Karlsruhe Institute of Technology

Directories for GPU4FS

Bachelor's Thesis
submitted by

Lennard Kittner

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisors:	Peter Maucher and Lukas Werling

4. Dezember 2022 – 4. April 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, April 4, 2023

Abstract

Non-Volatile Memory (NVM), especially Intel Optane, is a promising addition to conventional DRAM-based memory. However, Intel Optane has some drawbacks, namely high CPU usage due to high access times when writing to Intel Optane. This thesis has two goals. The first is to add new and improved directories to GPU4FS, a file system demonstrator, which aims to address some of the problems of Intel Optane by using a GPU as a file system accelerator. The second is to determine if a GPU-accelerated files system is feasible. We evaluate the GPU directories and the feasibility of a GPU-accelerated file system by comparing it to Ext4, a conventional Linux file system. The results show that in all but one of our benchmarks, Ext4 outperforms the GPU directories. The exception is the creation of long directory chains where the GPU directories are faster than the C++ file system library. We only evaluate synthetic benchmarks that do not reflect real-world file system use cases. Therefore, we concluded that a GPU-accelerated file system is feasible in typical file system use cases, which also involve reading and writing files. However, there are still possibilities to reduce latency and increase performance.

Contents

1 Introduction

Modern High-Performance Computing (HPC) necessitates ever-increasing memory capacity [22]. However, memory is expensive, and higher memory capacity also increases static and dynamic power consumption [19]. One solution to these problems is to use Non-Volatile Memory (NVM) instead; one example would be Intel Optane. However, Intel Optane has its own problems and drawbacks, namely high CPU usage due to high access times when writing to Intel Optane [29] [31]. Maucher [17] tried to address some of these problems by proposing a file system that runs primarily on an external accelerator to free up CPU resources. In his thesis, Maucher implemented a demonstrator of the proposed file system using an off-the-shelf GPU as an accelerator. So far, directory creation is primarily handled by the CPU, and the directories are linked-list based.

This thesis aims to port tasks like file lookup and directory creation to the GPU, further relieving the CPU. We present a design and implementation of a new H-tree-based directory data structure on the GPU. An H-tree is a combination between a tree and a hashmap. Hashmaps offer fast lookups using the parallelism of the GPU, and the tree structure helps to reduce the space overhead of large hash tables.

In our evaluation, we compare the latency and speed of directory creates and lookups to Ext4 [16], a conventional Linux file system running on the CPU. We measured a base latency of 0.012 s every time the shader is executed and new commands are sent to the GPU, which is 390 to 1000 times higher than the latency experienced by Ext4. However, longer benchmarks that reduce the impact of the base latency also reduce the performance difference. We also propose how the performance gap between Ext4 and the GPU directories can be narrowed.

This thesis is divided into three parts. The first part contains Chapters 2 and 3 and presents background knowledge and prior work relevant to this thesis. Chapters 4 and 5 make up the second part, which describes the design and implementation of the new directories. In the remaining Chapters 6, 7, and 8, we evaluate the new directories, propose future work, and conclude this thesis.

2 Background

This section introduces GPU4FS [17], the file system we aim to extend, data structures that can be used for file lookup, and other modern file systems and how they handle directories and file lookup. GPU4FS was explicitly designed with Intel Optane and GPUs in mind; therefore, this section will also present some information about Non-Volatile Memory and GPUs.

2.1 GPU4FS

GPU4FS [17] is a file system demonstrator that aims to address some of Intel Optane's shortcomings. GPU4FS uses a GPU as a file system accelerator to prevent CPU stalls caused by Intel Optane [29]. The goal of GPU4FS is to offload the file system management and all of the communication with Intel Optane to the GPU, thus reducing CPU usage. So far, GPU4FS can queue multiple commands and execute them on the GPU. These commands are: setting memory regions to a specific value (memset), copying memory regions (memcopy), writing files (file_path), and setting meta information for the shader execution (meta_info).

Currently, when a file is written to the storage medium, the CPU must create its inode and the inode of the encapsulating directory. After that, the inodes and file contents are copied to the storage medium by the GPU. Finally, GPU adds the file's inode to the directory.

Directories in GPU4FS are linked lists of directory entries; each entry consists of four fields: the block pointer to the file's inode, an offset to the next entry, the length of the file name, and the file name string itself. A linked list has the advantage of being space efficient and easy to implement. However, the file lookup in large file systems will be slow because the time complexity will be $O(n \cdot m)$, where n is the number of directories that have to be checked, and m is the average size of a directory.

2.2 H-Tree

The original H-tree was proposed by Phillips [20] in 2001 for Ext2 [23]. It is a tree with up to three levels. Every directory entry has a hash of its name as a key and the leaf blocks store lists of directory entries. A first level index block contains an ordered array of hash values, each pointing to a leaf block. The hash values in the first level block are the lower bound of all hash values in the leaf block they point to. The root level also contains an array of hash values that point to first level blocks. The arrays and leaf blocks must be sufficiently large to ensure that three levels are enough.

2.3 NVC-Hashmap

Schwalb et al. [25] have implemented a hashmap specifically designed with NVM in mind. Their goal was to use the hashmap for in-memory databases, but the same data structure could be used to manage files in a file system. They proposed using split-ordered lists.

Generally, a hashmap consists of multiple buckets, each holding elements with the same hash. A split-ordered list is a forward-linked list where buckets are

pointers into the list. The first element of each bucket is a dummy element, and new elements are placed in the list behind the dummy element of the corresponding bucket. Since forward-linked lists and dummy elements are used, no locking is required to update the hashmap [25].

2.4 File systems

The task of a file system is to manage the raw bits and bytes of an underlying storage system. A file system makes the raw data accessible by mapping it to files and folders. How a file lookup is handled depends on how files and folders are organized inside the file system.

Btrfs. Btrfs [24] is a Linux file system that uses balanced trees (B-trees) as its primary data structure. A directory in Btrfs contains two sorted lists. These lists hold so-called `dir_items`. A `dir_item` contains the name of a file and an ID to locate the file's contents. Both lists contain identical entries, but the first is sorted by filename hash and the second by inode sequence number. The first is used for path lookups, and the second for bulk operations like backups [24]. Btrfs uses B-trees to look up entries in these sorted lists.

Xfs. Xfs is a file system primarily used in scientific applications [11]. It is designed for computer systems with high CPU counts and large disk arrays. Like Btrfs [24], Xfs stores directory entries in B+-trees. However, if a directory only contains a few items, these items will be stored as a simple, unsorted list instead [11].

Ext4. Ext4 [16] is the default Linux file system. Unlike other file systems that use B-trees, such as Btrfs [24] and Xfs [11], Ext4 uses constant-depth H-trees [20] to store directory entries.

2.5 High-performance storage and Non-Volatile Memory

Non-Volatile Memory (NVM) is a new and promising type of storage; one example would be Intel Optane [22]. Unlike conventional volatile system memory, NVM does not need to periodically refresh its data, resulting in lower standby power. Furthermore, Intel Optane offers higher density and a lower cost per bit compared to DRAM [19]. Although Intel Optane offers a higher read speed than NVMe SSDs, the access latency can still be around 3-20 times higher than that of DRAM [19]. Especially writes require significantly more time and power than reads [31]. Additionally, Maucher [17] and Peng et al. [19] have shown that many simultaneous requests to Intel Optane cause the bandwidth to decrease. This decrease in bandwidth and the longer access times cause the CPU to stall, resulting in higher CPU utilization [29].

