File Size (MB)	24	240	28	4

Table 6.1: Maximum achieved bandwidth for different software RAID systems, given the best set of parameters, respectively. Higher is better.

CPU Usage To compare the CPU usage we wrote 20 60 MB files to BTRFS, the EXT4 file system on MD RAID, ZFS and GPU4FS in a RAID-0 configuration. To make the observation period long enough, we repeat the benchmark 20 times back to back. We record the global CPU utilization to measure the utilization of both the writing process and the kernel. The global CPU utilization encompasses the utilization of every core of both CPUs (both NUMA nodes), where 100 % means all cores of both CPUs are fully utilized. Due to the different write bandwidths, the processes run for different lengths of time. We display only an extract in Figure 6.6 because of the highly periodic utilization of the 20 repetitions. We recorded the global CPU utilization every 0.1 seconds, because this is the smallest resolution that our benchmark library supports [4].



(a) Comparing global CPU utilization of different software RAID systems.



(b) Global CPU utilization when GPU4FS is running. The purple line indicates a fully utilized single core.

Figure 6.6: Global CPU utilization for writing 20 files to RAID-0. 50 % means all cores of one CPU are fully utilized, and 100 % means all cores of both CPUs are fully utilized.

For the first 5 seconds, GPU4FS utilizes half of a single core to set up Vulkan and to compile the shader. GPU4FS has close to zero CPU utilization while the shader is running. After writing the files, GPU4FS utilizes half a single core to queue the files for the next iteration. On average, GPU4FS has a global CPU utilization of 2 %.

Both BTRFS and MD RAID fully utilize all cores of one CPU (NUMA node) for writing the files. Between benchmark runs, the CPU utilization drops for both RAID systems. The average global CPU utilization of BTRFS is 39 % and 38 % for MD RAID. For ZFS, we assume that we get the high peaks for writing the files to the file system cache and the lower plateaus reflect the utilization for the `fsync` calls. We can also see that the utilization drops between benchmark runs. For ZFS, the average global CPU utilization is 27 %.

Compared to the CPU based RAID systems, GPU4FS decreases the global average CPU utilization 13.5 times compared to the competitor with the lowest CPU utilization (ZFS). However, ZFS RAID cannot compete with GPU4FS in terms of bandwidth, but BTRFS can. Therefore, it is reasonable to compare the global CPU utilization of GPU4FS to BTRFS instead

of ZFS. Compared to BTRFS, GPU4FS decreases the average CPU utilization by a factor of x18.

6.5 RAID-1

We begin the evaluation of our RAID-1 implementation by optimizing the number of workgroups and invocations per workgroup that best utilize the write bandwidth of the two Optane DIMMs in a RAID-1 configuration. We perform this optimization because in RAID-1 every write is duplicated, therefore we cannot directly transfer the configuration from the memcopy or the RAID-0 benchmarks. Subsequently, we evaluate our write bandwidth using the established parameters for both Optane and DRAM. To conclude this section, we compare our findings to other software RAID systems in terms of bandwidth and CPU utilization.

6.5.1 Parameter Exploration

First, we need to find the ideal combination out of amount of workgroups and invocations per workgroup that best utilize the bandwidth of both Optane DIMMs in a RAID-1 configuration. In RAID-1 for every VRAM buffer read, two disk writes follow. This changed access pattern requires us to (re-)optimize the workgroup configuration again.

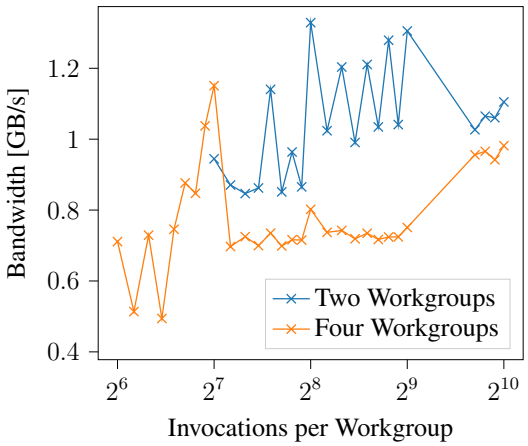
As argued in 6.1, ideally only one workgroup writes to a single DIMM at a time, therefore we dispatch two workgroups. Alternatively, we dispatch one workgroup per partition, totaling to two workgroups per Optane DIMM, to compare the bandwidth between both approaches.

We start the search at a total count of 256 invocations, as this resulted in the highest bandwidth of 1.4 GB/s per DIMM in the memcopy benchmark 6.1. We choose this number to still achieve acceptable bandwidth in the worst case that all invocations write to the same Optane DIMM at the same time. Again, we choose 1024 as the highest amount of invocation per workgroup, as this is the limit imposed by GLSL. To find the highest bandwidth, we run the file benchmark 6.2.2 to write four 512 MB files to RAID-1 chunks for different amounts of invocations per workgroup for two and four workgroups.

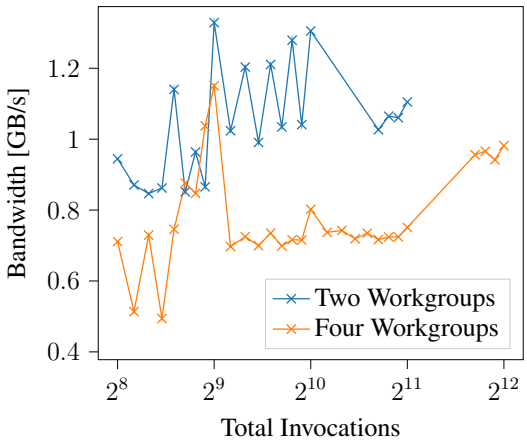
The achieved bandwidth for both two and four workgroups and different number of invocations per workgroup can be found in Figure 6.7. Note that, as previously mentioned in the context of RAID-1, every byte is duplicated. We are interested in the bandwidth for writing the file and not interested in the actual raw write bandwidth to the disks. Therefore, only the file bytes are included in the bandwidth calculation.

Like for RAID-0, two workgroups perform better than four workgroups on Optane. For two workgroups, the highest bandwidth of 1.33 GB/s is reached for 256 invocations per workgroup, which totals to 512 invocations. This is half of total invocations used in RAID-0. This makes sense as for every read data word from VRAM, two Optane writes follow to write the data word to both mirrors.

Another interesting observation is that the peak for four workgroups (1.17 GB/s) is achieved when using 128 invocations per workgroup, which totals to 512 invocations. Even tough the total number of invocations is the same, the bandwidth achieved by two workgroups is around 160 MB/s higher. This could be due to scheduling, more balanced disk accesses or when more



(a) Bandwidth for x invocations dispatched per workgroup.



(b) Bandwidth for x total invocations dispatched.

Figure 6.7: Bandwidth when writing four 512 MB RAID-1 files to two Optane DIMMs partitioned as four DAX devices. Two and four workgroups were dispatched.

workgroups are used, workgroups have to wait longer on average to acquire the exclusive lock to modify the chunk tree. But without profiling, we have no way to investigate this further.

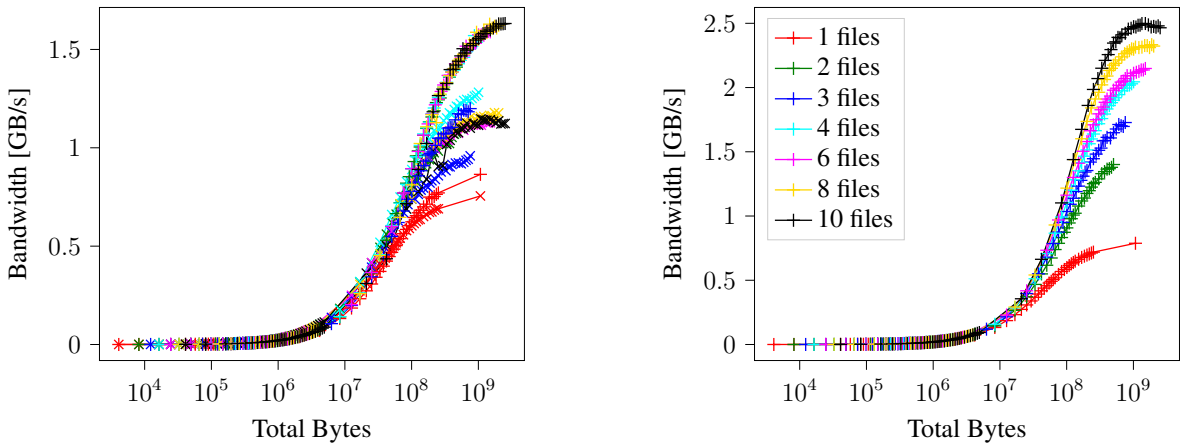
Analog to RAID-0 the shader always times out for invocations counts between 512 and 832 per workgroup for both two and four workgroups. We did not investigate this further.

