Karlsruhe Institute of Technology

Extending GPU4FS with Advanced File System Functionalities

Bachelor's Thesis
submitted by

cand. inform. Nico Rath

to the KIT Department of Informatics

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

29. April 2023 – 29. August 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, August 29, 2023

Abstract

Persistent non-volatile memory (PMem) gained more and more attention over the years. When compared to SSDs, it shines with lower latency and higher read throughput. However, its write bandwidth is a major drawback. As PMem is connected via the memory bus, the OS does not detect an IO-bound task; which lets CPU cores stall more likely. Shifting PMem interaction to an accelerator is one way to resolve this problem. GPU4FS is a graphics card accelerated file system. However, it does not provide any advanced features. In this thesis, we extend GPU4FS with checksum and deduplication functionality. While checksums protect a file system's content against corruption, deduplication can save storage by detecting duplicate data blocks. Both functionalities are suitable for a GPU, as their implementation is parallelizable. Our results show that GPU acceleration of those features becomes more feasible the more load the file system faces. Thus, a hybrid mode with mixed CPU/GPU utilization seems like a promising concept.

Persistenter, nicht-flüchtiger Speicher ist ein neu aufkommendes Feld im Bereich der Speichergeräte. Dieser glänzt durch niedrige Latenz und hohe Leserate. Ein Nachteil des sogenannten PMems ist seine vergleichsweise niedrige Schreibbandbreite. Da PMem über den Speicherbus an die CPU angebunden wird, erkennt das Betriebssystem einen interagierenden Prozess als CPU gebunden; was zu einem stallen der involvierten Kerne führt. Das Auslagern dieser Interaktion auf einen Beschleuniger kann dieses Problem lösen. GPU4FS implementiert ein Dateisystem vollständig auf der GPU, dem allerdings erweiterte Funktionen fehlen. Wir erweitern GPU4FS um Checksum und Deduplication Funktionalitäten, welche aufgrund ihrer Parallelisierbarkeit für eine GPU-Beschleunigung geeignet sind. Während eine Prüfsumme (Checksum) fehlerhaft gespeicherte Dateien identifizieren kann, bietet Deduplizierung (Deduplication) mehr Speicherplatz durch Duplikatelimination. Unsere Ergebnisse zeigen, dass die GPU bessere Ergebnisse liefert, wenn sie mit großen bzw. vielen Dateien zu tun hat. Wenn sie kleine bzw. wenige Dateien bearbeiten soll, ist sie der Flaschenhals. Daher schlagen wir weitere Forschung zu einem Hybrid-Modus vor, der die CPU unter niedriger Last verwendet und erst bei steigender Last auf die GPU umschaltet.

Contents

Chapter 1

Introduction

In recent years, modern computing systems gained more and more performance in terms of Central Processing Unit (CPU), Random Access Memory (RAM), and also secondary storage speed [1]. Within the last decades, CPUs have comprised not only one but multiple cores to work on multiple processes simultaneously [2]. In terms of Flynn's Taxonomy [3], multicore CPUs follow the "Multiple Instruction, Multiple Data" (MIMD) architecture with a relatively low level of parallel processing. Modern multicore CPUs for example, contain 8 to 32 cores on average, which allow a fully parallel execution of 8 to 32 instructions [4]. A so-called "Single Instruction, Multiple Thread" (SIMT) architecture is implemented by modern Graphics Processing Units (GPUs) [5]. The main purpose of a GPU is image rendering, which is a highly parallel task [6]. Although they can compute sequentially, GPUs are designed to process hundreds or thousands of instructions simultaneously. In contrast to CPUs, GPUs contain many small cores instead of a few big cores [7].

Modern file systems not only allow writing and reading data from and to a disk, they provide more advanced features. Btrfs [8] or ZFS [9] as modern Copy-On-Write (COW) file systems implement some advanced features like deduplication, snapshots, mirroring, or striping. As most file systems generally divide a disk into blocks with a fixed size, most of the file system tasks are done in parallel fashion: One specific task (checksumming, deduplication, read) usually processes multiple blocks. These tasks are capable of processing the involved blocks independently of each other. However, CPUs are designed to work on sequential operations. In addition, modern file systems require more compute resources from a CPU to provide their advanced features [10]. Consequently, those resources cannot be used for application-related tasks. Thus, GPUs with their parallel processing nature, could be the perfect accelerator for those file system features while reducing stress on the CPU.

Non-Volatile Memory (NVM) is one type of secondary storage. Intel Optane [11] as prominent representative is designed to provide DRAM-comparable

performance [12] with parallel access. The fastest type of Intel Optane connects directly to the memory bus of some Intel CPUs, allowing them to access Optane the same way as main memory [13]. Optane's write performance is not as fast as modern main memory, especially under moderate load. Thus, CPU cores are likely to stall [13, 10]. Shifting those file system tasks to another processing unit allows the CPU to work on application-related tasks instead of waiting for the storage device.

Therefore, the evaluation of GPU-based file systems seems promising. The file system GPU4FS [10] uses the GPU to accelerate file system calls, with NVM as its main storage target. However, GPU4FS currently does not provide advanced features. The goal of this thesis is to extend GPU4FS with checksum capabilities and to implement deduplication as one use case of checksums. Furthermore, we target minimal overhead and want to evaluate the feasibility of those features in terms of performance. Additionally, we contribute to GPU4FS's original goal of reducing as much stress on the CPU as possible.

This thesis is organized as follows: Chapter 2 introduces the necessary background about file systems, checksums, deduplication, and GPU programming. Chapter 3 continues with an overview of the original GPU4FS as well as related and well-known file systems. In Chapter 4, we introduce our file system design, which includes both checksumming and deduplication. Chapter 5 provides implementation details about the ported checksum algorithm as well as the file system features. Within Chapter 6, we evaluate the features and summarize further implications for GPU4FS.

Chapter 2

Background

The following section introduces important background for this thesis. We initially give a short summary of different storage devices, introduce important taxonomies of a file system and outline the state-of-the-art checksum algorithms and deduplication. Furthermore, we explain GPU-related concepts in combination with thesis-related aspects of the GPU framework Vulkan [14].

2.1 Storage Devices

The variety of available storage devices evolved significantly in recent years. Not only shifted the primary storage medium from slow Hard Disk Drives (HDDs) to fast Solid State Disks (SSDs), new storage technologies also emerged. This section gives an overview of various storage devices, which we mainly differentiate by their addressing granularity.

2.1.1 Block-Addressable Devices

Block-addressable devices are the most common non-volatile storage devices — at least in the desktop market. Well-known representatives of this area are HDDs and SSDs. While an HDD consists of spinning disks, an SSD consists of NAND flash cells. The HDD's spinning disks are accessed via a moving read/write head. By design, an SSD is much faster than an HDD, especially when it comes to random access [15].

An HDD is typically accessed via SATA [16]. While some modern SSDs still use a SATA controller, the overhead of the SATA protocol started to dominate their access time. Thus, so-called Non-Volatile Memory express (NVMe) SSDs evolved. Those are connected via PCIe bus, which provides much higher speeds than SATA; one single PCIe 3.0 lane provides 1 GB/s, while SATA3 provides a maximum of

600 MB/s [17]. The design goals of NVMe drives were low latency, performance, and parallelism. This implies future relevance for an accelerated file system [17].

As the name implies, block-addressable devices cannot be accessed byte-individually: The Operating System (OS) must request a contiguous, hardware-defined area of bytes instead of individual bytes. For example, a typical HDD sector consists of 512 bytes. Thus, requesting one byte on the application layer translates to a sector request on the physical layer. Furthermore, writing bytes to disk results in rewriting the entire sector, not just single bytes [18].

2.1.2 Byte-Addressable Devices

While block-addressable devices are suitable for storing large, contiguous data patterns, a byte-addressable device is designed to be updated more frequently with scattered access patterns. Thus, one common representative of byte-addressable memory is Dynamic Random Access Memory (DRAM). DRAM provides volatile storage space, which is way faster than any block-addressable device. Typically, the slower block-addressable devices hold system-related data which is loaded into DRAM during system boot or on purpose. This allows an application to benefit from DRAM's faster speeds [18].

In recent years, a new evolution in the world of byte-addressable devices blurred the line between the two aforementioned device types: Persistent Memory (PMem) and, most importantly, Intel Optane ® Dual Inline Memory Modules (DIMMs). Those storage technologies provide access in byte granularity, while persisting data non-volatile. Additionally, Optane provides access latencies in the order of 500ns, which is comparable to DRAM and even faster than NVMe. Optane DIMMs are connected via memory bus, which makes them accessible the same way as DRAM. This mitigates protocol-related overhead [19].

However, Intel Optane faces one major downside compared to DRAM and NVMe SSDs: Writes to an Optane DIMM are relatively slow, resulting in a maximum bandwidth of around 2 GB/s per DIMM [13]. As Optane is connected to the memory bus of the CPU, the OS does not detect an I/O task. Thus, operating cores are stalling, while the OS is not doing a context switch. This results in wasted valuable CPU time [20].

2.2 File Systems

After covering the variety of storage devices, we take a closer look at their logical organization via file systems. A file system is an essential software part of the OS, which is responsible for storing any files on storage devices in a reliable and efficient manner. Excluding some exceptions (e.g., tmpfs [21]), the data will be persisted. The file system exposes an Application Programmable Interface (API) towards applications, allowing them to use its capabilities. Contrary to main memory where each process receives its own address space, a file system's design assumes that the stored persistent data needs to be shared between multiple processes [18].

2.2.1 Organization

As file systems are implemented in software, a variety with different capabilities exist. While the file system hides the implementation details from user space by providing a standardized interface like POSIX [22], its internal details are important for this thesis. We will cover some of them in the following [18].

On-Disk Structures

To organize the contents of a storage device, the file system introduces specific on-disk structures. These not only organize bytes logically into units commonly known as files or directories, but are also fundamental for implementing advanced features like checksumming or deduplication. However, not all file systems need to implement the following structures similarly.

An **inode** is the root component of a persisted file. It has a unique identification number and stores file-related information. Typical examples are type (e.g., file), length, permissions, and references to the actual file content [18].

Directories organize inodes by referencing the inode number through so-called hard links. An inode can be referenced by many directories, with different file names associated. Thus, the file name is stored inside the directory entry, not the inode [18].

As file systems were originally developed for block devices, they organize content and their own structures in so-called **blocks**. Such a block is the smallest addressable unit on disk. As a block device not always has enough free contiguous blocks for a whole file, those blocks could be scattered across the drive. Thus, so-called **block pointers** address a block uniquely, and store optional metadata. Block pointers can then be used to reference the file content within an inode. As an inode is typically fixed-size, it provides limited space for referencing block

pointers. Some existing implementations solve this problem by utilizing a multi-level hierarchy of block pointers, similar to a memory page table [23, 24]. This hierarchy does not allocate data blocks directly within the inode, it references so-called **indirect block pointers** which preserve storage for more block pointers [18].

To reduce the amount of block pointers, some implementations refer to the concept of **clusters** [25] or **extents** [23]. Those are an either fixed or variable sized aggregate of contiguous disk blocks. The concept of extents is found in many file systems, although they use different terminology to refer to the same principle [18].

When mounting the file system, the driver usually needs knowledge about some file-system-related configuration. This configuration is stored in the so-called **superblock**. The superblock is stored in a fixed location on-drive. This allows the file system driver to load its content safely, as it can load it from a fixed address regardless of the drive's size or type [18].

Digression: Kernel Space vs. User Space

Within OSes, a process's execution context is divided into two categories: The kernel space and the user space. Communication between those two spaces happens through so-called system calls. Such a call allows a user space process to invoke a kernel space functionality. Consequently, a system call transfers control into the OS by executing a "trap" hardware instruction and raising the privilege level to kernel mode [18]. The control-taking kernel level thread has nearly no limitations regarding memory access and security principles. It has the same access privilege as the OS itself — allowing it to access arbitrary memory addresses and manage the hardware. Therefore, applications with many kernel interactions and a trusted behavior are loaded into the kernel space rather than the user space. Candidates for a kernel space execution are e.g., device drivers or file systems. As they provide and manage access to a whole I/O device, file systems tend to access arbitrary memory areas often. [18]. Thus, implementing the file system in kernel space is common practice.

However, user space file systems like Aerie [26], Strata [27] or SplitFS [28] emerged in recent years. User space file systems try to shift as much work as possible from the kernel into the user space to reduce kernel-related overhead. To interact with a user space file system, an application interacts directly with the file system driver and not with the OS. Instead, the user space file system decides when to interact with the OS, which allows it to reduce overhead.

2.3 Checksum Algorithms

Checksum algorithms have one property in common: They produce a fixed-length bit pattern from an arbitrary-sized input sequence. However, the purpose of this fixed-length bit pattern varies based on the chosen checksum algorithm: While some algorithms are designed to solely detect bit flips [29, 30] — bits whose state changed unintentionally [31] — the checksum family also includes cryptographic checksum algorithms. Their purpose is to produce a unique bit pattern which references an input sequence [32]. As one major goal of this thesis is to design checksum (sum) functionality for a novel file system, we take a closer look at different checksumming techniques in the following section. Initially, we cover some error-checking and bit-flip-detecting algorithms. We then introduce cryptographic checksums and their construction techniques, with one algorithm depicted in detail. Finally, we conclude with a discussion about the introduced mechanisms and explain the reason for choosing one particular algorithm within this thesis.

2.3.1 Cyclic Redundancy Check

Cyclic Redundancy Check (CRC) is an error-checking code which is mainly used within data transmission systems. It consists of an amount of parity bits, which are calculated over a bit input sequence. Those parity bits represent the validity of the checksummed data and can be recalculated for validation purposes [29].

In context of a CRC computation, the input bit sequence is interpreted as polynomial. To calculate the parity bits, a generator polynomial is introduced. The parity bits are calculated by multiplying the input polynomial by x^n, where n is the degree of the generator polynomial. Afterwards, the algorithm divides the result of that multiplication by the generator polynomial. The division's reminder represents the parity bits of the input sequence [29]. As a binary multiplication or division is equivalent to a left or right shift, those operations can efficiently be implemented in software or even in hardware circuits (so-called LFSRs) [33].

CRC comes in many flavors, with different error-checking capabilities. Some examples are CRC16 or CRC32. The number in their names represent the degree of the generator polynomial [29]. A generator polynomial generally has better error-checking capabilities the higher its degree is [34].

2.3.2 Fletcher Checksum

A Fletcher checksum is significantly easier to compute than a CRC code while giving nearly equivalent error detection properties [30]. Many common file systems implement Fletcher's sum [9, 8].

Fletcher's algorithm calculates two sums over its input bytes, sum_1 and sum_2. The algorithm computes iteratively by splitting its input into K bit long blocks. The sums are calculated modulo M, whereas $M = 2^K$ or $M = 2^K - 1$ according to Fletcher. To compute sum_1, each block is consecutively added modulo M. sum_2 is then computed by taking $sum_{1,i}$ after block i was added, and adding $sum_{2,i-1}$ to it. $sum_{2,i-1}$ represents the value of sum_2 after the $i-1$-th iteration. Finally, $sum_{1,n}$ and $sum_{2,n}$ are appended to represent the $2 \cdot K$ long checksum value [35].

2.3.3 Cryptographic hash functions

Cryptographic hash functions are hash functions which have special properties, especially in terms of security. A cryptographic hash function is either a Message Authentication Code (MAC) or a Manipulation Detection Code (MDC). While the former utilizes a secret key to encrypt a public message, the latter does not require any kind of secret key. As the name implies, an MDC is an encryption method which detects manipulation within the data it shall protect. A MAC on the other hand guarantees that a message originates from a specific author. We want to detect a data manipulation rather than validating a block's original author. Therefore, MAC functions are not relevant for the purpose of this thesis [32].

Let H be an MDC, x an input of arbitrary length, and $H(x)$ the hash value of x produced by H. H can then be subdivided into multiple categories [32, 36]. For this thesis, Collision Resistant Hash Functions (CRHF) [37, 38] are relevant, which are a subset of One Way Hash Functions (OWHF) [39].

H is an **OWHF** as defined by Merkle [39] if it satisfies the following conditions[36]:

- x can be of arbitrary size
- H outputs values with fixed length
- $H(x)$ is polynomial time computable when H and x are given
- Pre-image resistance: Finding x is computational infeasible if H and $H(x)$ are known
- Second pre-image resistance: Finding x and x' such that $H(x) = H(x')$ is computational infeasible if H and $H(x)$ are known

H is called a **CRHF** if it is an **OWHF** and satisfies the following definition, given by Merkle [36, 40]:

- Hash collisions: Finding x and y that resolve to the same hash value ($H(x) = H(y)$) is computational infeasible if only H is known

In contrast to the properties of an OWHF, the aforementioned hash collision property of a CRHF reduces the amount of known information further — knowing only H is a stronger condition than knowing H and $H(x)$ for an input value x. Our takeaway from these definitions is that within a CRHF, a collision occurs with "much fewer" probability than within an OWHF. Only if the hash function is collision-resistant, it can be called a **cryptographical** hash function.

Cryptographic Hash Functions are constructed in many ways. In the following, we explain well-known construction techniques.

Merkle-Damgard Iterative Hashing

The Merkle-Damgard construction technique was proposed by Merkle [38] and Damgard [41] independently. Their technique relies on the fact that a collision-resistant hash function f with a fixed-length input can be used to hash a variable-length input.

The Merkle-Damgard technique divides the variable-length input b in equal length sub-blocks b_i. Their length is congruent to the input length of f. The hash value $f(b_{i+1})$ of b_{i+1} is then calculated by applying the hash function f to the block b_{i+1} with respect to the hash value $f(b_i)$ of block b_i. As block b_0 has no predecessor, the construction technique relies on an initialization vector (IV). This procedure repeats iteratively from b_0 to b_n, whereas the last hashing step outputs the final hash value $f(b)$ of b. A more formal definition is given by Sobti et al [36].

Many well-known cryptographic hash functions utilize the Merkle-Damgard construction technique. Some examples are MD5 [42], SHA-1 [43], and the SHA-2 [43] family. Although it is one of the most used construction techniques, it faces some drawbacks regarding security [44, 45, 46]. Therefore, new construction techniques evolved.

HAIFA

The Hash Iterated Framework (HAIFA) method overcomes many drawbacks from the Merkle-Damgard construction technique. HAIFA utilizes the same approach as the Merkle-Damgard technique, with some modifications: It mainly introduces the number of hashed bits as well as a salt value into the hash function. Those modifications to the Merkle-Damgard technique mitigate its weaknesses against (second) pre-image collisions [47].

Sponge Construction

The Sponge construction technique also follows an iterative construction scheme. It works on arbitrary input and output sizes. A sponge construction operates in two

phases — the absorbing and the squeezing phase. The first phase divides the input in fixed-length blocks, and absorbs them iteratively into a hash state. Each block absorption involves a pseudorandom permutation of the hash state. Afterwards, the squeezing phase receives the fully absorbed hash state. It iteratively outputs a fixed amount of bits from the received state. Each iteration involves another permutation of the hash state before outputting the next set of bits. This squeezing process repeats iteratively until the outputted hash is long enough [48, 36].

SHA-3 is one exemplary family which uses the Sponge construction. It provides various different variants [43].

2.3.4 BLAKE3

BLAKE3 [49] is an evolution of the BLAKE2 [50] hash family. It targets 128-Bit security and is therefore considered collision-resistant [49]. 128-Bit security means that a colliding hash could be computed within 2^{128} operations [51].

Basic Procedure

BLAKE3 compresses its input in fixed-size chunks of 1024 bytes. Those are processed independently. Every chunk is further split into a set of blocks, containing 64 bytes each. BLAKE3 uses those blocks to calculate a chunk's hash value using the HAIFA method. Their hash values are used afterwards to calculate the final hash value of the input. BLAKE3 supports inputs of a length up to 2^{64} bytes [49], which is congruent to the maximum supported file size of GPU4FS[10].

In the following, we explain the BLAKE3 algorithm in detail. We start by explaining the construction of a chunk's hash value, which is depicted in Algorithm 1. Afterwards, we explain the so-called **Merkle-Tree** construction, which produces the input's hash value. An exemplary Merkle-Tree construction for a 4 KiB input is additionally depicted in Figure 2.1.

Detailed Procedure

BLAKE3 uses HAIFA to combine the 64 byte large blocks to a chunk checksum. To keep track of the compression's progress, BLAKE3 holds the intermediate results of a chunk in a stateful way. The authors call their implementation of the HAIFA method "Chunk Chaining Values", and describe the algorithm, which is sketched in Algorithm 1, in the following way: Let $s_{i,k}$ be the state which compresses i blocks of the k-th chunk. Starting with an 256-Bit initialization state $s_{0,k}$ and the 512-Bit long initial block $b_{0,k}$ of each chunk k, every iteration considers the content of block $b_{i,k}$ to update state $s_{i,k}$ to state $s_{i+1,k}$. The state $s_{0,k}$ consists of a predefined initialization vector IV, the position of chunk k in the inputstream, the

amount of bytes within the block $b_{0,k}$, and 32 bits of domain-specific information (Lines 3 and 5 to 14). While the initialization vector corresponds to the initialization routine of the Merkle-Damgard construction technique, the latter three initialization objects originate from the HAIFA technique. To process all blocks of a chunk, the algorithm needs to iterate at most 16 times. This is the outer loop (Line 3). The state $s_{i+1,k}$ is constructed by passing the state $s_{i,k}$ and the block $b_{i,k}$ to a compression function G (Line 16). This function iterates over this state-block combination seven times, which we call the inner loop (Line 14). Each inner iteration considers a permutation scheme (Line 17), which permutes $b_{i,k}$ before applying G again. Before continuing with the next outer iteration, the state $s_{i,k}$ goes through an XOR procedure (Lines 19 to 21) and finally results in the new state $s_{i+1,k}$. The inputted block $b_{i,k}$ is also called the message block, which we refer to in later chapters.

After constructing $s_{i+1,k}$, the algorithm continues with the next outer iteration. The "Chunk Chaining Values" technique replaces the last 256 bit of $s_{i+1,k}$ with values 0–3 from IV, the position of chunk k in the inputstream, the amount of bytes within the block $b_{i+1,k}$, and 32 bits of domain-specific information, before the procedure repeats (Lines 5 to 14). As each state $s_{i,k}$ is only dependent from its predecessor $s_{i-1,k}$, but not from any other states $s_{i,k'}$, the "Chunk Chaining Values" process is fully parallelizable. The final state $s_{n,k}$ corresponds to the hash value of the chunk k [49].

Each chunk's hash value $s_{n,k}$ is then arranged in a binary-tree fashion. For all k, the $s_{n,k}$ values are leaf nodes in this binary tree, located on layer l_n. Their parent nodes $sp_{1,k}$ on layer l_{n-1} consists of the compressed values of the pairs $(s_{n,2k}, s_{n,2k+1})$. To compress those pairs, the authors feed them as message block into BLAKE3's compression function, with a newly initialized state. Those parent nodes are themselves compressed in pairs $(sp_{1,2k}, sp_{1,2k+1})$ to create their parent nodes $sp_{2,k}$ on layer l_{n-2}, up to the root node. The root node on layer l_0 finally contains the hash value of the input. Such a binary tree construction is also called a Merkle-Tree [52, 49].

The authors propose a multi threading approach for BLAKE3. This approach follows the divide-and-conquer paradigm and builds the Merkle-Tree in a top-down fashion. It starts by splitting the input into a fully occupied left and a remaining right part. Here, the left subtree always contains a number of chunks greater than or equal to the number of chunks in the right sibling. The procedure continues until the input is split into chunks. After applying the "Chunk Chaining Values" approach, the backtracing happens by following the recursion stack [49]. This approach fits a fork-join concurrency model, which is provided by multi threading libraries as OpenMP [53] or Rayon (Rust) [54].

Algorithm 1: Chunk Chaining Values Algorithm. See Figure 2.1 for its usage in BLAKE3

Data: chunk c_k with $|c_k| \leq 1024$
Result: $s_{16,k}$ — The hash value of chunk c_k

/* State, empty at beginning */
$s \leftarrow []$;
/* Subdivide c_k in blocks $b[i]$ with $|b[i]| \leq 64$ byte */
$b \leftarrow divide(c_k)$;
/* Initialize start state */
$s[0;7] \leftarrow IV[0;7]$;
/* Outer Loop */
for $i \leftarrow 0$ **to** $b.length() - 1$ **do**
  /* Initialize outer loop values */
  $s[8;11] \leftarrow IV[0;3]$;
  /* Fill 64 bit value k in two 32 bit words */
  $s[12;13] \leftarrow k$;
  $s[14] \leftarrow |b[i]|$;
  if $i == 0$ **then**
    /* First iteration is chunk start */
    $s[15] \leftarrow CHUNK\_START$;
  else if $i == b.length() - 1$ **then**
    /* Last iteration is chunk end */
    $s[15] \leftarrow CHUNK\_END$;
  else
    /* No domain information within chunk */
    $s[15] \leftarrow 0$;
  end if
  /* Inner Loop */
  for $j \leftarrow 0$ **to** 7 **do**
    $G(s, b[i])$;
    $permute(b[i])$;
  end for
  /* Final XOR to construct hash value from state */
  for $j \leftarrow 0$ **to** 7 **do**
    $s[i] \leftarrow s[i] \oplus s[i+8]$;
  end for
end for
return $s[0;7]$;

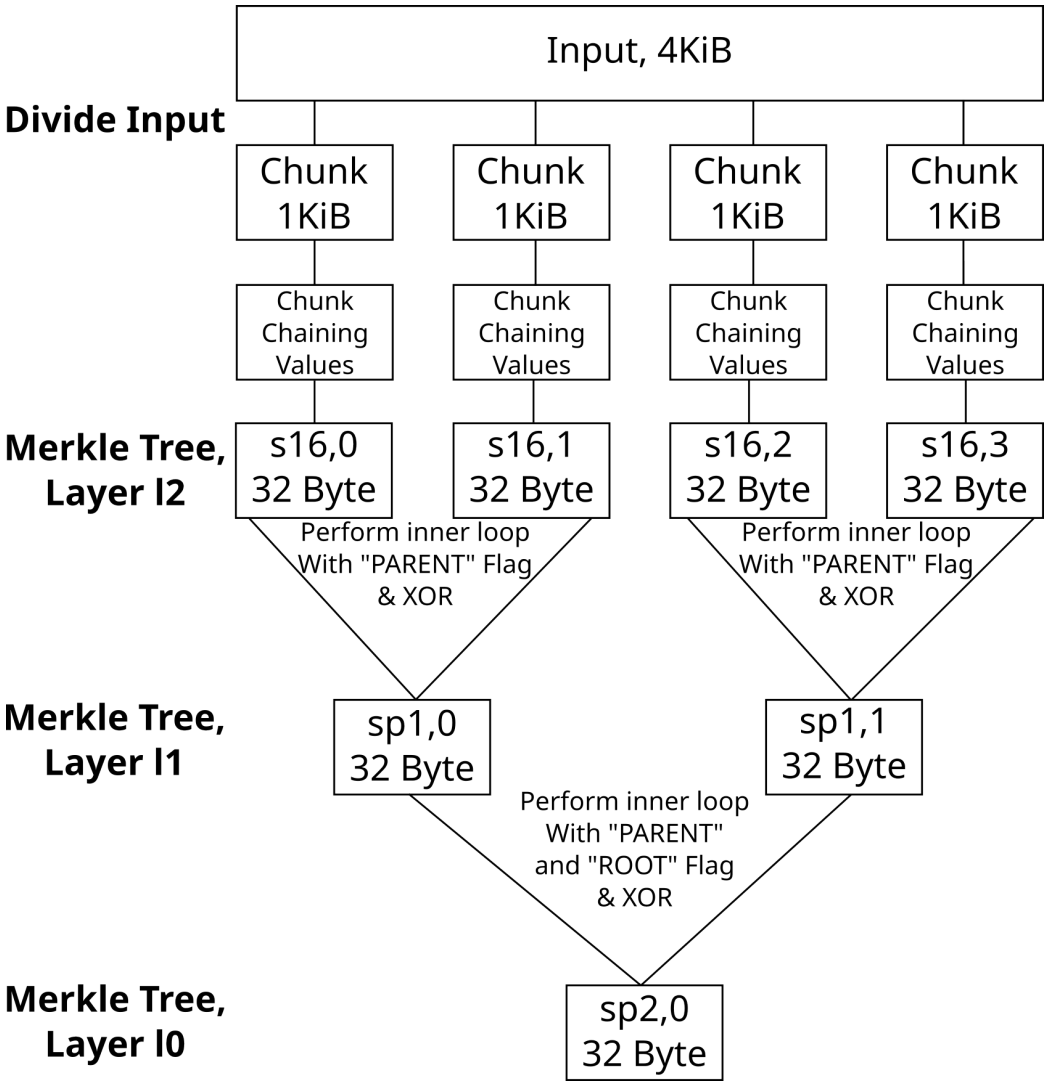


Figure 2.1: Blake3 Merkle-Tree construction for 4 KiB input. See Algorithm 1 for a detailed explanation of “Chunk Chaining Values”.

2.3.5 Discussion

The outlined checksum algorithms are just a subset of all existing algorithms. As there is not one checksum algorithm which satisfies all conditions, modern file systems like Btrfs [8] or ZFS [9] provide many checksum algorithms. Thus, the user is responsible for selecting an appropriate algorithm which matches the contextual use case. BLAKE3 seems to be the most promising algorithm for our use case. According to the authors, it has an "unbound degree of parallelism [...] that scales up to any number of SIMD lanes" [49] — which matches a GPU's execution model. SHA256 [43], MD5 [42] or the recent standard SHA-3 [55, 56] are calculated sequentially, presumably as their design is intended for CPU computation. In addition, BLAKE3 shows higher speeds than well-known cryptographic hash functions, and especially its predecessor [57]. Its superiority over CRC checksums is given by its applied context; cryptographic hash functions are fundamental for further features, e.g., deduplication, as outlined in Section 2.4.

2.4 Deduplication

In context of a file system, the main purpose of deduplication is to save storage space by eliminating duplicated blocks, files, or bytes [58]. The following section motivates the usage of deduplication within GPU4FS, and describes existing implementation techniques. We conclude with a discussion about the several construction techniques.

2.4.1 Motivation

As the amount of data is growing rapidly [59, 60], more and more research took place in organizing files on drive efficiently. This led to further development in the field of deduplication, making it suitable for several use cases: Cloud services have utilized deduplication for years, as they are prone to data duplicates across different users [60, 61]. Backups are very likely to contain redundant information, which is why modern backup solutions like Restic employ deduplication [62].

Even the desktop world provides use cases for deduplication, as the trend in the Linux world goes to containerization, immutable distributions, and sandboxed applications [63, 64]. Examples of the immutability trend are given by Flatpak [65] or Docker [66], whose main purpose is to sandbox applications into their own container. This container brings all required dependencies for the application and prohibits access to foreign content per default. However, many applications share some similar dependencies, which need to be persisted multiple times. Here, deduplication could save space by mitigating the amount of dependency duplicates.

Especially with Fedora's announcement to shift its focus to immutable distributions [63], we believe that deduplication is an important feature for a modern file system.

Deduplication can be implemented inline or post-processing [67]. While an inline deduplication dedups synchronously right before a corresponding disk write, a post-processing deduplication can act asynchronously at any point in time.

2.4.2 Chunk Sizes

Deduplication is applicable on specific granularity levels, e.g., chunk-level [68, 69] or file-level [70].