2.6 GPU-Architecture

The GPU's primary purpose is graphics processing; therefore, its architecture is designed with that goal in mind. However, GPUs can also be used for general-purpose computing using APIs like CUDA [18], OpenCL [10], and Vulkan [9]. A GPU is a multithreaded single instruction multiple data (SIMD) processor [12]. That means a GPU consists of multiple processing cores (SIMD processors), where each core operates on multiple pieces of data. The part of the SIMD processor that operates on a single piece of data is called a SIMD lane.

The DRAM located on the graphics card is called VRAM. Each SIMD lane has its own private section of off-chip DRAM called private memory [12]. The SIMD lanes can also access memory shared among lanes of a single SIMD processor called local memory; for communication between SIMD processors, a shared memory section called GPU memory can be used [12].

The programs running on the GPU are called shaders. When programming shaders, each SIMD lane looks like its own thread with its own program counter and variables.

The number of SIMD lanes running on the same SIMD processor is called a workgroup. A workgroup can be larger than the number of physical SIMD lanes of the processor. If so, multiple threads will be launched on the same processor.

2.7 OpenGL Shading Language (GLSL)

GLSL is a programming language used to write compute shaders. GLSL's syntax is very similar to C. However, unlike C, GLSL has no pointers, and to modify function parameters, GLSL has the keyword `inout` [14]. Another limitation is that GLSL only allows arrays where the length is known at compile time, and the length of arrays as function parameters has to be known at compile time as well [14]. GLSL has built-in atomic functions and barriers for locks, synchronization, and calculation with multiple threads [14]. Some examples that are often used throughout the program code of this thesis are: `atomicMin`, `atomicAdd`, `atomicExchange`, `atomicCompSwap`, and `barrier`. The first three atomic functions all take two parameters `mem` and `data`, they then calculate the minimum or sum of `data` and `mem` and write it to `mem`, or in the case of exchange directly write `data` to `mem`. After that, they will return the previous value of `mem`. The read and write happens atomically. `atomicCompSwap` also returns the previous value of `mem` but only writes `data` to `mem` if `mem` is equal to a third parameter named `compare`. The `barrier` synchronizes the execution and memory accesses of the SIMD lanes.

3 Related Work

This section presents prior work related to file systems, NVM, NVM data structures, and general-purpose GPU computing.

3.1 File Systems

The task of a file system is to manage the storage system and also give users the ability to store files and folders.

GPU4FS [17] is a file system demonstrator that aims to address some of Intel Optane's shortcomings. It is also the file system to which this thesis aims to add directories.

Btrfs [24] is a Linux file system that uses balanced trees (B-trees) as its primary data structure. We considered using B-trees as our main directory data structure. However, we decided against it due to its complexity and the fact that it is hard to take full advantage of the multithreaded nature of GPUs.

Ext4 [16] is the default Linux file system and uses H-trees for its directories. The H-tree is also the data structure used in our directories.

Xfs [11] is a file system primarily used in scientific applications. Xfs inspired us to store small directories as linked lists.

NOVA [30] is a log-structured file system that aims to maximize performance on hybrid systems with volatile and non-volatile memory while providing strong consistency guarantees. Unlike conventional log-structured file systems, in NOVA, every inode has its own linked list log. They chose logs per inode to improve concurrency and a linked list because of the good random-access performance of NVM. However, when an operation that involves multiple inodes, e.g., a move between Directories, is performed, journaling is used to update the logs atomically. NOVA stores its lookup data structures in DRAM to keep the in-NVM data structures simple and efficient.

Tmpfs for NVM Kim et al. [15] proposed that a modified version of Tmpfs [27] could be well suited as NVM file system. We will call the modified version of Tmpfs Tmpfs4NVM. The original Tmpfs is an in-memory file system designed for storing temporary files for applications. According to them, Tmpfs4NVM performs better than conventional file systems when used as an in-memory file system. They tested this by benchmarking Tmpfs4NVM and Ext4 using Ramdisk as the storage medium. They say Tmpfs4NVM is faster because it does not use reliability techniques a conventional file system would use, such as journaling, shadow paging, and checkpointing. These techniques are required when information is cached in volatile memory, but not to the same extent when using NVM. However, they also say that reliability mechanisms between the NVM and CPU cache are required when using NVM file systems.

3.2 NVM

In [21], Puglia et al. present the current state of NVM research. They write about the adaption of file systems and databases for NVM, but also the most common issues and most significant challenges of NVM and proposed solutions. They do this by analyzing, comparing, and categorizing various NVM-related studies.

In [31], Xue et al. talk about the characteristics, challenges, and opportunities of three leading technologies used for NVM, i.e., phase-change memory (PCM), Spin-transfer torque RAM (STT-RAM), and multi-level cell (MLC) STT-RAM. PCM is the technology Intel Optante is based on [28].

3.3 NVM Data Structures

Recently some work has been done to design new or modify existing data structures to take advantage of NVM and to alleviate its shortcomings [25] [6].

An example is the hashmap designed by Schwalb et al. [25]. Their goal was to create a hashmap that guarantees consistency even in the case of power failure while offering comparable performance to B+-Trees. Some of their ideas, like split-ordered lists, are used in our H-tree.

Bittman et al. [6] modified existing data structures to minimize bit flips. According to them, minimizing bit flips is more important than minimizing writes. The reason is that writes to phase-change memory (PCM), used for NVM, consumes (relatively) significant power and wears out the cells. Thus, the controller often only writes to the PCM if the value actually changes, i.e., a bit flip occurs. The authors present two data structures a hashmap and an XOR linked list. The hashmap reduces bit flips by avoiding zeroed keys and checking the Hemming distance of free slots before inserting new elements. The XOR linked list is doubly-linked, but the forward and backward pointers are XORed, so one pointer less has to be stored and potentially updated per list element.

3.4 GPU File System

Silberstein et al. [26] implemented a software layer called GPUfs, which allows the GPU to access files on the host machine. Their goal is to give GPU programs a POSIX-like API to interact with the file system. For example, GPUfs allows developers to map files into the GPU memory using the `gmmap` call. Unlike GPU4FS [17], GPUfs still relies on the CPU for its file system operations. GPUfs offers an API for the GPU but does not enable the GPU to talk directly to the storage medium.

4 Design

This chapter will present the complete implementation of the new directories. Currently, directories in GPU4FS [17] are linked lists. A linked list has the advantage of being space efficient and easy to implement. However, the file lookup in large directories will be slow compared to a tree-based approach. With a linked list, potentially, every file inside the directory has to be checked. With a tree managing the linked list, on the other hand, only a few files have to be checked because the tree makes it possible to reach later parts of the list without the need to scan all previous files inside the list.

The data structure inside directories is the most important part of the file lookup. After looking through file systems and data structures, three candidates are up for consideration: B-tree [24], H-tree [20], and Hashmap [25].