To conclude, we achieve the highest bandwidth on Optane on RAID-1 when dispatching 256 innovations per workgroup and two workgroups.

6.5.2 File Write

Now that we determined that 256 innovations per workgroup and two workgroups yield the highest bandwidth on two Optane DIMMs in a RAID-1 configuration, we can evaluate our RAID-1 implementation for different number of files and file sizes. We use the file write benchmark 6.2.2 to record the resulting bandwidth. We plot the total file bytes written against the bandwidth. We compare the achieved bandwidth between Optane and DRAM for the same input parameters (Figure 6.8a). Afterwards we compare the results to DRAM with optimized input parameters (Figure 6.8b).

We first discuss the results on Optane (Figure 6.8a). We reach a peak bandwidth of around 1.3 GB/s, which is close to the 1.4 GB/s peak achieved per Optane DIMM in memcopy. Because every write is duplicated to two both Optane DIMMs, it makes sense that we are close to the maximum bandwidth of one Optane DIMM. The fact that we don't quite reach the maximum bandwidth per Optane may be due to the file system and RAID overhead. The results also suggest, that the writes of both workgroups are relatively evenly balanced across the two Optane DIMMs. Considering the maximum bandwidth of RAID-0 is 2.4 GB/s, one would expect that on RAID-1 the bandwidth should roughly half to 1.2 GB/s. Considering that we achieve 1.3 GB/s, we even achieve 100 MB/s more.



(a) Two workgroups with 256 invocations each, writing to Optane (x) and DRAM (+).

(b) Ten workgroups with 1024 invocations each, writing to DRAM.

Figure 6.8: Bandwidth when writing x file bytes contained in k files, to four partitions in a RAID-1 configuration.

We observe lower bandwidths for the file amount one (760 MB/s) and three (960 MB/s) analog to RAID-0 because one workgroup has to idle. Interesting is the gap between the bandwidth achieved when writing four files is a bit higher (around 100 MB/s) than for other multiples of the amount of workgroups. This is not surprising as we optimized the bandwidth for four files.

When writing with the same input parameters and workgroup configuration to DRAM (Figure 6.8a), we nearly get the same trend and behavior as on Optane, but the bandwidth on DRAM is on average around 90 MB/s higher. This is exactly the opposite to RAID-0, where the bandwidth on Optane was around 90 MB/s higher than on DRAM. We don't have an explanation for the deviating behavior.

To prove that neither the GPU nor our shader code bottlenecks the bandwidth, we dispatch 10 workgroups, each of the maximum workgroup size of 1024 invocations, to write to DRAM (Figure 6.8b). We observe that the bandwidth is also halved on DRAM than compared to RAID-0. Otherwise, the behavior is analog to RAID-0. The peak of 2.5 GB/s is reached when writing ten files with ten workgroups. Again we can see that with every additional active workgroup we can achieve more performance, suggesting that we are not exploiting DRAM's available bandwidth. This is due to the limit of 1024 invocations per workgroup.

For RAID-1, we can conclude that we are able to almost sustain the write bandwidth to a single Optane DIMM from the memcopy benchmark 6.1, while introducing redundancy by copying the data to another DIMM.

6.5.3 Comparison to CPU RAID Implementations

Next, we compare GPU4FS's achieved bandwidth and CPU usage to BTRFS, the EXT4 file system on MD RAID and ZFS also running the same two Optane DIMMs in RAID-1

configuration. For each of the software RAID systems we choose the default configuration and did not optimize any parameters ourselves.

Bandwidth Analog to RAID-0 6.4.3, to evaluate and compare the bandwidth, we use the file write benchmark 6.2.2 to write one to ten files to each RAID system. For GPU4FS, we use the previous results from the file write benchmark on Optane 6.8a. We present a plot for the achieved bandwidth for one file and ten files (Figure 6.9). The highest achieved bandwidth for each file system can be found in Table 6.2.

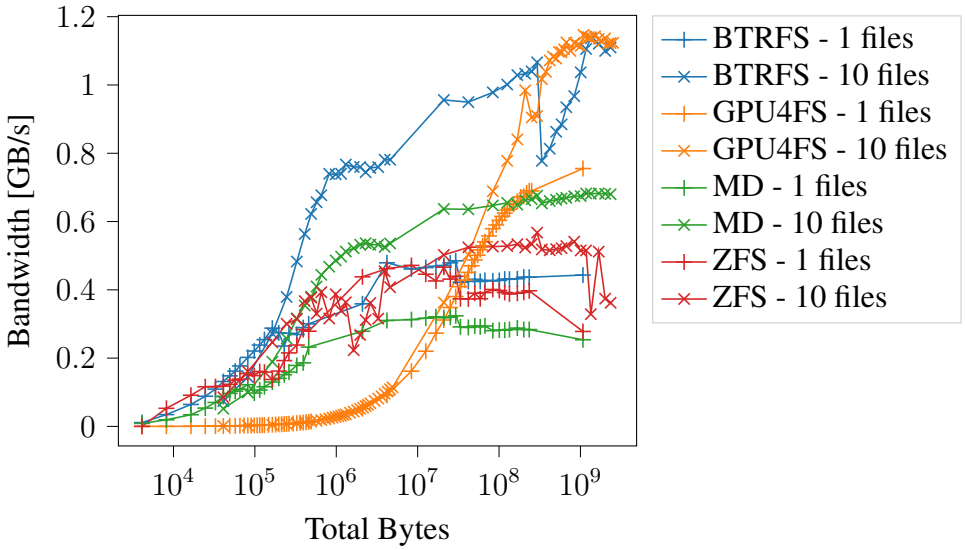


Figure 6.9: Bandwidth for writing 1 and 10 files to a RAID-1 disk array for different software RAID systems.

First, we can see that for all other RAID systems the bandwidth for small files is significantly higher than for GPU4FS. Similar to RAID-0, this is due to the high start up latency that we will discuss later. For all RAID systems the bandwidth is halved than compared to RAID-0, which is expected due to the write duplication.

For ten files, GPU4FS reaches the highest bandwidth (1.15 GB/s), closely followed by BTRFS (1.14 GB/s). Both achieve around twice the bandwidth of ZFS and MD RAID. For one file, GPU4FS outperforms the next best contestant (BTRFS) by 400 MB/s.

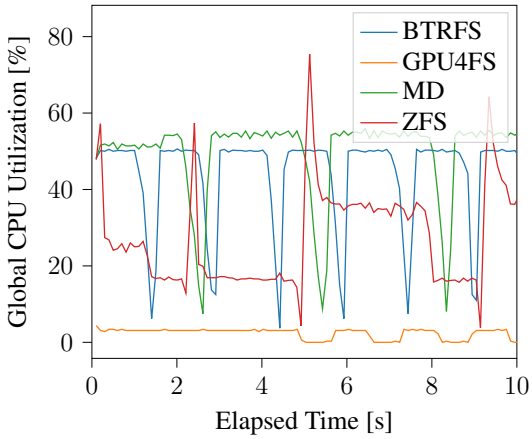
Theoretically, two threads should have been enough to sustain the bandwidth (2 GB/s) of two Optane DIMMs running in a RAID-1 configuration. However, no CPU-based RAID achieves close to 2 GB/s. Again, a reason for the low performance could but that these software RAID systems were designed for block devices, and running them on byte addressable Optane in `FSDAX` mode results in suboptimal performance. Nevertheless, by this comparison and by reaching around 2.5 GB/s on DRAM, we can conclude that GPU4FS is competitive in terms of write bandwidth and even outperforms its competitors on Optane in a RAID-1 configuration.