The chunk-level is able to utilize a definable fixed [68] or variable [69] chunk size. Thus, the given input stream is divided into those chunks and deduplicated afterwards. A fixed-size chunk-level deduplication works as follows: After splitting the input into the according chunk size, a hash function is applied to generate a so-called fingerprint of all chunks. This fingerprint represents the chunk uniquely. Thus, a cryptographic checksum is mandatory, as the collision resistance property outlined in Section 2.3.3 guarantees the uniqueness of the fingerprint. If an equal chunk enters the deduplication process afterwards, its fingerprint matches the already stored fingerprint of the prior chunk. Thus, the duplicate is detected. However, fixed-size chunking is prone to the boundary-shift problem [69]: As the chunk boundaries are fixed, an input is always subdivided the same way. This means that if an equal input stream is modified by inserting some bytes at its beginning, the positions of all following bytes change. Thus, the modified input stream has no similar chunks to its unmodified variant, leading to zero matches during the deduplication process.

To address this problem, the technique of variable-sized chunking evolved. Content-Defined Chunking (CDC) is the technique which resolves the boundary-shift problem. CDC splits based on the content within the input stream, not at predefined boundaries [69]. One way to implement this technique is via Rabin fingerprints [71] — a non-cryptographical hash function whose output can be used to detect if some given chunk-splitting conditions are satisfied. Restic for example defines those conditions as satisfied if the lowest 21 bits of the Rabin fingerprint are zero [62]. Although Rabin fingerprinting is the most common technique to implement CDC, there exist a variety of faster algorithms [72, 73], which even include GPGPU variants [74, 75]. After determining the variable-sized chunks, the procedure continues analogously to a fixed-size chunk deduplication.

In contrast, a file-level deduplication [70] detects whole files as duplicates. Thus, a file's data is not split into chunks and fingerprinted independently, but the overall file is fingerprinted and stored as duplication candidate.

2.4.3 Discussion

The proposed techniques have their pros and cons. Although a file-level deduplication imposes less overhead, its deduplication rate is worse compared to a chunk-level deduplication. A chunk-level deduplication on the other hand needs more computational power. Moreover, a variable-sized chunking is computational more intensive than a fixed-size chunking. However, the performance and especially deduplication ratio of all proposed techniques depend on their applied context. Microsoft's researchers found that "whole-file deduplication is a highly efficient means of lowering storage consumption"[58], while the amount of research in chunk-level deduplication also indicates the relevance of that technique.

2.4.4 Hashing Techniques

As proposed in the prior section, a chunk's unique identifier is computed via a collision-resistant hash function. However, storing the resulting key-value combinations efficiently is another important topic in the field of deduplication. This section introduces techniques for storing those pairs efficiently, with the ability to retrieve them fast. The techniques from this section founded our data structure from Section 4.2.

Extendible Hashing

Extendible hashing is a technique which builds dynamically sized hash tables. Its basic principle uses a single, so-called directory to store parts of the hash key. The directory size determines the length of the stored key. Each directory entry references one so-called leaf, whereas a leaf can be referenced by multiple directory entries. A leaf's purpose is to store all checksums with a common prefix, which is also present in the referencing directory entries. A key's value is retrieved by looking it up in the directory. This happens by truncating the key's length to the key length of the directory. The directory responds with the leaf, which then holds the key-value combination. If a leaf gets too small, the directory size is increased and the leaf is split in two — which includes partly rehashing [76].

CCEH

Cacheline-Conscious Extendible Hashing (CCEH) is a hashing structure which advances extendible hashing. It uses the same directory structure as extendible hashing, but points to so-called segments rather than leaves. Those segments are able to group multiple leaves together. To identify a leaf within a segment, CCEH uses the least significant bits (LSBs) of the complete key. A segment is identified by the most significant bits (MSBs) [77].

2.5 GPU Programming

A GPU operates fundamentally differently from a CPU. Whereas the CPU was primarily designed for sequentially-fashioned tasks, the GPU's design was specialized for image processing [78]. Image processing tasks are mostly SIMD-fashioned: One operation, applied to many pixels. Thus, a GPU provides many independently operating "core-counterparts" which are, compared to a CPU core, less powerful. However, their increasing computational potential led to research in non-graphics related tasks and founded the terminology of the so-called General Purpose Graphics Processing Unit (GPGPU) technique [78]. The following section introduces core aspects of GPU programming, starting with a short architectural overview of GPU architectures. We afterwards introduce the framework which is used within GPU4FS to configure and program the GPU.

2.5.1 Towards GPU Architectures

To understand some of our implementation details, it is important to know the basic architecture of a GPU chip. A GPU is a complex chip, not to mention the different terminologies between different vendors. Nevertheless, all GPUs utilize the same fundamental concepts. Therefore, we give a short and generally applicable overview of thesis-relevant GPU concepts without diving into vendor-specific details.

Figure 2.2 depicts a general applicable GPU architecture. We chose to use AMD RDNA3 terminologies in our explanations [79], but the concepts apply to any vendor. A GPU is connected to the CPU via the Peripheral Communication Interface express (PCIe) bus. This bus is a bidirectional communication channel used for data exchange. The GPU itself is divided into many distinct cores, the **Work-group Processors** (WGP). Each WGP consists of one or more **Compute Units** (CU), which perform the actual calculations. From a hardware perspective, those CUs can be seen as the GPU pendant to a CPU core, with the main difference that they consist of multiple programmable work elements — so-called single instruction, multiple data (SIMD) lanes. SIMD lanes of a CU execute the same instruction, but on different parts of the data. A CU typically consists of 32 to 64 SIMD lanes. The software counterpart to the CU is a so-called **wave** — a program which can be executed by the CU. Thus, a wave must match the size of a CU. A CU can execute different waves concurrently, while the wave provides the instructions and holds its program context. Each WGP provides a local cache (LDS), while each CU contains some registers. However, the LDS is not accessible across WGPs. Thus, the GPU provides a global L2-Cache [80, 79]. Some architectures extend the L2-Cache with additional cache hierarchies. One example is RDNA3's global data share (GDS) [79]. To receive data, the GPU is either able to communicate with

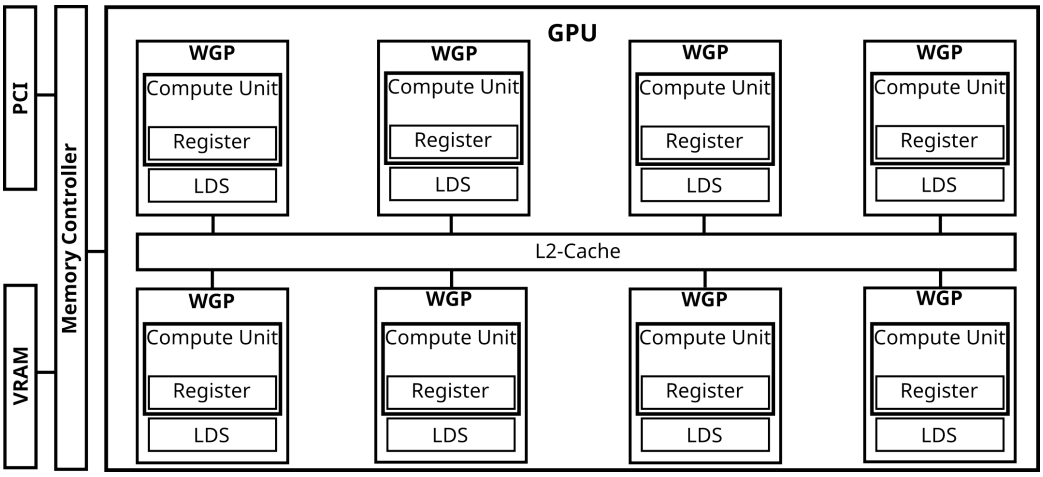


Figure 2.2: Simple GPU architecture, inspired by AMD's RDNA3 and NVIDIA's Ada Lovelace architectures. Sources: [80, 79].

DRAM via PCIe bus, or uses its own dedicated Video Random Access Memory (VRAM).

One important register is the Program Counter (PC). It indicates the wave's current instruction to execute. The PC is CU-local, which means that a wave can only issue one instruction at one time. This means that the SIMD lanes within a wave cannot be programmed independently, but must all execute the same instruction [81]. Different control flows between SIMD lanes within a wave lead to **divergent branches**. For resolving a divergent branch, the GPU executes one side of the branch via the condition-matching SIMD lanes, saves its results, rolls back, and executes the other side of the branch afterwards — sequentially. This means that divergent branches can slow down the CU by a factor of up to wavesize [82].

2.5.2 Programming Model

Based on the variety of specific GPU architectures, a standardized binary format with vendor-independent commands evolved: SPIR-V [83]. SPIR-V translates GPU programs (shaders) from a high-level language into a standardized form. We use GLSL [84] as high-level language, which is then translated via Google's glslc compiler into vendor-independent SPIR-V code [85].

As a GLSL shader is not bound to a specific GPU architecture, it is the programmer's responsibility to define the amount of SIMD lanes within a CU. This means that a shader can consist of larger "logical" CUs than the actual GPU's "physical" CUs are. Thus, GLSL introduces new terminologies to abstract from the actual GPU architecture: A logical CU is called a **workgroup**. This workgroup must be

translated onto physical CUs when executing the shader. Regarding Figure 2.2, a workgroup may consist of more SIMD lanes than the physical CU. Thus, the workgroup is split across different CUs of the same WGP. Depending on the shader and its memory dependencies across the workgroup, the LDS cache is more often involved to synchronize across different CUs. Reducing its involvement is thus one opportunity to increase a shader's performance; which is possible via GLSL's **subgroup** feature. Subgroups partition a workgroup logically into real CUs. Their size maps exactly to the size of a WGP's CU. In other words, a subgroup can be interpreted as a wave. This enables the programmer to explicitly use physical CUs with their local registers, while having the ability to synchronize larger workgroups over the LDS cache.

2.5.3 Vulkan

To actually run a GPU shader, a CPU-side configuration framework is required. This framework deals with the GPU driver in kernel space, and provides safe access of GPU functionalities from user space. Vulkan [14] is a graphics- and GPGPU-computing [78] API which does exactly that.

Over the last 30 years, OpenGL [86] was the leading standard in open source GPU programming. OpenGL follows a state machine architecture, in which it tries to generalize and hide computational complexity of different hardware. Although this was suitable for early graphics applications and overhead from several translation layers was negligible, GPUs evolved in terms of performance and different feature sets. Therefore, Vulkan, as a more lightweight API, was invented.

Within Vulkan, the developer has more control over the lower level details. In contrast to OpenGL, the driver does not make any assumptions or validations by default. It assumes that the application developer is providing every information explicitly. This allows the developer to configure, for example, memory accesses on a much lower level [87]. Additionally, Vulkan does not follow the state machine pattern of OpenGL: Instead of calling one library function at a time which changes the internal application state, Vulkan bundles different instructions together and sends one bundle at a time. Although Vulkan is capable of both graphics and compute shader execution, we focus only on the compute shader functionality of Vulkan, as the graphics part is not relevant for this thesis.

Overview

Vulkan uses various structures to employ a GPU. Before an application can actually issue a GPU command, it must discover all connected PCIe devices capable of Vulkan functionality. After choosing the **physical device**, the application must

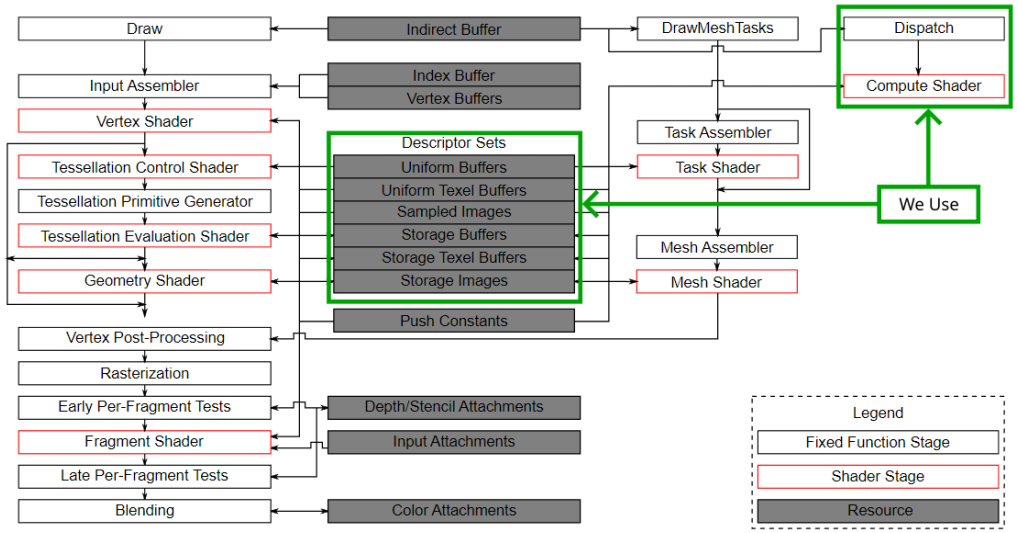


Figure 2.3: Overview of the Vulkan pipeline. The leftmost side represents the graphics part, while the rightmost side is the compute part. Source: [91], all green objects are modifications.

create a **logical device** from it. The logical device is an actual instance of the physical device which can consume the application's Vulkan commands. After creating a logical device, the application can dispatch **command buffers** through a **queue**. A queue in Vulkan is logically a connection path from the CPU to the GPU, which is capable of processing different types of commands [88, 89].

A **command buffer** records commands which shall be executed by a logical device [90]. For simplicity, we define a **compute command** as a combination of the following three Vulkan commands: Pipeline definition, Descriptor Set Layout definition, and dispatch.

The **pipeline definition** of a compute command is simpler than its graphics counterpart. While the latter involves several processing stages, a compute command uses only one stage of the so-called **Vulkan pipeline**. The Vulkan pipeline is outlined in Figure 2.3. It is important to mention that our terminology of the word "pipeline" is not equivalent with Vulkan's definition: We define a GPU4FS pipeline as the parallel execution of multiple software stages (shaders), where each GPU4FS command goes through those software stages according to a specific order. This definition is similar to a RISC pipeline [92]. Vulkan on the other hand defines a pipeline as a configured variant of the Vulkan pipeline, which has several stages either enabled or disabled. We only need the "compute shader stage" within GPU4FS, which, according to Khronos, "consist of a single static compute shader and the pipeline layout" [93]. Consequently, Vulkan does not allow multiple different, interleaved executions of the "compute shader stage" within its definition

of a pipeline.

The pipeline definition of a compute command only defines the actual shader to execute, its "main" function and the required pipeline layout with according shader resources [93, 94].

Shader resources are VRAM- or DRAM-backed allocations which can be used throughout a shader execution. Vulkan provides several types of shader resources, from whose the **buffer** is relevant within this thesis. A shader resource is represented by a descriptor on the CPU side [95].

After binding a pipeline definition, the application needs to back the defined shader resources with actual memory allocations. The **Descriptor Set Layout** defines the shader resources of a compute command, which are bound to the previously set pipeline layout. [95].

The last step of a compute command is the **dispatch**. As mentioned previously, a command buffer is executed via a specific queue, belonging to a specific logical device. The dispatch defines the actual amount of used workgroups which shall execute the previously set pipeline layout with the defined descriptor set layout. Each workgroup spawns its own instance of the shader, which leads to additional independently started invocations [96].

A command buffer can consist of multiple compute commands. Those can either be dependent or independent of each other. However, dependent compute commands must be synchronized appropriately. Compute commands can either be execution- or memory-dependent. We focus on the latter dependency, as this is the relevant case within this thesis. Vulkan provides many ways of synchronizing between and even within compute commands. While the former involves the CPU side and thus the Vulkan framework, the latter relies on the GLSL specification [97].

Memory Coherency and Synchronization between Commands

Vulkan provides several mechanisms for resolving dependencies between commands; including fences, semaphores, events, and pipeline barriers [97]. The two relevant synchronization mechanisms in our case are fences and pipeline barriers, which we outline in the following.

A **fence** signals the host a completion of a command buffer. Before passing a command buffer to a queue, the Vulkan fence is passed as an additional parameter. Afterwards, the CPU can execute a blocking wait call on the fence to await the command buffer's termination. After passing the fence, the CPU can be sure that all operations from the command buffer where executed and are visible [97].

A **pipeline barrier** on the other hand allows fine-granular synchronization within a command buffer. Pipeline barriers can either be inserted between or within pipeline stages of the Vulkan pipeline. Thus, they can be used to declare a dependency between different compute commands within a command buffer [97].

Memory Coherency and Synchronization within a Command

Synchronization between commands is not the only relevant case. As mentioned in Section 2.5.3, a dispatch which specifies multiple workgroups spawns different shader instances. Workgroups compute independently of each other, which makes it necessary to be able to synchronize them appropriately. A dispatch-wide synchronization differentiates two cases [98]:

1. Synchronization within a workgroup
2. Synchronization between workgroups

GLSL provides `memoryBarrier()` primitives, which ensure the first of both cases. These guarantee the completion and relative ordering of memory accesses within a workgroup. However, they **do not** guarantee any visibility between workgroups, as the workgroup's execution order is unspecified. In other words, those barriers are useful to synchronize workgroup-local SIMD lanes [84, 98].

The second case tends to be an uncommon case within a compute shader. At least, GLSL does not provide any primitives which would allow a synchronization between workgroups directly. Thus, we developed a workaround for ensuring visibility between workgroups, which is further examined in Chapter 5. For now, we introduce the relevant concepts for our workaround.

One important precondition for a workgroup-wide synchronization is to declare a buffer **coherent**. If the compiler encounters such an annotated buffer, it assumes that the content of this buffer could be changed by dependent shader invocations and generates appropriate SPIR-V code [99]. The **volatile** classifier implies an even stronger condition, which includes that a buffer's content could be changed at any time from an external source [99]. However, the GLSL specification is not precise on their distinction. Further research led us came to the conclusion that coherently mapped buffers imply coherency between dependent shader invocations, whereas dependent means in-order in the Vulkan pipeline or between workgroups. Volatile buffers on the other hand shall be used for coherency between independent shader invocations, which includes different compute commands [100]. Nevertheless, both memory accesses are slower than non-coherent respectively non-volatile accesses [84]. However, the GLSL language specification notes that those qualifiers are just a precondition for valid visibility; they do not ensure the relative ordering of memory accesses. Thus, ensuring synchronization between workgroups involves some kind of ordering.

Furthermore, another synchronization-related concept exists: **Atomics**. GLSL provides several operations which are guaranteed to modify a buffer's content atomically. All atomic operations work on GLSL types only. This means that no user-defined structs can be modified atomically. However, atomic functions are a fundamental concept which prevent race conditions in certain scenarios [84].

Chapter 3

Related Work

This chapter introduces related work to our topic, which inspired our design. We introduce the most essential aspects of GPU4FS initially, which is the foundation of this work. We then give a short overview of well-known file systems and explain their solutions to our outlined problems.

3.1 GPU4FS

In the following, we will introduce the current state of GPU4FS. GPU4FS is an inode-based file system which is mainly designed for organizing Intel Optane DIMMs [10]. The biggest difference to a common file system is its implementation on a GPU rather than a CPU. GPU4FS uses the Vulkan [14] framework to fulfill file system tasks on the GPU. The main goal of GPU4FS is to reduce stress on the CPU by moving computational-intensive work from the CPU to the GPU [10].

3.1.1 Essentials

GPU4FS was designed as a file system accelerator for modern storage, particularly for Intel Optane. However, as GPU4FS uses standard Linux functionality for mapping the Optane DIMM onto the GPU's VRAM [10], it can be modified to address NVMe drives. This generalizability is another motivation for our integration of additional file system features.

As mentioned in Section 2.5, GPUs are primarily configured in user space, which explains the decision of designing a user space file system. To support commands that need OS support[1], GPU4FS introduces a "trusted component" application with privileged permissions.

[1] e.g., mmap() [101] to map memory into a process' address space

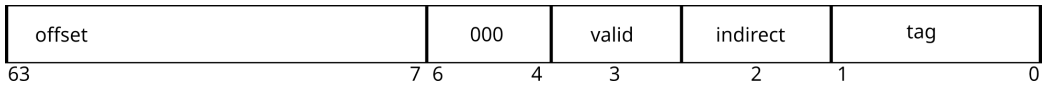


Figure 3.1: Block pointer in GPU4FS. The block pointer is 64 bit in size. It provides 57 bits for the address offset, three unused bits, one bit indicating its validity, one bit indicating an indirect block reference, and two bits indicating the page type of the referenced block. Own drawing, inspired by: [10]

Communication between a CPU process and the GPU-side of the file system is handled through a GPU4FS command buffer, which has nothing in common with a Vulkan command buffer from Section 2.5.3: Each user space process holds its own set of command buffers, which are mapped GPU-visible within a shared memory region. A user space process triggers a file system action by creating and inserting a command descriptor into the command buffer. The GPU then detects a new entry within the buffer, executes the command and signals completion back to the requesting process. [10].

GPU4FS stores files in clusters on the NVM drive, so-called **pages**. It provides three different page sizes, which are congruent to x86-64 MMU page sizes[2] [102]. Each page stores one or more **blocks**, which contain either file metadata or real file data. When storing a file, it is thus split up to fit into one or more blocks, which are located on one or more pages. GPU4FS uses block pointers to address these blocks [10].

3.1.2 Block Pointer Design

The smallest element in GPU4FS is a block pointer. As mentioned previously, those are used to identify a block uniquely while holding some essential metadata. This metadata determines the type of the referenced block. The block pointer design is depicted in Figure 3.1. A file in GPU4FS is made up of several blocks, which can be organized in an indirection hierarchy [10].

3.1.3 Block Design

GPU4FS provides three block flavors: Inode, indirect block, and data block.

The file system uses an inode to reference the actual file. The inode size is either 128 bytes or 256 bytes; depending on the configuration in the superblock. As its current demonstrator implements inodes with a size of 128 bytes, we assume the same inode size throughout our proposals. Such an inode reserves space for

[2] 4 KiB, 2 MiB or 1 GiB

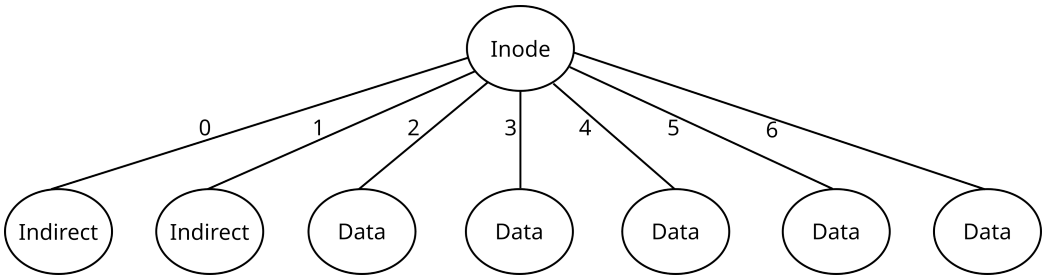


Figure 3.2: GPU4FS's file pointer structure. The figure represents the pointer structure for a file which stores two indirect blocks and five data blocks in the inode. While the data blocks reference actual file content, the indirect blocks are filled with further block pointers.

seven block pointers, which can either point to an actual data block or to an indirect block. An indirect block references further block pointers [10]. This indirection hierarchy allows unlimited file sizes in theory. The amount of block pointers within an indirect block is determined by its page size. An indirect block can hold one of the following options:

- 2^4 block pointers if its inode sized (128 byte)
- 2^9 block pointers if its small page sized (4 KiB)
- 2^{18} block pointers if its large page sized (2 MiB)
- 2^{27} block pointers if its huge page sized (1 GiB)

The current implementation of GPU4FS uses only inode-sized pages to store indirect blocks [24]. Figure 3.2 depicts an exemplary block pointer structure, corresponding to an actual file in the file system.

3.2 EXT4

The fourth extended file system (EXT4) is the fourth evolution of the family of EXT file systems. EXT4 evolved from EXT3 [103] and exists in mainline Linux since version 2.6.19 [23].

EXT4 is a kernel space file system which uses 48-bit-long block addresses. Within EXT4, files are stored in extents. Especially HDDs benefit from extents, as they reduce I/O latency by avoiding random reads. Furthermore, EXT4 uses journaling for its crash consistency [23, 104].

EXT4 was a big inspiration for GPU4FS's block design [10, 23]. GPU4FS's indirect block approach is inspired by both EXT3 and EXT4: While the indirect

block approach itself can be seen as inspired by EXT3 [103], the variable page size is somewhat related to EXT4's extends [23]. EXT4 uses CRC checksums to secure its metadata and journal, but not its data blocks [23, 105]. It also does not support deduplication directly [23]. However, there exist approaches on patching deduplication into the file system externally [106].

3.3 Btrfs

The B-Tree file system (Btrfs) is a kernel space file system which uses fundamentally different on-disk structures than GPU4FS [10] and EXT4 [23]. A Btrfs formatted file system uses COW B^+-Trees for storing the actual data. Atomic block pointer updates in combination with COW ensure crash consistency [8].

Btrfs manages a file system within a forest of trees. This forest contains four primary tree structures:

- **Subvolumes** store actual user files and directories
- **Extent allocation trees** provide extent-flavored free storage
- **Checksum tree** which holds one checksum item per allocated extent
- **Chunk and device trees** which allow RAID functionality

A leaf node within Btrfs stores three relevant file system data structures: A block header, an array of items, and an array of data elements. Those leaves represent e.g., a file or a directory entry [8].

A leaf's block header stores metadata like CRC sums, flags, generation number, and so on. An item within a Btrfs leaf node is fixed-size and consists of three fields: key, offset, and size. In addition, each item references a corresponding data element in the aforementioned data array. The key determines the item's purpose within the file. One possible key value is `inode`. The corresponding data element holds the actual data of the inode. Another important type is the `extent` key. An item of type `extent` allocates an extent from the **extent allocation tree** and stores its reference within an extent `[item, data]` pair. Btrfs groups multiple blocks together in an extent. A leaf can hold multiple extents, as on-disk fragmentation does not always ensure large enough extents. Each leaf allocates one checksum item per allocated extent, which references an entry in the **checksum tree**.

Btrfs does not support deduplication: Its authors state that "Due to the memory requirements, it might be a feature only fit for high-end servers" [8]. However, tools that extend Btrfs with deduplication functionality exist [107]. According to the Btrfs documentation, those deduplication tools utilize a byte to byte comparison rather than the existing checksums [107].

Bees

Bees is a daemon which runs continuously to identify duplicated data. It performs a full file system scan and stores all found blocks in a fixed-size hash table, with an LRU strategy. During this scan, fully duplicate extents are immediately detected and removed. Partly duplicated extents can be detected after the full file system scan: Bees then tries to write the unique parts of both extents, and references the duplicate blocks from one of the already existing extents. If Bees detects a duplicate block within an extent, it additionally examines the nearby blocks in the files which contain the matched block [108].

Duperemove

Duperemove is a tool which detects duplicated extents within different files. It does not run as a daemon but rather as an application which needs to be triggered manually. Duperemove takes a list of files as input and finds duplicated extents within all of those files. Fulfilling this task involves three steps: Duperemove initially discovers all submitted files, hashes their content in $blocksize$ length chunks, and stores the hashes in a database. This database serves as input for further deduplication invocations, allowing Duperemove to detect files which did not change. The second step uses the built database to create a list of duplicate extents. Step three triggers the final deduplication step utilizing the results from step two. Duperemove does not support partly duplicated extents [109].

3.4 ZFS

ZFS is a COW kernel space file system, mainly used for server storage purposes. Its main focus lies on data integrity, simple administration, and support for immense capacity. ZFS follows an indirect block pointer approach for referencing large files. The overall file system structure is maintained in a large tree, with the data blocks as its leaves. To understand its structure and complexity, we briefly describe the involved ZFS components when issuing a POSIX system call in the following:

The involved ZFS components are depicted in Figure 3.3. Initially, ZFS receives the system call through its ZFS POSIX layer. This layer translates the POSIX call into a ZFS-valid structure, called the Object Transaction Interface. The transferred object is then passed to the Data Management Unit (DMU). The DMU translates the received object further into a data virtual address. This virtual address is then passed to the Storage Pool Allocator (SPA). The SPA uses this virtual address for allocating blocks on the physical device. As there could be many physical devices present within a ZFS context, the SPA is responsible for choosing the correct device for allocation. ZFS uses so-called vDevs as storage devices.

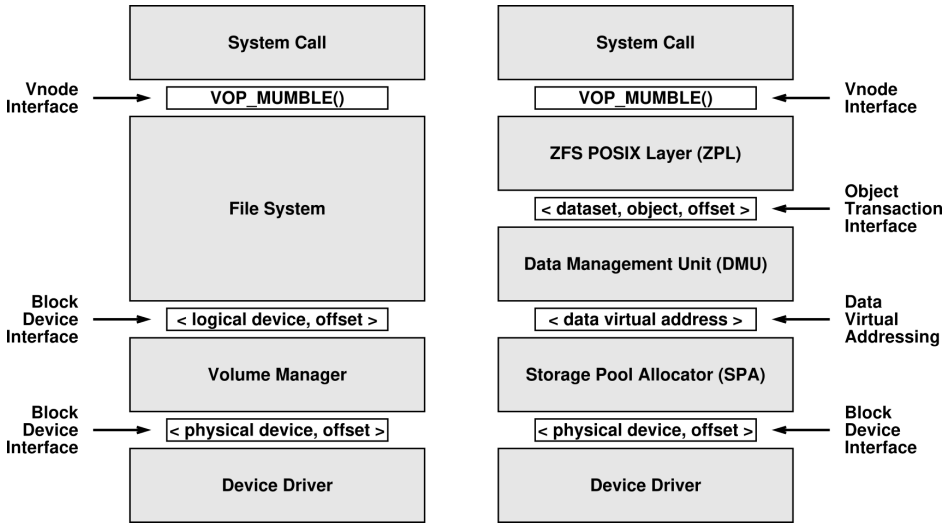


Figure 3.3: ZFS layer overview. The left side shows a traditional file system block diagram, while the right side depicts the ZFS block diagram. Source: [9].

A vDev is a virtual device which bundles different capabilities. For example, a vDev is created by the ZFS administrator, who specifies capabilities like mirroring, disk concatenation, and further device-related concepts. Finally, the vDev is backed with real devices, which persist data physically. The SPA thus allocates blocks from a vDev, which executes its configured abilities [9].

3.4.1 Checksumming

Defining data integrity as main goal makes error detection and correction capabilities one of the core components of ZFS. The file system checksums every allocated block in the file system tree. ZFS provides a block pointer which is 128 bytes in size, with reserved space for a 256-bit checksum [110]. Each ZFS block stores the block pointers of its children and consequently their checksums. The only block which stores its own checksum is the root of the file system tree — the so-called **überblock**. Any create or update operation on a file involves multiple checksum calculations, propagating in a bottom-up fashion up to the überblock. It is important to mention that each update operation involves rewriting all involved block pointers, coming from ZFS's COW compliance. After writing the actual data to disk, ZFS calculates the newly allocated block's checksums. It then writes those checksums into freshly allocated parent indirect blocks, which must further be referenced by fresh indirect blocks. This procedure propagates recursively up

to the überblock. Writing the überblock atomically to complete a file operation ensures crash consistency [9].

3.4.2 Deduplication

ZFS implements an inline deduplication. Therefore, each file write considers additional entries from a so-called **deduplication table (DDT)** [111]. ZFS deduplicates fixed-size ZFS blocks, whose size is configurable to up to 1 MiB at pool creation [112]. However, the community discusses if CDC is integrable into ZFS [113].