The B-tree offers good worst-case performance for directory lookups, $O(\log(n))$, where n is the number of subdirectories [20]. Additionally, B-trees are more space efficient than H-trees and hashmaps because they do not have to store lookup tables. On the other hand, B-trees are very complex [20].

H-trees were designed to be much simpler than B-trees while still offering similar performance [20]. In contrast to B-trees, H-trees have a limited depth, significantly simplifying splitting and balancing. However, the index blocks have to store lookup tables for the leaves; the original H-tree paper [20] proposed 4 KB per index block and a maximum depth of three.

According to their benchmarks, the hashmap proposed by Schwalb et al. [25] has worse single-threaded search and insert performance than a B-Tree but performs better than a B-tree in multithreaded tests. The hashmap is also simpler to implement than the B-tree and H-tree. However, a lookup table has to be stored similarly to an H-tree. Fortunately, the hashmap stores its entries as a forward linked list, so directories with few elements could be stored as a simple linked list, and if the directory gets large enough, the lookup table could be added. This is very similar to Xfs, which also stores small directories as lists [24].

We believe combining hashmap and H-tree provides the best balance between complexity, performance, and additional data to be stored. Additionally, it works well for multithreaded access, which is crucial for a GPU. Our data structure will store entries inside split-ordered lists like the hashmap. However, since a hashmap lookup table would be too big, an H-tree is used to index the list instead. Figure 2 shows a draft of the data structure.

Another critical factor is the hash function. The hash function should be fast and efficient, and the output values should be evenly distributed to minimize linear search. We believe a hash function that has already proven its effectiveness should be used rather than developing our own hash function. Considering this, the hash function used by Ext4 is a good choice. Ext4 uses a modified md4

hash that hashes strings to 32 or 64 bit numbers [13]. We use 32 bit hashes in our implementation because our H-tree will manage at most 65.025 hash regions. Thus, $2^{32} = 4.294.967.296$ possible hash values are sufficient. It also allows for smaller tree block entries and, therefore, more entries per tree level.

4.1 Directory and H-tree

A directory consists of an optional H-tree and a linked list of leaf blocks. The H-tree is only added to sufficiently large directories. A linked list is used for lookups in small directories, resulting in less space overhead for small directories. The Directory entries are stored inside a split-ordered forward-linked list. The linked list contains dummy entries that signal the boundaries of the hash regions. The hash regions are ordered by their lowest hash, and divided by dummies containing the lowest hash of the hash region, but the entries between dummies are unsorted. If the list grows too large new leaf blocks are added to the leaf block list. Currently, the H-tree is limited to three levels: root, first, and leaf. Each entry inside the H-tree has a hash value less or equal to the lowest hash inside the hash region they represent. Entries inside the root level block can point directly to dummies inside the leaf blocks or to a first level block. Entries inside the first level blocks always point to dummies inside the leaf blocks. Figure 1 shows two example directories, one managed by an H-tree and one without an H-tree.

In its current configuration, a hash region has a maximum size of 256, the tree is limited to a depth of three, and an H-tree will only be created after 256 entries have been inserted into a directory. This configuration allows an H-tree to manage $255 * 255 * 256 = 16.646.400$ files. The directory can store more files, but we can only give hard runtime limits up to 16.646.400 files; after that, performance can degrade because hash regions can become larger than 256 entries resulting in a longer linear search. Changing the size of hash regions or the value of entries after which the H-tree is added is simple. However, changing the maximum depth of the tree requires changes to the block-splitting code. The changes are relatively simple and centered around the possibility that with more than three levels, a block split can lead to the parent block also splitting, which is impossible with only three levels because the parent block is always the root.

Locking is handled through a reader-writer lock in the inode of directories. We considered locking on the hash region level, but the changes to the H-tree when a hash region splits affect the entries of other hash regions. This problem may be solved in the future through more sophisticated locking. One approach would be to use optimistic locks on the allocation bitmap and split-ordered list because they can be updated atomically and use exclusive locks on parts of the H-tree if a hash region splits. Optimistic locks work by first reading some value and then only write a new one if the old value has not changed since the last read. The

operation is tried again if the value changes between the read and the write access. An exclusive lock, on the other hand, guarantees access to a resource to a single party until the lock is released. This allows multiple reader and writer to access the directory simultaneously and only restrict access if a hash region splits.

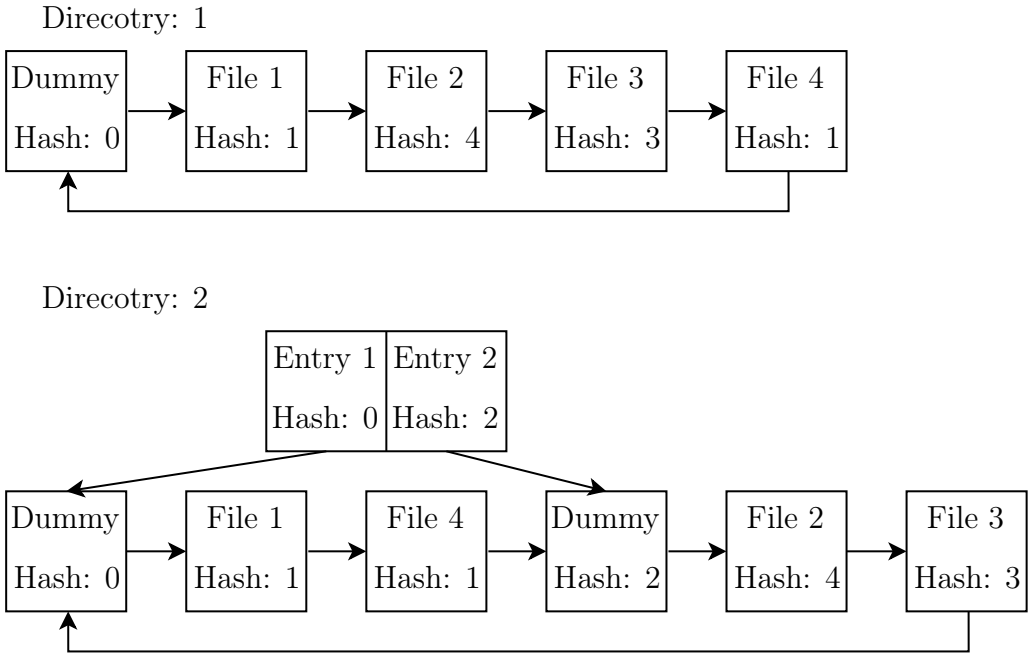

Figure 1: Directories one and two both show the same directory with four different files. In this example, each file has a hash from one to four. The hash of the dummies and entries represents the starting point of their hash region. Directory one is not managed by an H-tree. It has only one dummy, and hash region, which starts at hash zero, and the files have no particular order. Directory two is managed by an H-tree. There are two hash regions inside the directory. The first hash region starts at hash zero and the second at hash two. Each hash region has its own dummy and entry inside the H-tree. The hash region starting at zero only contains files whose hash is greater or equal to zero and smaller than two. The second hash region only contains files with a hash of at least two. The files inside the hash regions have no particular order.

4.1.1 Entry Lookup

To determine if a directory contains a given file, first, the file's name is hashed. Second, the first leaf block is checked to determine whether an H-tree manages the directory; if not, a linear lookup will be performed.