CPU Usage Analog to RAID-0 6.4.3, to compare the CPU usage we wrote 20 60 MB files on BTRFS, the EXT4 file system on MD RAID, ZFS and GPU4FS in a RAID-1 configuration.

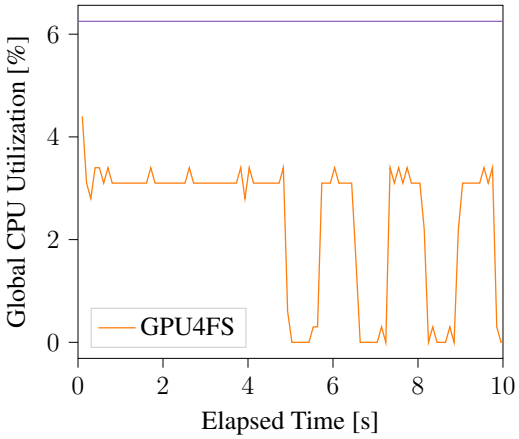
	BTRFS	GPU4FS	MD RAID + EXT4	ZFS
Max Bandwidth (GB/s)	1.13683	**1.28219**	0.68266	0.56745
Files	10	4	4	10
File Size (MB)	128	240	160	28

Table 6.2: Maximum achieved bandwidth for different software RAID systems, given the best set of parameters, respectively. Higher is better.

To make the observation period long enough, we again repeat the benchmark 20 times back to back and record the global CPU utilization. The global CPU utilization encompasses the utilization of every core of both CPUs (both NUMA nodes), where 100 % means all cores of both CPUs are fully utilized. Again, we display only an extract in Figure 6.10 because of the highly periodic utilization of the 20 repetitions.



(a) Comparing global CPU utilization of different software RAID systems.



(b) Global CPU utilization when GPU4FS is running. The purple line indicates a fully utilized single core.

Figure 6.10: Global CPU utilization for writing 20 files to RAID-1. 50 % means all cores of one CPU are fully utilized, and 100 % means all cores of both CPUs are fully utilized.

We can see that the CPU usage of GPU4FS follows the same course as for RAID-0. Again, for the first 5 seconds, GPU4FS utilizes half a single core to set up and has close to zero CPU utilization while the shader is running. On average, GPU4FS has a global CPU utilization of 2 %. The utilization is very similar to RAID-0 since the CPU does not perform any RAID management tasks.

Both BTRFS and MD RAID fully utilize one CPU for writing the files. Between benchmark runs, the CPU utilization drops for both RAID systems. The average global CPU utilization of BTRFS is 43 % and for MD RAID 49 %. For ZFS, we assume that we get the high peaks for writing the files to the file system cache and the lower plateaus are the utilization for the `fsync` calls. We can also see that the utilization drops between benchmark runs. For ZFS, the average CPU utilization is 29 %.

Compared to the CPU based RAID systems, GPU4FS decreases the global average CPU utilization 14.5 times compared to the competitor with the lowest CPU utilization (ZFS). However, ZFS RAID cannot compete with GPU4FS in terms of bandwidth, but BTRFS can. Therefore, it is reasonable to compare the global CPU utilization of GPU4FS to BTRFS instead of ZFS. Compared to BTRFS, GPU4FS decreases the average CPU utilization by a factor of x21.5.

6.5.4 Rebuild

Next, we evaluate how long a rebuild of a RAID-1 disk array takes. We run the rebuild benchmark 6.2.3, that performs a file write, zeros one disk and records the time taken to rebuild the disk array. We plot the total amount of repaired bytes against the elapsed repair time. We use the same workgroup configuration as for writing.

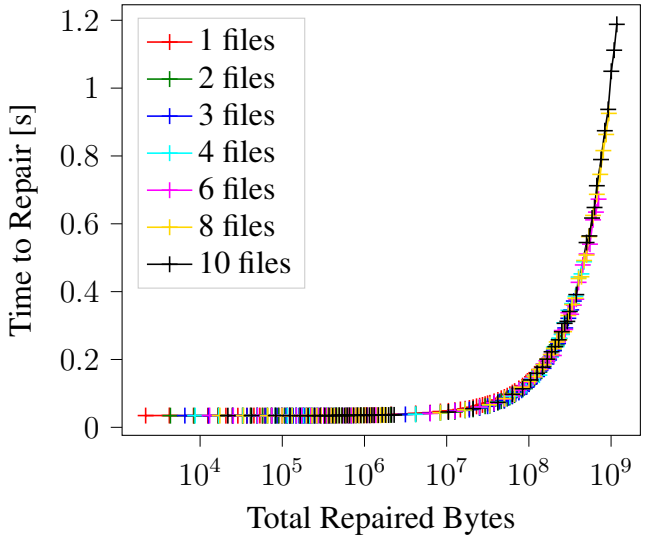


Figure 6.11: Time to repair for x RAID-1 bytes in k files. Two workgroups with 256 invocation each were dispatched.

On Optane (Figure 6.11) the time to rebuild linear depends on the number of bytes. The curves correlate for the different numbers of files. Thus, we can conclude that time to rebuild mostly depends on the amount of bytes to rebuild and not on the amount of files. This suggests that the overhead to walk the chunk tree, to copy the logical chunk descriptor and to insert a new logical chunk descriptor into the local tree is neglectable. The main work is copying the data from one mirror to another mirror.

We can use the time to repair and the amount of rebuilt bytes to calculate the rebuild bandwidth. The maximum resulting bandwidth is around 1 GB/s suggesting that we lose around 400 MB/s over writing to a single Optane DIMM in the memcopy benchmark 6.1. When rebuilding two Optane DIMMs in a RAID-1 configuration, we read data from one DIMM and copy it to the other DIMM. Either the reduced bandwidth means that the read time of the source DIMM significantly impacts the rebuild time or that too many invocations write to the

target Optane DIMM at a time and overwhelming it. Unfortunately, we did not have enough time to investigate this.

We also ran the rebuild benchmark on DRAM. The execution time of the shader was 0.05s on average. When repairing 1 GB of data, this would result in a write bandwidth of 20 GB/s surpassing the bandwidth of the PCIe interconnect. Naturally, this phenomenon is impossible to occur. Examination using CPU verification showed that the zeroed disk was not successfully repaired. We suspect that due to a synchronization problem on DRAM, the workgroups never iterate through the chunk list. The workgroups conclude that they reached the end of the list and complete their execution. Thus, no chunk is repaired. Unfortunately, we could not find the root cause for the problem. However, we can guarantee that our repair shader code works because on Optane none of these problems exist and the verification passes successfully.

We can conclude that on Optane we need close to a second to recover one gigabyte of data of the lost partition for RAID-1.

6.6 RAID-5

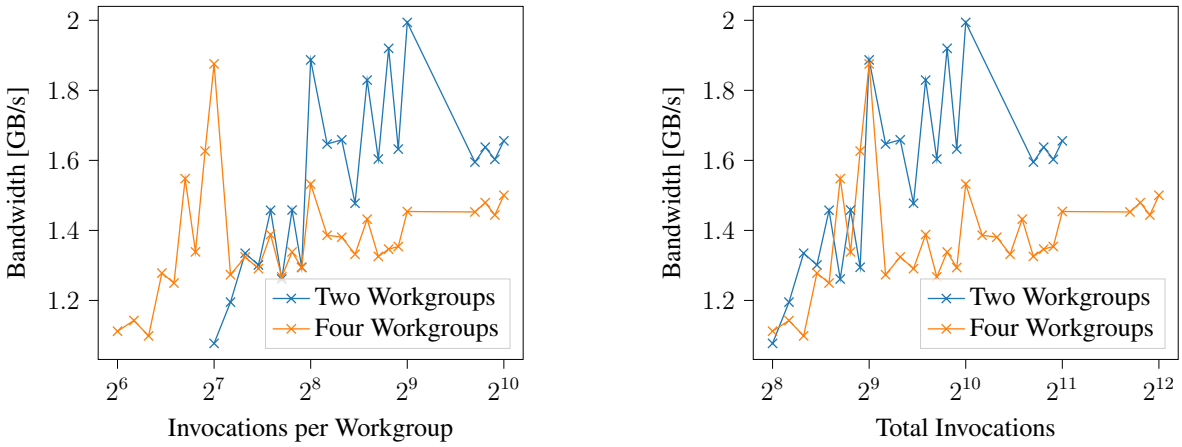
We begin the evaluation of our RAID-5 implementation by optimizing the number of workgroups and invocations per workgroup that best utilize the write bandwidth of the two Optane DIMMs in a RAID-5 configuration. We perform this optimization because due to the additional overhead of the parity calculation, it may be necessary to dispatch more invocations compared to RAID-0 to utilize the two Optane DIMMs. Subsequently, we evaluate our write bandwidth using the established parameters for both Optane and DRAM. To conclude this section, we compare our findings to other software RAID systems in terms of bandwidth and CPU utilization.