In terms of DDT entry size, there are various different claims: Matt Ahrens claims that the DDT consists of entries which are either 192 bytes (in-memory) or 168 bytes (on-disk) in size [114]. His claims are available on the official OpenZFS homepage. However, the recent OpenZFS documentation outlines that the in-memory size of a DDT entry is slightly more than 320 bytes nowadays, but does not mention any different on-disk sizes [112]. In both cases, a DDT entry is identified by a ddt_key. This ddt_key uses a 256-bit cryptographical checksum. ZFS allows SHA256, SHA512, Skein, and BLAKE3 as checksums for deduplication [115].

Although the DDT is structured according to the extendible hashing technique [76] on-disk, the DDT uses an AVL tree for organizing its cached in-memory entries [116, 114]. An AVL tree is a balanced binary tree [117].

Each DDT entry represents a previously written block on disk. Every time a file is written to disk, ZFS invokes the deduplication step: It compares all checksums of this file's blocks with all saved DDT entry keys. If any DDT entry has the same key, respectively same checksum, the corresponding file block is either directly considered a duplicate (ZFS property `on`) or further validated through a byte-wise comparison (ZFS property `validate`) [118]. Based on the result of the deduplication step, the block is either written or discarded. The former case leads to the insertion of a new DDT entry into the DDT table. In the latter case, ZFS writes a reference to the block behind the matching DDT entry into the file's block pointer structure [116].

As every file write invokes a DDT table walk, ZFS suggest holding the entire table in DRAM [119]. However, given the size of a DDT entry, the authors of a widely known storage software[3] suggest 1 to 3 GB of RAM per 1 TB of data. Additionally, they mention that ZFS deduplication is a CPU-intensive task and reduces the overall throughput of the ZFS partition [111].

[3] TrueNas [120]

3.5 Additional File Systems

Next to the three aforementioned file systems, there exist a variety of other file systems, each having different capabilities. This section gives an overview of further file systems with relevance for GPU4FS.

3.5.1 NOVA

The NOn-Voltaile memory Accelerated (NOVA) is a kernel-space file system which was designed for managing non-volatile memory. It is a log-structured file system, which utilizes parallelism by giving each inode its own log. Its logs are implemented as a linked list, as the random access performance of NVM is higher than of any other persistent storage medium. Thus, a full linked list is extended via a next pointer at the end. Additionally, it utilizes radix trees in DRAM to perform search operations quickly and to reduce on-disk data structures. However, NOVA neither proposes design approaches for checksumming nor for deduplication [121].

3.5.2 User Space File Systems

As mentioned in Section 2.2.1, the concept of user space file systems became popular in recent years. Their functionality is mainly implemented in user space; they only involve the kernel if it is unavoidable. Aerie [26], Strata [27], and SplitFS [28] are some examples of user space file systems. They provide a user space library which applications link against. This library provides functionality which maps a file read only into a requesting process's address space.

Strata performs a write by adding the write request to an update log. This update log is then processed by Strata's kernel component [27]. Aerie on the other hand allocates space for a received write call via its kernel component, and lets the user space library write the content directly [26]. Conversely, SplitFS utilizes a staging file in user space, which is updated by the write call. The `fsync()` call flushes this staging file onto drive [28].

The user space library either intercepts system calls [27, 28] or provides a library which provides corresponding calls [26]. Commonly, their user space library ensures synchronized and coordinated access, and goes into the kernel only if its necessary [26, 27, 28]. Whereas Aerie and Strata introduce a trusted component for kernel interaction [26, 27], SplitFS uses the ext4 DAX module [28].

As with NOVA, those file systems do not include checksum or deduplication capabilities. Thus, they are relevant for the overall thematic, but not primarily for this thesis.

Chapter 4

Design

Whilst GPU4FS [10] implements basic write and copy operations, it lacks advanced file system functionalities. This chapter details a design of checksumming and deduplication for GPU4FS. A working checksum implementation is a pre-condition for implementing deduplication. Therefore, the checksum design process is detailed prior to the deduplication design. After proposing their architecture, a discussion about alternative design approaches justifies our decisions for both checksumming and deduplication.

4.1 Checksumming

This section proposes a checksum approach for GPU4FS. Furthermore, we discuss alternative designs. Our approach covers an optimized design for a GPU-based block read and write. We describe our checksum approach only for an initial file write, as GPU4FS does not currently support file updates. However, Section 5.1.3 gives theoretical advice on how to update checksums appropriately.

A satisfying checksum approach within GPU4FS is given if it fulfills the following properties:

- Parallelization: Parallel computation in terms of SIMT
- Overhead: Low checksum overhead on disk
- Extensibility: Independent of current page sizes and chosen checksum algorithm

Our implementation uses BLAKE3 [49] as checksum algorithm. Selecting the checksum algorithm for a GPU4FS-formatted partition is possible within the superblock: It provides appropriate configuration fields [10]. It is important to mention that our proposed file system design is independent of the chosen algorithm.

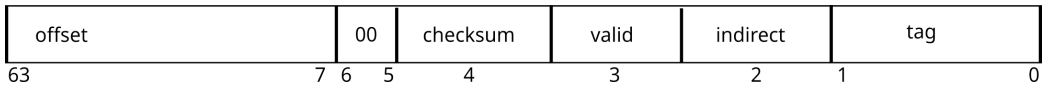


Figure 4.1: Modified block pointer for GPU4FS which supports checksumming. The modified block pointer consists of two instead of three unused bits. One bit is used to identify a checksum block.

Our design is capable of different checksum algorithms, preferably algorithms which support 16 or 32 byte checksums to reduce internal fragmentation.

4.1.1 Block Pointer Design

As GPU4FS utilizes block pointers to reference data blocks [10], we decided to introduce a new checksum block into the file system. A checksum block is identified by a new bit flag inside the block pointer, as seen in Figure 3.1. We use one of the currently unused bits to implement this bit flag. A block pointer is defined as a checksum block pointer if its checksum bit flag is set to "1". The extra bit flag makes it easier to debug the checksumming approach, and eases the read process of a file. However, it could be omitted in most cases, with one exception. We outline this exception in Section 4.1.2, after explaining the overall design.

4.1.2 Block Design

Checksum blocks are either stored on 128 byte, 256 byte, 4 KiB, 2 MiB or 1 GiB pages. For simplicity, we differentiate between 128 byte- and 256 byte pages by calling them inode- or mini page sized, respectively. As an inode provides 7 block pointers, their corresponding checksums fit onto a mini page.

Although we solely use mini pages to provide space for checksum blocks in our implementation, a checksum block is not bound to a mini page. We chose this page size as GPU4FS's current demonstrator only supports inode-sized indirect pages. However, GPU4FS's theoretical design is also capable of 4 KiB, 2 MiB or 1 GiB indirect pages [10]. In case of such an allocation, our checksum blocks also fit on those larger page sizes to reduce overhead. We outline the necessary changes in Section 4.1.2, after explaining our overall checksumming design for mini pages.

As this thesis implements the BLAKE3 checksum algorithm, one mini-page-sized checksum block can hold up to 8 checksums. Figure 4.2 depicts such a checksum block. The "self sum" field of a checksum block represents its own checksum, which is calculated after the block was fully filled. To ensure a correct self sum calculation, its 32-byte-long position within the checksum block needs to be zeroed out prior to the self sum calculation.

To integrate the checksum structure into the actual file system, we modify GPU4FS's current file pointer structure. Based on the file pointer structure outlined in Section 3.1.3, we modify the block types "inode" and "indirect block": Both data structures use formerly free block pointers to point to checksum blocks. Given GPU4FS's recursive indirect block pointer implementation [10], this design does not reduce the maximum file size in theory — although we reduce the total amount of addressable data blocks, which is inevitable.

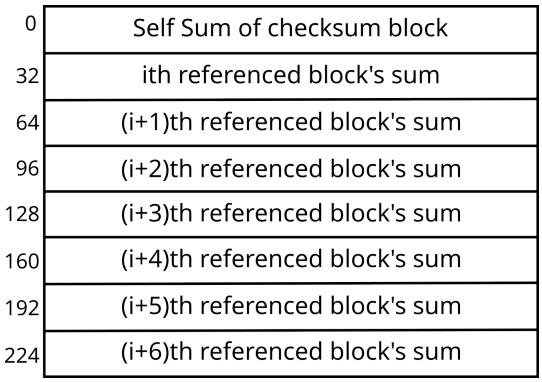


Figure 4.2: Internal structure of a checksum block, stored on a mini page. The checksum block always follows the depicted structure: The first checksum is the block's self sum, the remaining checksums are calculated over foreign blocks.

Inode Extension

Starting with the inode block, there are eight checksums which need to be stored: Seven block checksums induced by the seven block pointers inside the inode, and one checksum coming from the inode itself.

Therefore, the checksums coming from an inode block can be stored within one mini-page-sized checksum block. This checksum block is referenced through the first of the seven inode block pointers and stores its self sum, the inode's sum, and the six sums of the six remaining block pointers. Figure 4.3 depicts the modification of an inode's block pointers. The remaining six block pointers can either point to real data blocks ("D0" to "D4" in Figure 4.4) or indirect blocks ("I0" in Figure 4.4). To reduce potential internal fragmentation, an inode which uses only two of its available block pointers is also able to allocate an inode-sized checksum block.

According to Section 3.1.3, an indirect block stores further block pointers. To checksum those further block pointers, the required indirect block extension is outlined in the following.

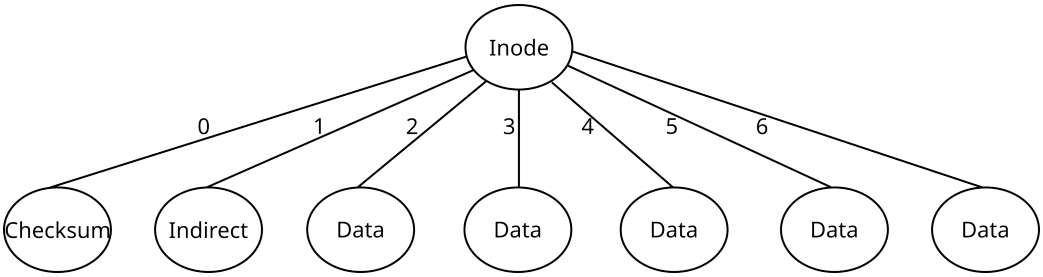


Figure 4.3: Modified file pointer structure of an GPU4FS inode. In contrast to GPU4FS's original file pointer structure outlined in Figure 3.2, the first block pointer now points to a checksum block. Per page size construction, the checksum block provides space for all 8 checksums which are required within an inode. Figure 4.4 details this figure.

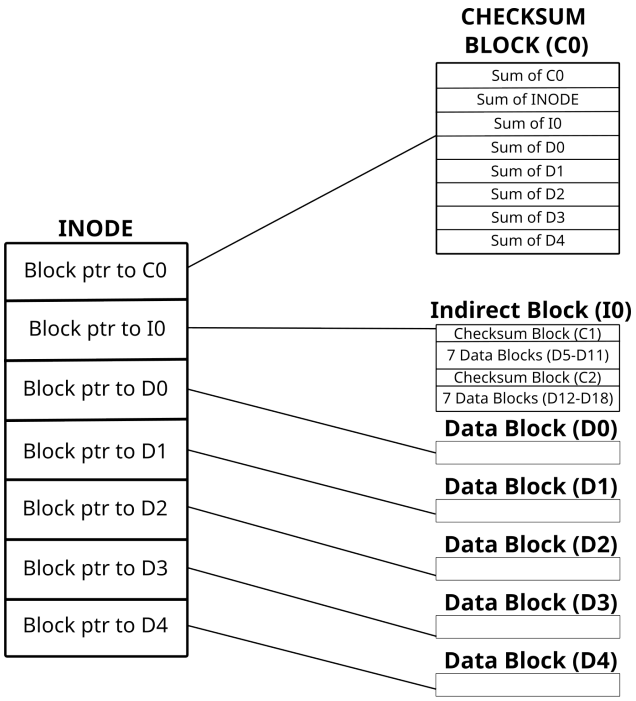


Figure 4.4: Modified file pointer structure of an GPU4FS inode from Figure 4.3 in detail. Notice that this is an exemplary inode; an inode can hold between zero and six indirect block pointers. "I0" stores references to data blocks in the depicted example. However, it could also reference a mixture of indirect block pointers and data blocks, or indirect block pointers only. Our design supports both scenarios.

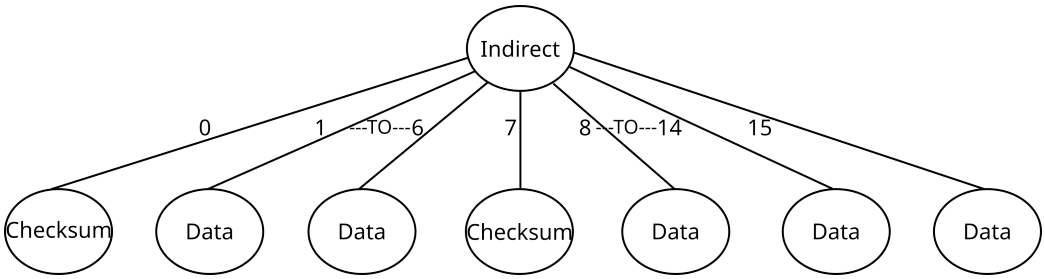


Figure 4.5: Modified pointer structure of an GPU4FS indirect block, which is inode-sized. Inode-sized indirect blocks hold up to 16 further block pointers. As one checksum block is 256 bytes in size and can thus hold eight 32-byte checksums, it is necessary to reference two checksum block pointers to store all 16 checksums. Figure 4.6 details this figure.

Indirect Block Extension

Inode-sized indirect blocks can store up to 16 block pointers, which are either pointing to real data blocks or to more indirect blocks. Thus, it is necessary to store up to 16 checksums for one indirect block. With BLAKE3 as checksum algorithm, one indirect block needs 512 bytes of checksum storage. Given the size of a mini page, it is inevitable to store more than one checksum block in the indirect block. Figure 4.5 proposes the new indirect block structure, which is detailed by Figure 4.6: Pointer zero and pointer seven are both used to point to checksum blocks, while all other 14 block pointers follow the original block pointer semantic. Thus, a checksum block from an inode-sized indirect block stores its own sum in position zero, and the checksums of the next contiguously referenced blocks in the remaining seven positions. As with inodes, the indirect block is also able to allocate inode-sized checksum blocks, or a combination of both.

As an indirect block potentially stores more than one checksum block, the "self sum" field of a checksum block can be used differently; instead of storing its own self sum, it might store the self sum of a foreign checksum block, which increases the integrity. In other words: An indirect block which uses more than one checksum block can distribute their self sums across its different checksum blocks. Such a scenario is exemplary outlined in Figure 4.6, and formalized in the following: Let d_0 be a data block and c_0 be a checksum block which stores the checksum of d_0. If c_0 now has a corrupted bit in both its self sum and the stored sum of d_0, there is no way to detect if d_0 did suffer from corruption or the checksum block c_0 stores a flipped checksum. However, if another checksum block c_1 stores the sum of c_0, this block can be used to validate if c_0 is intact or corrupted. With this extension, three instead of two dependent blocks need to be corrupted before a corruption cannot be detected unambiguously.

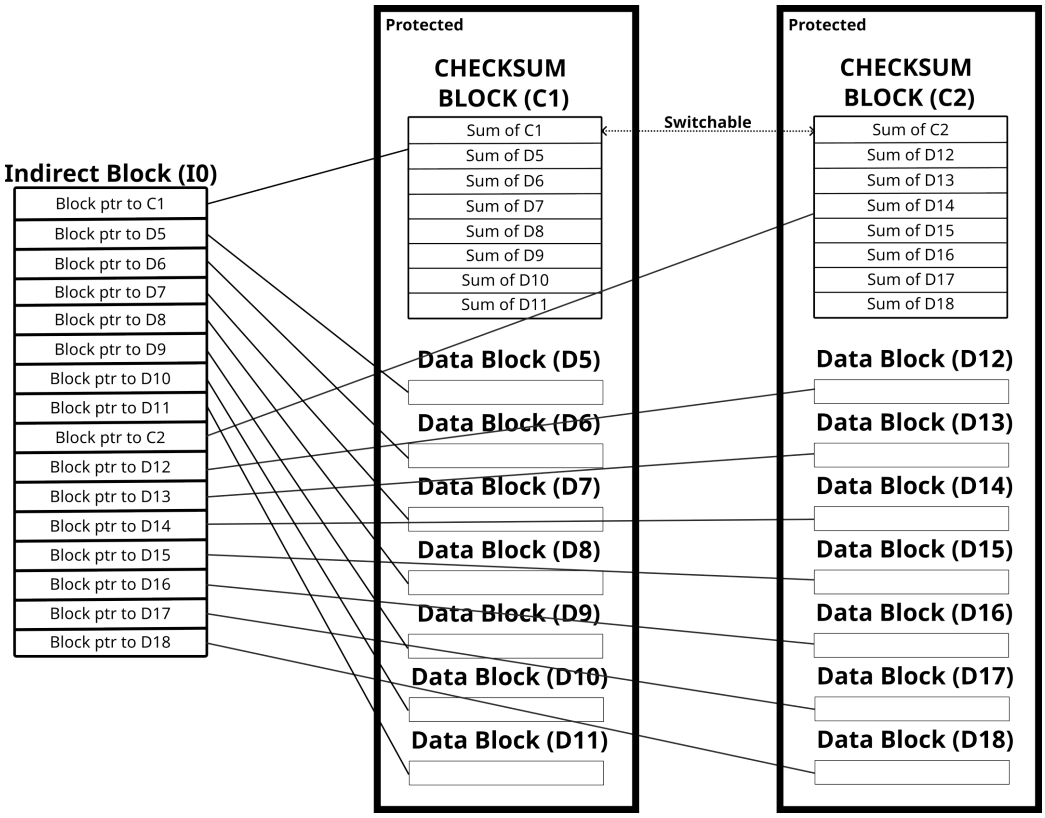


Figure 4.6: Detailed file pointer structure of an GPU4FS indirect block from Figure 4.5. This figure extends Figure 4.4. “I0” references 14 data blocks “D5 to D18”, whose checksums are stored in C1 and C2, respectively. This is just an example; “I0” could also reference further indirect blocks, which would then follow the depicted design themselves.

Large Indirect Blocks

As stated previously, GPU4FS is able to allocate indirect blocks which are larger than an inode-sized page. Thus, we propose two different approaches on extending the checksumming approach for different indirect block sizes in the following. While approach static uses only mini pages for storing checksum blocks, approach dynamic allows different page sizes for a checksum block.

Within **approach static**, every position in the indirect block which is a multiple of eight references a checksum block. This implementation allows an extension to larger indirect block sizes directly, by following the implied mathematics. Approach static leads to the following amount of mini-page-sized checksum blocks, with respect to the GPU4FS page sizes:

- Inode-sized indirect blocks: 2^1 checksum blocks
- Small-page-sized indirect blocks: 2^6 checksum blocks
- Large-page-sized indirect blocks: 2^{15} checksum blocks
- Huge-page-sized indirect blocks: 2^{24} checksum blocks

Approach dynamic on the other hand allows checksum blocks on page sizes different from mini pages. This means that e.g., an indirect block which is small-page-sized could utilize four small-page-sized checksum blocks instead of 2^6 mini-page-sized checksum blocks. The degree of an indirect block's occupancy makes it feasible to dynamically use a combination of small-page-sized and mini-page-sized checksum blocks in combination. This principle can be applied to the other GPU4FS page sizes.

Approach dynamic must use a specific target metric to determine the amount of checksum blocks. Depending on the selected metric, the previously modified block pointer with its checksum flag (regarding Section 4.1.1) eases its implementation, especially when using a mix of different metrics.

One feasible metric is to target the least possible overhead. For determining which checksum block size combination provides the least overhead for a given file, the approach solely needs knowledge about the file size. As a checksum block reduces an indirect block's free block pointers, the file could need more or less indirect blocks, depending on the different checksum block sizes. To be able to allocate enough block pointers, each upper indirect block must be allocated in advance of its potential child blocks. The algorithm then allocates checksum blocks in a greedy fashion: Starting with huge-page-sized checksum blocks, the algorithm allocates as many huge-page-sized checksum blocks as it can fully occupy. Then, it proceeds with large-page-sized checksum blocks, then small-page-sized checksum blocks, and finally places the remaining blocks onto one or more mini-page or inode-sized checksum blocks.

4.1.3 Discussion

The outlined checksum approach was built with our three design goals in mind: Parallelization, overhead and extensibility. Additionally, we modified the already existing file system structure as little as possible. In the following, we will discuss differences to common file system implementations and outline an alternative checksum approach for GPU4FS.

Fundamental Differences to Related Work

ZFS uses a block pointer which is 128 bytes in size [110]. However, GPU4FS's block pointer is only 8 bytes in size [10]. Therefore, ZFS's implementation — storing checksums inside the block pointers — is not easily adaptable to GPU4FS. Increasing the block pointer size of GPU4FS would not just break current file system usage; it would also affect the current page sizes and lead to more internal fragmentation. An inode would consequently not consume 128 bytes, its storage requirements would increase to at least 200 bytes when storing only one block pointer inside the inode. 200 is not a multiple of two, which leads to changes in the overall address format.

Compared to ZFS, the overhead of our checksumming approach is higher: ZFS has a checksum overhead of 32 bytes per block pointer. Our checksum approach comes with the same overhead when considering full or half-full indirect blocks. In those cases, the allocated mini-page-sized checksum blocks are fully filled. All other cases, however, introduce internal fragmentation. This is unavoidable as GPU4FS needs to allocate pre-defined page sizes which are, per block pointer construction, at least 128 bytes in size. The largest internal fragmentation within our checksum approach is given if an indirect block is only filled with one block pointer. In that case, the remaining 14 block pointers are unused. Although we need to allocate one mini page only to checksum the indirect block's first eight block pointers, the internal fragmentation expands to $256 - 32 - 32 = 192$ bytes. 192 bytes is the largest internal fragmentation inside a checksum block when using inode-sized indirect blocks and mini-page-sized checksum blocks. When allowing inode-sized checksum blocks, the maximum possible internal fragmentation reduces to $128 - 32 - 32 = 64$ bytes.

Btrfs utilizes a fundamental different approach on storing data checksums. Within Btrfs, an additional data structure is used to store checksum items only. This so-called checksum tree is stored independently of the actual file's data [8]. Additionally, Btrfs checksums all tree blocks using a CRC32 checksum [8, 122].

As pointed out in Section 3.3, Btrfs uses so-called extents to store and organize file data. Those extents can be variably sized — which makes a static checksum approach as in our case unattractive. A separate tree structure is technically possible

within GPU4FS by selecting GPU4FS's block pointer as unique key for the tree structure. However, it would introduce more complexity into the file system: A separate tree requires multiple SIMD lanes to walk the tree structure in parallel to the real file pointer structure, which makes load balancing a file read difficult. Additionally, a file write would require to walk the checksum tree for allocating fresh checksum blocks, introducing additional $\mathcal{O}(\log n)$ operations to the already expensive checksum calculation.

EXT4, which design was an inspiration for GPU4FS [10], provides checksumming only for block groups and its journal [105, 23]. In its current state, EXT4 plans to integrate checksumming for the extent tail, the allocation bitmaps, the inodes and potentially the directories. Given its age and further development in more recent file systems, the reasonability of those modifications is questionable. EXT4 uses a CRC16 checksum for block groups, and a CRC32 checksum for its journal [105]. EXT4 can thus not detect bit flips on individual file blocks, which makes its implementation unattractive for GPU4FS.

Alternative Checksum Blocks — The Linked-List Approach

To reduce the amount of checksum block pointers within an indirect block, an alternative implementation is also feasible. Such an implementation could utilize a special inode-sized block, as depicted in Figure 4.7. This "special checksum block" is referenced by the first block pointer inside an inode or an indirect block. The alternative approach requires a distinction between the special checksum block and the actual checksum block. An actual checksum block stores checksums only and is thus similar to our implemented checksum block from Section 4.1.2. As the actual checksum block provides only space for a fixed amount of checksums, we need the ability to reference multiple actual checksum blocks. While our approach from Section 4.1.2 borrows multiple block pointers to reference multiple checksum blocks, the alternative approach uses the special checksum block: The "Next Block" block pointer of a special checksum block can reference another special checksum block, which allows chaining multiple contiguous special checksum blocks together. Therefore, it is possible to utilize various pages sizes for the actual checksum block. Holding an indirect block's checksums would consequently require four inode-sized or two mini-page-sized checksum blocks, chained by four respectively two special checksum blocks. The self sum field inside a special checksum block ensures its validity, while the "Sum Of Checksum Block" field holds the corresponding actual checksum block's self sum. The remaining 48 bytes are reserved for future usage. Compared to our chosen approach, such a linked-list approach is considered bad in terms of our design goals: The chaining approach forces the GPU to process the checksums sequentially. While the chosen approach can retrieve all checksums within one indirect layer fully parallel with one

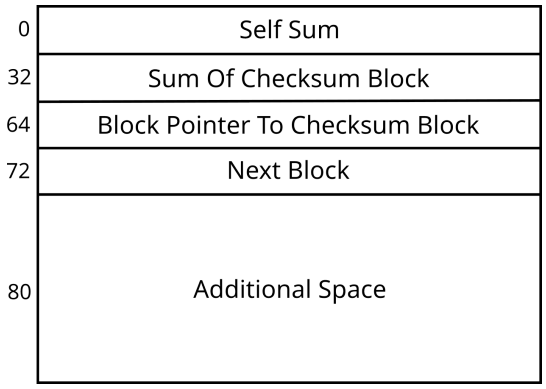


Figure 4.7: Special checksum block, forming an alternative checksumming approach. The special checksum block holds a self sum, the sum of the actual checksum block, a block pointer to the actual checksum block, a block pointer to a further special checksum block and provides 48 additional bytes for further usage. The depicted special checksum block is referenced within an inode or indirect block instead of the real checksum block.

distributed operation, a special checksum block's next pointer can only be resolved after fetching the special checksum block fully from drive. Although the actual checksum block can be retrieved in parallel with the next special checksum block, the process is highly sequential: For example, an inode-sized indirect block would require four sequential fetches each. Furthermore, the special checksum block would store the "Inode Sum" redundantly. With regard to the wasted additional space, the actual internal fragmentation of a special checksum block is high. In fact, the total overhead is always at least 128 bytes more than our chosen approach, as a special checksum block introduces 128 bytes regardless of the actual checksum block's size. For example, a fully occupied indirect block needs at least $128 * 2 + 256 * 2 = 768$ bytes of additional storage. Our chosen approach comes with a maximum overhead of $2 \cdot 256 = 512$ bytes to checksum an indirect block. Storing more actual checksum block pointers inside one special checksum block is not feasible as their self sum would exceed the special checksum block's size. Furthermore, extending its size is not sensible, as this would potentially aggravate the consumed overhead. Nevertheless, the major argument against the linked-list approach is its sequential nature. Therefore, this approach was not chosen.

4.2 Deduplication

With an integrated checksum approach, GPU4FS is ready for proposing a deduplication design. This thesis implements inline deduplication, as seen in ZFS [111]. This has one major reason: As inline deduplication reduces the amount of file writes, we found it more suitable for the bandwidth-limited Optane DIMM than a post-processing deduplication. Based on the research from Section 2.4, we chose to adapt the fixed-size chunking approach into GPU4FS: Its page sizes are the perfect chunking candidate. Thus, we call our design block-level deduplication, as it operates on the various block sizes of GPU4FS.

After proposing our design, we sum up with a discussion about alternative approaches and compare our implementation with already existing implementations.

A satisfying deduplication design should fulfill the following properties:

- Fast: Inline deduplication needs low latency and quick responses
- Parallel: High degree of independent parallelization
- Overhead: As few additional blocks as possible
- Extensibility: Design independent of block sizes, allowing e.g., CDC in the future [72]

4.2.1 File System Design

Inline deduplication requires a fast deduplication decision during every file write, while providing a persistable on-disk design. Therefore, we introduce a variety of new structs into the GPU4FS file system driver:

Deduplication Table (DDT)

We borrow the term deduplication table (DDT) from ZFS [9], as it fulfills the same purpose within GPU4FS. The DDT is a structure which stores all deduplicated blocks efficiently. We explain its structure in the following.

From a logical perspective, the DDT of GPU4FS follows the structure of a binary tree. Thus, GPU4FS's superblock references the root node of the DDT tree. Per construction, this root node contains two child nodes, which are each capable of having two further child nodes, and so on. This feature is recursive, which allows to reference an infinite depth of nodes, respectively deduped blocks. Leaf nodes within the DDT tree reference an actual data block on drive, whereas inner nodes reference a subtree of DDT entries with a common checksum part. The detailed structure of our DDT tree is outlined in Section 4.2.1.

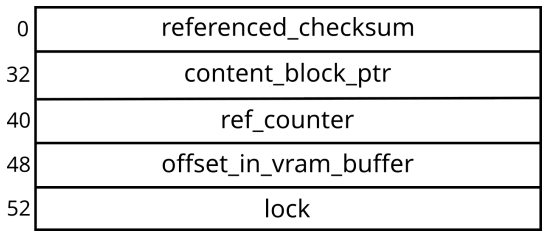


Figure 4.8: Design of a DDT entry in VRAM. Each DDT entry represents either a data block or a DDT inner node on drive. The “referenced_checksum” field stores either common checksum bits if the DDT entry is an inner node, or the full checksum if the DDT entry is a leaf. The “content_block_pointer” is a block pointer. Each DDT entry provides an 8-byte-long “ref_counter”, indicating the amount of pages it dedups. The “offset_in_vram_buffer” and “lock” integers are runtime components. They are only filled if “content_block_pointer” points to a further DDT inner node, respectively the current node is locked.

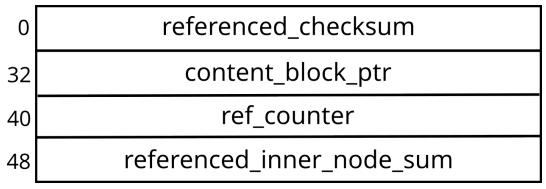


Figure 4.9: Design of a DDT entry on drive, which is similar to Figure 4.8. This DDT entry replaces the runtime components with a “referenced_inner_node_sum”. This sum is a checksum over the referenced DDT block, with 16 byte length.

Each node within the DDT tree consists of 64-byte DDT entries, outlined in Figure 4.8. Physically spoken, this makes up a total of 128 bytes per DDT tree node. Thus, a tree node fits perfectly onto an inode-sized page of GPU4FS [10]. Selecting a GPU4FS page size for tree nodes brings further benefits: A GPU4FS block pointer can reference a DDT tree node without any modifications to i.a., the type field. This allows smaller DDT entries, as there is no need for storing two block pointers of different size — the "content_block_pointer" can either reference a real data block or another DDT tree node. Furthermore, writing and reading a DDT tree node is efficiently possible, as DDT tree nodes can be treated as normal data blocks on the physical layer. Additionally, GPU4FS does not need to hold all DDT entries in VRAM: Fetching them from drive is as easy as fetching a normal data block. With a suitable eviction strategy, the maximum VRAM used for deduplication is limited. Additionally, a more advanced memory allocator for data blocks can also serve DDT tree node requests with no need for modification.

In terms of overhead, we face 64 bytes of additional space per data block in the file system. Given the most recent results from Dinneen et al., the log normal median of file size distribution is 9 KiB across their measured dataset [123]. 9 KiB fit onto three GPU4FS small pages, which then need $64 \cdot 3 = 192$ bytes for storing the DDT entries. Given the fact that we need to allocate inode-sized pages, we round those number up to 256 bytes, which corresponds to two allocated pages per file. If we consider 1 TiB which is distributed as outlined, we end up with $2^{40} \div 2^8 = 2^{32}$ bytes of DDT entries, which is a total of 4 GiB. It is important to mention that 64 bytes overhead per such an allocation are induced by the GPU4FS page sizes, not our design.