For the linear case, the lookup is completed, so we describe the lookup when an H-tree manages the directory. Next, the root of the H-tree is checked to determine the hash region which contains the file's hash. To be more specific, each entry inside the H-tree block has a hash value representing the lowest hash value of the referenced hash region, so we want to find the entry with the highest hash value still smaller than the hash of the file. Here we take advantage of the many SIMD lanes of the GPU. An H-tree block can have up to 255 entries, and each lane checks an equal portion of the entries. Therefore if the shader runs with 64 SIMD lanes per workgroup, each lane has to check at most four entries. Each lane searches the entry inside their portion of entries with the highest hash value still smaller than the hash of the file. Then all lanes combine their results by calculating the highest hash, thus determining the hash region containing the file's hash.

If the entry points to another H-tree block, the same process of determining the right hash region is repeated for that H-tree block. If the entry points to a dummy inside a leaf block, a linear search starts at that dummy. Our H-tree is limited to a depth of three; therefore, at most, two H-tree blocks must be checked before reaching a dummy.

The linear search stops if the next dummy is reached, i.e., the current hash region ends. The configuration we tested splits hash regions with more than 256 entries. Thus, the linear search has to check at most 256 entries. However, a hash region can become larger than 256 entries if the H-tree is full or more than 256 hash collisions occur. In this case, the linear search may check more than 256 entries.

4.1.2 Entry Creation

The file insertion uses the same code as the file lookup to determine the hash region in which the file will be inserted. Then, space inside the directory is allocated; if all leaf blocks are full, a new empty leaf block is created and added to the linked list of leaf blocks. After that, a directory entry referencing the file is inserted into the linked list immediately after the dummy of the hash region.

The hash region is split if it contains 256 files after the insertion. Since the entries inside hash regions are unsorted, a pivot hash that splits the hash region evenly has to be determined. We calculate the pivot hash by taking advantage of the high SIMD lane count of the GPU. There are 256 files inside the hash region. Therefore 256 hashes have to be checked. First, each lane calculates the rank of an equal portion of these hashes. After that, the hash between the two most middle hashes is chosen as the pivot hash. Then a new dummy is inserted into the directory, and the pointers of the entries inside the hash region are updated according to whether their hash value is greater or smaller than the pivot hash. Next, a reference to the new hash region has to be inserted into the H-tree.

If the directory has no H-tree, a new root block with two entries is created. The first entry points to the hash region starting at zero, and the second entry points to the hash region, which starts at the pivot hash.

Otherwise, the entry is added to the tree structure. Here two things can happen: a new level has to be added to the tree, or a tree block has to be split. Both operations are done the same way as in the original H-tree by Phillips [20]. When splitting a tree block, half the entries are copied to a new H-tree block, and a new entry is inserted into the root block. Adding a new level is done by creating a new root block with an entry that points to the old one.

Currently, if the root of a two level tree is full, a warning is sent to the user, signaling that the directory tree is full. When this happens, files can still be inserted into the directory. However, tree blocks can not be split anymore. Thus, if a hash region's splitting requires a tree block to be split, no split will occur.

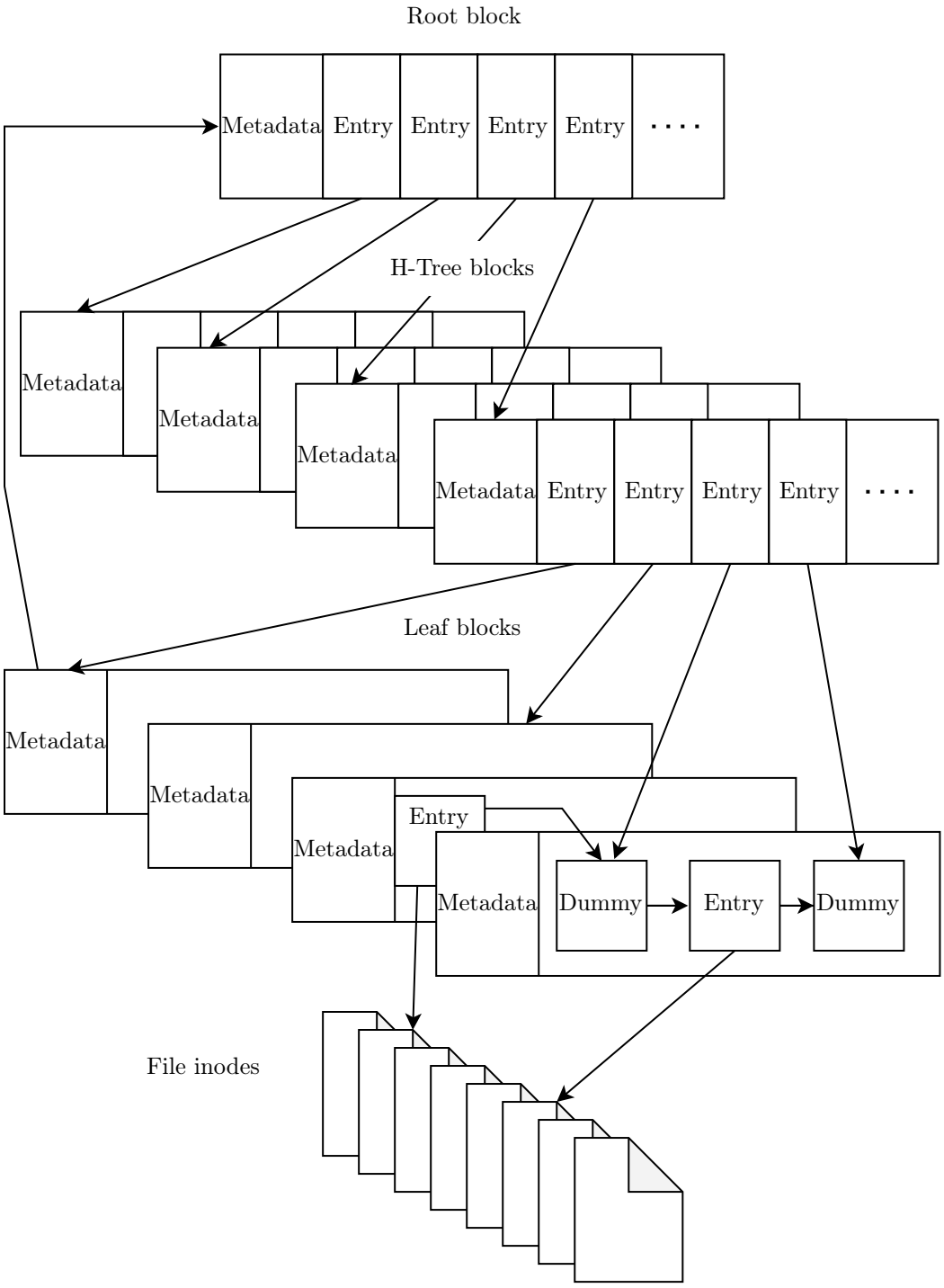

Figure 2: This is an overview of the data structure used inside directories. It shows the two tree levels and the leaf level, along with some directory entries, dummies, and inodes.

4.2 Comparison to the original H-tree

This section will explain the differences between the H-tree used for GPU4FS [17] and the original H-tree designed by Phillips [20].