6.6.1 Parameter Exploration

First, we need to find the ideal combination out of amount of workgroups and invocations per workgroup that best utilize the bandwidth of both Optane DIMMs in a RAID-5 configuration. A new optimization is necessary because the RAID-0 configuration cannot be transferred directly due to the potential overhead caused by the parity calculation. If the parity calculation takes considerable time, it might be that a speed-up can be achieved if more invocations are dispatched. This requires us to (re-)optimize the workgroup configuration again.

As argued in 6.1, ideally only one workgroup writes to a single DIMM at a time, therefore we dispatch two workgroups. Alternatively, we dispatch one workgroup per partition, totaling two workgroups per Optane DIMM, to compare the bandwidth between both approaches.

We start the search at a total count of 256 invocations, as this resulted in the highest bandwidth of 1.4 GB/s per DIMM in the memcopy benchmark 6.1. We choose this number to still achieve acceptable bandwidth in the worst case that all invocations write to the same Optane DIMM at the same time. Again, we choose 1024 as the highest amount of invocation per workgroup, as this is the limit imposed by GLSL. To find the highest bandwidth, we run the file benchmark 6.2.2 to write four 512 MB files to RAID-5 chunks for different amounts of invocations per workgroup for two and four workgroups.



(a) Bandwidth for x invocations dispatched per workgroup.

(b) Bandwidth for x total invocations dispatched.

Figure 6.12: Bandwidth when writing four 512 MB RAID-5 files to two Optane DIMMs partitioned as four DAX devices. Two and four workgroups were dispatched.

The achieved bandwidth for both two and four workgroups and different number of invocations per workgroup can be found in Figure 6.12. When using for disks in a RAID-5 configuration, we need to write a parity stripe every three data stripes. Again, we only consider the bandwidth for writing the file and not the actual raw write bandwidth to the disks. Therefore, the parity bytes are not considered in the bandwidth calculation.

Like for RAID-0 and RAID-1, two workgroups perform better than four workgroups on Optane.

For two workgroups, analog to RAID-0 the bandwidth increases for less than 512 invocations per workgroup up to the peak of 2 GB/s at 512 invocations per workgroup. For more invocations per workgroup, the bandwidth decreases.

For four workgroups, we reach the peak of 1.9 GB/s at 128 invocations per workgroup, which aligns with the local peak when using 256 invocations per workgroup and two workgroups.

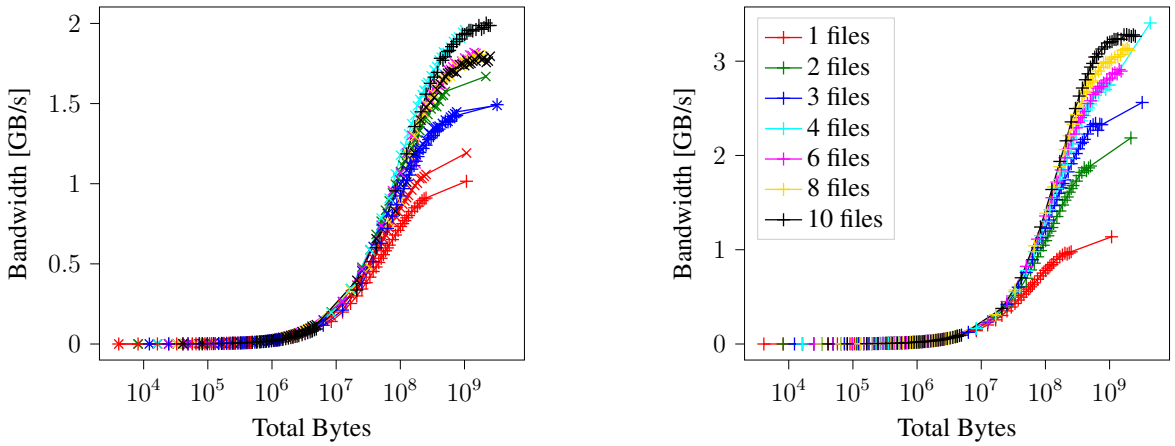
Analog to RAID-0 and RAID-1 the shader always times out for invocations counts between 512 and 832 per workgroup for both two and four workgroups. We did not investigate this further.

To conclude, we achieve the highest bandwidth on Optane on RAID-5 when dispatching 512 innovations per workgroup and two workgroups. This is the same configuration as for RAID-0, which indicates that the parity calculation does not add significant overhead. We investigate this further in the next section.

6.6.2 File Write

Now that we determined that 512 innovations per workgroup and two workgroups yield the highest bandwidth on Optane, we can evaluate our RAID-5 implementation for different number of files and file sizes. We use the file write benchmark 6.2.2 to record the resulting bandwidth. We plot the total file bytes written against the bandwidth. We compare the achieved bandwidth

between Optane and DRAM for the same input parameters (Figure 6.13a). Afterwards we compare the results to DRAM with optimized input parameters (Figure 6.13b).



(a) Two workgroups with 512 invocations each, writing to Optane (x) and DRAM (+).

(b) Ten workgroups with 1024 invocations each, writing to DRAM.

Figure 6.13: Bandwidth when writing x file bytes contained in k files, to four partitions in a RAID-5 configuration.

We first discuss the results on Optane (Figure 6.13a). The maximum achieved bandwidth is around 2 GB/s four files. That the peak is reached for four files is not surprising, as we previously optimized the invocation count for four files. Considering that we achieved a maximum bandwidth of 2.4 GB/s for RAID-0, we lose around 400 MB/s for calculating and writing the parity stripe. The maximum bandwidth for six, eight and ten files is around 1.8 GB/s which is $\frac{3}{4}$ of RAID-0 2.4 GB/s. 1.8 GB/s is the expected bandwidth, considering that for every three data stripe, we need to write one parity stripe for RAID-5. Because the maximum achieved bandwidth is 2 GB/s, we know that we lose less than $\frac{1}{4}$ of RAID-0's bandwidth. We can conclude that the actual parity calculation does not add much overhead, and the bandwidth is decreased because we need to write the parity stripe.

Again, we observe lower bandwidth for the file amount 1 and 3 than for multiples of the number of workgroups analog to RAID-0 and RAID-1 because one workgroup has to idle.

Next, we write with the same input parameters and workgroup configuration to DRAM (Figure 6.13a). The highest bandwidth is achieved for 10 files with 2 GB/s, which closely matches the trend of four files on Optane. For three files, the curve on DRAM and Optane overlap. For one file, the bandwidth on Optane is around 200 MB/s higher than on DRAM. For more than one file, the curves for the same amount of files are closer together on Optane and DRAM.

To prove that neither the GPU nor our shader code bottlenecks the bandwidth, we dispatch 10 workgroups, each of the maximum workgroup size of 1024 invocations, to write to DRAM (Figure 6.13b). For DRAM (Figure 6.13b) the bandwidth also decreased to around $\frac{3}{4}$ of RAID-0's bandwidth. This supports the assumption that the GPU is mainly busy writing and the parallel parity calculation has little influence on the runtime. The peak bandwidth is achieved for four files at 3.5 GB/s.