Ensuring DDT Integrity

The runtime-only fields are an important aspect of our DDT design. As the fields "offset_in_vram_buffer" and "lock" are only relevant if the according DDT entry was loaded into VRAM, we can use their space on drive differently: Our design suggests a DDT entry on drive as depicted in Figure 4.9. On drive, each DDT entry stores the checksum of the block behind the referenced "content_block_ptr". This "referenced_inner_node_sum" is suitable for validating the DDT entries before using them. During a write-back of a DDT entry onto drive, GPU4FS can update the checksum of this DDT entry. This design has further advantages: The sum within each DDT entry protects its referenced subtree, if the DDT entry is an inner node.

The DDT entry provides enough space to store a 128-bit-wide cryptographic checksum. Therefore, we suggest using the already implemented BLAKE3 algorithm for checksumming DDT entries. With regard to security, a truncated BLAKE3 128-bit-wide checksum is collision resistant with 64-bit security [49].

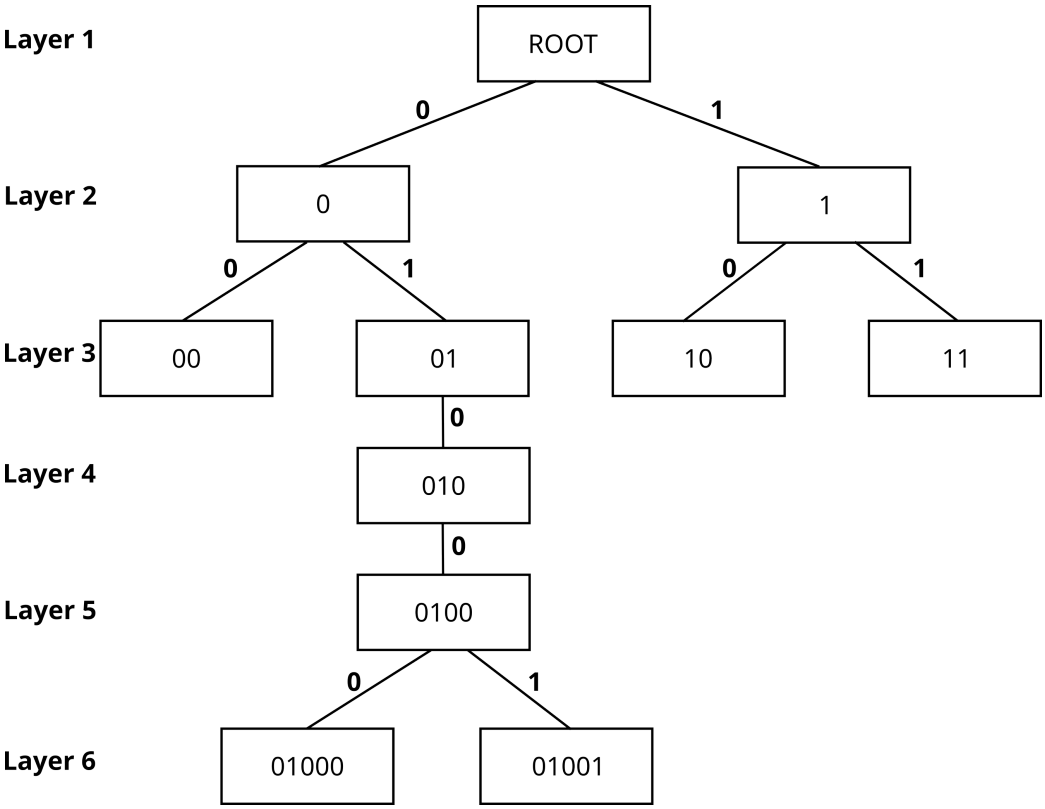


Figure 4.10: Exemplary DDT binary tree. The depicted figure shows a DDT with 5 referenced block pointers. The numbers on the edges represent the binary value of the layer-indicating checksum bit, while the values inside the nodes represent the relevant part of the stored checksum. Each leaf references one data block uniquely.

The first collision is therefore expected after 2^{64} stored blocks, which ought to be enough for our purpose; 2^{64} is an upper limit for the maximum addressable amount of blocks within GPU4FS [10].

Our demonstrator will not implement the DDT self sums. However, it is an important aspect for future work.

Allocating the DDT Tree

Trees need some kind of keys, which identify its entries uniquely. Those keys are used during traversation and allow the algorithm to choose the next inner node. Our DDT structure utilizes checksum bits as keys. We initially describe a binary tree implementation to describe our principle of keys. As a binary tree would waste

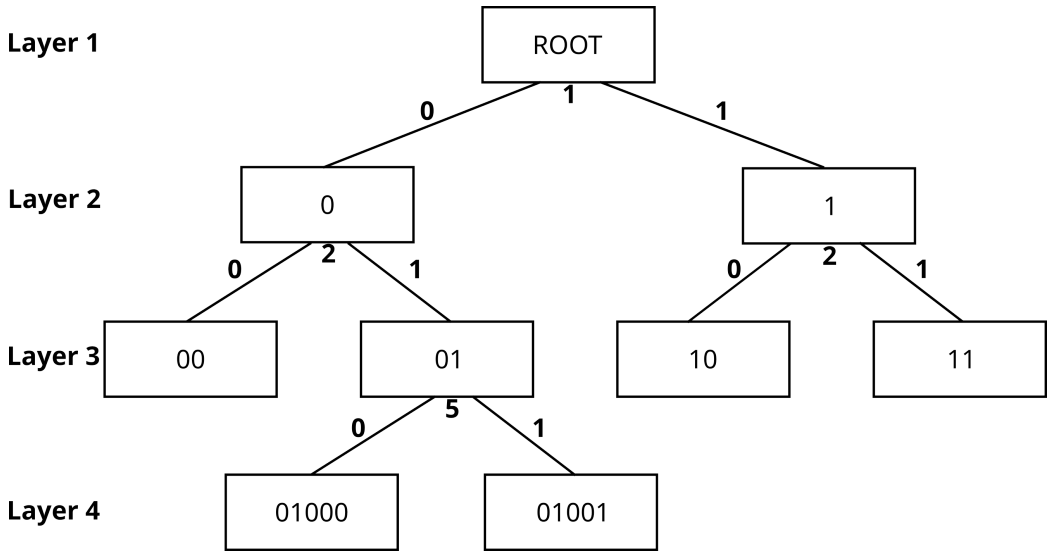


Figure 4.11: DDT binary tree from Figure 4.10 as binary radix tree. The figure outlines the radix property: The inner nodes “010” and “0100” were replaced as they have a common prefix. To identify the common prefix of a node’s referenced subtree, the numbers underneath the nodes are used. They represent the amount of similar checksum bits within the current layer.

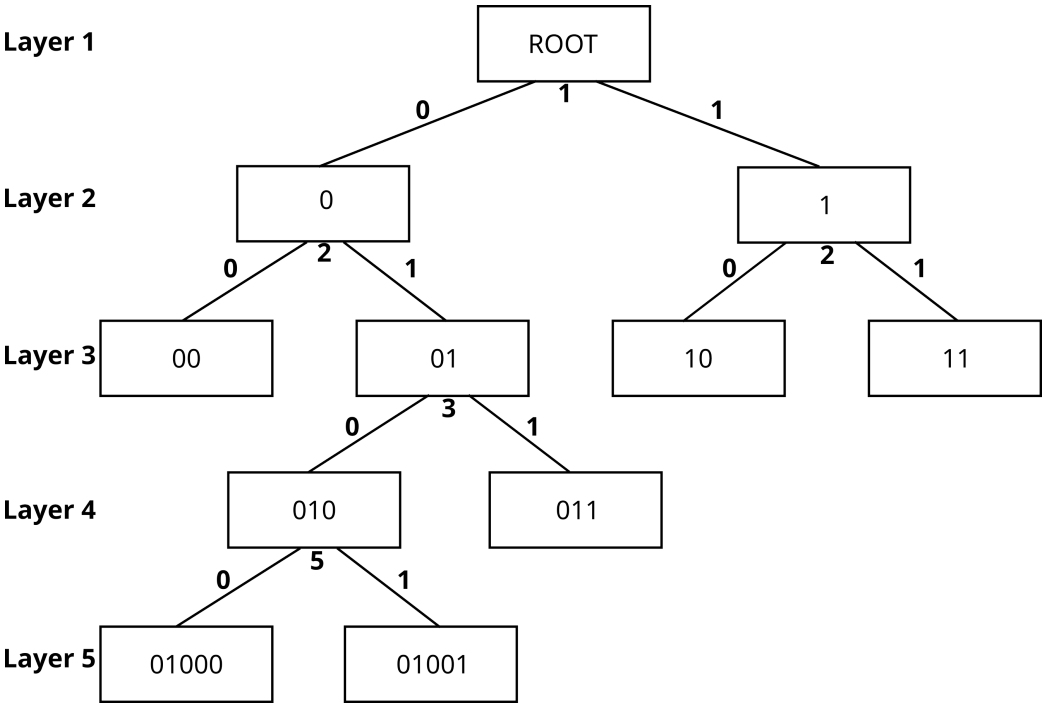


Figure 4.12: DDT binary radix tree from Figure 4.11 after adding the checksum "011". The inner node "01" was amended with an additional checksum bit, moved one layer downwards, and became the inner node "010". The former position of "01" was then initialized with a new inner node. This node uses checksum bit 3 instead of 5 for determining the child to traverse. The new "01" inner node references leaf "011" in the right child, while the moved inner node "010" is its left child. The child nodes of inner node "010" are unmodified.

space, we refer to an optimized variant afterwards.

Within a **binary tree**, the DDT tree is traversed in the following way: Starting from the root node, the first checksum bit of a data block determines if the tree traversal recurses into the left or the right child. While a value of "0" indicates the left child, its complementary value "1" indicates the right child. This procedure continues for each layer l_i using the i-th checksum bit. An exemplary binary tree is depicted in Figure 4.10.

However, using only one bit to determine the next recursion candidate has one downside: We can construct an example where we insert two checksums into our binary tree structure which are only different in their 256th checksum bit. Although the checksums only differ in one bit, we must allocate 255 further nodes. As an internal node consumes one inode-sized page, its allocation introduces an overhead of 128 bytes. This means that we would waste 255 inode-sized pages. Additionally, the allocated nodes force pointer chasing of an unnecessary long pointer chain, where each node refers to only one child.

Thus, a binary tree alone is not feasible: Instead of utilizing the i-th bit as key for the layer l_i, we can jump over a sequence of common checksum bits. This means that each child which is the only child within its parent is merged with the parent. This data structure is known as radix tree, which has shown promising within PMem [124]. The outlined technique of interpreting the bits as string and arranging them in a radix tree is related to the leaf organization in **extendible hashing** [76]. In the following, we explain our usage of the extendible hashing technique from Section 2.4.4.

Consider Figure 4.11 for an exemplary scenario. The according decision bit for layer l_4 is chosen by selecting the first different bit of the referenced two checksums as corresponding key — the 5th bit in the depicted example. However, a later allocation could differ in its 3th bit with the aforementioned checksums. To be able to resolve such a collision, we need to store the common checksum part of the compactified inner node. The collision scenario involves the preceding common checksum part to find the index of the first colliding inner node bit with the new checksum n_s. The colliding inner node n_c on layer l_i is then moved one layer down according to its common checksum value, and relocated within a newly introduced inner node n_n. The former position of the aforementioned inner node n_c on layer l_i is occupied by n_n. This new inner node n_n uses the colliding bit to decide between the newly introduced checksum and the moved inner node. Finally, the new inner node n_n on layer l_i receives the common checksum part of n_c and n_s. The resulting tree is depicted in Figure 4.12. As each insertion requires up to one new node allocation, our DDT design introduces at most 128 bytes per data block.

Each modification of the DDT affects only a small, fixed number of tree nodes. Locking is therefore applied on a node level. This ensures parallel computation to a high degree, as only a small set of nodes is locked for a short amount of time.

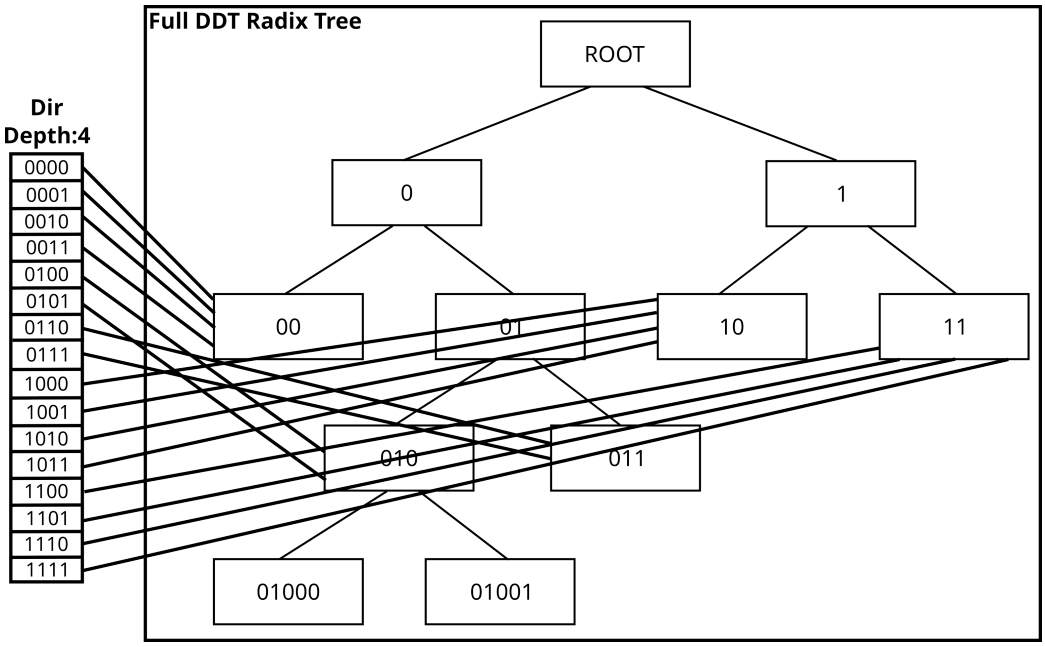


Figure 4.13: DDT binary radix tree from Figure 4.12 with directory. The directory stores direct references to DDT inner nodes, which reduces tree traversal.

With our design of the DDT tree, we are able to store DDT entries on drive efficiently. However, looking up the DDT tree requires $\mathcal{O}(\log n)$ operations, as we currently do not use any caching strategy. Extendible hashing as in Section 2.4.4 suggests a directory, which references its leaves via parts of the key. Within our design, those leaves are DDT inner nodes, referencing a subtree. Thus, we are able to use a directory of any size, which points to an DDT inner node of the tree. This way, we do not need to start our traversal at the root node, but directly in the correct subtree. Changes within the DDT may include changes in the directory, which consistency is guaranteed via atomic operations. A visualization of the directory is given in Figure 4.13. Based on the principle of extendible hashing [76], the directory is only allowed to point to an inner node whose stored key's length is less than or equal to the directory's key length.

Depending on the size of the directory, this results in a lookup time of $\mathcal{O}(1)$. However, the constant runtime is a trade-off between allocated VRAM for the directory and subtree depth to traverse. Contrary to the original extendible hashing, we do not limit the amount of entries within a leaf [76]. Therefore, there is no need for expanding the directory after a specific amount of insertions; we can initialize it either fixed-size or expand it by some heuristics. The directory itself can either be built dynamically after mounting the file system, or stored in a contiguous region on drive and loaded into VRAM. To provide high speeds, the directory should always be swapped into VRAM.

Given time constraints, we were not able to implement the directory structure. However, we expect its implementation to speed-up deduplication.

4.2.2 Discussion

The introduced deduplication design satisfies our four design goals: It is fast, locks on the fine-granular node level, introduces the least possible overhead, and is extensible to a high degree. If a future development of GPU4FS decides to increase the inode-page size from 128 bytes to 256 bytes, the proposed binary radix tree is substitutable with a 4-ary radix tree. The 4-ary radix tree holds four child nodes — which increases the size of a DDT node from 128 bytes to 256 bytes. The radix property works the same way, although the key now utilizes at least two bits instead of one bit.

Although the overhead of a file-level deduplication would be smaller, its deduplication ratio would also decrease. As mentioned in Section 2.4.3, the deduplication ratio of the applied technique relies on the surrounding context.

Towards Extensibility

As our design utilizes checksum bits for identifying blocks on drive, the deduplication design is not bound to a specific block size: A DDT entry could also store a reference to another data structure, which might hold `(block-pointer, offset, length)` triples. The DDT entry itself then consists of the checksum of the data behind those triples. Thus, the deduplication design supports CDC chunking, although this would introduce more overhead in terms of computation and storage. However, its evaluation is future work.

Fundamental differences to related work

ZFS was a great inspiration for our proposed design. It implements deduplication as an AVL tree in RAM, using the "extendible hashing" approach on-disk [112, 116]. The AVL property ensures that the referenced subtrees within each node differ in their height by only one layer [117]. This property makes the AVL tree a balanced tree. Nevertheless, balancing a binary tree comes with its costs: As a modification-induced imbalance might occur on a much higher level [125], the entire tree must be locked. Additionally, the AVL tree has no compressing properties similar to the radix tree. Thus, we chose to use a radix tree.

As with checksumming, **Btrfs** again uses a fundamentally different approach for deduplication. The core of Btrfs even does not support deduplication, as outlined in Section 3.3. Instead, the Btrfs documentation suggest the two supported tools Bees and Duperemove, which patch deduplication into the file system. Both tools implement a post-processing deduplication, which Btrfs refers to as "out-of-band" deduplication [107]. This is contrary to our design, as we provide an inline deduplication. The tool Duperemove takes a list of files as input and dedupes them by identifying duplicate extents. It must be invoked by the user and does not run as a daemon [109]. Duperemove relies on a red-black tree as its DDT [109]. Additionally, Duperemove subdivides its files in configurable long chunks, which are then checksummed and deduplicated. Bees on the other hand is designed to run as a daemon, which incrementally dedupes new data by utilizing a Btrfs tree search. Bees works on the whole file system, which is rather related to our implemented deduplication variant than Duperemove's procedure [108]. Differently to Duperemove, Bees uses a fixed-size hash table [108]. Thus, the DDT layout of Duperemove is more related to our design than the DDT layout of Bees. Finally, Bees examines the nearby area of a duplicate block to find larger duplicate areas [108]. This is again contrary to Duperemove and our design, as GPU4FS scans all blocks independently of each other — similarly to Duperemove. Duperemove on the other hand allows a configurable block size to dedup, whereas GPU4FS's deduplication design is limited to the page sizes.

As mentioned in Section 3.2, **EXT4** does not support deduplication at the moment. Thus, it cannot be compared to GPU4FS's deduplication design.

Alternative DDT Design — Simple Lookup Cache

Alternatively to a radix tree, the DDT could also be represented by a cache structure [126]. The full 256 checksum bits then are the key of the direct-mapped cache, where each unique entry references one data block pointer. The GPU4FS superblock could point to the address of the DDT lookup cache. This approach comes with major downsides: Utilizing a single direct-mapped cache would need 2^{256} entries to address all existing checksum values. Providing space for all of them is necessary as all checksum values could occur at any time with equal probability. Implementing the cache fully-associative is no option, as the lookup time would grow to $O(N)$. As one goal within our deduplication is performance and low latency for deduplication lookups, a linear table search of a potentially 2^{256} entries large page table is not suitable.

Alternative DDT Design — Hash Table

Advancing the direct-mapped cache approach to a hash table is another possible option. Again, the GPU4FS superblock could point to the address of the DDT hash table. Hash tables have, similar to direct-mapped caches, an access, insertion, and deletion time of $\mathcal{O}(1)$. The hash table stores entries in so-called buckets. Its hash function is our cryptographical checksum. As our hash table is not 2^{256} bytes large, hash collisions are unavoidable. Those could either be resolved via open addressing, or via separate chaining [127].

Open addressing moves elements inside the hash table if a collision occurs. The size of the hash table determines the amount of checksum bits which are used for addressing a value's table entry location. Thus, the hash table's size should be a power of 2. A colliding element is then stored on an alternative, free location. This approach comes with one major downside: The more elements the hash table stores, the denser it is occupied. At some time, the table needs to be relocated to provide more space. However, such a relocation requires the complete table to be locked, which violates our "parallel" goal. Additionally, if we expand the hash table more than 128 bytes at a time, we waste space in advance without knowing that this space is actually needed. This violates our "overhead" goal. Expanding the hash table by only 128 bytes locks the table more often, which reduces throughput and violates the "parallel" goal.

Separate chaining requires the hash table to store lists inside its buckets. After determining a value's bucket, it is stored inside the accompanying list. Implementing the lists naively would increase the lookup time of our hash table with a

growing amount of entries: As the hash table has a fixed size, its lists inside the buckets grow over time. Depending on their implementation (e.g., a linked-list), they seek a linear lookup time — which violates our "fast" goal. Trading the linear lookup time with a hash table resize is not easily possible, as we would change the hash key's size. This would require a revalidation of all bucket entries with moves from one bucket to another. This not only violates our "fast" goal, it also violates the "parallel" goal as we need to lock the overall structure during relocation. To mitigate this problem, the hash table could be allocated "larger" initially. However, this would again violate the "overhead" goal. From all outlined alternative approaches, separate chaining is most related to extendible hashing in its original variant. Nevertheless, our adaption of extendible hashing does not build fixed-size buckets which are accessed through the directory, but a tree which's inner nodes can be looked up within the directory.

Both outlined techniques identify the resize operation as costly. Although there exist approaches on reducing its overhead [128], subsequent work found that their performance is not significantly better than already existing approaches [77].

Towards Speed-Up Data Structures

Our design is able to use several speed-up data structures, which improves lookup time even further. One promising approach are CCEHs[77]. As with extendible hashing, we do not integrate this technique directly but adapt its concept.

Instead of placing two DDT entries onto an inode-sized page, we are able to place 64 DDT entries onto a small page. This decreases the depth of our radix tree, as the small page can use six checksum bits to address its entries. However, allocating bigger pages to store DDT nodes introduces more internal fragmentation, as cryptographical checksums are expected to collide with equal probability. The selection of DDT page sizes is thus a trade-off between performance and overhead. However, one sensible way to use larger page sizes is to compactify the DDT tree: If a subtree of depth six is fully occupied, we are able to group the according inode-sized pages together into one small page. The same argument holds for large pages and, potentially, huge pages. The advantage of this optimization is that we save pointer chasing operations and thus increase the overall throughput: As the DDT tree becomes denser, its upper levels are more likely to be fully or nearly fully occupied. The overhead of such a compactification is negligible, as we introduce a constant amount of one additional block pointer to perform the compactification. Additionally, we are able to reconstruct the original pointer structure from its compactification by walking the subtree and recreating the inode-sized pages within each layer. This way, we ensure to support a rollback if potential deletions of the DDT remove values from a compactified subtree. However, we do not implement this compactification and leave it at that as theoretical advice.

Chapter 5

Implementation

After designing the two file system features, their integration into GPU4FS is mandatory for evaluating some design decisions. The following section details the implementation process of checksumming, and deduplication afterwards. We describe the implementation process in the chronological order as it actually happened: A suitable BLAKE3 implementation is necessary for a checksum integration, while deduplication depends on a working checksum integration. Each section concludes with further work, describing additional implementation effort to improve our design. Throughout the whole chapter, the terminology "shader" is used and implies a compute command as introduced in Section 2.5.3.

5.1 Checksumming

The following section covers the implementation of GPU4FS's checksum functionality using GLSL shaders [129]. Initially, we describe the implementation of the depicted BLAKE3 [49] checksum algorithm on the GPU. Afterwards, we implement the theoretical checksum design from Section 4.1. Finally, we describe an approach on how to update and delete our checksums efficiently, and sum up with a discussion about alternative implementation approaches.

5.1.1 Checksum Algorithm — BLAKE3

Calculating a BLAKE3 checksum follows the outlined steps from Section 2.3.4:

1. Split input in 1024 byte large chunks and checksum them independently
2. Arrange those checksum results as leaves in a Merkle-Tree
3. Backtrace the Merkle-Tree to the root value

4. Output the root value as final hash value

The following section details our implementation of those four steps. We extend our approach within every section until we finally cover all preconditions for a GPU4FS integration. The BLAKE3 authors proposed two approaches for implementing step one on SIMD hardware efficiently. One approach tends to be more efficient on smaller input lengths, while the second targets larger input sizes. Both BLAKE3 implementations follow their "Chunk Chaining Values" suggestion, covered in Section 2.3.4.

Basic Structures

Our BLAKE3 implementation uses multiple data structures. The most essential structures are explained in the following:

The **state array** holds the internal state of each compressed chunk. Since BLAKE3's checksum can be calculated in parallel, we need to allocate storage to store the calculation results from multiple SIMD lanes. Within BLAKE3, 1024 byte large chunks can be compressed independently of each other, producing a 32 byte hash output for each. Those hash values are then combined in the Merkle-Tree fashion, as explained in Section 2.3.4. To share all results with all SIMD lanes within a workgroup, the state array must be declared "shared". GLSL then provides the `memoryBarrierShared()` call to ensure visibility between all SIMD lanes within a workgroup: All modifications prior such a barrier must be visible to all SIMD lanes within a workgroup after passing the barrier [84].

The **parallel hashing** struct represents the internal state of a single processing SIMD lane. This means that within a parallel hashing instance, the GPU stores the calculation results of one single local chunk calculation. Adding another state variable in addition to the state array allows state modifications during a chunk's compression process more easily while encapsulating the invalid results during the compression process. After calculating a chunk's sum, the internal state is moved from the parallel hashing struct to the shared state array variable.

In the following, we introduce the two different approaches for fulfilling step one of the checksum calculation. Those approaches are derived from the BLAKE3 paper. While the former approach is suitable for small input sizes, the latter provides better performance for larger inputs [49].

Implementing Approach One

BLAKE3's first approach arranges the internal state words $v_0 - v_{15}$ into four 128-bit vectors. Each of those four vectors contains four contiguous state words of the initial internal state, starting at state v_0. The approach then applies the compression function G to those four vectors. Afterwards, a diagonalization step

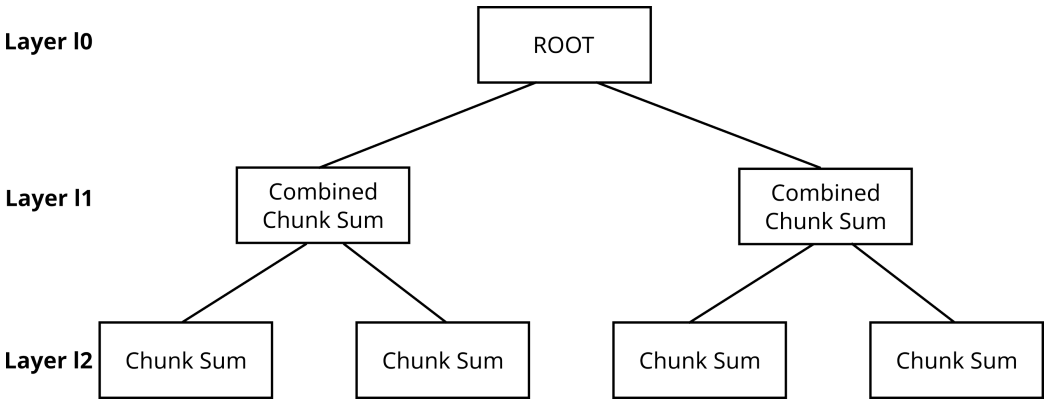


Figure 5.1: Exemplary Merkle-Tree, representing the structure of a BLAKE3 checksum. The depicted Merkle-Tree holds four leaves, each representing 1024 bytes of input length. The tree corresponds to a small page's checksum calculation.

rearranges those words so that each diagonal now forms a column. We used the `atomicExchange()` function to update the contents within a shared state array variable [49].

As approach one mixes the internal states of multiple chunks during its diagonalization step, we must ensure workgroup-wide memory consistency during compression. Our experiments showed that approach one performs worse than approach two, which we believe originates from the synchronization overhead. Thus, we utilize approach two in all scenarios.

Implementing Approach Two

Approach two suggests compressing one chunk per SIMD lane. Implementing the second approach utilizes the parallel hashing struct to store a chunk's state locally. After applying the compression function G and the permutation scheme according to BLAKE3's definition, each SIMD lane holds the hash value of its assigned chunk. This hash is then written back into the state array variable. Approach two follows the SIMD programming principle, which prevents dynamic branching. Thus, all SIMD lanes are able to work truly parallel.

Backtracing the Merkle-Tree

Step two now rearranges the results from step one as leaf nodes in a Merkle-Tree. We define $c_{l,i}$ as the Merkle-Tree node i, located on layer l. The lowest layer which accommodates the leaf nodes is called $lmax$. Initially, step two inputs two

neighbor leaves $c_{lmax,i}$ and $c_{lmax,i+1}$ with i being an even number as message block into BLAKE3's compression function G. Those inputted values are then processed in the same way as a real chunk block was processed before. Using approach two, another 32-byte checksum $c_{l-1,i}$ is calculated. This process continues within every layer l until the algorithm reaches layer $l0$. Figure 5.1 visualizes the backtracing up to the root node.

Backtracing a tree is efficiently done using a top-down recursion approach. The lowest recursion level would calculate the chunk sums utilizing the approach from Section 5.1.1, and the recursion stack would handle all the backtrace referencing automatically. As outlined in Section 2.3.4, the top-down approach was originally proposed by the authors, which found it suitable for a fork-join concurrency model on the CPU [49]. Sadly, GLSL does not allow recursion on the GPU [84]. Therefore, we cannot build the tree using the advantages of stack-based recursion. We solved this problem by constructing the tree in a bottom-up fashion: After building layer n, we backtrace one layer upwards until we reach the root node. Without recursion, backtracing a non-full tree is not trivial, especially when exhausting the parallelization potential. Given the properties of the BLAKE3 Merkle-Tree pointed out in Section 2.3.4, we split the tree in its fully occupied left side and its potentially not fully occupied right side. The left side can be processed in parallel using $\#chunks\ in\ left\ subtree$ SIMD lanes initially. Every backtracing step from layer i to layer $i-1$ halves the amount of compressing SIMD lanes. Within the right side, we backtrace all chunks $c_{l,i}$ if they have a neighbor $c_{l,i+1}$. In that case, we create a new inner node $c_{l-1,i}$. If $c_{l,i}$ has no neighbor, it becomes $c_{l-1,i}$. This procedure allows a bottom-up backtracing of arbitrary-sized BLAKE3 Merkle-Trees without recursion stack.

Processing Large Input Sizes

The approach from Section 5.1.1 is capable of hashing arbitrary input lengths in theory. Practically, it is limited to the workgroup size. In our case, the amount of SIMD lanes within a workgroup is 256. This means that this approach can process up to $2^{10} \cdot 2^8 = 2^{18}$ KiB. Within GPU4FS, however, we need to checksum page sizes which are 2 MiB or even 1 GiB in size. This motivates further research of backtracing larger input sizes.

A compute shader limits the allocatable space for global variables [130]. It is therefore not possible to allocate a large state array, providing enough space to satisfy our requirements. Even if there were no allocation limits: Storing all leaf's hash values of e.g., a huge page at the same time would require an array with $2^{30} \div 2^{10} = 2^{20}$ entries. This corresponds to $2^{20} \cdot 2^5 = 2^{25}$ bytes of storage, which is 32 MiB in total.