The original H-tree entry is 64 bit in size and consists of a 32 bit hash and a 32 bit logical pointer. In our implementation, the block pointer is expanded to 64 bit, and another 64 bit are used for the 32 bit hash and some metadata; the metadata is padded to 32 bit, so the entries are 64 bit aligned. Since the new tree entries are twice as large, one tree block can only store 255 instead of 508 entries. The two entries can be seen in Figure 3. The leaf block is the most significant difference between the H-tree presented in this thesis and Phillips' [20] H-tree. In Phillips' [20] H-tree, every hash region has its own fixed-size leaf block; this has some problems he admits in his paper. Our H-tree aims to address these problems by using a single split-ordered list that spans across leaf blocks instead. This approach defines a hash region through a dummy element inside the list instead of a fixed-sized leaf block.

Possible waste of space inside block: If a block has only a narrow range of hash values, space inside the block could be wasted because a 4 kB block is too large for the number of entries. Our H-tree does not have this problem because leaf blocks are independent of hash regions and can be used by all hash regions.

Leaf blocks with very few entries: Phillips' leaf blocks are 4 kB in size, and the name of a file can be as large as 256 B, that means in a worst-case scenario, less than 16 directory entries could be stored in one block. The number of leaf blocks is limited, so a directory with long file names can store fewer files than a directory with short file names. The hash regions in our H-tree are not limited to a fixed amount of disk space; instead, an arbitrarily predetermined number of entries can be stored in every hash region.

Extended search if too many hash collisions occur: Hash regions can become larger than a single leaf block if too many entries have colliding hashes. Phillips alleviates this by signaling a collision and also filling the leaf block of the following hash region. As mentioned before, our hash regions are not limited to a fixed amount of disk space, so this is not a problem. However, lookups may take longer if more entries collide than are generally allowed inside a hash region.

The need to copy entries when a block splits: To split hash regions, Phillips sorts the entries inside the hash region and then copies half of the entries over to the new leaf block. We do not sort the entries but instead choose a hash that splits the regions evenly, which was also mentioned in the original paper as a possible way for splitting hash regions. This reduces the time complexity of the split from $O(n \log(n))$ to $O(n)$, where n is the number of files inside the hash region. After that, we update the pointers of the entries instead of copying them. This reduces writes to NVM, which is crucial because of the significant time and

power writes to NVM require [31].

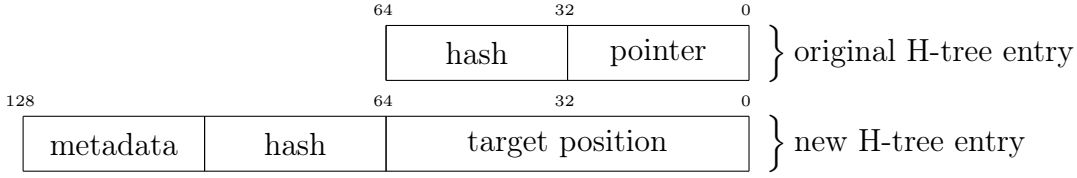

Figure 3: The original and the new H-tree entries.

4.3 Lookup Cache

The lookup cache reduces the number of NVM accesses, thus improving lookup latency. The cache maps the hash of a path component to its name, a pointer to its inode, and a pointer to the inode of the encapsulating directory. The name is necessary to detect hash collisions. The pointer to the encapsulating directory is necessary to determine whether the correct directory is being searched because file names are not unique across directories. Thus resolving a given path with n path components requires n accesses to the cache. We considered using the hash of the whole path instead of only the hash of a single path component, which would reduce the required accesses to only one cache access per path. However, in contrast to path components where the name has a fixed length, a path has no length limitation. This complicates storing the whole path, which is necessary to detect hash collisions. Another disadvantage of caching the whole path is that relative paths, which are part of a longer path that is already cached, still cause cache misses.

At the moment, the cache is located inside the local memory of each workgroup. Therefore each workgroup has its own cache, and the size of the cache is limited to the size of the local memory. The local memory is limited to a few kilobytes, whereas the VRAM can be as large as multiple gigabytes [12]. In the future, the cache should be moved to the VRAM and shared among workgroups. However, this would also require locking so the cache stays consistent.

4.4 On-Disk Data Structures

Here we present the blocks and entries that make up the new directories. A block contains 4 kB of continuous data and comprises a header and a set of entries. An entry can vary in size and purpose but is always inside a block. A directory is a file, so the directory's inode points to the blocks the pointer can be seen in Figure 4. Figure 2 shows an overview of the whole data structure.

4.4.1 Block Pointer and Position

The tagged block pointer of GPU4FS [17], as seen in Figure 4, is used in the inode and lookup cache. GPU4FS is a 64 Bit file system, meaning nearly all offsets, counters, and sizes are 64 Bit in size; this includes the block pointer. Maucher [17] chose 64 bit because of Intel Optane's 64 bit-crash consistency and atomicity guarantees.

Files systems have so-called blocks. A block is the smallest amount of data that can be read or written; it also depends on the target storage medium. In GPU4FS, the block sizes are inspired by the DRAM page sizes commonly seen on x86-64 systems. Thus, GPU4FS uses 1 GB, 2 MB, 4 kB, and 256 B or 128 B blocks. It is configurable whether an inode is 256 B or 128 B in size.

By design, each block on the drive is 128 B aligned. Thus, the last seven bits of the block pointer are always zero. These bits are used for a two-bit tag field indicating the size of the referenced data, a valid flag, and an indirection flag indicating whether the pointer is pointing to more pointers.

Everywhere except for the inode and lookup cache positions are used instead of tagged block pointers. A position is a 64 B aligned offset, therefore more precise than the tagged block pointer. This precision is unnecessary when addressing blocks because the smallest block is 128 B. However, most data inside blocks is 64 bit or 32 bit in size and 64 bit aligned.

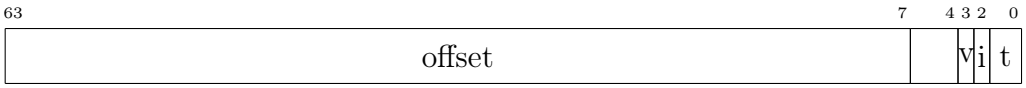

Figure 4: The tagged block pointer. The first 57 bits are used for the byte offset. The following three bits are unused. The next two bits signal whether the pointer is valid (v) and whether it points indirectly (i) to more pointers. The remaining two bits show the tag (t), which holds information about the size of the referenced data.

4.4.2 Directory Leaf Block

As the name suggests, the directory leaf block is on the lowest level of the H-tree structure and houses the directory entries. Leaf blocks have a size of 4 kB to reduce wasted space in small directories.

The header of the leaf block contains the position of the root of the H-tree, the position of the next leaf block, and the allocator. If no H-tree exists, the field is set to the position of the leaf block itself. The rest of the leaf block contains the linked list of directory entries. The leaf block and leaf block entries use global positions instead of local positions or offset because the linked list of directory entries can

span multiple leaf blocks scattered across the disk.