For RAID-5, we can conclude that we are able to sustain more than $\frac{3}{4}$ of RAID-0 write bandwidth. This means we lose less than $\frac{1}{4}$ of the bandwidth because we need to write the parity stripe. Additionally, we can conclude that the parity calculation does not add any significant overhead.

6.6.3 Comparison to CPU RAID Implementations

Next, we compare GPU4FS's achieved bandwidth and CPU usage to BTRFS, the EXT4 file system on MD RAID and ZFS also running the same two Optane DIMMs in RAID-5 configuration. For each of the software RAID systems we choose the default configuration and did not optimize any parameters ourselves.

Bandwidth Analog to RAID-0 and RAID-1 6.4.3, to evaluate and compare the bandwidth, we use the file write benchmark 6.2.2 to write one to ten files to each RAID system. For GPU4FS, we use the previous results from the file write benchmark on Optane 6.13a. We present a plot for the achieved bandwidth for one file and ten files (Figure 6.14). The highest achieved bandwidth for each file system can be found in Table 6.3.

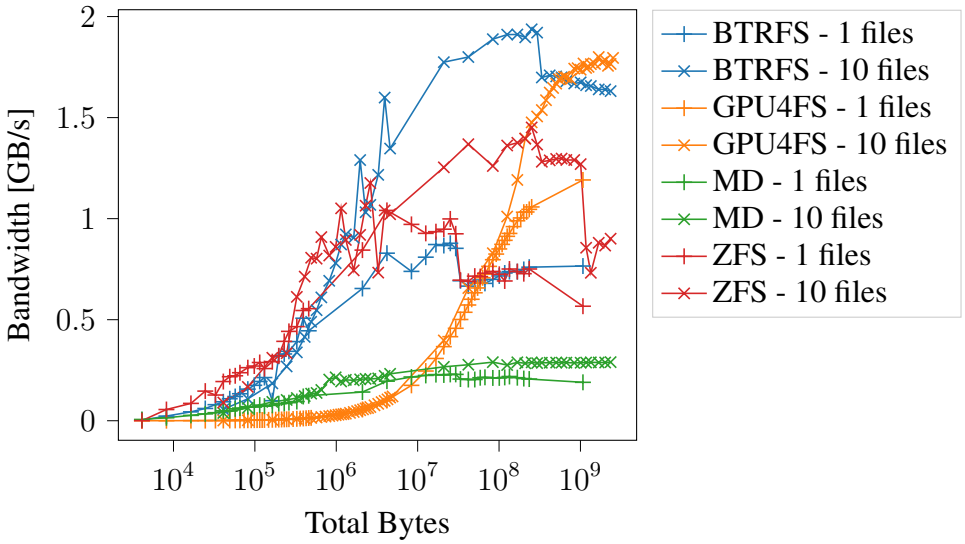


Figure 6.14: Bandwidth for writing 1 and 10 files to a RAID-5 disk array for different software RAID systems.

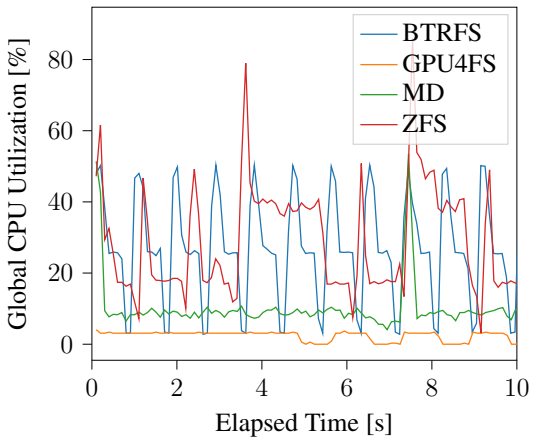
MD RAID only achieves around 300 MB/s regardless of the amount of files. MD RAID achieved around 2 GB/s on RAID-0 and 700 MB/s on RAID-1, thus the bandwidth of RAID-5 should be between the two unless the parity calculation is the bottleneck. BTRFS achieves the highest bandwidth with 1.94 GB/s for ten file, which proves that the CPU has enough compute power to calculate the parity. For ten files, GPU4FS maximum bandwidth is lower than BTRFS's bandwidth, but overall GPU4FS achieves the highest bandwidth of 1.96 GB/s. ZFS achieves nearly the same bandwidth as on RAID-0, which suggests that the writes are more balanced across the two Optane DIMMs for RAID-5.

To reiterate, these software RAID systems were designed for block devices, and they are not optimized for byte addressable Optane in `FSDAX`. Nevertheless, by this comparison and by reaching around 3.5 GB/s on DRAM, we can conclude that GPU4FS is competitive in terms of write bandwidth and even outperforms its competitors on Optane in a RAID-5 configuration.

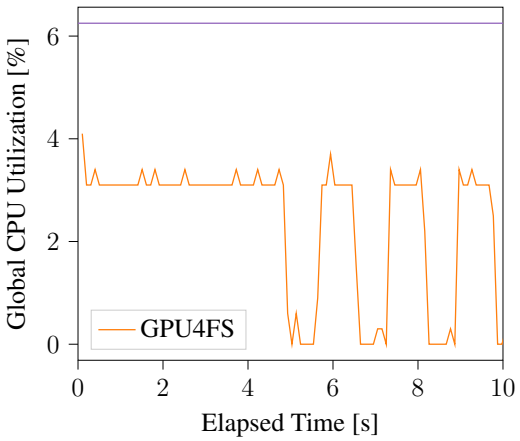
	BTRFS	GPU4FS	MD RAID + EXT4	ZFS
Max Bandwidth (GB/s)	1.93705	**1.96069**	0.29007	1.45145
Files	10	4	10	10
File Size (MB)	24	240	224	24

Table 6.3: Maximum achieved bandwidth for different software RAID systems, given the best set of parameters, respectively. Higher is better.

CPU Usage Analog to RAID-0 and RAID-1 6.4.3, to compare the CPU usage we wrote 20 60 MB files on BTRFS, the EXT4 file system on MD RAID, ZFS and GPU4FS in a RAID-1 configuration. To make the observation period long enough, we again repeat the benchmark 20 times back to back and record the global CPU utilization. The global CPU utilization encompasses the utilization of every core of both CPUs (both NUMA nodes), where 100 % means all cores of both CPUs are fully utilized. Again, we display only an extract in Figure 6.15 because of the highly periodic utilization of the 20 repetitions.



(a) Comparing global CPU utilization of different software RAID systems.



(b) Global CPU utilization when GPU4FS is running. The purple line indicates a fully utilized single core.

Figure 6.15: Global CPU utilization for writing 20 files to RAID-5. 50 % means all cores of one CPU are fully utilized, and 100 % means all cores of both CPUs are fully utilized.

We can see that the CPU usage of GPU4FS follows the same course as for RAID-0 and RAID-1. Again, for the first 5 seconds, GPU4FS utilizes half a single core to set up and has close to zero CPU utilization while the shader is running. On average, GPU4FS has a global

CPU utilization of 2 %. The utilization is very similar to RAID-0 and RAID-1 since the CPU does not perform any RAID management tasks.

For all the CPU based RAID system, we can clearly see the plateaus in the utilization where the `fsync` syscall are handled. For MD RAID, one iteration takes around 7 seconds.

The average of 8 % for MD RAID is reached during the `fsync` plateaus, which makes sense as MD RAID spends most of its time in the kernel, presumably to calculate the parity. Contrary to RAID-0 and RAID-1 we can also see the kernel time for BTRFS at around 25 % utilization. The average global CPU utilization of BTRFS is 26 %.

For ZFS, we assume that we get the high peaks for writing the files to the file system cache and assume the lower plateaus are the utilization for the `fsync` calls. We can also see that the utilization drops between benchmark runs. For ZFS, the average CPU utilization is 32 %.