To solve this issue, the properties of a Merkle-Tree come to rescue. Let m be a

large Merkle-Tree with depth k. We consider a Merkle-Tree "large" if it exceeds the size of the state array. A subtree in this context is a part of the large Merkle-Tree which represents the compression structure of 256 chunks. The only size-exempted subtree instance is the last possible subtree within a dispatch, holding less than 256 leaves. This means that every other Merkle-Subtree ms_i with depth $l \leq k$ must have depth $l = 8$ while holding exactly $2^l = 256$ leaves. A subtree's leaf on depth l originates from the original tree's depth k. Those full subtrees are independently compressible up to their own root node $msroot_i$. The $msroot_i$ of a Merkle-Subtree ms_i is an internal node of the original Merkle-Tree, located at depth $k - (l - 1)$. It satisfies the requirement of not having any neighbor within its Merkle-Subtree ms_i and is therefore the subtree's root node.

Within our approach, we can store up to 256 chunk compressions inside the state array. By not exceeding this size, the procedure would output only one $msroot_0$ value which would represent the hash value of the input. As we exceed this size in GPU4FS, we have further subtrees with further $msroot$ nodes, as outlined previously. Therefore, the proposed modification does not return the backtrace result $msroot_0$ but rather stores it as $msroot\_first_0$ inside the **first level array**. The algorithm then continues with the next chunk values, which either form another full subtree with 256 leaves or the last subtree in a dispatch with less than 256 leaves. Their results are also not returned but stored as $msroot\_first_i$ inside the first level array. The first level array can hold up to 256 Merkle-Subtree hash values — which corresponds to a total input size of up to 64 MiB. The proposed algorithm is recursive, which means that the same procedure applied to a full state array can be applied to a full first level array, compressing its values into a second level array. When allocating 64 further second level array places, the algorithm can handle input sizes up to 4 GiB. A last step backtraces the results from the **second level array** array by utilizing the approach from Section 5.1.1. Figure 5.2 represents the outlined scenario.

Parallelizing across Workgroups

Utilizing the approach from Section 5.1.1, it is possible to compress input sizes of any length. However, this approach requires shared variables to allow backtracing. Shared variables within GLSL are only valid between multiple SIMD lanes within a workgroup, but not between different workgroups [84]. Even when considering VRAM storage, GLSL does not provide synchronization barriers which allow a coherent VRAM access across different workgroups [84]. Vulkan on the other hand allows synchronizing between different shaders [131]. The BLAKE3 compression shader is thus split into two shaders, whose execution order is outlined in Figure 5.3: The first shader $S1$ is responsible for the chunk compression and subtree backtracing proposed in Section 5.1.1, while the second shader $S2$ backtraces the

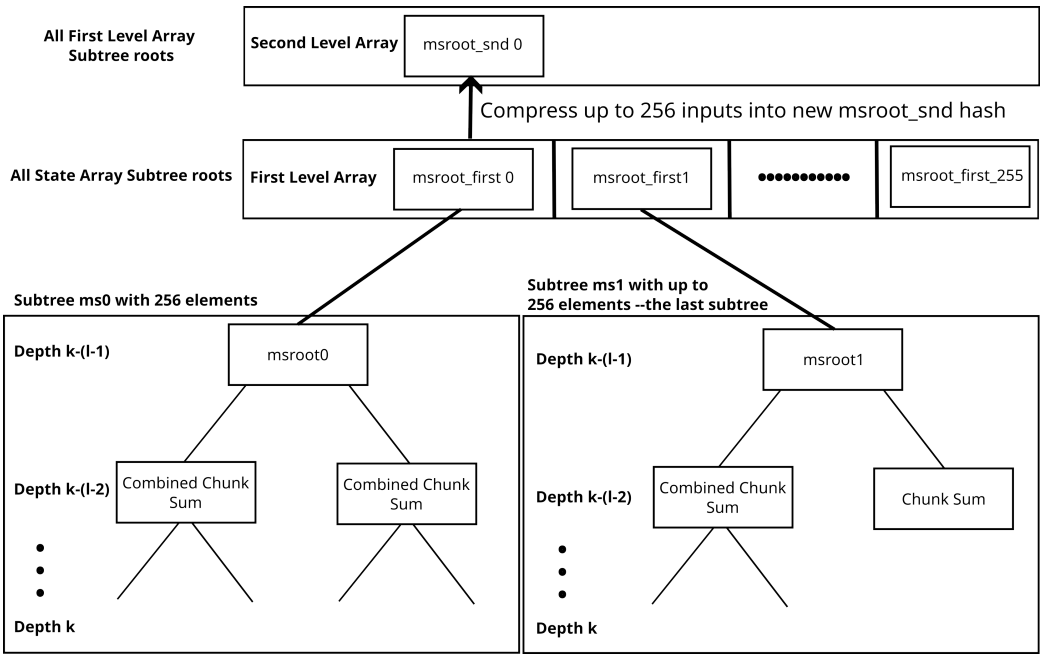


Figure 5.2: Exemplary compression of a state array which is fully occupied. Each subtree is either a fully occupied subtree with 256 elements, or the last subtree with ≤ 256 elements. The up to 256 hash values inside the first level array represent the hash values from the subtrees ms_0 to ms_{255}. The first level array is cleared by compressing its elements into the second level array. Afterwards, the next batch of subtrees ms_{256} to ms_{511} can be processed. This procedure continues until there are no subtrees ms_i remaining. In a final step, all elements within the second level array are compressed to the final hash value.

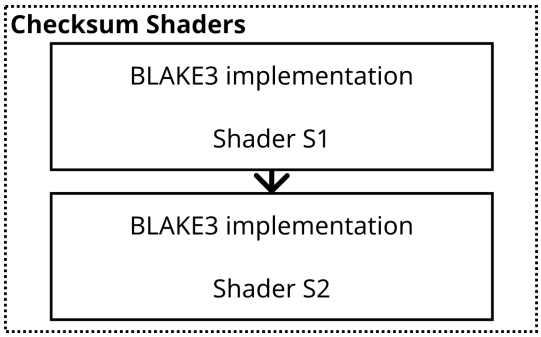


Figure 5.3: The two BLAKE3 shaders S1 and S2. S1 calculates the workgroup-local results while S2 uses them to backtrace workgroup-crossing. The arrow in between indicates a Read-After-Write (RAW) conflict and thus their execution order.

results from the first shader invocation in one final step. $S1$ can utilize multiple different workgroup sizes, which needs not be a power of 2. This property is important as most modern GPU's do not provide 2^n compute units [132].

Our implemented scheduling algorithm in $S1$ assigns a balanced amount of elements to each workgroup w_i in a greedy fashion, which ensures an efficient usage of the GPU's compute units. Within the execution of $S1$, each workgroup processes 2^i input bytes. The scheduling algorithm distributes three input sizes across the workgroups, which we call packages: The large package size, the small package size, and the remaining package size. As mentioned in Section 5.1.1, backtracing is only allowed between subtrees whose $msroot_i$ nodes are located on the same layer in the original Merkle-Tree. Therefore, it is necessary to introduce those three package sizes. We define the small package size to be one power of two less than the large package size. This means that a large package size of 2^i bytes results in a small package size of 2^{i-1} bytes. We then assign the workgroups either large or small package sizes. Only the last workgroup w_n within a dispatch receives a remaining package size amount of bytes. Allowing a non-power of 2 as remaining package size allows compressing sizes of arbitrary input length without destroying Merkle-Tree's properties. The algorithm distributes as many large packages to workgroups as there are enough bytes left to fill all other workgroups fully up with small packages. This means that our package size splitting point i ensures that all workgroups w_i to w_{n-1} process 2^{i-1} bytes, while all workgroups w_0 to w_{i-1} process 2^i bytes. After distributing the packages to the workgroups, the approach from Section 5.1.1 is applied by every workgroup individually, placing their results in a VRAM-mapped buffer. This buffer is called **workgroup result array**.

Shader $S2$ then receives this workgroup result array after appropriate synchro-

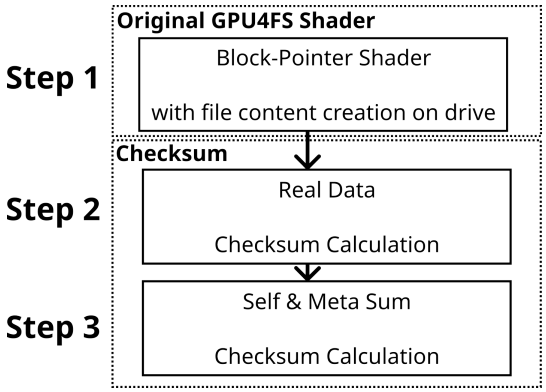


Figure 5.4: The shader pipeline which calculates a file's checksum within GPU4FS, utilizing the design from Section 4.1. Each box represents a shader call, while each arrow represents their execution order and RAW conflicts.

nization. Its only purpose is to backtrace the workgroup result array with respect to the three different subtree depths, which are induced by the three package sizes. $S2$ receives $\#workgroup$ many subtrees, which corresponds to the amount of chunks to compress within $S2$. Based on the outlined properties of the Merkle-Tree construction technique, $S2$ can utilize the backtracing approach from Section 5.1.1. Depending on the splitting point i, $S2$ must additionally compress some small packages starting at the subtree from former workgroup w_i to ensure a fully occupied left subtree. This is the case if the splitting point i is less than $\#workgroup \div 2$. Afterwards, the approach from Section 5.1.1 is applied to the workgroup result array, which was potentially further compressed to satisfy the left-subtree-property.

5.1.2 File System Integration

After porting BLAKE3 to the GPU, the checksum design from Section 4.1 integrates checksumming into GPU4FS. Writing a file's checksums to NVM is done within three steps and in accordance to the design from Section 4.1:

1. Reserving blocks and configuring block pointers within a file's inode and indirect blocks

2. Calculating real data, inode, and indirect block sums and writing them to the corresponding checksum blocks

3. Calculating the checksum block's self sums and storing them according to the block format from Figure 4.2

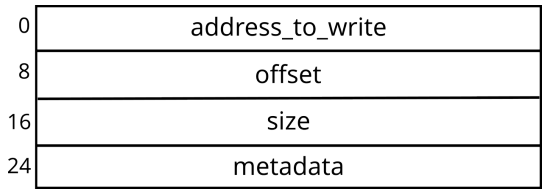


Figure 5.5: **Checksum control struct** which controls step two and three of the checksumming approach. A checksum control struct consists of four 8 byte unsigned integers. The "address_to_write" field specifies the address where to write the calculated checksum on drive. Furthermore, the struct defines the referenced data with length "size" via an "offset" into a mapped buffer. The field "metadata" holds multiple smaller values, for example an identifying "buffer id". The selectable buffers are either "File (0)", "NVM (1)" or "Command (2)". Those buffers are further described within GPU4FS [10].

The shader pipeline (with our pipeline definition from Section 2.5.3) is outlined in Figure 5.4. Its stages are explained in the following. The pipeline suffers from RAW conflicts, which require an appropriate memory synchronization. We outline our implemented synchronization mechanisms in Section 5.3.

Configuring Block Pointers

In its current implementation, GPU4FS uses one compute shader to implement its commands [10]. We need to implement checksum calculation only for its command file write, as all other commands do not perform any checksum-related tasks. The single compute shader processes the file write command through the following four steps [10]:

1. Writing file data to disk
2. Writing inode to disk
3. Configuring block pointers inside inode and indirect blocks
4. Updating the directory

We must intercept at step three of the file write command: Configuration of a file's block pointers must happen in accordance with our design from Section 4.1. Therefore, we modify the aforementioned step and add some logic which reserves an inode's zeroth, and every indirect block's zeroth and seventh block pointer. Every reservation is then stored in a block-type-matching, VRAM-mapped **checksum**

config buffer. These buffers hold an array of checksum control structs, whose design is outlined in Figure 5.5. The "block pointer" shader provides five different checksum config buffers, each separated by the following types:

- Inode and indirect block checksum (meta checksum config buffer)
- Self checksum (self checksum config buffer)
- Small page checksum (small checksum config buffer)
- Large page checksum (large checksum config buffer)
- Huge page checksum (huge checksum config buffer)

Each checksum config buffer holds its own array of **checksum control structs**. A distinction between self sums and other sum types is necessary as a checksum block's self sum can only be calculated after its content was fully written to disk. However, the content of a checksum block is not known prior to step two of our checksum write approach. The distinction between inode, small, large, and huge page checksums is necessary to synchronize our shader pipeline properly. Additionally, it ensures compliance with the original GPU4FS file write design, which is good in terms of backwards compatibility. Both arguments are further outlined in Section 5.3. Furthermore, their distinction allows better load balancing of different page sizes, as outlined in Section 5.1.2.

The config buffers do not rely on any kind of sorting, which makes their processing more flexible. This is additionally compliant with GPU4FS's idea of out-of-order command execution [10].

Calculating Real Block, Inode, and Indirect Block Sums

With filled config buffers, the next step is calculating the actual checksums. Step two calculates all data, inode, and indirect block sums, whose checksum control structs are stored inside the checksum config buffers. We refer to inode and indirect block sums as "meta" sums, respectively "meta" pages. To calculate the checksums, we extend the BLAKE3 implementation from Section 5.1.1. The checksum calculation of real, inode and indirect block sums must support the four different GPU4FS page sizes which were outlined in Section 3.1.3: 128 byte, 4 KiB, 2 MiB, and 1 GiB. In Section 5.1.1, we argue that each workgroup is capable of processing about 262 KiB of input length. This means that shader S2 only needs to be invoked for 2 MiB and 1 GiB pages. Our extension to the already existing BLAKE3 implementation covers this case distinction: If a page's size is 128 bytes or 4 KiBs, it is scheduled on one workgroup only, utilizing shader S1. If a page's size is 2 Mibs or 1 GiBs, it is schedulable on $\min(\#wg, page\_size \div 2^{18})$

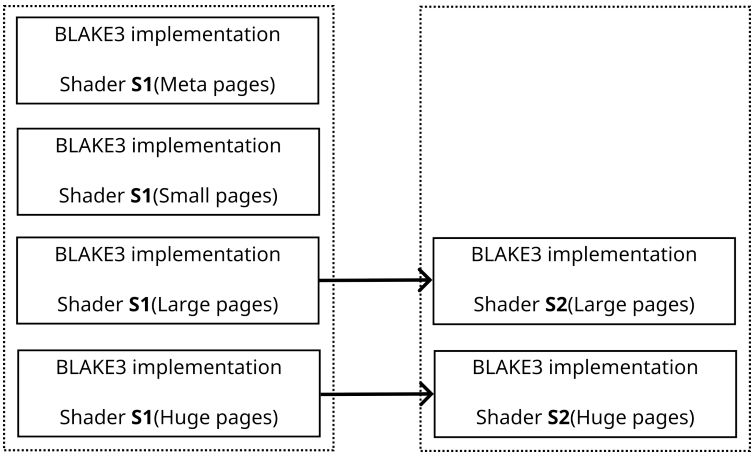


Figure 5.6: Detailed shader call procedure of step two from our checksumming shader pipeline. Four instances of S1 calculate the workgroup-local results according to Section 5.1.1, whereas the two S2 dispatches backtrace the results of their corresponding S1 shader according to Section 5.1.1. The arrows in between indicate RAW conflicts.

many S1 workgroups. The argument $page\_size \div 2^{18}$ within $\min$ ensures that each workgroup has at least 262 KiB to compress — according to Section 5.1.1. This distributes as much work as possible onto the GPU's different compute units, allowing a high throughput.

Distributing the different page sizes onto different sizes of workgroups requires some effort. Although we describe the procedure of checksumming one file with the shader pipeline, GPU4FS is designed for multiple dispatches of file writes [10]. Therefore, step two must also support scenarios where e.g., a large page follows a small page, which requires the shaders to collect enough workgroups for scheduling a large page. However, this would involve communication between workgroups, which is — as outlined in Section 2.5.3 — not officially supported. Therefore, we implement step two within four S1 and two S2 dispatches, which is visualized in Figure 5.6. Each of them map their type-matching checksum config buffer, as described in Section 5.1.2. The four S1 dispatches are responsible for compressing either huge or large pages into their own workgroup result array, or writing small or inode page sums directly to NVM. As only the pairs of S1 and S2 shaders are dependent of each other, the checksum calculation of all page sizes works in an out-of-order fashion. We only need to ensure that an S2 shader blocks until its corresponding S1 shader provides enough input. Resolving this RAW conflict is explained in Section 5.3. Additionally, a dependent pair of S1 and S2 shader could be moved independently of all other pairs within the pipeline. This property is

especially important for deduplication, as outlined in Section 5.2.1.

This modification also introduces more flexibility for configuring GPU4FS to an application's workload: The amount of workgroups for checksumming each page size can be adjusted by the mounting application. Thus, an application which normally deals with many small files can dispatch more workgroups for checksumming small pages, and the other way around. Mapping the corresponding checksum control buffers into the according S2 dispatch allows a correct decision of the number of workgroup result array elements for each entry in the checksum control buffer.

Calculating the Checksum Block's Self Sums

Step three is responsible for calculating the first entry within every checksum block: Its self sum. Their calculation process is simpler than the process from Section 5.1.2, as there is only one single page size to consider. With respect to Section 4.1.2, a self sum calculation is scheduled on one workgroup. Our procedure maps the **self checksum config buffer** as checksum config buffer, and starts another run of S1.

As mentioned in Section 4.1, the checksum design is able to not only use mini-page-sized checksum blocks. In case of a checksum block allocation larger than 4 KiB, it is necessary to involve a dispatch of shader S2. However, this is not part of this thesis, but important for future development of GPU4FS.

5.1.3 Future Work — Update and Deletion

As a file system needs to update and delete files at certain times, we propose a theoretical approach on how to support those tasks within checksumming. However, this thesis's demonstrator neither implements nor evaluates those tasks — we leave our advice for future work.

Update

Supporting block updates involves two main tasks: Updating a block's sum and updating its indirection path. Both processes imply no fundamental differences to our proposed implementation: The checksumming shader pipeline from Figure 5.4 receives information from the block pointer shader. Based on that information, the checksumming process takes place. Thus, by implementing the update process in the block pointer shader which is responsible for configuring block pointers and a file's indirection hierarchy, we automatically gain support for our checksum update mechanism: The block pointer shader fills the according checksum config buffers

with information about the modified data block(s), as well as the meta and self blocks which must be rechecksummed.

Supporting updates in general introduces a new command into GPU4FS's list of supported commands. It may be derived from the file write command [10], but should provide specific information and offsets of the blocks which need to be updated.

Deletion

Deleting a checksum is a trivial task, as this task's complexity lies in directory updates: Deleting a file means decrementing the hardlink counter of its inode to zero. A zeroed hardlink counter means that this inode is no longer referenced within the file system and can be overwritten. Therefore, all blocks belonging to that inode — especially our checksum blocks — are automatically freed and can be reallocated. There is no need for overwriting the stored checksums. However, the reallocation process needs a more advanced block allocator [10], which is not context of this thesis.

5.1.4 Future Work — Outlook

Our checksumming approach works within the original demonstrator of GPU4FS. However, it is capable of some extensions, which we advise in the following.

Verification of Checksums

Although checksums are now introduced into the file system, we do not utilize them for checking the actual validity of the file content. As the original demonstrator does not implement the ability to read files through the GPU, we had no opportunity of integrating a checksum validation for the GPU side. Thus, we suggest implementing the verification of checksums when reading blocks from disk by simply recalculating a block's sum and validating it against the stored value. The checksum verification can utilize our proposed implementation with minor changes, applying it either after reading a file's content fully or in an asynchronous fashion.

RAID

BLAKE3 ensures unique fingerprinting of GPU4FS blocks. This feature is not only relevant for deduplication, but can also be used to implement RAID functionalities [18]: When detecting a corrupted block, a RAID 1 could utilize a checksum to

detect if the same block is intact on a mirror drive and trigger a replication. RAID functionality is already a work-in-progress within our team.

More Checksum Algorithms

In Section 2.3, we introduced multiple families of checksum algorithms. Cryptographic checksums were relevant to ensure deduplication support, but may be too expensive for simply protecting against bit flips. As our approach was specifically designed to be decoupled from the utilized checksum algorithm, we encourage the evaluation of simpler algorithms like Fletcher's sum.

5.2 Deduplication

This section details our deduplication implementation, which follows the design of Section 4.2. We initially describe modifications to the GPU4FS pipeline which was outlined in Section 5.1.2. Afterwards, we outline our implementation of the deduplication shader, including further practical aspects. We conclude with theoretical advice for updating and deleting a DDT entry and motivate future work.

5.2.1 Preparing the Deduplication Shader

Before we are able to integrate a new deduplication shader, the implementation from Section 5.1.2 needs some modifications. Currently, we generate our block pointers in the initial block pointer shader and write the file content immediately on drive. Afterwards, we proceed through the checksum shader invocation, which calculates block checksums over data and checksum blocks. However, deduplication depends on already calculated checksums for deciding if a block is actually written to drive. Otherwise, the according block pointer needs to point to an already existing block while not writing the duplicate. As our implementation from Section 5.1.2 calculates the checksums after writing the actual data on disk, deduplication cannot be implemented straightforwardly. Implementing deduplication without any modifications to the shader pipeline would introduce high write overhead, as duplicated blocks would be written unnecessarily to drive. Instead, we decide to enrich the shader pipeline with the **write-data** shader. Its purpose is to write the actual file data onto drive. Decoupling the write-data process from the block pointer creation allows the pipeline to utilize a higher amount of workgroups for writing blocks to drive than for their block pointer creation — which brings more parallelism.

Figure 5.7 depicts the modified shader pipeline. The original GPU4FS shader is now only responsible for configuring the block pointers and directory entries.

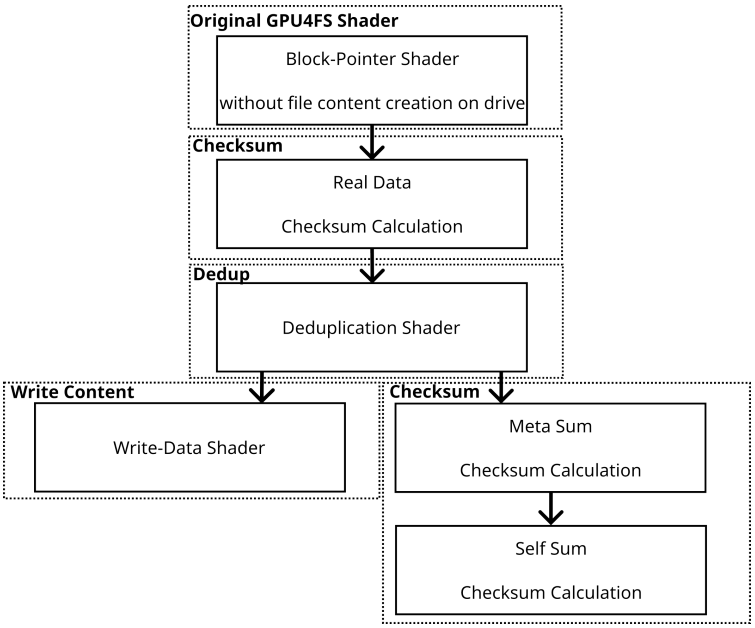


Figure 5.7: Shader pipeline of GPU4FS with integrated deduplication shader. The arrows indicate RAW conflicts.

However, the data behind the configured block pointers is not written to drive immediately: The block pointer fills a buffer in VRAM instead. The so-called **staging buffer** holds an array of **staging structs** (Figure 5.8). Those staging structs represent a block pointer on drive, combined with the content it points to. The write-data shader is able to use this information when writing the data to drive. The only values written by the block pointer shader are the inode and its indirect blocks, filled with block pointers. Those values are required regardless of the deduplication shader's decision.

As a potential deduplication decision leads to different entries within an inode or indirect blocks, we cannot calculate their checksum in advance to the deduplication shader. Thus, the calculation of meta sums is dependent of the deduplication step — which justifies the new "Meta Sum Checksum Calculation" step after the deduplication. As the self sums of those checksum blocks depend on calculated meta sums, we decided to execute the self sum shader completely after the meta sum shader. Although the self sums of real data blocks could be calculated after the "Real Data Checksum Calculation" step, two different self sum invocations are not sensible in terms of overhead. Thus, all self sums of a file write are calculated within the "Self Sum Checksum Calculation" step. As the meta and self sum calculations do not influence the write-data shader, we can execute them in parallel. Figure 5.7 indicates this behavior by placing both shaders next to each

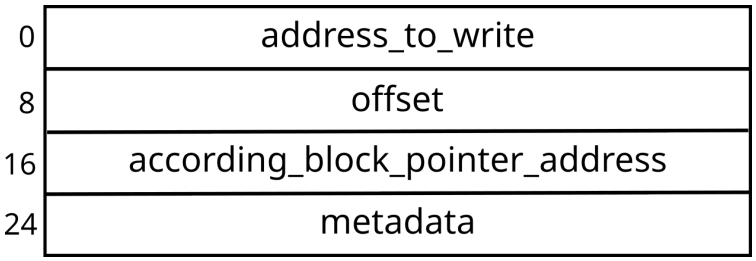


Figure 5.8: Staging struct in detail. It consists of the "address_to_write", an "offset" into a VRAM buffer, the "according_block_pointer_address" of the backing block pointer and a field with metadata. The "metadata" field includes e.g., the size.

other, without an additional RAW conflict between them.

With these modifications, we decoupled the block pointer creation from the actual content creation on disk and ensured a correct checksum calculation. This brings further benefits for a pipelined execution, which we outline in Section 5.3.2. However, we still need to pass all block's checksums to the deduplication shader. As the deduplication shader is executed after the checksum calculation, we fill another buffer named **DedupInfoBuffer** with **DedupInfoBufferStructs** during the checksum calculation. Those structs are nothing more than combinations of checksum values, an offset in our staging buffer and a metadata field. This offset ensures that the deduplication shader can determine the corresponding staging buffer entry for a calculated checksum. DedupInfoBufferStructs are 48 bytes in size each. With those modifications, we are able to integrate deduplication.

5.2.2 Deduplication Shader

With the design from Section 4.2 in mind, we implement a new deduplication shader. This shader's main purpose is to decide wether a staged block needs to be written or is already present on drive. The latter case reconfigures the corresponding block pointers, and signals the write-data shader to not write the duplicate. To fulfill this purpose, the deduplication shader needs access to the staging buffer and all corresponding checksum values. Thus, GPU4FS executes the deduplication shader after checksumming the data blocks, but prior to the write-data shader.

Block Distribution across Workgroups

The first step within the deduplication shader is to distribute a file's blocks across all dispatched deduplication workgroups. Each workgroup is able to start a deduplica-

tion when it has received eight DedupInfoBufferStructs. This number is explained in the following. The block distribution across workgroups either follows a concurring or a cooperative approach. Selecting one of those two approaches happens implicitly by selecting one of the RAW synchronization mechanisms, which are outlined in Section 5.3. For now, it is important to know that each deduplication workgroup is able to start the deduplication process as soon as it was assigned at least eight DedupInfoBufferStructs.

Shader Implementation

After a deduplication workgroup acquired enough DedupInfoBufferStructs, the actual deduplication takes place. Within the responsible workgroup, each subgroup acquires one of the workgroup's assigned DedupInfoBufferStructs. Although the deduplication process would only need one SIMD-lane per DedupInfoBufferStruct, we utilize complete subgroups of 32 SIMD lanes. This decision is based on the structure of the GPU and the high potential of divergent branches: As explained in Section 2.5, different control flows between SIMD lanes of a subgroup lead to branch divergence. Such a divergent branch is resolved by executing the groups of divergent SIMD lanes sequentially. As we explain in the following, our deduplication process involves four distinct cases which occur relatively evenly within a growing DDT. Therefore, parallelizing over SIMD lanes would introduce a high degree of branch divergence. However, as each subgroup, respectively wave, has its own instruction pointer, we have no divergent branches during the actual deduplication process if we parallelize over subgroups instead of SIMD lanes.

We describe our shader implementation by explaining the process of inserting a block b's checksum chk into the DDT.

After acquiring an entry from the DedupInfoBuffer, a subgroup proceeds by walking through the existing DDT. Within each layer, the subgroup decides which checksum bits of chk are relevant to walk through the structure. After walking the tree down to its leaves, the subgroup decides between two cases:

1. DDT leaf is empty
2. DDT leaf is occupied

The former case is the simplest case in our algorithm: If the indicated DDT leaf is empty, a dedup table entry is filled according to Section 4.2.1. The latter case checks the DDT entry's stored checksum against chk: If both match, it found an entry to dedup and increments the "ref_counter" of the matching DDT entry by one. However, if they do not match, this step involves a relocation of the already existing DDT leaf. The algorithm then replaces the leaf with an inner node and utilizes the nearest different checksum bit of both, the former DDT leaf and chk, to

address them uniquely in the new inner node. The exact behavior of this relocation was specified in Section 4.2.1.

As we use our compactified DDT tree from Section 4.2.1, a subgroup must also validate each inner node it traverses against chk. Inserting a new DDT entry could involve a relocation within an inner node. This is the third case which could occur whilst inserting a new DDT entry. After utilizing the relevant checksum bit of chk to decide a recursion into the right or left child, a subgroup validates the checksum bits in the child's "referenced_checksum" field. This field stores the prefix checksum bits which are equal in the underlying subtree, and is depicted in Figure 4.8. Before continuing its traversal, the subgroup decides between two cases again:

1. DDT inner node's referenced_checksum is equal to same-length prefix of chk

2. DDT inner node's referenced_checksum is not equal to same-length prefix of chk

Again, the former case is the simpler case of both. If the chosen child node's subtree has the same checksum prefix than chk, the subgroup recurses into the child without any modifications. The latter case involves a relocation of this child node, according to Section 4.2.1: The subgroup removes the inner node from the DDT structure and integrates a new inner node. This inner node has two children, of which one points to the previously removed inner node (utilizing the distinct checksum bit) and the other one points to chk. The new inner node then stores a common prefix of its children, which must be shorter than the common prefix of the relocated inner node.

GLSL neither allows vectors nor dynamic sized arrays [84]. Thus, we must store all allocated DDT entries in a fixed-size, contiguous array of type **dedup inner node**. A dedup inner node is the struct which represents a DDT inner node, having two DDT entries as children. Consequently, a **DedupTableEntries** buffer allocates this array in VRAM. To allocate from that array, we introduce a small buffer called **DedupTableEntriesOffset**. This small buffer is only 72 bytes in size and holds nothing more than the DDT tree root and an integer counter "last_offset". This "last_offset" counter indicates the next free entry in our VRAM representation of the DDT. Each insertion into the DDT structure atomically increases the "last_offset" field and thus ensures that each index is uniquely acquired by exactly one subgroup. Those atomic operations are implemented through GLSL's atomic functions [84].

Ensuring a fine-granular locking of our DDT entries is crucial to exploit much parallelism. As mentioned in Figure 4.8, we use the "lock" field of each DDT entry to lock it during a traversal. Each subgroup locks one DDT entry — the currently traversed one — at a time. This lock remains acquired during all required

operations to that DDT entry, especially relocation. The subgroup releases its lock immediately when continuing its traversal into a child node. As the index of an inner node does not need to correspond to its logical location in the DDT tree, we do not need to move all DDT entries physically when a subgroup allocates a new inner node and relocates the currently locked one. This decoupling of logical and physical location allows a subgroup to lock and process an inner node, while another subgroup may be processing the inner node's subtree at the same time — without interruption.

```
1   bool lock_block_ptr_atomically(uint parent_offset, bool
        direction) {
2       int res;
3       //Depending on direction, lock either left or right child
4       if (direction) {
5           //Ensure atomicity through atomic compare swap
6           res = atomicCompSwap(ddt_entries.parent_entries[
                parent_offset].left_node.locked, 0, 1);
7       } else {
8           res = atomicCompSwap(ddt_entries.parent_entries[
                parent_offset].right_node.locked, 0, 1);
9       }
10      return res == 0;
11  }
12
13  dedup_block_result dedup_block(){
14  ...
15      bool is_locked = false;
16      do {
17          //Lock child node of "current_inner_node" of either "
                left" or "right" direction
18          is_locked = lock_block_ptr_atomically(
                current_inner_node, direction);
19          //Only continue if successfully locked
20          if(is_locked) {
21              //Continue with actual work on DDT entry
22              ...
23              //Unlock previously locked child node. This method
                    is equivalent to lock_block_ptr_atomically, but
                    swaps the "0" and "1" in atomicCompSwap
24              unlock_block_ptr_atomically(current_inner_node,
                    direction);
25          }
26      } while(!is_locked);
27  }
```

Listing 5.1: Spinlock around actual deduplication procedure. The "do/while" loop is our actual spinlock, which tries to acquire a DDT entry atomically. It can only proceed if it acquired the entry successfully.