The allocator is one crucial design aspect of the leaf block. It has to be able to handle directory entries of different sizes because, depending on the file name, a directory entry can vary in size. The size of an entry can be anywhere between 33 B to 288 B because an entry contains 32 B of metadata and the file name, which is at most 256 B. Chapter 4.4.3 describes the directory entry in greater detail. It should also be possible to free and reallocate space so that file deletion can be added in the future. Furthermore, the allocator should minimize wasted space inside allocation units and the leaf block itself. With this in mind, we chose a bitmap-based allocator with an allocation size of 16 B, which means 256 possible allocations inside the 4 kB leaf block. A bitmap-based allocator divides the total space, in this case, a 4 kB leaf block, into equally sized chunks, in our case 16 B in size. To allocate the nth 16 B chunk inside the leaf block, one has to set the nth bit inside the allocation bit map to one. However, the first 48 B are used for the leaf block's header, leaving 253 possible allocations. Therefore, the first three bits of the bitmap are always zero and can be used for something like locking. The leaf block can be seen in Figure 5.

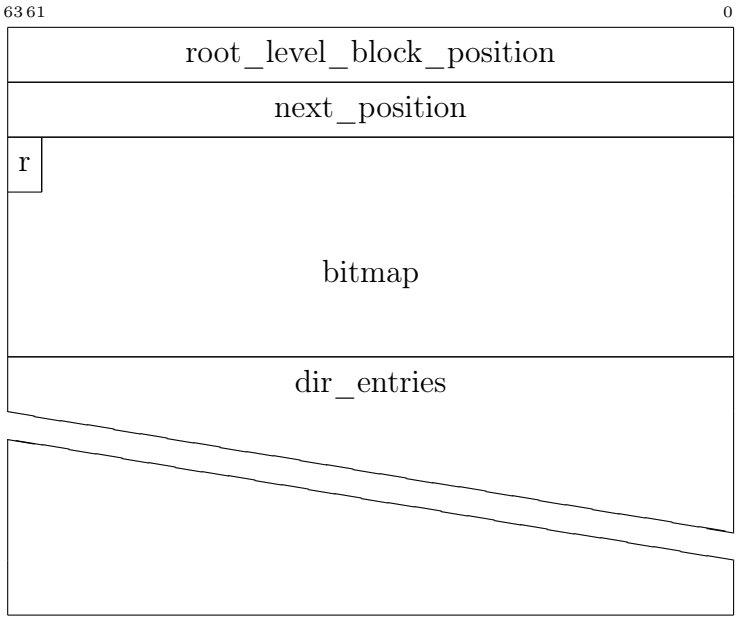

Figure 5: The directory leaf block. The header contains the position of the H-tree, the position of the next leaf block, and a bitmap for allocation. The r represents the reserved bits of the bitmap. The remaining space contains the directory entries.

4.4.3 Directory Entry

Every file inside a directory has its own directory entry. It contains all the file's relevant information, as seen in Figure 6. In addition to the file name, file name length, and inode position, the entry also contains a hash of the file name. The hash is included to speed up name comparisons when looking up files and to avoid recalculating the hash every time the hash region splits, which is necessary to determine the hash that splits the hash region.

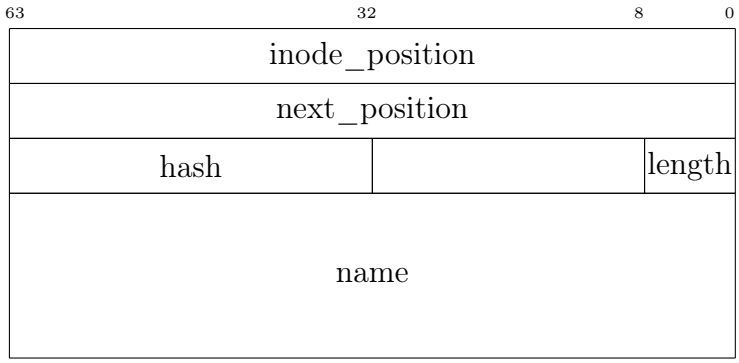

Figure 6: The directory entry. It contains the position of the inode of the underlying file or directory, a next pointer, the hash and length of the file name, and the name itself.

4.4.4 Dummy Entry

The dummy entry, as seen in Figure 7, is a special kind of directory entry. Instead of referring to a file, it signals the beginning of a new hash region. That means all directory entries that come after a dummy have the same or a higher hash value. The structure of the dummy is very similar to a regular directory entry. However, there are some differences. The first eight bytes are unused because the dummy does not belong to any file. Furthermore, the name field is used to count the entries inside the hash region belonging to the dummy. Therefore, the name length is always set to zero.

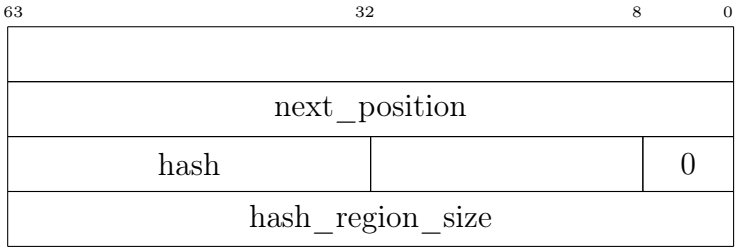

Figure 7: The dummy entry block. It is a special kind of directory entry. The first eight bytes are unused, then comes the next pointer, hash, and length, just like the directory entry. However, the length has to be zero, and the field after that is used for the size of the hash region.

4.4.5 H-tree Block

The H-tree is used to speed up file lookups in directories. Its design is straightforward; each H-tree block, as seen in Figure 8, is 4 kB in size, and the size of an H-tree block entry is 16 B, which means one block can theoretically store 256 entries. However, the first 8 B store the number of entries inside the block, and the second 8 B are padding. Therefore, only 255 entries can be stored in one block.

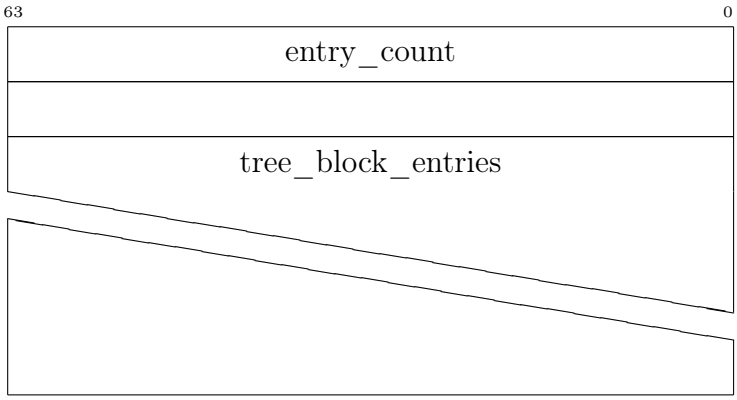

Figure 8: The H-tree block. The first eight bytes store the number of entries, the next eight are unused, and the remaining space contains the tree block entries.

4.4.6 H-tree block Entry

Inside the H-tree block is the H-tree block entry, as shown in Figure 9. It maps a hash region to either another H-tree block or a dummy entry inside a directory leaf block. The first 8 B are split evenly between metadata and hash. Currently,

the metadata only signals whether the entry points to a dummy or another H-tree block. The last 8 B is the position of either a dummy or an H-tree block.