Compared to the CPU based RAID systems, GPU4FS decreases the global average CPU utilization 4 times compared to the competitor with the lowest CPU utilization (MD RAID). However, MD RAID cannot compete with GPU4FS in terms of bandwidth, but BTRFS can. Therefore, it is reasonable to compare the global CPU utilization of GPU4FS to BTRFS instead of MD RAID. Compared to BTRFS, GPU4FS decreases the average CPU utilization by a factor of x13.

6.6.4 Rebuild

Next, we evaluate how long a rebuild of a RAID-5 disk array takes. We run the rebuild benchmark 6.2.3, that performs a file write, zeros one disk and records the time taken to rebuild the disk array. We plot the total amount of repaired bytes against the elapsed repair time. We use the same workgroup configuration as for writing.

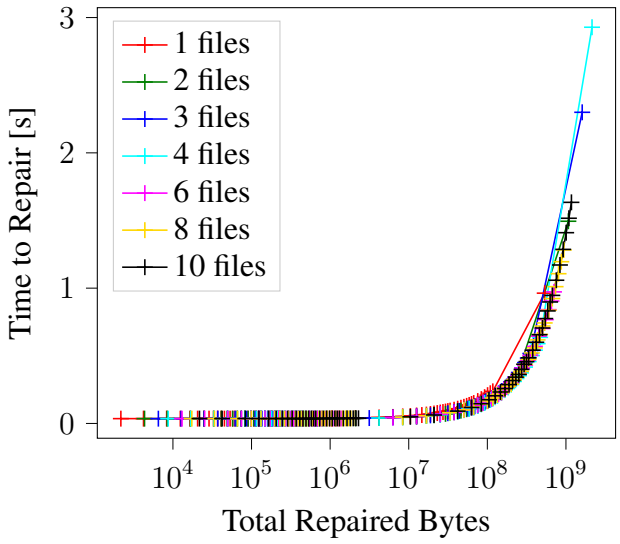


Figure 6.16: Time to repair x RAID-5 bytes in k files. Two workgroups with 256 invocation each were dispatched.

The synchronization problem that occurred on DRAM for the RAID-1 repair persist for RAID-5. Therefore, we only present results on Optane (Figure 6.16).

Analog to the RAID-0 rebuild, the time to rebuild linear depends on the number of bytes. The curves correlate for the different numbers of files. Thus, we can conclude that time to rebuild mostly depends on the amount of bytes to rebuild. The fact that we observed this correlation independently for RAID-1 and RAID-5 strengthens our thesis that the rebuilding of meta information such as chunk tree and logical chunk descriptors hardly affects the repair time.

Although we achieved higher write bandwidth on RAID-5 than on RAID-1, the rebuild time increased. Compared to RAID-1 rebuild, we need to read in 1.5 times more words, but only need to write 0.5 of the words written during a RAID-1 rebuild. We can rule out the XOR operation to be responsible for that slow down, as the file write benchmark suggests that it does not have a huge impact. The slowdown could be due to the fact that the RAID-1 rebuild logic offers higher locality and sequentiality, therefore benefitting from caching effects. But without profiling, we have no means to find out where the bottleneck is.

We can use the time to repair and the amount of rebuilt bytes to calculate the rebuild bandwidth. The resulting maximum rebuild bandwidth is 500 MB/s. We can conclude that on Optane we need close to two seconds to recover one gigabyte of data of the lost partition on RAID-5.

6.7 Latency

Besides bandwidth, latency is the other relevant metric for a file and RAID system. The shader setup, the queuing of files and finally the writing requires writing through different buses which takes time. Determining the latency is important to know the limiting factors of the bandwidth.

To analyze the latency, we plot the total written file bytes against the runtime of the file write for each RAID level, both on Optane and DRAM for the same input parameters. The plot 6.17 shows the runtime when writing ten files.

For both Optane and DRAM, we see that RAID-1 is the slowest and as expected, doubles the runtime when compared to RAID-0. The runtime for RAID-5 lies between them, but is closer to RAID-0. On DRAM, we observe a higher base latency.

First, we analyze the latency on Optane. For all RAID levels, the base latency is around 35 ms, which is significantly higher than the 1.5 ms measured by Maucher [23].

Next, we want to determine the cross-over point for each RAID level. The cross-over point specifies the minimum number of bytes that must be written for the bandwidth to no longer be dominated by the base latency. In the case of RAID-1, the cross-over point is reached at approximately 8 MB. Meanwhile, for RAID-5, this threshold is doubled, to 16 MB. Lastly, for RAID-0, the cross-over point is at around 28 MB.

Interestingly enough, for DRAM the base latency increases to 48 ms. This is interesting as Optane has a higher latency than DRAM [20], so the effect should be the other way around. Considering that the rebuild did not work on DRAM, there seems to be an underlying issue when accessing DRAM from the GPU on our test configuration. The cross-over points on DRAM shifts to 18 MB for RAID-2, 50 MB for RAID-5 and 56 MB for RAID-5.

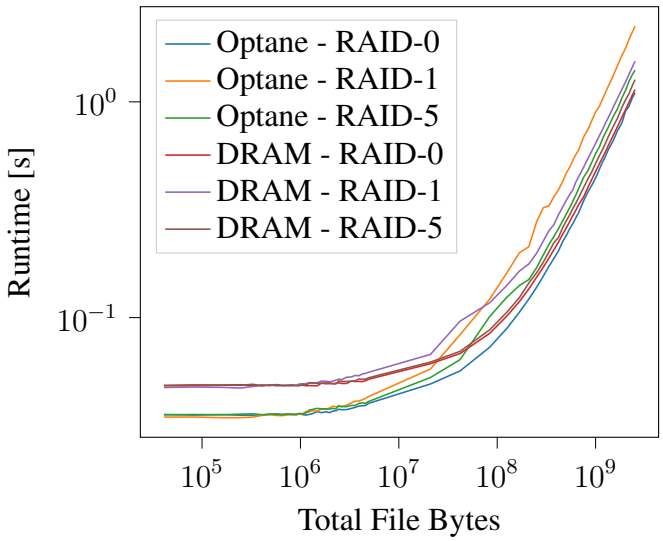


Figure 6.17: Runtime when writting x bytes split into 10 files to logical chunks of different RAID levels.

Either our system adds a lot of latency for the chunk allocation, acquiring the exclusive lock and insertion into the chunk tree, or the additional latency is introduced by the changed hardware. The comparison between Optane and DRAM shows that the hardware configuration has an impact on the latency. Our focus was on bandwidth maximization, and we consider latency minimization as future work.

6.8 Discussion

In this section, we discuss our evaluation and conclude whether we achieved our goal of designing a GPU based RAID system that can compete in terms of write bandwidth with CPU based RAID systems while simultaneously decreasing CPU utilization.

6.8.1 Performance Limitations

Due to the limited write performance (1.4 GB/s instead of 2 GB/s) from the GPU to Optane, it is difficult to properly generalize the benchmark results. We believe that the deviating behavior to Maucher's benchmarks [23] is due to the combination of a different GPU and a doubled PCIe lane count, leading to bundled write requests that overwhelm the Optane DIMM in our configuration. Therefore, it is necessary to benchmark the performance again on a different hardware configuration that can reach the maximum bandwidth on Optane from the GPU.

Even the CPU-based software RAID could not achieve the maximum performance on the two Optane DIMMs. Theoretically, the CPU RAID systems should be able to achieve 4 GB/s in a RAID-0 array on two DIMMs, because we were able to achieve 2 GB/s per Optane DIMM in a CPU-based memcopy benchmark. The CPU based memcopy results prove that neither the CPU nor the DIMMs are the cause of the limited performance. Instead, the limited performance

on the CPU could be due to the software RAID systems writing with too many kernel threads and thus overload the Optane DIMMs. Therefore, one could try to tune the software RAID systems manually by configuration adjustments to lower the thread count instead of using the default configurations optimized for block devices.