Our locking procedure utilizes a spinlock to synchronize between subgroups. The implemented lock is outlined in Listing 5.1. Synchronizing between subgroups within a workgroup is supported by GLSL natively: As outlined in Section 2.5.3, the `memoryBarrier()` primitives guarantee synchronized memory between SIMD lanes within a workgroup, and thus also subgroups. However, synchronizing SIMD lanes between workgroups is more complicated, as mentioned in Section 2.5.3 and Section 5.1.1: GLSL does not provide any synchronization primitives between workgroups [84]. Atomic functions on the other hand guarantee that a modified value is visible to all invocations afterwards, even between different workgroups [84]. Additionally, GLSL provides the **volatile** memory flag, which forces the shader to load and store directly to global memory. In combination with an atomic locking field and our spinlock procedure from Listing 5.1, we found a way to guarantee memory visibility between workgroups within GLSL code.

5.2.3 Future Work — Update and Deletion

For providing update and deletion support within deduplication, we introduced the "ref_counter" field into the DDT entry. This field is depicted in Figure 4.8. As the deduplication shader increases the "ref_counter" with each successful deduplication, an **update** process can decrement the field of the former referenced block pointer atomically by one. Thus, the deduplication shader needs knowledge about both the old and the new block pointer and checksum of the modified block; which can be provided via an additional staging struct stored in the staging buffer. As the "offset_in_staging_array" field of a DedupInfoBufferStruct is 64 bit large, we can simply split the field into a lower and an upper half with 32 bit each. Both halves then reference a staging struct each: While the lower half points to a staging struct which holds the old block pointer and checksums, the upper half points to the staging struct which contains the updated information. The old checksum can then be retrieved from drive, while the new checksum comes from the priorly executed checksum shader. The deduplication shader then decrements the DDT entry which references the old block pointer and processes the new staging struct as described in Section 5.2.2. This process depends on the modifications from Section 5.1.3, meaning that GPU4FS needs the new command "FILE_UPDATE".

The deduplication shader handles a block **deletion** analogously — with the only difference that there is no new staging struct next to the old staging struct. If the "ref_counter" field reaches the value zero, the DDT entry identifies no block on drive anymore and can be freed.

However, our demonstrator neither implements update nor deletion support. We leave these implementation details for future work.

Digression: The Boundary-Shift Problem

As we use fixed-size chunking, the deduplication process is prone to the boundary-shift problem [69]: Inserting some bytes at the start of a file could lead to potential shifts throughout all files, which means that most blocks within the file would be classified new. However, we can address this problem within GPU4FS: If a block overflows during an insertion, GPU4FS is able to shift the overflow onto a new GPU4FS block, without touching the following file blocks. This way, they remain untouched, which means that a potential deduplication stays persistent. To indicate if this procedure should be applied, we could introduce a "dedup" bit flag, similar to the checksum bit flag from Section 4.1.1. Depending on the amount of set "dedup" flags in an area, the update process is able to decide which update strategy it should apply. Although this procedure introduces more internal fragmentation, we gain another benefit from that consideration: Only few blocks must be changed on PMem, which reduces write amplification. However, the implementation and evaluation of this idea is future work.

5.2.4 Future Work — Outlook

Our demonstrator implements only a subset of potential deduplication features. Thus, we outline some future work in the following:

Paging

One important aspect of our implementation is the ability to swap DDT entries in and out. Given the low latency of Intel Optane and the small size of a DDT entry, we assume that evicting DDT entries and exchanging them with PMem is fast — at least faster than with any other persistent memory device. Implementing a drive-backed pagination brings further benefits, as we ensure consistency with the underlying PMem as soon as an entry is no longer held in VRAM. Although our demonstrator does not implement this feature, we have some theoretical advice on how to implement it.

The "offset_in_vram_buffer" field from Figure 4.8 indicates if a DDT entry is present in VRAM. It is zeroed out if the corresponding entry is swapped onto PMem. The subgroup which fetches the entry from drive sets the corresponding "lock" field, which reduces additional reads from the DIMM and thus stress. Given the granularity of our DDT entries, however, it seems sensible to hold the most traversed pages (which are the "upper" DDT-tree levels) always in VRAM, and to swap only within the "lower" levels. The crossover point between upper and lower level is user definable or could even utilize dynamic heuristics. This crossover point is also affected by the size of the extendible hashing directory, which was

introduced in Section 4.2.1. It is also feasible to implement a hybrid mode, where the lower levels are always exchanged with PMem. With a working directory implementation as described in Section 4.2.1, it could be sensible to hold solely the directory in VRAM, and modify DDT entries directly on PMem. However, the outlined principles need further research.

To provide an independent allocator for the DDT entries, the GPU4FS superblock provides a "reserved" field. As mentioned in Section 4.2, we are also able to use the default linear allocator of GPU4FS to allocate space for DDT entries. However, with future development of an advanced block allocator, we do not need this linear allocator anymore.

Paging — The Vulkan Way

Vulkan provides the ability to declare a device-local buffer pageable [133]. If the Vulkan device supports this extension, it is able to swap out buffers from VRAM to DRAM if the VRAM is under pressure. Thus, this feature may be utilized to enhance the paging process from Section 5.2.4, but needs further research.

Indirect Block Deduplication

The fact that equal data comes in patterns led to more advanced deduplication approaches, e.g., CDC [72]. However, CDC within GPU4FS raises more questions about the overall file system, as the overall block pointer design and indirection hierarchy could need modifications. Thus, we suggest implementing indirect block deduplication in addition to data block deduplication. This feature is not sensible within our demonstrator, as the demonstrator only supports one indirection layer. However, files with a higher indirection degree as well as potential different indirect block sizes could benefit from an indirect block deduplication.

Checksumming DDT Entries

Our DDT design is already prepared for integrating DDT self sums, as outlined in Section 4.2.1. Thus, we leave this advice for future work.

Unmount and Remount

The problem of spilling the DDT entries onto PMem was already outlined. The same procedure supports a graceful unmount process: When **unmounting** the file system, one final step before committing the unmount is to write back all modified DDT entries from VRAM to PMem. A DDT modification is detected by calculating all DDT entry self sums and comparing them with their counterpart on drive: A VRAM entry must be written over a PMem entry if their self sums diverge.

As the DDT sums are recursive, it is important to write the DDT entries to drive in a bottom-up fashion. After flushing the leaves, their parents can be checksummed and flushed, then their parents can be checksummed and flushed, and so on.

Depending on the mount time of the file system, the amount of modified DDT entries can be huge. Thus, a continuously running daemon could reduce the amount of non-flushed DDT entries by writing back some modified entries periodically — e.g., when the file system idles.

Remounting the file system is straightforward, as the DDT could either be prefetched from PMem (to some extent) or lazy loaded on purpose. Depending on a potential directory implementation as in Section 4.2.1, the remount could also involve a load or full rebuild of the directory.

5.3 Resolving RAW Conflicts

The outlined implementation does not clarify how to solve the RAW conflicts. According to Section 2.5.3, Vulkan provides many synchronization mechanisms. Our implementation needs to synchronize between different workgroups, especially between different commands. Thus, we have multiple options for resolving the RAW conflicts. This section distinguishes two resolution options: We refer to them as the Vulkan approach and the GPU4FS approach. While the former uses standard Vulkan functionality, the latter focuses on GPU4FS structures and uses the overall GPU4FS command design. It is important to mention that the Vulkan approach is not capable of the shared command buffer idea from GPU4FS [10]. This led to the development of the GPU4FS approach, which we motivate further in Section 5.3.2.

5.3.1 The Vulkan Approach

Vulkan's synchronization mechanisms include fences and pipeline barriers, as introduced in Section 2.5.3. While a fence is suitable for synchronizing different command buffers, a pipeline barrier synchronizes memory accesses within a command buffer. Our final pipeline, as outlined in Figure 5.7, has the flexibility to use both of them interchangeably, with minor changes to the implementation.

The pipeline can be implemented via five different command buffers. Those can be synchronized via fences. In fact, this means that each arrow between the stages in Figure 5.7 represents a fence in code, ensuring the termination of the conflicting step. It is important to mention that, according to Section 5.1.2, our checksumming procedure utilizes two shaders within its command buffer, not one. Consider Figure 5.3 for those additional RAW conflicts. However, the RAW-indicating arrows can be resolved by splitting the S1 and S2 shaders into two further command buffers, which are then synchronized via fences. Consequently, the "Real

Data Checksum Calculation" step consists of two instead of one command buffer if both S1 and S2 are needed. This is the case for large and huge page's checksums.

On the other hand, the pipeline could utilize one common command buffer for all shader invocations. In this case, a fence is no longer appropriate as it only synchronizes between but not within command buffers. Thus, a pipeline barrier comes to help: All RAW conflicts can be resolved by putting a pipeline barrier at exactly those positions where the aforementioned resolution put fences. This ensures not only a memory-coherent pipeline, but also coherency within a multi-shader stage as the checksum calculation. Additionally, the implementation is able to execute the write-back shader truly parallel to the meta sum and self sum calculations, as induced by Figure 5.7.

Our implementation of the Vulkan approach utilizes one command buffer in combination with pipeline barriers, as pipeline barriers do not involve busy-waiting on the CPU. Fences require a CPU thread to stall, which is against the goals of GPU4FS [10]. As the Vulkan approach needs to know all scheduled GPU4FS commands before starting the shaders, it is able to distribute all commands equally across all available workgroups, within all shader stages. Thus, it is the cooperative synchronization approach which assigns each workgroup its subset of commands in advance.

5.3.2 The GPU4FS Approach

Although the proposed mechanisms from the prior Section 5.3.1 are feasible for situations where GPU4FS is invoked as a one-time program, they are not compliant with its original design: GPU4FS should run as a daemon rather than a one-time program, having a command buffer per accessing application [10]. This means that each application which requires access to the file system originally runs the startup routine once, with one invocation of the original compute shader. GPU4FS uses only one compute shader within its original implementation from Section 3.1, and provides daemon-like behavior by utilizing an endless loop within the shader. This endless loop always polls for new command buffer entries, and dispatches them onto a workgroup. When using the RAW resolving mechanisms from Section 5.3.1, the endless loop would never allow the block pointer shader to terminate. Thus, both the fence and the pipeline barrier afterwards would never trigger, which would never allow our following shaders to execute. Additionally, a pipelined per-file execution of the shaders is not possible due to the pipeline barriers. To solve this issue, we put each stage from the shader pipeline from Figure 5.7 in one Vulkan command, without any Vulkan-based synchronization mechanism. Then we adapt the endless loop from GPU4FS's initial shader into our shaders: Each dispatched shader runs in an endless loop, and acquires commands from the GPU4FS command buffer. As Vulkan commands can run independently on

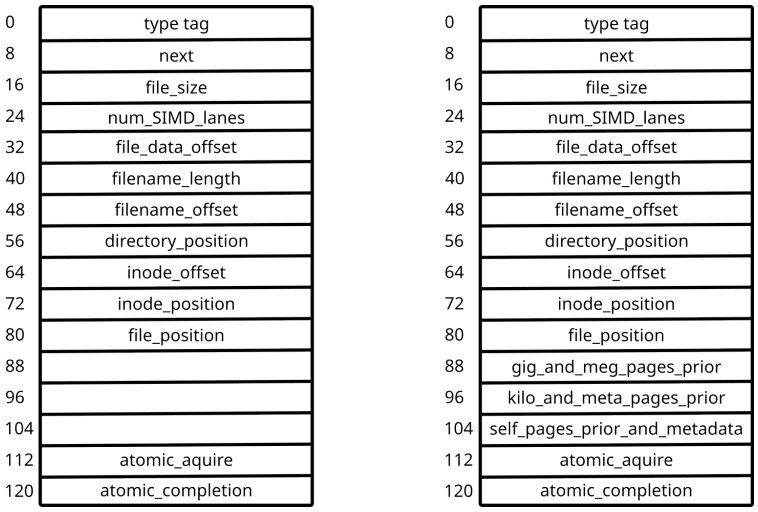


Figure 5.9: "File write" command descriptor from GPU4FS. The left side represents the original "File write" command descriptor [10], while the right side depicts our modification.

the GPU, they do not hinder each other to proceed. This modification raises two former solved issues:

1. How to guarantee ordered per-file execution of our shaders?
2. How to guarantee memory synchronization across our shaders?

Both Vulkan mechanisms which solve these issues are no option anymore, as the endless loops would never allow a shader to terminate.

To solve those issues, we assign each shader type one unique identification bit. This means that especially our checksum shaders need seven bits — five for their S1 and two for their S2 variant — as their exact type must be distinguishable. We then modify the "atomic_acquire" and "atomic_complete" field's usage. These are integers within a file write command descriptor [10]. The file write command descriptor is depicted in Figure 5.9. The fields were originally designed for indicating a command acquisition by a workgroup, respectively its completion [10]. In its current implementation, the fields are used as a binary bit flag [24]. However, the fields are 8 bytes in size and therefore capable of holding 64 bits. Within our modification, the "atomic_acquire" signals a command's current processing state: If a shader successfully acquires a command, its "atomic_acquire" field is "binary or"-ed with the shader's identification bit. Using GPU4FS's block assignment algorithm [10], a command is dispatched only once within each shader. The "atomic_complete" field on the other hand signals a shader step completion: If a command is fully

processed by a shader, its "atomic_complete" field is "binary or"-ed with the shader's identification bit. This signals the following shader that it can dispatch this command, resulting in another "atomic_acquire" modification by the subsequent shader. A command is then fully processed after its "atomic_complete" field has the write-data shader and the self sum shader bit set. Contrary to the Vulkan approach, the GPU4FS approach does not assign commands to workgroups uniquely: Here, the workgroups are responsible for acquiring commands by themselves. This makes the GPU4FS approach the concurring approach.

Although this procedure works in theory, we have one last problem to discuss: Cache coherency. A written value is not necessarily visible to other shader invocations. However, we can adapt our DDT entry locking approach from Section 5.2.2: By declaring all mapped buffers **volatile**, we ensure memory coherency across workgroups [98, 99]. The relative ordering of memory accesses is guaranteed by `memoryBarrier()` primitives [84] and simple spinlocks: Each shader step spinlocks on the "atomic_completion" field of a command it wants to acquire. The shader is then only allowed to proceed if the matching shader flag in that field was set by the previous shader. Afterwards, the shader tries to acquire the command via an atomic "or" on the command, using its unique identification field. To signal its completion, it sets the appropriate bitfield in the "atomic_completion" field after issuing a `memoryBarrier()` call — which allows the next shader to continue. The principle is sketched in Listing 5.2. Our file write command descriptor outlined in Figure 5.9 introduces the following fields into its previously empty positions: "gig_and_meg_pages_prior", "kilo_and_meta_pages_prior", and "self_pages_prior_and_metadata". The former two fields hold the amount of already processed gig, meg, kilo or meta pages. As a reminder, meta pages in our context are indirect blocks or inodes. Those indicators are mainly used by our shader implementations to identify positions in all internal buffers uniquely. As we allow out-of-order execution, the shaders do not need to process all commands in the same order. Thus, it is important that we assign each file write command its own budget of free array positions which are not dependent from a shader's dispatch ordering. Therefore, the introduced fields indicate the exact amount of previously processed blocks, each separated in different flavors. The entries each are 32 bit wide, as we expect that after 2^{32} pages the first scheduled block is done and the buffers can be cyclically refilled. The "metadata" field is general purpose and uses only one bit in our current implementation, which we currently use to enable or disable deduplication on file write command level.

```
1     int acquire_new_block(int block) {
2     //... use code from maucher[10]
3
4     // spinlock until command is ready for current shader
5     // atomic operations are not removed by compiler
6     while ((atomicAdd(config.data[block_offset +
          completion_offset],0) & (FLAG_PREVIOUS_SHADER)) == 0);
7
8     // workgroups tries to acquire, atomic operation ensures that
          only one workgroup succeeds
9     if ((atomicOr(config.data[block_offset +
          already_acquired_offset], FLAG_CURRENT_SHADER) &
          FLAG_CURRENT_SHADER) == 0) {
10        new_block = block;
11        return 0;
12    }
13    //... use code from maucher[10]
14 }
15 void dispatch(int start_block) {
16    //... use code from maucher[10]
17    memoryBarrier();
18    barrier();
19    //signal command processing done
20    if (local == 0) {
21       atomicOr(config.data[block * block_size +
             completion_offset], FLAG_CURRENT_SHADER);
22    }
23    memoryBarrier();
24    barrier();
25    //... use code from maucher[10]
26 }
```

Listing 5.2: Command acquisition process, schematically sketched. The "do/while" loop is our actual spinlock, which tries to acquire a DDT entry atomically. It can only proceed if it acquired a block successfully. Otherwise, it tries over and over.

Distributing Checksum Calculations

The GPU4FS approach in its presented manner works if each command only needs one workgroup within a shader stage. However, we face the situation that a large or huge checksum calculation needs more than one workgroup per data block. As we do not specify a fixed amount of workgroups, we need an even more fine-granular assignment algorithm, which allows the dispatched workgroups to distinct which part of the block they are working on. Our implementation introduces an additional buffer per shader, the so-called **ProcessingInfoBuffer**. This small buffer holds

two integer arrays: The stage and the done array. The index in both arrays is given by the currently selected command number, which simply is an integer number. If a workgroup is about to acquire a block belonging to a command, it atomically increases the stage value of this command. If the stage value was zero, the workgroup additionally is responsible to initialize the done field which belongs to the command with the highest possible stage value for this command. In terms of checksumming, this is either the total amount of large pages or the total amount of huge pages. The done field is then atomically decremented after each workgroup finished its execution. As we support out-of-order execution, we cannot assume that the workgroups finish in a first-in-first-out fashion. Thus, the done field ensures that the workgroups do not set the "atomic_completion" field until each workgroup processed its assigned subset fully. This way, we are able to count the amount of workgroups working on a block, and assign each workgroup the correct block to work on.

Towards Out-Of-Order Execution

The GPU4FS approach already involves out-of-order execution of the shaders. However, we can benefit from even more out-of-order execution by utilizing the several introduced structs from Chapter 5. With respect to the final pipeline from Section 5.2.1, we are able to start the write content stage of a command as early and parallel as possible: The block pointer shader inserts a staging struct into the staging buffer as soon as the according block pointer was written. Thus, we can use one unused metadata bit of that struct to indicate its readiness for a write-data dispatch. Depending on the activation state of deduplication, this is either the case after writing the staging struct into the staging buffer, or after passing the deduplication shader. Thus, a system with disabled deduplication is able to write a block as soon as its block pointer was created. As write content then runs parallel to checksumming, this hides latency and reduces the overall execution time. A system with enabled deduplication on the other hand could benefit from a speculative write content mechanism. However, speculative writing needs further investigation by future work.

The deduplication principle benefits from an allowed out-of-order execution of the overall deduplication process. We know from Section 5.2.2 that each dispatch of a deduplication shader needs at least eight assigned DedupInfoBufferStructs to perform one deduplication iteration. Thus, we are able to indicate their readiness via the metadata field, which can be set by the checksumming process. The deduplication shader is then able to spinlock on both the DedupInfoBufferStructs and the according staging struct, and proceeds after it acquires eight ready blocks. This procedure utilizes the same "dispatch" and "acquire_new_block" principle as outlined in Listing 5.2, with the exception that it does not rely solely on the

"atomic_acquire" and "atomic_completion" fields of a GPU4FS command. A command's "atomic_acquire" field is set after all containing blocks were acquired by a deduplication instance, as well as a command's "atomic_completion" field is set after the last deduplication shader completes its processing of a command's blocks. To ensure that each deduplication shader instance acquires eight different ready blocks, we use the idea of the **ProcessingInfoBuffer** again. In context of deduplication, the done field of a command is initialized with the maximum amount of deduplication cycles for a command, which is a multiple of eight. This amount can be calculated via the file size, stored in the associated GPU4FS command.

Chapter 6

Evaluation

In the following, we present a detailed evaluation of our newly introduced features. Initially, we cover the raw checksumming performance and compare it with its GPU4FS implementation. Afterwards, we benchmark the deduplication process isolated. As we introduced a pipeline into GPU4FS, we benchmark the overall design with an integration test of our new pipeline, and compare the results to former results from GPU4FS [10].

6.1 Testing Methodology

To benchmark our features, we utilize Vulkan's "timestamp" query functionality [89]. This functionality allows us to write a timestamp on the GPU after passing a specific stage within a command buffer. In depth, the timestamp queries insert an execution dependency between its prior and its following commands, with respect to the specified stage for the timestamp query. This stage must exist in the Vulkan pipeline, as outlined in Section 2.5.3. Thus, we insert a query before and after each command submission. We define the "all commands" stage as our stage flag. This ensures that the commands we want to time passed all stages within the Vulkan pipeline [89]. We also declare a memory barrier in front of both timestamp writes, to ensure that all memory operations were written. It is important to mention that our GPU4FS related evaluations mapped the GPU4FS command buffer on VRAM instead of DRAM. This mitigates PCIe bandwidth limitations, although it does not prevent them completely. If not mentioned differently, we mapped all persistent buffers in the "tmpfs" environment which is backed by DRAM. Furthermore, we inserted a cleanup shader which cleans all buffers before executing any benchmark.

Our evaluation process ran on three different machines. We outline their technical details in the following:

6.1.1 Machine “Optane”

The first machine is our target machine. It consists of the following technical specifications:

- Dual Intel ™ Xeon ® Silver 4215 CPUs @ 2.5 GHz, 16 threads each.
- 128 GB of 16 GB DDR4 DIMMs at 2400 MT/s, 64 GB per CPU.
- 512 GB of Intel Optane memory at 2400 MT/s, DDR4-socket-compatible. Distributed into four 128 GB DIMMs and 256 GB per CPU.
- AMD RX 6800 GPU, with AMD’s RDNA2 architecture. It consists of 60 Compute Units, 16 GB VRAM, and a memory interface of 512 GB/s width. Connected via PCIe 3.0.

6.1.2 Machine “RX7900XTX”

Specifically for benchmarking the pipeline’s performance, we introduce machine “RX7900XTX”. It consists of:

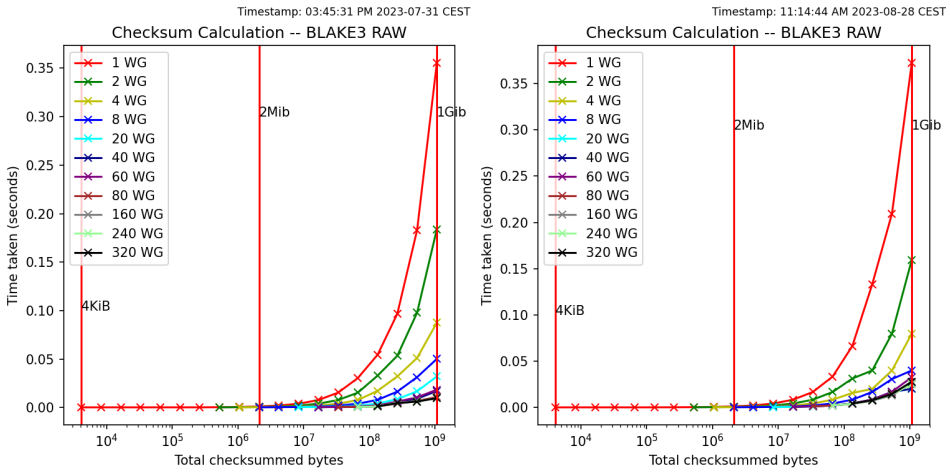
- AMD Ryzen ™ 7 3700X, 8 cores with SMT (16 in total) @ 3.6 GHz.
- 8 GB of DDR4 at 2400 MT/s, single channel configuration.
- AMD RX 7900XTX GPU, with AMD’s RDNA3 architecture. It consists of 96 Compute Units, 24 GB VRAM, and a memory interface of 800 GB/s width. Connected via PCIe 3.0.

6.1.3 Machine “Laptop Nvidia”

For evaluating our results not only on AMD hardware, but also on Nvidia graphics cards, we introduce machine “Laptop Nvidia” with the following capabilities:

- AMD Ryzen ™ 9 6900HS, mobile processor, 8 cores with SMT (16 in total) @ 3.3 GHz.
- 32 GB of DDR5 at 6400 MT/s, dual channel configuration.
- Nvidia RTX 3050 Ti mobile, based on a GA107 Ampere Chip. It consists of 20 Compute Units, 4 GB VRAM, and a memory interface of up to 195 GB/s width. Connected via PCIe 4.0

It is important to notice that machine “RX7900XTX” and “Laptop Nvidia” are not Intel Optane ready. Thus, we cannot benchmark the overall GPU4FS performance on them. However, those machines still show interesting performance details of our implemented design.



(a) BLAKE3 on Machine "RX 7900XTX"

(b) BLAKE3 on Machine "Laptop Nvidia"

Figure 6.1: BLAKE3 on GPU, comparing various workgroup sizes. The algorithm performs better, the more workgroups it has available.

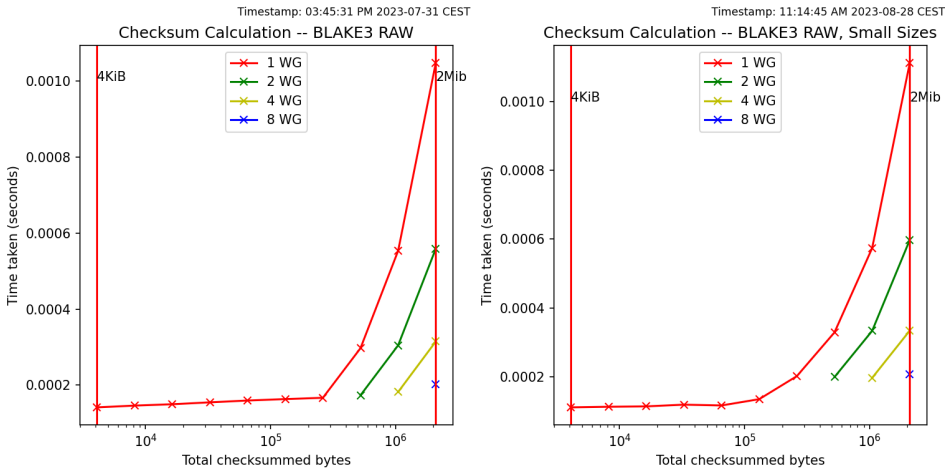
6.2 Checksumming

This section evaluates our implemented checksumming approach. As outlined in Section 5.1.1, the process of checksum integration involved the implementation of BLAKE3 on the GPU. This implementation is independent of its GPU4FS integration. Thus, we initially evaluate the implemented checksum algorithm. Afterwards, we compare the GPU4FS integration to the theoretical performance of our algorithm.

6.2.1 Checksumming — Raw Algorithm

Our BLAKE3 implementation includes various steps, which were outlined in Section 5.1.1. We show the performance of the overall algorithm in the following section.

Figure 6.1 depicts the raw BLAKE3 algorithm on machines "RX7900XTX" and "Nvidia Laptop", dispatched with different workgroup sizes. Figure 6.2 details the leftmost sides of both subfigures from Figure 6.1. We observe that the execution time rises linearly with the amount of checksummed bytes. However, we can reduce the execution time by increasing the amount of processing workgroups, which leads to more parallelization. This is exactly what we expected. Regarding Section 5.1.1, additional workgroups are only sensible when one workgroup utilizes its SIMD lanes fully and would need another iteration to continue computation. As we



(a) BLAKE3 on Machine "RX 7900XTX"

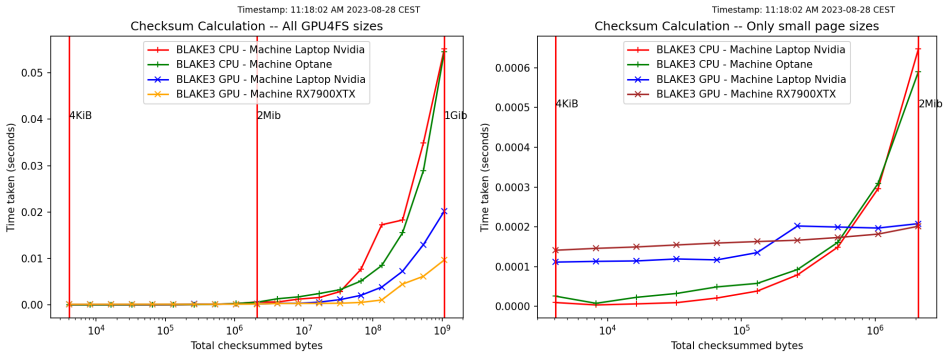
(b) BLAKE3 on Machine "Laptop Nvidia"

Figure 6.2: BLAKE3 on GPU, comparing various workgroup sizes. Zoomed view of Figure 6.1. We observe that eight workgroups are the sweet spot for checksumming 2 MiB.

implement approach two of BLAKE3 [49] (compare Section 5.1.1), this is the case on exactly 256 chunks or $2^8 \cdot 2^{10} = 2^{18}$ bytes of input. Thus, we benchmark additional workgroups only when the algorithm can use them.

Figure 6.3 compares our measured performance results with the BLAKE3 Rust reference implementation, provided by the original BLAKE3 authors. We benchmarked the original implementation and switched to its multithreaded variant after the single threaded variant was slower. This was the case at around 1 to 2 MiB. The depicted GPU results represent the best results from Figure 6.1. Our results show that the GPU implementation of BLAKE3 outperforms the reference CPUs at around 1 to 2 MiB — the crossover point where we needed to use the CPU's multi threading variant. Thus, it is questionable if the multi threading implementation of the original variant works performantly. We verified our CPU results through the official CLI tool "b3sum", which showed comparable results in the higher end. The lower end of "b3sum" showed higher results, which we believe to originate from precision problems within bash's time command and additional overhead.

As Figure 6.3b shows, the GPU implementation is slower for small input sizes when compared to the CPU implementation. However, this is not surprising: The reference implementation on the CPU is able to use native AVX instructions. Contrary, the GPU code must go through multiple translation layers before it is compiled down to the GPU's ISA. Additionally, the small sizes fit the execution model of a CPU better, as the few cores of the CPU can deal better with small



(a) BLAKE3 — All sizes (b) BLAKE3 — Small sizes only

Figure 6.3: BLAKE3 on various GPUs, compared with the reference implementation on CPU. The markers indicate the GPU4FS page sizes. We observe that large size calculations are better on GPU, although the reference implementation on CPU performs better for small sizes.

input sizes. However, as GPU4FS only needs an algorithm which can checksum GPU4FS page sizes, we are able to build optimized variants for small and large pages. Their performance benefits are outlined in Section 6.2.2.