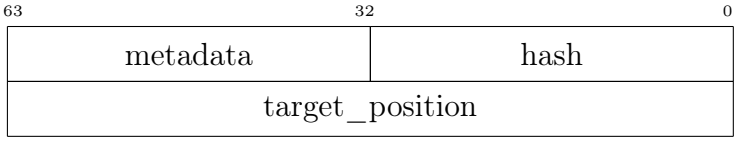

Figure 9: The H-tree block entry. It contains some metadata, a hash representing the underlying hash region, and the position of a dummy or H-tree block.

4.5 Cache

The lookup cache consists of cache entries. Every entry has a fixed size of 272 B. The first 256 B store the file name, and the last two 8 B fields contain tagged block pointers to the encapsulating directory's inode and the file's inode. The file name hash is used to locate the cache entry inside the cache and therefore does not need to be stored inside the entry. The cache entry can be seen in Figure 10.

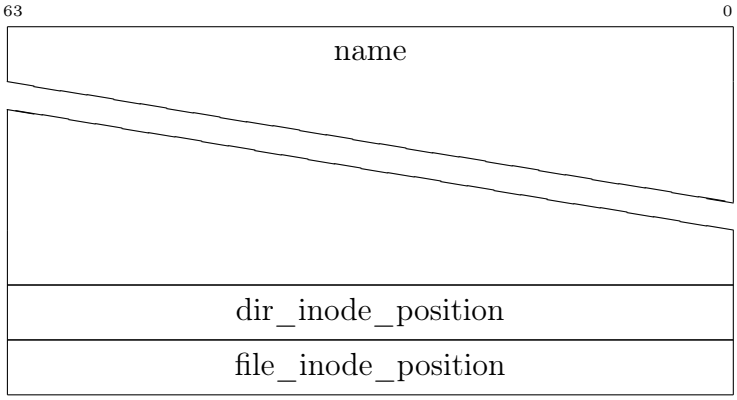

Figure 10: The cache entry. The first 256 bytes store the file name, then comes a tagged pointer to the inode of the encapsulating directory and a second tagged pointer to the file's inode.

5 Implementation

This chapter describes the implementation of the new directories for GPU4FS [17]. We first ported ext4's hash function to GLSL and then implemented parsing a path string on the GPU. After that, we implemented the split-ordered list and the H-tree.

The implementation targets x86-64 Linux Systems with modern NVIDIA or AMD GPUs. We used the same tools as the creator of GPU4FS, meaning the graphics API used is Vulkan [9], the shaders are implemented using GLSL [14], and Googles glslc [7] GLSL [14] to SPIR-V [8] compiler. The CPU-side implementation uses C++-20 [3] and is compiled using g++ [4].

The new directories for GPU4FS are tested on these systems

- a desktop PC with an Intel Core™ i7-4770K processor and a dedicated NVIDIA GeForce GTX 1070 GPU,
- a server with an Intel Xeon E5-2618L v3 processor and a dedicated AMD Radeon RX 6600 GPU,
- and a server with two Intel Xeon Silver 4215 processors, a dedicated AMD Radeon RX 6800, and four DIMMs of Intel Optane NVM, with a combined capacity of 512 GB. This is also the machine used for the benchmarks in our evaluation in Chapter 6.

5.1 Commands

In GPU4FS [17], command descriptors are first written into a command buffer shared by the CPU and GPU and then executed by the GPU. This section presents commands of GPU4FS that have been modified and new commands that were previously not part of GPU4FS.

The new file lookup and directory creation commands use response codes to signal successful execution or to provide more information in case of an error. The response codes have the following meanings:

0. The execution was successful.
1. The file already exists.
2. The H-tree is full.
3. A data pointer could not be added to the directory's inode because all indirection blocks are full.
4. The file could not be found.
5. A path component could not be found.
6. An invalid data pointer was encountered.

7. More than 256 hash collisions were detected.

8. One or more file system invariants have been violated.

5.1.1 The Metadata Command

GPU4FS [17] requires the command buffer to start with some metadata that is shared for all executions. The metadata configures whether to use separated execution, and the number of SIMD lanes that should be used, which are the same usages as presented in the GPU4FS thesis [17]. Separated execution gives work-groups the ability to execute commands independently of each other.

In the past, GPU4FS created the root directory with the CPU, but now the GPU is used instead. Furthermore, we added a new flag called `create_root` that signals whether or not to create a new root directory. Without this flag, a new root directory would be created every time the GPU4FS compute-shader is executed, making all previously created files and directories inaccessible. Another approach would be to set a flag inside the superblock that signals if the root directory already exists. The modified metadata command can be seen in Figure 11.

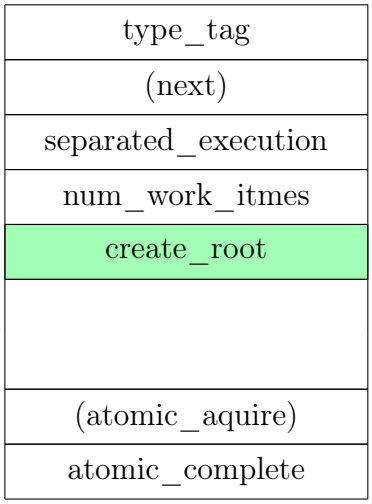

Figure 11: The GPU4FS [17] metadata command descriptor. It contains one boolean flag signaling separated execution, the number of SIMD lanes, and a new flag signaling whether to create a new root directory (highlighted in green). The `next` and `atomic_aquire` fields are unnecessary for the metadata command descriptor because it is always the first command executed by every lane.

5.1.2 The File Create Command

The new file create command can create and insert files into the H-tree-based directories. There was already a command to create files in GPU4FS [17], and although they use different kinds of directories, the new command descriptor is entirely backward compatible with the old one; the only difference is the `type_tag` and that some fields are no longer used. The `inode_position` and `file_position` were previously used by the CPU to specify where to write the inode and file but are no longer needed because Maucher [17] has since ported the block allocator to the GPU. Furthermore, the `directory_position` was used by the CPU to tell the GPU in which directory the file should be inserted; this is also no longer needed because the GPU is now capable of resolving a given absolute path and finding the directory in which the file should be inserted on its own. The file create command can be seen in Figure 12, and more information about the unchanged parameters can be found in the GPU4FS thesis [17].

Main Functionality: To write a file to disk, the GPU has to complete five steps:

1. Write the file data to disk,
2. write the inode to disk,
3. set the block pointers,
4. resolve the path,
5. update the directory

The first three steps are unchanged from the original file write command from GPU4FS and are, therefore, not described in this thesis. On the other hand, the fourth and fifth steps are new and thus described in more detail. The steps can be executed in any order, but step four has to happen before step five because the path has to be resolved before the directory can be updated.

Resolve the path: The function `resolve_path()` is used to resolve the path. The function takes as input the position of the root inode, command block, path offset, and path length. The position of the path offset and path length are required, so the function can parse different command descriptors. If the file exists, it will return the position of the file's inode, the position of the dummy of the hash region the file is located in, the file name length, the file name hash, and the file name itself. If a path component can not be found, the lookup will stop, and all information needed to create the missing file and resume the lookup will be returned. This information includes the position of the inode of the last directory that was found, the position of the dummy of the hash region in which the missing

file should be inserted, the length of the name of the missing file, the hash of the name of the missing file, the name of the missing file itself and some state that can be used to resume the lookup. The file create command does not use the lookup resume feature, but it is necessary for the directory creation. The initialization of `resolve_path()` can be seen in Listing 1. To resolve the path, we loop over it and look up each path component as seen in Listing 2.