Despite our efforts, we cannot be sure that all processes involved in the write process of the CPU RAID systems are actually pinned to the CPU to which the target Optane DIMMs is attached. That pinning does not work reliably can be seen in the example of ZFS. ZFS reaches global CPU utilization rates well above 50 % and thus definitely uses both CPUs (both NUMA nodes). Write accesses from the remote CPU to the target DIMM can also be a reason for the limited performance from the CPU. Therefore, it can also be worthwhile to benchmark the CPU RAID systems on a machine with only one CPU. Generally, we never used the second CPU for our benchmarks, and we did not use the remote Optane DIMMs due to inconsistent performance. Since the setup with two CPUs only causes complexity and hard to track hardware behavior, our recommendation for future evaluations is to switch to a system with only one CPU.

However, the comparison between performance of GPU4FS and CPU based RAID system on Optane will always be unfair, since the CPU RAID systems were designed and optimized for block devices and GPU4FS for Optane. Additionally, the `fsdax` mode required for the CPU RAID systems causes additional overhead. In order to be able to fairly compare the performance and to get any adoption, GPU4FS must support other storage media such as SSDs, especially considering that Optane has been discontinued [27]. In addition, GPU4FS should implement the POSIX file system API [2] to be able to conduct standardized file system benchmarks.

6.8.2 Unexpected Behavior

Due to synchronization problems on DRAM, the rebuild did not work reliably. However, the shader code responsible for the rebuild worked without any problems on Optane. Unfortunately, at present, we lack a clear explanation for this unexpected behavior. Nonetheless, an investigation of the root cause is essential to understand and address this issue, ensuring that similar synchronization problems can be reliably avoided in future implementations.

While optimizing the invocation count, for each RAID level the shader is timed out between invocations counts between 512 and 832 per workgroup for both two and four workgroups. Our first guess is that the Optane DIMM is overloaded with write requests, but this cannot be true because the shader does not time out for invocation counts above 832. We have no explanation for this behavior either. We suspect that there is a problem in the graphics card drivers or that a hardware bug occurs for this workgroup configuration. In general, it would be recommended to switch from GLSL to a shader language that is designed for general purpose computing on the GPU and not originally designed for graphics processing. Changing the shader language could be the solution to many of the synchronization problems.

Comparing the file write bandwidth between Optane and DRAM on the same input parameters workgroup configuration shows a diverging behavior. In Maucher's evaluation, the bandwidth curves for both storage media overlap for writing one file, and DRAM is clearly faster for more files [23]. In our case, however, the bandwidth curves almost never overlap

and inexplicably the file write sometimes finishes faster on Optane and sometimes on DRAM. Due to this inconsistency and the limited bandwidth (1.4 GB/s) on an Optane DIMM, we can conclude that the different GPU produces different performance behavior, for example due to varying caching effects.

6.8.3 Latency

We have focused only on bandwidth maximization and have been able to achieve satisfactory bandwidths considering the mentioned limitations. However, the performance of the RAID system is not yet optimal, especially in terms of latency. Low latency ensures that data stored in the RAID array can be accessed and retrieved quickly. This is crucial for applications and systems that require fast access to data. A low latency is especially important for small files, because the base latency cannot be amortized over the transfer of many bytes. Setting up the shader, queuing the files and finally writing through the different buses takes significant time. Compared to the CPU based RAID system, the latency of GPU4FS is too high, with a minimum latency at around 35ms. In order for GPU4FS to provide acceptable performance for small files, the latency needs to be minimized.

Another interesting effect is that the base latency we measured on DRAM is higher than on Optane. This is interesting as Optane has a higher access latency than DRAM [20], so the effect should be the other way around. Considering that the rebuild did not work on DRAM and the other effects explained in the previous section, there seems to be an underlying issue when accessing DRAM from the GPU on our test configuration.

6.8.4 Evaluation of our Goal

For RAID-0, we achieve a write bandwidth of 2.4 GB/s on two Optane DIMMs. We showed that, compared to a memcopy to a single DIMM, we can nearly double the bandwidth by using two Optane DIMMs in a RAID-0 configuration. The comparison to CPU-based RAID systems shows that, for RAID-0 on Optane, GPU4FS is at least 100 MB/s faster than the next best competitor. GPU4FS lowers the average global CPU utilization by a factor of x18 compared to its competitors on RAID-0.

For RAID-1, GPU4FS also outperforms its competitors by more than 100 MB/s and reduces global CPU utilization by a factor of x21.5. We showed that RAID-1 only looses 100 MB/s over RAID-0 while introducing redundancy. With a maximum bandwidth of 1.3 GB/s on RAID-1 we are close to the maximum write bandwidth to one Optane DIMM (1.4 GB/s). Compared to RAID-0 and RAID-5, we demonstrated the need to halve the invocation count to maintain an acceptable bandwidth because every write is duplicated.

For RAID-5, GPU4FS also has the highest bandwidth with 1.96 GB/s but is closely followed by one competitor (BTRFS). GPU4FS also reduces the average global CPU utilization by a factor of x13 for RAID-5. Our evaluation of RAID-5 showed that the parity calculation does not add significant computational overhead. This is an important finding because it lets us conclude that efficient RAID-6 Reed-Solomon coding is also a feasible with our parallel distributed parity calculation approach. We showed that lose less than $\frac{1}{4}$ of RAID-0's the maximum bandwidth on RAID-5, because we need to write the parity stripe.

Two more noteworthy findings are: The time to rebuild linearly depends on total bytes and repair, being independent of the number of files regardless of the RAID level. The file write bandwidth, mostly depends on the total file bytes and the number of files does not have a significant impact, as long as the number of files is a multiple of number of dispatched workgroups. That the number of files has no significant effect on the bandwidth implies that the overhead for allocating chunks and acquiring the exclusive chunk lock is neglectable.

Regardless of the RAID level, GPU4FS never utilizes more than half of a CPU core. In the CPU usage analysis, we could see that GPU4FS needs around 5 seconds, to set up Vulkan and to compile the shader. GPU4FS has close to zero CPU utilization while the shader is running. On average, GPU4FS has a global CPU utilization of 2 %.

GPU4FS can outperform its competitors in terms of bandwidth on Optane, but the comparison is limited to our test configuration due to the decreased performance, as explained above. Nevertheless, we showed that we can easily reach 4.1 GB/s on DRAM in the RAID-0 configuration. That means the GPU would have enough performance to fully utilize both Optane DIMMs if the full bandwidth of 2 GB/s per DIMM was achievable from the GPU. We conclude that, at least on Optane, GPU4FS is competitive in terms of bandwidth, if not better than the benchmarked CPU RAID systems. At the same time, GPU4FS lowers the average global CPU utilization by a factor of x13 to x21 depending on the RAID level. Therefore, we can conclude that, on Optane at least, we have achieved our goal of designing a GPU-based RAID system that competes with CPU-based systems in terms of bandwidth while significantly reducing CPU utilization.

7 Future Work

In this chapter, we present topics that are candidates for follow-up work in the field of a GPU-accelerated file system with RAID capabilities.

7.1 RAID-6

In our work, we introduced a design approach for allowing invocations to independently encode RAID-6 parity and proved the efficiency of this approach for RAID-5 6.6.2. The evaluation of this encoding scheme for RAID-6 is still open. Furthermore, exploring how the workgroup's invocations can work together to efficiently solve a linear equation system on Galois Fields for data recovery is still an open topic. Therefore, we consider research into GPU-accelerated RAID-6 a worthwhile research topic.

7.2 Latency Optimization

The evaluation has shown that by offloading RAID management tasks to the GPU, we can achieve competitive bandwidths 6.8. However, it is important for a file/RAID system to have a short base latency to handle requests as fast as possible. As we have shown in the latency evaluation 6.7, the base latency of the current demonstrator is quite high at around 35ms. Therefore, we consider latency minimization to be an important issue and recommend conducting research in this area.

7.3 Checksums

As highlighted in the background section 2.2.2 and explored in related work 3, checksums are a crucial addition to any RAID system. Data corruption can occur at the bit level, making checksums an essential tool for detecting such corruption efficiently. The parallel nature of GPUs makes them well-suited for accelerating checksum calculations. In the event of data corruption detection, the redundancy information encoded into the full-stripe can be employed for data recovery. Therefore, researching GPU-accelerated checksums and their integration into our RAID subsystem appears to be an appealing research direction.