Speaking of the reference implementation, we also took a closer look on the results from the original BLAKE3 authors. They propose that their algorithm provides a throughput of up to 140 GB/s for 32 MiB when executed on an Intel Xeon Platinum 8275CL with 48 threads [49]. This is questionable, as we measured different results with their provided code. We also found another GPGPU implementation which provides execution times on an Intel Xeon Platinum 8358 CPU with 64 threads. Their benchmarks show similar results to our depicted measurements [134]. We do not know how the BLAKE3 authors achieved those speeds, but do question them at this point. However, we can imagine that they somehow removed the overhead coming from a DRAM fetch from their measurements, but cannot prove this theory yet.

Takeaway

The outlined results have implications for our GPU4FS configuration. As Figure 6.2 outlines, we should use one single workgroup for checksumming small page sizes and eight parallel workgroups for checksumming large page sizes. From a theoretical perspective, this makes sense as those workgroup sizes are capable of calculating the checksums without one inner iteration: One workgroup has 256 SIMD lanes, which lets it compress up to $256 \div 4 = 64$ small pages. In contrast, eight workgroups are capable of compressing $2^{18} \cdot 2^3 = 2^{21}$ bytes, which is exactly

the size of a large page. Theoretically, a huge page checksum calculation would need $2^{30} \div 2^{18} = 2^{12}$ workgroups, which cannot be scheduled truly parallelly on current GPUs. As our results imply, the BLAKE3 algorithm works more efficiently when distributing it across more workgroups — if it can employ them with data. Thus, we chose to select 256 workgroups as an appropriate workgroup size for calculating huge page checksums — especially as the BLAKE3 summer is capable of backtracing up to 256 workgroups without using a loop.

6.2.2 Checksumming — GPU4FS Integration

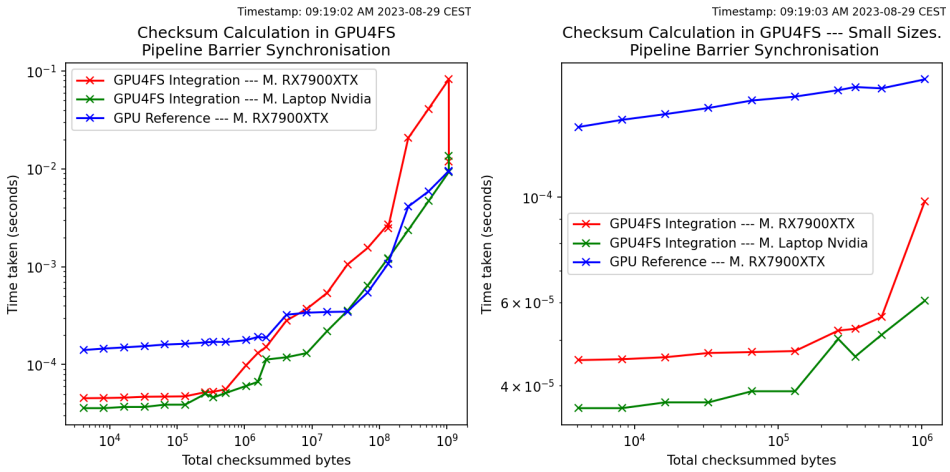
Given the results from Section 6.2.1, we can evaluate our GPU4FS integration of BLAKE3. The GPU4FS integration of BLAKE3 does not need the capabilities to checksum arbitrary input sizes — we only need to support the five GPU4FS page sizes. Thus, the GPU4FS integration of BLAKE3 provides optimized variants for some of those page sizes.

As presented in Section 5.1.2, the GPU4FS integration of BLAKE3 has a RAW dependency coming from the block pointer shader. We proposed two ways of resolving those conflicts in Section 5.3; the Vulkan Approach as in Section 5.3.1 and the GPU4FS approach as in Section 5.3.2. We present only the former approach in the following. Our decision relies on the fact that the GPU4FS approach was designed for out-of-order execution of the overall GPU4FS pipeline. Regarding Section 5.3.2, this forces bypassing compute-unit local caches, which are fast. As we benchmark one shader in isolation and not the overall pipeline, we do not benefit from out-of-order execution. Our isolated measurements do not include the Meta and Self shaders as their performance is, based on their sizes, equivalent to the small page shader.

To ensure correctness of the stored checksums, we walked the file system on the CPU and compared the checksum block's values with their expected values. This validation process was enabled up to the maximum amount of currently supported indirect blocks within a file. Larger files could not be stored; we therefore could not validate their stored checksums.

Checksumming with the Vulkan Approach

Figure 6.4 depicts the checksum integration with the Vulkan Approach as synchronization mechanism. Regarding Figure 5.3 and its depicted checksumming shaders S1 and S2, we started $\lceil \#kilo\_pages \div 64 \rceil$ workgroups of the S1 variant for checksumming small pages, 32 workgroups of the S1 variant for checksumming large pages, and 256 workgroups of the S1 variant for checksumming huge pages. While the amount of small page dispatches ensures that each workgroup needs at most one processing iteration, the amount of large and huge page dispatches are a trade-off



(a) All sizes (b) Small sizes only

Figure 6.4: Comparison of our BLAKE3 algorithm vs. the adapted GPU4FS integration of our BLAKE3 algorithm. We observe that the page-optimized integration performs better than our reference implementation for small pages, and on machine "Laptop Nvidia" also for large pages. Additionally, machine "RX7900XTX" shows strange behavior for large pages.

between performance and overhead — especially on machine "RX7900XTX". The large and huge calculations involve one workgroup performing the S2 shader each.

The results indicate clearly that checksumming all kinds of small pages, up to the theoretical maximum amount of small pages within a file, is faster than our GPU reference implementation on machine "RX7900XTX". We chose machine "RX7900XTX" for reference as it hosts the fastest of our GPUs. Notice that the GPU reference implementation checksums a growing input size and thus has a growing, dependent Merkle-Tree, while our GPU4FS integration must only checksum pages independently of each other. Especially Figure 6.4b details the differences between those two situations: While the execution time of the GPU4FS integration stays roughly the same up to 131 KiB input size, the GPU reference implementation grows. The latter rise in the integration's execution time (Figure 6.4b) originates from the growing pressure on the memory bus, which we validated by removing the coherency-guaranteeing pipeline barrier before writing our last timestamp.

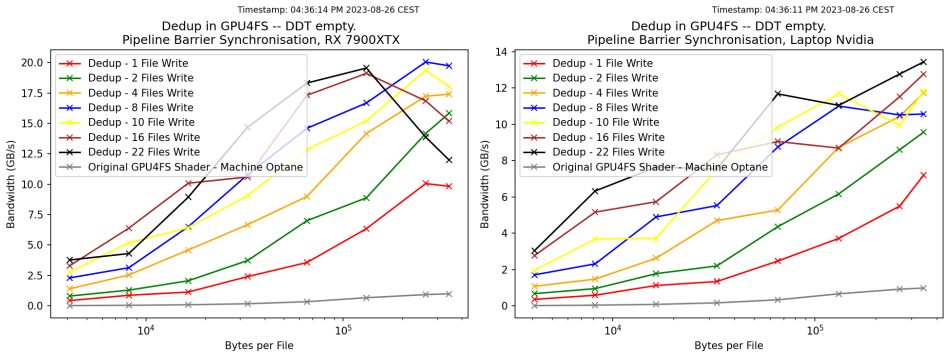
The measurements on machine "Laptop Nvidia" scale with especially large sizes (Figure 6.4a), which indicates that our algorithm operates efficiently on all potential combinations of input sizes. We also benchmarked the maximum amount of large pages possible within one file, which show no statistical outliers

on machine “Laptop Nvidia”. However, machine “RX7900XTX” shows divergent behavior which is not congruent to our measurements on machine “Laptop Nvidia”. Even when allowing more than 32 workgroups for processing large pages, we were not able to achieve the same performance as machine “Laptop Nvidia”. As the GPU of machine “RX7900XTX” consists of 96 CUs while the GPU of machine “Laptop Nvidia” consists of 20 CUs, this must originate from a combination of the overall system configuration of machine “RX7900XTX” and Vulkan-related issues. In fact, we visualized the PCI tree of machine “RX7900XTX” via `sudo lspci -vvv -t`. After finding a PCI chain with two switches in between, we observed that some of those switches reported a configured speed of PCIe 1.0 with a width of 1 lane. This might be the cause for our issue. Unfortunately, we do not have another test system available. Although the PCIe bus and RAM are minorly involved during the checksum calculation, we can hardly imagine that the huge differences originate from their different GPU architectures. We mainly based this assumption on the hardware-superiority of machine “RX7900XTX’s” GPU over machine “Laptop Nvidia’s” GPU and the reference performance of machine “RX7900XTX”.

However, we addressed our observations by measuring the Vulkan approach on another machine which is PCIe 4.0 ready and provides 64 GB of RAM, but has a slower AMD RX6800XT GPU. This machine shows the same linear asymptotic performance as machine “Laptop Nvidia” when checksumming up to 511 large pages. In combination with our results from machine “Laptop Nvidia”, we conclude that our algorithm performs well and suffers from the configuration of machine “RX7900XTX”.

Takeaway

We conclude that our checksum integration provides a high throughput, especially for large sizes. Additionally, its performance is higher than our GPU reference implementation within small sizes. However, we observed that machine “RX7900XTX” seems to be influenced by the amount of parallel large page calculations, especially as its integration performs worse than our results from the reference implementation as soon as it crosses the 8 MiB mark. Nevertheless, its performance for huge pages is as good as its reference implementation. In combination with the results from machine “Laptop Nvidia”, we assume that those issues come from the system configuration.



(a) Dedup with empty DDT on Machine RX7900XTX

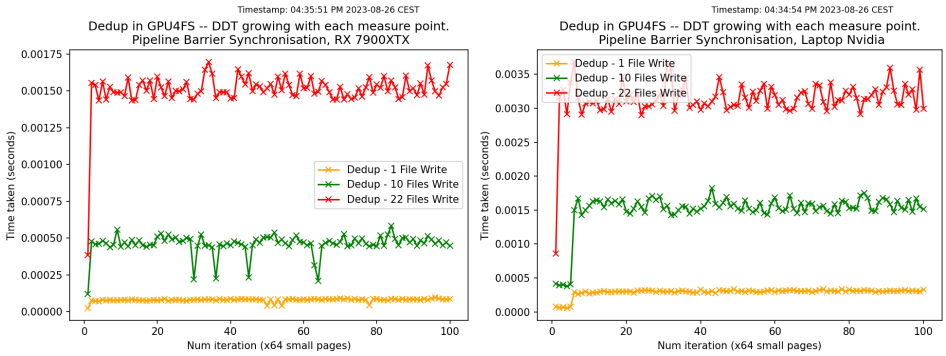
(b) Dedup with empty DDT on Machine Laptop Nvidia

Figure 6.5: Deduplication process with empty DDT; isolated measurement. The DDT performance is always higher than the original GPU4FS shader. Depending on the used GPU, it can sustain higher load with a growing throughput.

6.3 Deduplication

This section details our measurements for the isolated deduplication process. We initially measure the deduplication with an empty DDT and compare those results with a growing DDT afterwards. Both measurements involve randomized values for each measured amount of bytes. To ensure the correctness of our DDT approach, we validated all stored block pointers against their expected block pointers. Additionally, we implemented a DDT visualizer, which is able to visualize the DDT graphically. While the former ensures the correctness of the overall deduplication, the latter lets the user check if the DDT was built correctly. We benchmark the Vulkan approach only, given the same arguments as in Section 6.2: The GPU4FS approach is optimized for interleaving commands and out-of-order execution, from which the isolated benchmark does not benefit. Figure 6.5 shows the deduplication process isolated, starting with an empty DDT. The figures show the original GPU4FS shader performance for reference.

We observe that our deduplication process is capable of delivering higher speeds than the original GPU4FS shader. Additionally, the deduplication bandwidth increases with the amount of files to process up to a certain level: As each workgroup can process 8 pages in parallel without one inner iteration, the machine “Laptop Nvidia” is capable of processing a maximum of 160 pages and the machine “RX7900XTX” a maximum of 768 pages without iterating once — if we expect an optimal scheduling. Thus, “Laptop Nvidia” (Figure 6.5b) shows more variance within its results, as we exceed the maximum amount of workgroups much faster



(a) Machine “RX7900XTX” (b) Machine “Laptop Nvidia”

Figure 6.6: Deduplication process with growing DDT, up to 140,800 pages. The performance stays the same with a growing DDT.

than on machine “RX7900XTX”. However, machine “RX7900XTX” (Figure 6.5a) also looses performance after exceeding its capabilities. Within deduplication, there is no communication with the local RAM over the PCIe bus. This explains why we do not face any memory-limiting drops as in Section 6.2.2. Additionally, the deduplication process does not involve much branching when the DDT is empty. Therefore, the theoretically faster machine “RX7900XTX” outperforms the “Laptop Nvidia” GPU — which is what we expected.

Growing DDT

However, the more important performance indicator is the deduplication performance over time, as the DDT grows. To build a realistic scenario, we inserted more and more DDT entries into the table, and measured the time an insertion needs consecutively. Figure 6.6 depicts our measurements. We compare one, ten, and 22 file writes, with 64 small pages each. Thus, the highest number of DDT entries during our depicted measurements is 140,800. We even benchmarked 2000 contiguous insertions on machine “RX7900XTX”, which showed the same bandwidth as our measurements for 100 insertions.

As Figure 6.6 shows, we measure a steady deduplication performance with a low variance. The measured variance is not surprising, as we utilize spinlocks to synchronize the DDT accesses and insert randomized values into the table. Here, the decision from Section 5.2.2 shows its benefits: As we employ full subgroups only, we ensure that there are no divergent branches during the overall deduplication. Thus, the performance does not drop significantly with a growing DDT, although it does depend on the amount of files and the fairness of the scheduler. Those arguments also address the differences between machine “RX7900XTX”

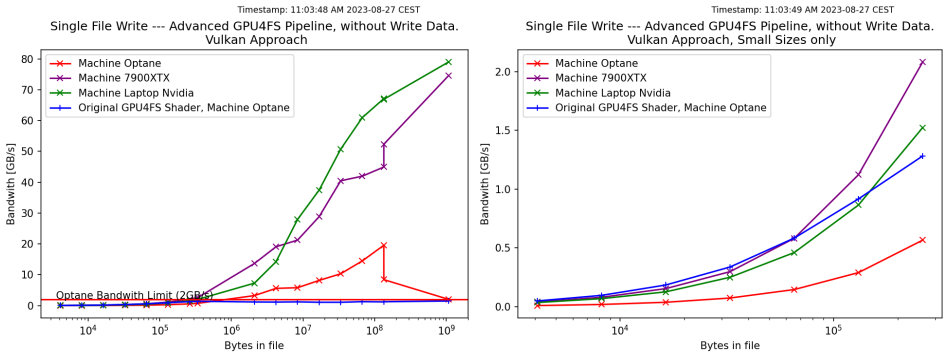
and machine "Laptop Nvidia", when it comes to different file sizes: As machine RX7900XTX consists of 96 Compute Units, which are capable of processing four SIMD32 waves [79] each, it can process up to $4 \cdot 96 = 384$ subgroups truly parallel. Thus, inserting one file (which consists of 64 small pages, processed by eight subgroups and thus one workgroup) produces a much lower difference between those two machines than inserting ten or 22 of those files in parallel.

Nevertheless, we observe an interesting behavior in the left values of both diagrams. The leftmost value of all measurements is lower, which is a bit odd. The first run clears the DDT, whereas subsequent runs do not clear the DDT. This is the only difference between those runs, which hints to a Vulkan-related issue with the memory allocator. However, that behavior is not solely originating from Vulkan-internal problems. As our implementation does not implement the directory from Section 4.2.1, a growing DDT implies more pointer chasing operations. Thus, an integration of the directory would speed-up the DDT process. However, pointer chasing alone does not explain why the first run is that much lower, compared to the second run. For isolating the issue's cause, we modified the heaviest of our benchmarks the following way: Instead of inserting 22 random files, we introduced a case distinction. Each "even" run of the benchmark inserts random files with random blocks, while each "uneven" run of the benchmark inserts the same file containing 64 similar blocks each. This benchmark showed that the deduplication process of the uneven runs took longer, with nearly the double amount of execution time. Thus, we conclude that our locking mechanism is a bottleneck. As our locking mechanism locks each page of the table exclusively, regardless if it is accessed read-only or read/write, the insertion of 22 files with 64 equivalent blocks has less parallelization potential. To solve this issue, we suggest upgrading our lock to a reader/writer lock.

Another interesting observation we made is that a larger DDT buffer influences the performance: With a bigger DDT buffer, the execution time is constantly higher, but shows the same asymptotic execution time as in Figure 6.6.

Takeaway

We saw that our deduplication process's performance was steady in a real-world scenario. Even 22 parallel file deduplications with 64 small-page-sized blocks each provided a steady throughput of around 3.8 GB/s on machine "RX7900XTX", which is nearly double the maximum bandwidth of Intel Optane. Furthermore, deduplicating one file of that size provides a throughput of around 3.31 GB/s on machine "RX7900XTX", which is again above the maximum bandwidth of Intel Optane. Those numbers are a lower limit, as the dedup's performance does not depend on the block sizes. However, we found that the simple lock has been a bottleneck that, according to our results, would be solved with a reader/writer lock.



(a) Measuring all sizes (b) Zoomed view on small sizes

Figure 6.7: Single file write of our advanced GPU4FS pipeline, depicted in Figure 5.7 — without write-data shader stage, utilizing Vulkan approach for synchronization. The pipeline outperforms the original GPU4FS shader at around 128 KiB–2 MiB.

6.4 Shader Pipeline

Although we measured our two features independently of each other, it is important to consider the overall performance of our shader pipeline. This section compares our shader pipeline with the results from GPU4FS's implementation prior to our changes [10].

We evaluate both the Vulkan approach and the GPU4FS approach, as presented in Section 5.3. Both approaches are initially evaluated without the write-data shader, to gain insights into the overall pipeline performance. Afterwards, an evaluation with the write-data shader follows. We then summarize implications for GPU4FS and give recommendations for further development.

6.4.1 The Vulkan Approach

This section presents the evaluation of the Vulkan approach, initially without and afterwards with activated write-data shader.

Without Write-Data Shader

Figure 6.7 details the performance of the Vulkan approach without writing actual file data. The only written data consists of checksums, inodes, indirect blocks, and directory entries — as those values where written prior to the write-data shader.

We observe that our implemented pipeline outperforms the Intel Optane bandwidth, depending on the used machine, at about 128 KiB to 2 MiB. The performance of machine "Laptop Nvidia" does not drop afterwards — it rises up to the

largest measured size. Machine “RX7900XTX” has a lower maximum throughput than machine “Laptop Nvidia”. This is not surprising, based on the results from Section 6.2.2. However, machine “Optane” performs poorly in terms of throughput, and does not show comparable results — although its GPU is theoretically faster than the GPU of machine “Laptop Nvidia”. Given the fact that two machines with two completely different GPUs perform significantly better lets us conclude that machine “Optane” suffers from the overall configuration of PCIe 3.0 slot, slow RAM, and old CPU — from which the first two facts also apply to machine “RX7900XTX”. However, all three GPUs outperform Intel Optane’s performance by far when crossing the 2 MiB mark.

As Intel Optane shines with low latency [13], it is important to take a closer look on the pipeline’s small page performance, as seen in Figure 6.7b. There, we observe that the Vulkan approach can compete with the original shader — at least on machine “RX7900XTX”. Machine “Laptop Nvidia” is slower than machine “RX7900XTX”, which stems from the overall higher load induced by the complete pipeline: The latency of scheduling the different tasks onto the fewer compute units with the Vulkan synchronization of multiple compute commands manifests in a lower throughput.

To classify our results, we measured the original GPU4FS shader on machine “Optane”, although machine “Optane” showed strange behavior within the complete evaluation process. This has two major reasons: Only machine “Optane” is, obviously, Intel Optane ready. Thus, machine “RX7900XTX” and “Laptop Nvidia” would not give any implications for comparing the write throughput on Intel Optane. Additionally, our pipeline puts high pressure on the overall system as well as the PCIe bus. The original GPU4FS shader has only three mapped buffers, one dispatched shader, and its most complex task is writing bytes onto NVM. Therefore, the original shader does not show the same strange behavior — although we do not reach the measured bandwidth limits from Maucher [10].

With Write-Data Shader

After evaluating the pipeline without writing any data to drive, we enable the write-data shader and run the same benchmark. Our measurements are depicted in Figure 6.8. Unsurprisingly, we observe a slowdown, especially for large pages. Machine “Optane” again performs poorly in terms of throughput, which was already observed in Figure 6.7. However, machine “RX7900XTX” and machine “Laptop Nvidia” are less influenced by the write data shader within small sizes. They start to slow down after the 2 MiB mark. This was expected, as both machine “RX7900XTX” and machine “Laptop Nvidia” write into DRAM with as many workgroups as there are pages. However, as soon as the benchmark switches to large pages, the write-data shader writes 2 MiB instead of 4 KiB per workgroup.

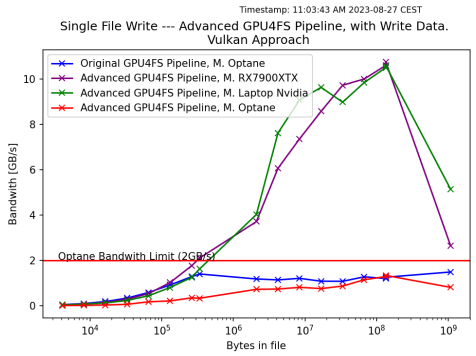


Figure 6.8: Single file write of our GPU4FS pipeline, depicted in Figure 5.7 — with write-data shader. Utilizing Vulkan approach for synchronization. The throughput drops in comparison to Figure 6.7, especially for large pages.

6.4.2 The GPU4FS Approach

Complementary to the Vulkan approach (Section 5.3.1), we present the results of the GPU4FS approach (Section 5.3.2) in the following section.

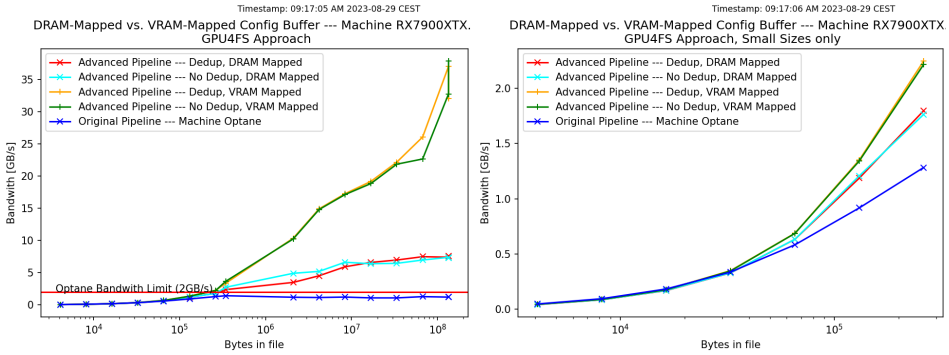
Towards PCIe Limitations

Our presented results tried to mitigate all limitations which do not originate from our design. This ensures that we benchmarked the overall design without being influenced by limitations that are not in our control. However, as the following section is our recommendation for further research, we compare the GPU4FS config buffer VRAM- vs. DRAM-mapped. Figure 6.9 depicts those results. We observe that a VRAM-mapped config buffer is highly favorable, especially when it comes to larger file sizes. This is not surprising, as the PCIe 3.0 bus of machine "RX7900XTX" significantly slows the pipeline down. However, mapping the command buffer on VRAM violates the original goal of GPU4FS, as it introduces copy operations on the CPU. This violation is easily resolved by utilizing GPU-side caches, as proposed in the original design [10].

Without Write-Data Shader

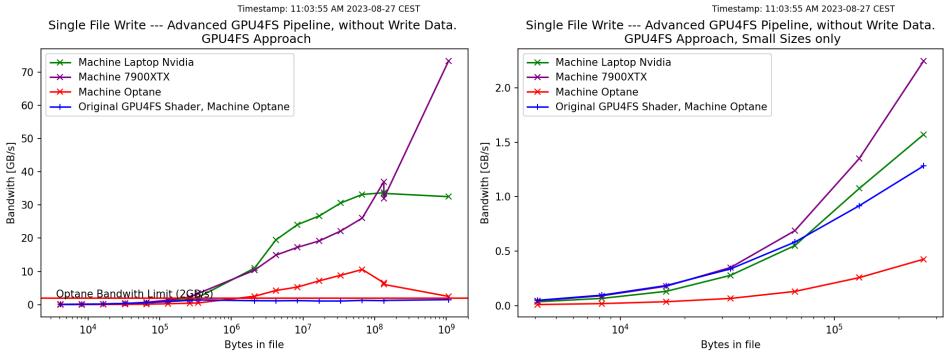
Figure 6.10 shows major differences in comparison to Section 6.4.1.

Machine "RX7900XTX" is always faster than or as fast as the original GPU4FS shader. Although the Vulkan approach showed this behavior already for large sizes (Figure 6.7a), the GPU4FS approach is additionally faster within small sizes on both machine "RX7900XTX" and machine "Laptop Nvidia" (Figure 6.10b). This



(a) Measuring all sizes (b) Zoomed view on small sizes

Figure 6.9: Advanced GPU4FS Pipeline without write-back, comparing config buffer location DRAM-mapped vs VRAM-mapped. The VRAM-mapped variant performs better than its DRAM-mapped counterpart.



(a) Measuring all sizes (b) Zoomed view on small sizes

Figure 6.10: Single file write of our advanced GPU4FS pipeline, depicted in Figure 5.7 — without write-data shader stage. Utilizing GPU4FS approach for synchronization. Machine “Laptop Nvidia” and machine “RX7900XTX” are faster than the Vulkan approach (Figure 6.7b) at small sizes.

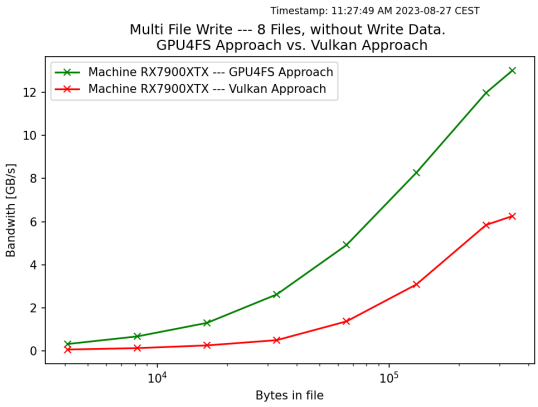
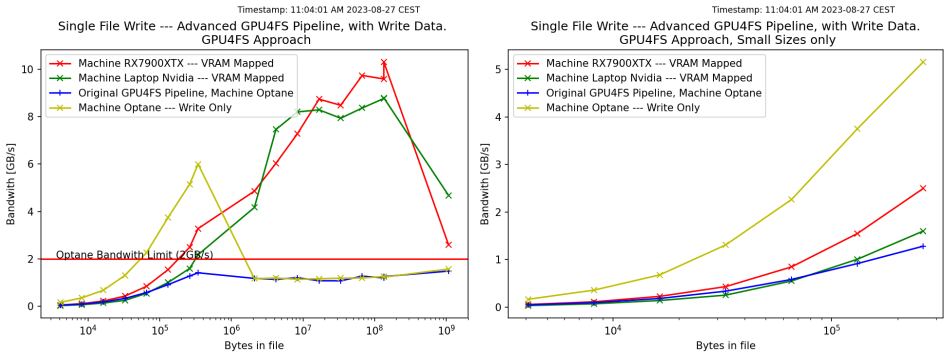


Figure 6.11: Multi file write (8 files) of our advanced GPU4FS pipeline — without write-data shader. Comparing GPU4FS approach with Vulkan approach. We observe that pipelining and out-of-order execution is favorable.

observation is emphasized within Figure 6.11: Writing multiple files benefits from a pipelined out-of-order execution, especially when it comes to smaller sizes. Contrary to the Vulkan approach, the buffers must not be synchronized over the whole system, but rely on consistency within VRAM [99, 135]. This mitigates PCIe and DRAM latency, which is another explanation for the higher throughput within small sizes. Additionally, we observe that the throughput of calculating larger sizes is lower with the GPU4FS approach (Figure 6.10a) than with the Vulkan approach (Figure 6.7a). This behavior must originate from the volatile-mapped buffers and the concurring nature of the GPU4FS approach.

With Write-Data Shader

To conclude our measurements, we present the GPU4FS approach with enabled write-data shader in Figure 6.12a. We compare those results with the original GPU4FS shader, and include a measurement of the maximum Optane throughput at a given size with disabled advanced features (yellow line). This maximum throughput is reached by using only the block pointer and the write-data shader, with the maximum amount of writing workgroups. We observe that our pipeline performs well, especially for large page sizes (Figure 6.12a). However, the maximum throughput of the Vulkan approach is not reached, which comes from the volatile-mapped buffers and concurring nature of the GPU4FS approach. Machine "Optane" tends to benefit from the pipelining of small page writes (Figure 6.12b, yellow), in comparison to the original implementation which uses only one workgroup for each file write (Figure 6.12b, blue). After rising up to a plateau of around

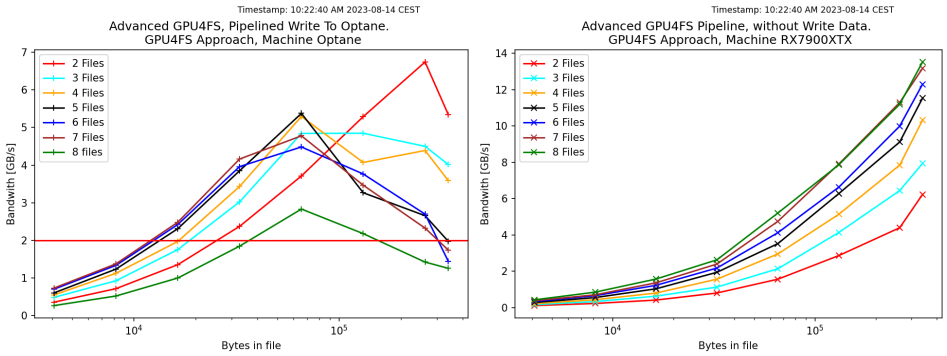


(a) Measuring all sizes (b) Zoomed view on small sizes

Figure 6.12: Single file write of our advanced GPU4FS pipeline, depicted in Figure 5.7 — with write-data shader stage. Utilizing GPU4FS approach for synchronization. The GPU4FS approach is faster than the Vulkan approach at small sizes (Figure 6.8), while Optane shows an unexpected high throughput for small sizes.

6 GB/s, the raw write performance drops dramatically down to around 1–1.5 GB/s. This could be caused by both a saturated Optane DIMM or a saturated GPU, although a saturated Optane DIMM is more likely [13]. However, after dropping once, the performance of the Optane DIMM stays constant in both pipelining and non-pipelining. This is also the crossover point where our pipeline exceeds the Optane DIMMs performance. It is important to notice that we cannot compare our results directly, as machine “RX7900XTX” does not write to Optane but DRAM. However, the yellow line indicates that at around 2 MiB, Optane is the bottleneck, not our pipeline. Although this is also the point where the write-data shader writes 2 MiB instead of 4 KiB per workgroup, we came to the former conclusion, as this is the expected maximum bandwidth for Optane [13].

Considering Figure 6.12b, we observe that the bottleneck for dealing with small files are the advanced features. However, with respect to Section 6.2.2, our benchmark results are influenced by many factors: Starting the pipeline needs some time, as we benchmark the setup process within the shaders. Additionally, we face unfair scheduling on the GPU and have no way of controlling the shader execution. Thus, spawning more pipeline stages involves spawning more spinlocking threads, which can influence our results. Thirdly, we had to bypass the lower level caches by setting our buffers volatile, and thus miss more performance. Furthermore, we faced some Optane-related uncertainties, which we detail in Section 6.4.3.