7.4 Crash Consistency

At present, GPU4FS lacks crash consistency, a fundamental requirement for any reliable file system. Achieving crash consistency is particularly challenging, especially in scenarios involving multiple disks where atomic updates are not possible. Implementing a copy-on-write approach for updating data stripes and utilizing logging for committing these updates to multiple disks could be a strategy to explore. Thus, we consider the pursuit for effective crash consistency mechanisms on the GPU an essential research topic.

7.5 TLB

As the file system runs for some time, certain chunks may be frequently accessed. To minimize slow disk accesses, their logical chunk descriptors and some nodes of the chunk tree could be cached in VRAM. Such a TLB structure could accelerate translation from logical pointers to physical offsets. Designing an efficient GPU cache structure that allows parallel access appears to be a promising avenue for further research.

7.6 Dynamic Parallel Allocation & Deallocation

One significant limitation currently is the absence of a functional block allocator in GPU4FS that supports parallel allocation and deallocation. Presently, the only way to achieve page-aligned allocations is by rounding up allocations to page boundaries, resulting in space wastage with no means of freeing memory. This limitation severely hampers the adoption of dynamic data structures. Therefore, we consider research into efficient parallel memory allocation and deallocation on the GPU a critical research area.

7.7 Rebalancing & Defragmentation

After the introduction of deallocation capabilities, disk fragmentation can become an issue. Implementing an online algorithm capable of merging existing chunks and relocating pages within the file system to reduce fragmentation would be a valuable research topic. The abstraction provided by the logical address space can be effectively utilized to move underlying physical chunks while updating the mapping, eliminating the need to modify logical pointers. Additionally, disk rebalancing becomes necessary when new disks are added to the array or existing disks need to be removed to maintain balanced disk utilization. Thus, we consider finding solutions for these two challenges as engaging research opportunities.

7.8 SDD Support and POSIX Compliance

As Optane is discontinued [27], GPU4FS needs to support other storage media. To gain user adoption and enable comparison to file systems implemented for block devices, GPU4FS

should support SSDs. GPU4FS strives to be a POSIX-compliant file system [2] and already incorporates necessary metadata in its data structures [23]. However, as of now, there is no implementation for the POSIX file system APIs available for GPU4FS. Such an implementation is necessary for wide adoption and would facilitate the use of standardized file system benchmarks for future evaluations, enabling fair and standardized comparison to other file systems. Therefore, we recommend implementing the required APIs.

8 Conclusion

In this thesis, we proposed and implemented a flexible RAID system, capable of managing RAID configurations at a granular file level that seamlessly integrates into GPU4FS-a novel GPU accelerated file system.

Historically, RAID systems have played a crucial role in enhancing both data performance and integrity. While hardware-based RAID solutions were popular for their performance benefits, they lacked the flexibility and portability of software-based alternatives. As CPUs evolved and gained processing power, software-based RAID systems emerged as viable options. However, tasks involving complex parity coding, essential for ensuring data redundancy, placed a significant computational burden on CPUs, particularly in configurations with double and triple parity. To confront this challenge, we proposed the integration of a GPU-based RAID system into GPU4FS, building upon the work by Maucher [23].

Our core design concept revolves around a logical address space managed by the GPU4FS file system, which allows for dynamic allocation of files and pages with specific RAID configurations. This approach provides the flexibility to tailor performance and redundancy settings to individual files and processes.

Central to our design was the exploitation of GPU's parallel processing capabilities to calculate parity information and handle write and read requests efficiently. Our goal was to create a GPU-based RAID system that can compete in terms of write bandwidth with its CPU-based counterparts while significantly reducing CPU utilization.

Across RAID-0, RAID-1, and RAID-5, our integrated system consistently achieved higher write bandwidths than the best CPU-based RAID solutions on Optane while simultaneously reducing the average CPU utilization by a factor of x13 to x21 depending on the RAID level. Our tests showed that GPU4FS's parity calculations introduced minimal computational overhead, reaffirming the efficiency of our GPU-centric approach.

In conclusion, our work showcases the potential of a flexible GPU based RAID system managed by a file system to meet the evolving demands of storage tasks. By harnessing the computational power of GPUs, we have successfully designed an innovative alternative to CPU-based RAID systems that can compete in terms of bandwidth while significantly reducing the CPU usage.

Bibliography

[1] mmap(2) - Linux manual page. URL: https://www.man7.org/linux/man-pages/man2/mmap.2.html (visited 08/24/2023).

[2] The Open Group Base Specifications Issue 7, 2018 edition. URL: https://pubs.opengroup.org/onlinepubs/9699919799/ (visited 09/17/2023).

[3] OpenGL Wiki. URL: https://www.khronos.org/opengl/wiki/Main_Page (visited 08/23/2023).

[4] psutil documentation — psutil 5.9.6 documentation. URL: https://psutil.readthedocs.io/en/latest/#psutil.cpu_percent (visited 09/16/2023).

[5] RAIDZ Types Reference. URL: https://raidz-calculator.com/raidz-types-reference.aspx (visited 08/16/2023).

[6] Introduction of B-Tree, April 2013. Section: Advanced Data Structure, URL: https://www.geeksforgeeks.org/introduction-of-b-tree-2/ (visited 08/31/2023).

[7] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[8] M. Blaum, J. Brady, J. Bruck, and Jai Menon. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44(2):192–202, February 1995. Conference Name: IEEE Transactions on Computers.

[9] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.

[10] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[11] Douglas Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979. Publisher: ACM New York, NY, USA.

[12] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In

Proceedings of the 3rd USENIX Conference on File and Storage Technologies, FAST'04, page 1, USA, 2004. USENIX Association. event-place: San Francisco, CA.

[13] Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. Gibraltar: A Reed-Solomon coding library for storage applications on programmable graphics processors. *Concurrency and Computation: Practice and Experience*, 23(18):2477–2495, 2011. Publisher: Wiley Online Library.

[14] Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. Accelerating reed-solomon coding in raid systems with gpus. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–6. IEEE, 2008.

[15] Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. Arbitrary dimension reed-solomon coding and decoding for extended raid on gpus. In *2008 3rd Petascale Data Storage Workshop*, pages 1–3. IEEE, 2008.

[16] Matthew L Curry, H Lee Ward, Anthony Skjellum, and Ron Brightwell. A lightweight, gpu-based software raid system. In *2010 39th International Conference on Parallel Processing*, pages 565–572. IEEE, 2010.

[17] Michael Gilroy and James Irvine. RAID 6 Hardware Acceleration. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6, August 2006. ISSN: 1946-1488.

[18] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[19] Brian Hickmann and Kynan Shook. ZFS and RAID-Z: The Über-FS? *University of Wisconsin–Madison*, 2007.

[20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, March 2019.

[21] Aleksandr Khasymski, M. Mustafa Rafique, Ali R. Butt, Sudharshan S. Vazhkudai, and Dimitrios S. Nikolopoulos. On the Use of GPUs in Realizing Cost-Effective Distributed RAID. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 469–478, 2012.

[22] Adam Leventhal. Triple-Parity RAID and Beyond: As hard-drive capacities continue to outpace their throughput, the time has come for a new level of RAID. *Queue*, 7(11):30–39, December 2009.

[23] Peter Maucher. GPU4FS: A Graphics Processor-Accelerated File System. Master's thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, 2022.

[24] 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*, pages 109–116, 1988.

[25] James S. Plank. A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems. *Software: Practice and Experience*, 27(9):995–1012, 1997.

[26] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage*, 9(3), August 2013. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[27] Ryan Smith. Intel To Wind Down Optane Memory Business - 3D XPoint Storage Tech Reaches Its End. URL: `https://www.anandtech.com/show/17515/intel-to-wind-down-optane-memory-business` (visisted 09/16/2023).

[28] Weibin Sun, Robert Ricci, and Matthew L. Curry. GPUstore: harnessing GPU computing for storage systems in the OS kernel. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 1–12, New York, NY, USA, June 2012. Association for Computing Machinery.

[29] David Teigland and Heinz Mauelshagen. Volume Managers in Linux. In *USENIX Annual Technical Conference, FREENIX Track*, pages 185–197, 2001.