(a) Writing to Optane (b) GPU4FS pipeline

Figure 6.13: File write with an increasing amount of files, comparing Optane's throughput with the GPU4FS pipeline's throughput. With a growing pressure, the pipeline reaches a higher throughput. In parallel, Optane's throughput reduces with a higher load.

6.4.3 Towards Low Pressure

Our results from the last section imply that Intel Optane provides a much higher throughput when it is under low pressure: The proposed bandwidth limit of 2 GB/s [13, 24] seems to only show up when the DIMM is under higher load. We found comparable results within a benchmark of parallel 4 KiB block writes [13], although it is not clear if its authors tested one or multiple interleaved Optane DIMMs. However, to address our hypothesis, we measured the Optane performance with an increasing load of 4 KB page writes. We compared those results with a multi-file write of our GPU4FS pipeline to find out where the crossover point of "GPU-based parallelization benefiting" and "Intel Optane bandwidth suffering" lies. To ensure that our changes were flushed to Optane, we mapped its address space via `mmap()` [101] with the "MS_SYNC" flag set. Our measurements are depicted in Figure 6.13 and indicate that Optane provides a high throughput when being under low pressure. However, as the pressure increases, Optane starts to drop performance, until it falls below the expected 2 GB/s bandwidth border. We also validated our measurements via a parallel writing implementation on the CPU, where we wanted to employ a `msync()` [136] call to flush the CPU caches to Optane. However, the `msync()` call always responded with an "Invalid Argument" error, which might indicate that the synchronization flag of `mmap()` is also not working properly. Thus, our observations need further research. Our next steps include debugging the `msync()` call as well as the `mmap()` call to see if the CPU flushes actually take place. One possible way to start debugging is by placing Kprobes [137] on the responsible kernel functions.

However, our results from Figure 6.13 indicate that the GPU4FS pipeline performs well under load and gets faster the more it is under pressure. We measured the load with the GPU4FS approach up to an amount of 8 files with 64 MiB each, and saw an increase of throughput with each file until we reached a maximum of 81 GB/s. This lets us conclude that the pipeline design is feasible for a high load.

6.4.4 Vulkan-related Issues

During our evaluation, we found that iterative executions of the pipeline within the same program instance result in different behavior. We covered this behavior already during our "growing DDT" test from Section 6.3. Especially machine "Laptop Nvidia" was influenced by an iterative run, reducing its throughput to a maximum of 4 (GPU4FS approach) to 9 (Vulkan approach) GB/s. However, machine "RX7900XTX" was majorly influenced by an iterative version of the small size benchmarks, which manifested in factor-3-worse results. Interestingly, machine "RX7900XTX" showed consistent results in the field of 2–64 MiB, but lost throughput to a maximum of 8 (GPU4FS approach) to 12 (Vulkan approach) GiB/s at 128 MiB. An explanation for these numbers is the L3 cache of its GPU [138], which might mitigate Vulkan-related overhead on VRAM. Overall, those observations let us conclude that the setup process influences the execution time, especially during subsequent runs in the same program instance. However, as GPU4FS is thought as a daemon, it must absolve the setup process only once. Although the iterative benchmarks showed lower throughput, they still crossed Optane's bandwidth limit at around 2 MiB of input size and thus imply the same qualitative results. However, those Vulkan-related issues need further investigation and validation, for which we suggest porting our implementation to another GPGPU API.

6.5 Discussion

After evaluating our implemented features, we found several implications for further GPU4FS development. This section summarizes our takeaways and gives further implications for the overall file system.

The evaluation of our checksumming approach leads to several conclusions: Processing large pages is not bottlenecked by our checksumming procedure. We showed that the approach scales with the amount of blocks to process and outperforms Optane by far when checksumming large or huge pages. The small page checksumming is bottlenecked by the checksumming speed of one checksum, if the `mmap()` [101] call works as assumed. However, we optimized the GLSL variant of small pages as much as possible. It faces no divergent branches and SIMD parallelization across all 256 lanes. Thus, the only way to potentially gain

more performance lies in porting the algorithm to a computational-optimized API like AMD HIP [139].

During its evaluation, the deduplication process showed consistent performance with a growing DDT. We observed that the throughput of the deduplication process stays consistent with an increasing amount of parallely processed blocks. Additionally, we identified the locking mechanism to be one bandwidth-limiting factor in some special cases, although we always observed a higher throughput than Intel Optane's maximum bandwidth. When it comes to the GPU4FS integration, deduplication delays the write-data shader's execution. It needs to wait for the checksum calculation before making a decision. Thus, the write-data shader also needs to wait for the checksumming and even deduplication process before it can start with its task. Compared to a raw checksumming implementation where the write-data shader starts immediately with its task, this costs additional performance. However, we encourage future work on doing speculative write backs to Optane, which may be interrupted and canceled by the deduplication shader.

Our measurements found that pipelining within GPU4FS is beneficial. However, given the restrictions from Vulkan and GLSL, we had to reimplement many techniques in a relatively hacky way. Our implementation faces bypassed caches, busy-waiting, and unfair GPU scheduling, which puts high pressure onto GPUs. However, our results showed that a pipelined execution is not just theoretically interesting. We found that many CPU-related concepts like preemption and inter-thread communication are required to implement file system features efficiently. Especially the "cooperative groups" feature from CUDA [140] would replace our spinlocks efficiently. Additionally, the unusual configurations of machine "RX7900XTX" and machine "Optane" should be revised to suit their GPU's theoretical capabilities. Furthermore, our observations from Section 6.4.4 encourage the evaluation of our pipeline with an API built solely for GPGPU — as e.g., AMD HIP [139].

6.6 Future Work

After discussing our evaluation results, we give advice for continuing our work. Those were already outlined in Section 5.1.3 and Section 5.2.3, which is why we present a summary of them.

Update and Deletion

Both update and deletion support are not implemented in GPU4FS, and therefore not within our thesis. During their implementation in GPU4FS, we motivate considering our advice for extending checksumming (Section 5.1.3) and deduplication (Section 5.2.3) with those capabilities.

Speculative Write-Back

Within Section 5.3.2 and Section 6.5, we motivated a speculative write-back. It should hide the latency of the deduplication process by writing data to Optane speculatively, with the possibility to get interrupted by the deduplication shader.

6.6.1 Checksumming

Section 5.1.4 outlines our future work suggestions for checksumming. Those include checksum validation during read, RAID functionalities as well as a validation of more checksum algorithms. Our first suggestion depends on a working read path within GPU4FS. Therefore, we motivate implementing the read path.

6.6.2 Deduplication

Our deduplication process is extensible with our suggestions from Section 5.2.4. They include two paging approaches for exchanging DDT entries with PMem, indirect block deduplication, and checksums for DDT entries. Additionally, we provide advice for the file system's unmounting and remounting procedures. Moreover, we motivated an extension from fixed-size chunking to variable-sized chunking in Section 4.2.2. Furthermore, we addressed the boundary-shift problem within GPU4FS's fixed-size chunking in Section 5.2.3.

During our evaluation, we found that the integration of an additional directory as described in Section 4.2.1 as well as a reader/writer lock provide performance benefits. Additionally, the directory structure extends the potential paging mechanisms, as it makes it feasible to exchange DDT entries directly with PMem without loading the DDT tree into VRAM. Therefore, we suggest an evaluation of both approaches.

Chapter 7

Conclusion

In this thesis, we extended the NVM file system GPU4FS with two advanced file system features: Checksumming and Deduplication. Our goal was to find out if the GPU is suitable for modern file system tasks, while reducing as much load on the CPU as possible. Design-related goals were little additional overhead as well as a high degree of parallelization and overall performance. One checksum-specific goal was to decouple the file system design from the used checksum algorithm.

Our checksumming design is able to utilize different checksum algorithms. After fetching a block from drive, the design immediately exposes its according checksum blocks without the need to follow a pointer chain. The design is furthermore extendible to larger indirect blocks, while giving the ability to reduce checksum-related overhead even further by allocating larger checksum blocks.

The deduplication design introduces only 64 bytes of overhead per stored block into the file system. Its design follows the extendible hashing strategy, which utilizes a radix tree to organize DDT entries efficiently. The extendible hashing strategy allows saving as many sequential pointer chasing operations as possible, while providing a fine-granular lock on SIMD-level to ensure the tree's consistency.

Finally, we conclude that our implementation performs great for large and huge pages, and sustains its performance under growing load. Although the new Optane numbers imply that the performance of a "small page, few files" insertion is dominated by the overall pipeline, we still believe that the pipeline's performance is enough to saturate most desktop systems — especially with appropriate caching strategies for small transactions. However, those small, few file transactions do not put much pressure on the CPU. Thus, we conclude that GPU acceleration is most sensible when there is moderate to high load to process. Smaller transactions can be handled by the CPU to saturate Optane's bandwidth. Thus, a GPU4FS hybrid mode might be the option of choice, which combines both high Optane throughput and less stalling CPU cores.

Bibliography

[1] R.R. Schaller. "Moore's law: past, present and future." In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665.

[2] Balaji Venu. *Multi-core processors - An overview*. 2011. arXiv: 1110.3535 [cs.AR].

[3] Michael J Flynn. *Flynn's Taxonomy.* 2011.

[4] AMD. *AMD Ryzen 7000-Serie Desktop-Prozessoren*. 2023. URL: https://www.amd.com/de/processors/ryzen#Ryzen%E2%84%A2-7000 (visited on 06/10/2023).

[5] Axel Habermaier and Alexander Knapp. "On the correctness of the SIMT execution model of GPUs." In: *Programming Languages and Systems: 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24-April 1, 2012. Proceedings 21*. Springer. 2012, pp. 316–335.

[6] Pedro Trancoso and Maria Charalambous. "Exploring graphics processor performance for general purpose applications." In: *8th Euromicro Conference on Digital System Design (DSD'05)*. IEEE. 2005, pp. 306–313.

[7] Fawad Murtaza. *AMD Compute Units vs. Nvidia CUDA Cores: What's the Difference?* 2021. URL: https://www.makeuseof.com/compute-units-vs-cuda-cores-whats-the-difference/ (visited on 06/10/2023).

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

[9] Jeff Bonwick et al. "The zettabyte file system." In: *Proc. of the 2nd Usenix Conference on File and Storage Technologies*. Vol. 215. 2003.

[10] Peter Maucher. "GPU4FS: A Graphics Processor-Accelerated File System." In: (2022).

[11] Intel. *Intel Optane Memory*. 2023. URL: https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html (visited on 07/20/2023).

[12] Ivy B Peng, Maya B Gokhale, and Eric W Green. "System evaluation of the intel optane byte-addressable nvm." In: *Proceedings of the International Symposium on Memory Systems*. 2019, pp. 304–315.

[13] Joseph Izraelevitz et al. "Basic performance measurements of the intel optane DC persistent memory module." In: *arXiv preprint arXiv:1903.05714* (2019).

[14] Khronos Group. *Khronos Vullkan Registry*. 2023. URL: https://registry.khronos.org/vulkan/ (visited on 05/20/2023).

[15] Vamsee Kasavajhala. "Solid state drive vs. hard disk drive price and performance study." In: *Proc. Dell Tech. White Paper* (2011), pp. 8–9.

[16] SATA-IO Board Members. *Serial ATA International Organization*. URL: https://sata-io.org/system/files/specifications/SerialATA_Revision_3_1_Gold.pdf (visited on 08/20/2023).

[17] Qiumin Xu et al. "Performance analysis of NVMe SSDs and their implication on real world databases." In: *Proceedings of the 8th ACM International Systems and Storage Conference*. 2015, pp. 1–11.

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

[19] Jian Yang et al. "An empirical guide to the behavior and use of scalable persistent memory." In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 2020, pp. 169–182.

[20] Lukas Werling, Christian Schwarz, and Frank Bellosa. *Towards Less CPU-Intensive PMEM File Systems*. 2021. URL: https://www.betriebssysteme.org/wp-content/uploads/2021/09/FGBS_Herbst2021_Folien_Werling.pdf (visited on 08/06/2023).

[21] Man pages. *tmpfs*. URL: https://man7.org/linux/man-pages/man5/tmpfs.5.html (visited on 07/11/2023).

[22] PASC. *The Open Group Base Specifications Issue 7, 2018 edition*. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/ (visited on 08/17/2023).

[23] Avantika Mathur et al. “The new ext4 filesystem: current status and future plans.” In: *Proceedings of the Linux symposium*. Vol. 2. Citeseer. 2007, pp. 21–33.

[24] Peter Maucher. *GPU4FS-Code*. 2022. URL: `https://git.scc.kit.edu/itec-os/research/maucher/gpu4fs-code` (visited on 05/29/2023).

[25] Various authors. *FAT*. URL: `https://www.win.tue.nl/~aeb/linux/fs/fat/fat-1.html` (visited on 07/11/2023).

[26] Haris Volos et al. “Aerie: Flexible file-system interfaces to storage-class memory.” In: *Proceedings of the Ninth European Conference on Computer Systems*. 2014, pp. 1–14.

[27] Youngjin Kwon et al. “Strata: A cross media file system.” In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 460–477.

[28] Rohan Kadekodi et al. “SplitFS: Reducing software overhead in file systems for persistent memory.” In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 494–508.

[29] Chris Borrelli. “IEEE 802.3 cyclic redundancy check.” In: *application note: Virtex Series and Virtex-II Family, XAPP209 (v1. 0)* (2001).

[30] Theresa Maxino. “Revisiting fletcher and adler checksums.” In: (2006).

[31] Ryan Mangipano. *Bit Flips: Was That a Zero or a One?* Dec. 15, 2009. URL: `https://www.itprotoday.com/cloud-computing/bit-flips-was-zero-or-one` (visited on 08/23/2023).

[32] Bart Preneel. “Cryptographic hash functions.” In: *European Transactions on Telecommunications* 5.4 (1994), pp. 431–448.

[33] Tenkasi V Ramabadran and Sunil S Gaitonde. “A tutorial on CRC computations.” In: *IEEE micro* 8.4 (1988), pp. 62–75.

[34] Craig Partridge, Jim Hughes, and Jonathan Stone. “Performance of checksums and CRCs over real data.” In: *ACM SIGCOMM Computer Communication Review* 25.4 (1995), pp. 68–76.

[35] John Fletcher. “An arithmetic checksum for serial transmissions.” In: *IEEE transactions on Communications* 30.1 (1982), pp. 247–252.

[36] Rajeev Sobti and Ganesan Geetha. “Cryptographic hash functions: a review.” In: *International Journal of Computer Science Issues (IJCSI)* 9.2 (2012), p. 461.

[37] I Damgard, Collision-Free Hash Functions, and Public-Key Signature Schemes. *EuroCrypt 87, LNCS, Vol. 304*. 1988.

[38] Ralph C Merkle. "One way hash functions and DES." In: *Advances in Cryptologyâ€"CRYPTOâ€™89 Proceedings*. Springer. 2001, pp. 428–446.

[39] Ralph Charles Merkle. *Secrecy, authentication, and public key systems.* Stanford university, 1979.

[40] Ralph C Merkle. "One way hash functions and DES." In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 428–446.

[41] Ivan Damgard. "A design principle for hash functions." In: *CRYPTO 1989* (1990), pp. 416–427.

[42] Ronald Rivest. *RFC1321: The MD5 message-digest algorithm*. 1992.

[43] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 180-4. Washington, D.C.: U.S. Department of Commerce, 2015. DOI: 10.6028/NIST.FIPS.180-4.

[44] Antoine Joux. "Multicollisions in iterated hash functions. Application to cascaded constructions." In: *Advances in Cryptology–CRYPTO 2004: 24th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 2004. Proceedings 24*. Springer. 2004, pp. 306–316.

[45] John Kelsey and Tadayoshi Kohno. "Herding hash functions and the Nostradamus attack." In: *Advances in Cryptology-EUROCRYPT 2006: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28-June 1, 2006. Proceedings 25*. Springer. 2006, pp. 183–200.

[46] Yevgeniy Dodis, Thomas Ristenpart, and Thomas Shrimpton. "Salvaging Merkle-Damgård for practical applications." In: *Advances in Cryptology-EUROCRYPT 2009: 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings 28*. Springer. 2009, pp. 371–388.

[47] Eli Biham and Orr Dunkelman. "A framework for iterative hash functions-HAIFA." In: *Cryptology ePrint Archive* (2007).

[48] Guido Bertoni et al. "Cryptographic sponges." In: *online] http://sponge. noekeon. org* (2011).

[49] Jack O'Connor et al. *Blake3 - One function, fast everywhere*. 2021. URL: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf (visited on 05/10/2023).

[50] Jean-Philippe Aumasson et al. "BLAKE2: simpler, smaller, fast as MD5." In: *Applied Cryptography and Network Security: 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings 11*. Springer. 2013, pp. 119–135.

[51] Jean-Philippe Aumasson. "Too much crypto." In: *Cryptology ePrint Archive* (2019).

[52] Ralph C Merkle. "A digital signature based on a conventional encryption function." In: *Advances in Cryptologyâ€"CRYPTOâ€™87: Proceedings 7*. Springer. 1988, pp. 369–378.

[53] OpenMP. *The OpenMP API specification for parallel programming*. URL: https://www.openmp.org/ (visited on 08/17/2023).

[54] Rayon-RS. *Crate rayon*. URL: https://docs.rs/rayon/latest/rayon/ (visited on 08/17/2023).

[55] Guido Bertoni et al. "Keccak sponge function family main document." In: *Submission to NIST (Round 2)* 3.30 (2009), pp. 320–337.

[56] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 202. Washington, D.C.: U.S. Department of Commerce, 2015. DOI: 10.6028/NIST.FIPS.202.

[57] Jack O'Connor et al. *Blake3 - One function, fast everywhere*. 2022. URL: https://github.com/BLAKE3-team/BLAKE3 (visited on 05/10/2023).

[58] Dutch T Meyer and William J Bolosky. "A study of practical deduplication." In: *ACM Transactions on Storage (ToS)* 7.4 (2012), pp. 1–20.

[59] Qinlu He, Zhanhuai Li, and Xiao Zhang. "Data deduplication techniques." In: *2010 international conference on future information technology and management engineering*. Vol. 1. IEEE. 2010, pp. 430–433.

[60] Wen Xia et al. "A comprehensive study of the past, present, and future of data deduplication." In: *Proceedings of the IEEE* 104.9 (2016), pp. 1681–1710.

[61] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. "Side channels in cloud services: Deduplication in cloud storage." In: *IEEE Security & Privacy* 8.6 (2010), pp. 40–47.

[62] Restic. *Restic*. 2023. URL: https://restic.net/ (visited on 08/05/2023).

[63] Matthew Miller. *Objective Review: Immutable variants are the majority of Fedora Linux in use*. URL: https://discussion.fedoraproject.org/t/objective-review-immutable-variants-are-the-majority-of-fedora-linux-in-use/79288 (visited on 08/05/2023).

[64] Trevor Dunlap, William Enck, and Bradley Reaves. "A Study of Application Sandbox Policies in Linux." In: *Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies*. 2022, pp. 19–30.

[65] Flatpak authors. *Flatpak*. URL: https://flatpak.org/ (visited on 08/05/2023).

[66] Docker authors. *Docker*. URL: https://www.docker.com/ (visited on 08/05/2023).

[67] Jibin Wang et al. "I-sieve: An inline high performance deduplication system used in cloud storage." In: *Tsinghua Science and Technology* 20.1 (2015), pp. 17–27.

[68] Sean Quinlan and Sean Dorward. "Venti: A new approach to archival data storage." In: *Conference on file and storage technologies (FAST 02)*. 2002.

[69] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. "A low-bandwidth network file system." In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 2001, pp. 174–187.

[70] William J Bolosky et al. "Single instance storage in Windows 2000." In: *Proceedings of the 4th USENIX Windows Systems Symposium*. Seattle, WA. 2000, pp. 13–24.

[71] Michael O Rabin. "Fingerprinting by random polynomials." In: *Technical report* (1981).

[72] Wen Xia et al. "FastCDC: A fast and efficient content-defined chunking approach for data deduplication." In: *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 2016, pp. 101–114.

[73] Wen Xia et al. "P-dedupe: Exploiting parallelism in data deduplication system." In: *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*. IEEE. 2012, pp. 338–347.

[74] Samer Al-Kiswany et al. "StoreGPU: exploiting graphics processing units to accelerate distributed storage systems." In: *Proceedings of the 17th international symposium on High performance distributed computing*. 2008, pp. 165–174.

[75] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. "Shredder: GPU-accelerated incremental storage and computation." In: *FAST*. Vol. 14. 2012, p. 14.

[76] Ronald Fagin et al. "Extendible hashing a fast access method for dynamic files." In: *ACM Transactions on Database Systems (TODS)* 4.three (1979), pp. 315–344.

[77] Moohyeon Nam et al. "{Write-Optimized} Dynamic Hashing for Persistent Memory." In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 2019, pp. 31–44.

[78] David Luebke et al. "GPGPU: general-purpose computation on graphics hardware." In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 2006, 208–es.

[79] AMD. *RDNA3 Instruction Set Architecture*. 2023. URL: https://www.amd.com/system/files/TechDocs/rdna3-shader-instruction-set-architecture-feb-2023_0.pdf (visited on 07/31/2023).

[80] Nick Evanson. *GPU Architecture Deep Dive: Nvidia Ada Lovelace, AMD RDNA 3 and Intel Arc Alchemist*. 2023. URL: https://www.techspot.com/article/2570-gpu-architectures-nvidia-intel-amd/ (visited on 07/31/2023).

[81] Mark Wyse. "Understanding GPGPU Vector Register File Usage." In: 2018. URL: https://api.semanticscholar.org/CorpusID:13107618.

[82] David M. Koppelman. *GPU Microarchitecture Note Set 6-Warps and Branch Divergence*. URL: https://www.ece.lsu.edu/koppel/gp/2020/lsli06-br-diverg.pdf (visited on 07/31/2023).

[83] Khronos. *SPIR Overview*. URL: https://www.khronos.org/spir/ (visited on 07/31/2023).

[84] Khronos. *The OpenGLÂ® Shading Language, Version 4.60.7*. URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf (visited on 05/31/2023).

[85] Google. *Shaderc*. URL: https://github.com/google/shaderc (visited on 07/31/2023).

[86] Khronos Group. *OpenGL Overview*. 2023. URL: https://www.khronos.org/opengl/ (visited on 05/20/2023).

[87] Benjamin Kenwright. "Getting Started with Computer Graphics and the Vulkan API." In: *SIGGRAPH Asia 2017 Courses*. SA '17. Bangkok, Thailand: Association for Computing Machinery, 2017. ISBN: 9781450354035. DOI: 10.1145/3134472.3136556. URL: https://doi.org/10.1145/3134472.3136556.

[88] Khronos. *Setup*. URL: https://vulkan-tutorial.com/Drawing_a_triangle/Setup (visited on 07/09/2023).

[89] Khronos Group. *Khronos Vullkan Registry*. 2023. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/ (visited on 07/09/2023).

[90] Khronos Group. *Command Buffers*. 2023. URL: https://registry.khronos.org/vulkan/site/spec/latest/chapters/cmdbuffers.html (visited on 07/09/2023).

[91] Khronos. *Pipelines — Common*. URL: https://registry.khronos.org/vulkan/site/spec/latest/_images/pipelinemesh.svg (visited on 07/03/2023).

[92] Andrew Waterman et al. "The RISC-V instruction set manual." In: *Volume I: User-Level ISAâ€™, version* 2 (2014).

[93] Khronos. *Compute Pipelines*. URL: https://registry.khronos.org/vulkan/site/spec/latest/chapters/pipelines.html#pipelines-compute (visited on 07/03/2023).

[94] Khronos. *Compute Shader*. URL: https://vulkan-tutorial.com/Compute_Shader#page_Compute-pipelines (visited on 07/09/2023).

[95] Khronos Group. *Resource descriptors*. 2023. URL: https://registry.khronos.org/vulkan/site/spec/latest/chapters/descriptorsets.html#descriptorsets-storagebuffer (visited on 07/09/2023).

[96] Khronos Group. *Dispatching Commands*. 2023. URL: https://registry.khronos.org/vulkan/site/spec/latest/chapters/dispatch.html (visited on 07/09/2023).

[97] Khronos Group. *Synchronization and Cache Control*. 2023. URL: https://registry.khronos.org/vulkan/site/spec/latest/chapters/synchronization.html (visited on 07/09/2023).

[98] Khronos. *Memory model*. URL: https://www.khronos.org/opengl/wiki/Memory_Model#Ensuring_visibility (visited on 07/09/2023).

[99] Type Qualifier (GLSL). *Khronos*. 2023. URL: https://www.khronos.org/opengl/wiki/Type_Qualifier_(GLSL) (visited on 07/11/2023).

[100] imported_obfuscator. *Difference between coherent and volatile qualifier*. URL: https://community.khronos.org/t/difference-between-coherent-and-volatile-qualifier/72254 (visited on 08/22/2023).

[101] Michael Kerrisk. *mmap(2) – Linux manual page*. URL: https://man7.org/linux/man-pages/man2/mmap.2.html (visited on 08/20/2023).

[102] Jason Power, Mark D Hill, and David A Wood. "Supporting x86-64 address translation for 100s of GPU lanes." In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2014, pp. 568–578.

[103] Stephen Tweedie. "Ext3, journaling filesystem." In: *Ottawa Linux Symposium*. Vol. 20. 0. Ottawa Congress Centre Ottawa, Ontario, Canada. 2000, p. 0.

[104] Various authors. *Ext4 Design*. 2014. URL: https://ext4.wiki.kernel.org/index.php/Ext4_Design (visited on 06/10/2023).

[105] Various authors. *Ext4 Metadata Checksums*. 2013. URL: https://ext4.wiki.kernel.org/index.php/Ext4_Metadata_Checksums (visited on 05/30/2023).

[106] Usha A Joglekar, Bhushan M Jagtap, and Koninika B Patil. "Deploying De-Duplication on Ext4 File System." In: *International Journal of Engineering Research and Technology* (2014).

[107] Various Btrfs authors. *Deduplication*. URL: https://btrfs.readthedocs.io/en/latest/Deduplication.html (visited on 06/11/2023).

[108] Zygo et al. *BEES*. URL: https://github.com/Zygo/bees (visited on 06/12/2023).

[109] Mark Fasheh et al. *Duperemove*. URL: https://github.com/markfasheh/duperemove (visited on 06/12/2023).

[110] Various contributors. *OpenZFS*. URL: https://github.com/openzfs/zfs/blob/master/include/sys/spa.h (visited on 05/28/2023).

[111] Truenas. *ZFS Deduplication*. 2023. URL: https://www.truenas.com/docs/references/zfsdeduplication/ (visited on 06/20/2023).

[112] OpenZFS. *Workload tuning*. URL: https://openzfs.github.io/openzfs-docs/Performance%20and%20Tuning/Workload%20Tuning.html (visited on 07/12/2023).

[113] Various GitHub users. *Add content-defined chunking for better deduplication*. URL: https://github.com/openzfs/zfs/issues/11400 (visited on 07/12/2023).

[114] Matt Ahrens. *Zero performance overhead OpenZFS dedup*. URL: https://openzfs.org/w/images/8/8d/ZFS_dedup.pdf (visited on 07/11/2023).

[115] OpenZFS. *Checksums and Their Use in ZFS*. 2023. URL: https://openzfs.github.io/openzfs-docs/Basic%20Concepts/Checksums.html (visited on 06/20/2023).

[116] Various OpenZFS authors. *OpenZFS - DDT.h*. 2023. URL: https://github.com/openzfs/zfs/blob/master/include/sys/ddt.h (visited on 06/20/2023).

[117] M AdelsonVelskii and Evgenii Mikhailovich Landis. *An algorithm for the organization of information*. Tech. rep. JOINT PUBLICATIONS RESEARCH SERVICE WASHINGTON DC, 1963.

[118] Oracle. *The dedup property*. 2010. URL: https://docs.oracle.com/cd/E19120-01/open.solaris/817-2271/gjhav/index.html (visited on 06/21/2023).

[119] Oracle. *ZFS Data deduplication*. 2014. URL: https://docs.oracle.com/cd/E37831_01/html/E52872/shares__shares__general__data_deduplication.html (visited on 06/21/2023).

[120] Truenas. *Truenas*. URL: https://www.truenas.com (visited on 06/20/2023).

[121] Jian Xu and Steven Swanson. "{NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main memories." In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 2016, pp. 323–338.

[122] Jan Kára. "Ext4, btrfs, and the others." In: *Proceeding of Linux-Kongress and OpenSolaris Developer Conference*. 2009, pp. 99–111.

[123] Jesse David Dinneen and Ba Xuan Nguyen. "How Big Are Peoples' Computer Files? File Size Distributions Among User-managed Collections." In: *Proceedings of the Association for Information Science and Technology* 58.1 (2021), pp. 425–429.

[124] Se Kwon Lee et al. "{WORT}: Write optimal radix tree for persistent memory storage systems." In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 2017, pp. 257–270.

[125] Dirk Pflueger. *Fundamental Algorithms*. URL: https://www5.in.tum.de/lehre/vorlesungen/fundalg/slides/fundalg08.pdf (visited on 07/12/2023).

[126] Alan Jay Smith. "Cache memories." In: *ACM Computing Surveys (CSUR)* 14.3 (1982), pp. 473–530.

[127] Donald Ervin Knuth. "Sorting and searching." In: *The art of computer programming* 3 (1998).

[128] Pengfei Zuo, Yu Hua, and Jie Wu. "{Write-Optimized} and {High-Performance} hashing index scheme for persistent memory." In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 461–476.

[129] Khronos. *Core Language (GLSL)*. URL: https://www.khronos.org/opengl/wiki/Core_Language_(GLSL) (visited on 05/30/2023).

[130] Khronos. *Compute Shader*. URL: https://www.khronos.org/opengl/wiki/Compute_Shader#Limitations (visited on 05/31/2023).

[131] Khronos. *Understanding Vulkan Synchronization*. Mar. 2021. URL: https://www.khronos.org/blog/understanding-vulkan-synchronization (visited on 05/31/2023).

[132] Wikipedia. *List of AMD graphics processing units*. URL: https://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units (visited on 05/31/2023).

[133] Khronos and various developers. *VK_EXT_pageable_device_local_memory(3) Manual Page*. 2023. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_pageable_device_local_memory.html (visited on 08/04/2023).

[134] Anjan Roy. *BLAKE3 on GPGPU*. 2022. URL: https://itzmeanjan.in/pages/blake3-on-gpgpu.html (visited on 08/15/2023).

[135] Philip Taylor. *Vulkan memory dependencies*. URL: https://github.com/philiptaylor/vulkan-sync/blob/master/memory.md (visited on 08/18/2023).

[136] Michael Kerrisk. *msync(2) – Linux manual page*. URL: https://man7.org/linux/man-pages/man2/msync.2.html (visited on 08/20/2023).

[137] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. *Kernel Probe (Kprobes)*. URL: https://docs.kernel.org/trace/kprobes.html (visited on 08/21/2023).

[138] AMD. *AMD Radeon RX7900XTX*. URL: https://www.amd.com/de/products/graphics/amd-radeon-rx-7900xtx (visited on 08/26/2023).

[139] AMD. *HIP Documentation*. URL: https://rocm.docs.amd.com/projects/HIP/en/latest/ (visited on 08/14/2023).

[140] Mark Harris and Kyrylo Perelygin. *Cooperative Groups: Flexible CUDA Thread Programming*. URL: https://developer.nvidia.com/blog/cooperative-groups/ (visited on 08/14/2023).