Update the directory: After the path has been resolved, the file can be inserted using the function `dir_insert_file()` and the information returned by `resolve_path()`. The `dir_insert_file()` first prepares the directory for file insertion and checks if the file already exists, as shown in Listing 3. Then, the function writes all necessary data to the disk and updates the existing directory. It will also split the hash region in which the file is inserted if it gets too large. This can be seen in Listing 4. Chapter 4 provides more information about the splitting process.

```
// shared variables
shared int64_t _path_buffer_256[32];
shared uint _path_component_hash;
shared int _path_component_length;

int resolve_path(uint root_inode_position, uint block_position,
    uint path_offset_offset, uint path_length_offset, inout uint
    inode_position, inout uint dummy_position, inout uint hash,
    inout int64_t name[32], inout int length, bool use_state, inout
    int current_element, inout int component_shift, inout int
    chars_scanned) {
    uint local = gl_LocalInvocationID.x;
    uint working_directory_position = root_inode_position;
    uint path_component_hash = 0;

    int return_code = SUCCSESS;

    if (!use_state) {
        component_shift = 0;
        current_element = 0;
        chars_scanned = 0;
    }

    int path_offset = int(config.data[block_position +
    path_offset_offset]) / sizeof_int64_t;
    int path_length = int(config.data[block_position +
    path_length_offset]);
    int path_elements = rounded_up_division(path_length,
    sizeof_int64_t);
    ...
```

Listing 1: The GLSL code to prepare the parsing and resolving of a path. First, the state is initialized with default values or a provided state if `use_state` is true. Then the path offset and path length are retrieved from the config buffer.

```
    ...
    while (chars_scanned < path_length) {
        _path_component_length = path_length - chars_scanned;
        if (local < path_elements - current_element && local<32) {
            // read from config buffer ...
            // calculate path component length
            for (int j = 0; j < sizeof_int64_t; j++) {
                if ((int(_path_buffer_256[local] >> (j*8)) & 0xFF)
↪   == 0x2F) {
                    atomicMin(_path_component_length,
                        int(local*sizeof_int64_t+j));
                    break;
                }
            }
        }
        groupMemoryBarrier();
        // update loop variables ...
        // skip leading "/"
        if (_path_component_length == 0) { continue; }
        // add padding ...
        int err = dir_lookup(working_directory_position,
↪   dummy_position, _path_component_hash, _path_buffer_256,
↪   _path_component_length);
        if (err != SUCCSESS) { ... }
    }
    ...
```

Listing 2: The GLSL code for parsing and resolving a path. It is part of the `resolve_path()` function. The loop ends if the path is fully parsed or a path component cannot be found. In each iteration, first, the path component is read from the config buffer. Each lane reads eight characters at once and writes them to the shared variable `_path_buffer_256` so all lanes inside the same workgroup can see the current path component. Depending on the previous path component, some shifting is necessary because accesses to the config buffer are 64 bit aligned, but path components are not. After that comes another loop in which each lane checks its eight characters for a slash (0x2F), i.e., the end of the path component. The position of the first slash is then communicated to the other lanes of the workgroup through the shared variable `_path_component_length`. Next, the hash is calculated by the first lane of the workgroup and communicated to the other lanes via the shared variable `_path_component_hash`. Finally, `dir_lookup()` is called to check if the path component exists. If so, the next iteration starts; otherwise, the loop ends.

```
int dir_insert_file(uint dummy_position, uint dir_inode_position,
↪   uint file_inode_position, uint hash, int64_t name[32], int
↪   length) {
    uint local = gl_LocalInvocationID.x;
    acquire_write_lock(dir_inode_position);

     // check if file already exists
    if (dir_lookup_linear(dummy_position, dir_inode_position, hash,
↪   name, length) == SUCCSESS) {
        unlock_write_lock(dir_inode_position);
        return FILE_ALREADY_EXISTS;
    }
    int type = 0;
    bool valid = false;
    bool indirect = false;
    uint dir_position =
↪   inode_get_position_from_first_data_pointer(dir_inode_position,
↪   type, valid, indirect);
    if (valid == false) {
        unlock_write_lock(dir_inode_position);
        return INVALID_DATA_POINTER;
    }
    bool has_H_tree = uint(nvm.to[dir_position]) != dir_position;
    uint space_needed = ENTRY_NAME_OFFSET +
↪   rounded_up_division(length, sizeof_int64_t);
    uint entry_position = 0;
    int err = dir_allocate_space(dir_inode_position, space_needed,
↪   entry_position);
    ...
```

Listing 3: The GLSL code to prepare the insertion of a file into an existing directory. First, the inode of the directory is locked using `aquire_write_lock()`. Second, `dir_lookup_linear()` is used to check if the file already exists. If not, the first leaf block position is read from the inode, and space is allocated inside one of the leaf blocks using `dir_allocate_space()`.

```
22      ...
23      const uint name_elements = rounded_up_division(length,
    ↪   sizeof_int64_t);
24      nvm.to[entry_position + ENTRY_INODE_POSITION_OFFSET] =
    ↪   int64_t(file_inode_position);
25      nvm.to[entry_position + ENTRY_HASH_OFFSET] = int64_t(hash) <<
    ↪   ENTRY_HASH_SHIFT;
26      nvm.to[entry_position + ENTRY_NAME_LENGTH_OFFSET] |=
    ↪   int64_t(length);
27      if (local < name_elements) {
28          nvm.to[entry_position + ENTRY_NAME_OFFSET + local] =
    ↪   name[local];
29      }
30      inode_increment_hardlink_counter(file_inode_position);
31      // insert file into list ...
32      if (uint(nvm.to[region_dummy + ENTRY_HASH_REGION_SIZE_OFFSET])
    ↪   < DIR_MAX_ENTRIES_PER_HASH_REGION) {
33          unlock_write_lock(dir_inode_position);
34          return SUCCSESS;
35      }
36      if (!has_H_tree) {
37          uint H_tree_block = get_free_block();
38          err = new_H_tree(dir_position, dir_inode_position,
    ↪   H_tree_block);
39      } else {
40          err = expand_H_tree(dir_inode_position, region_dummy);
41      }
42      unlock_write_lock(dir_inode_position);
43      return SUCCSESS;
44  }
```

Listing 4: The GLSL code to insert a file into an existing directory. It is part of the `dir_insert_file()` function. First, the file's information is written into the leaf block. The `local` variable is unique for each SIMD lane inside a workgroup; it counts from zero to the number of SIMD lanes per workgroup. Then the next pointer of the dummy of the hash region in which the file is inserted is updated. Finally, three things can happen depending on the directory and size of the hash region. If the hash region has less than 256 entries, the inode of the directory is unlocked using `unlock_write_lock()`, and the function returns. Otherwise, it is split using `expand_H_tree()`, or if no H-tree exists, a new one is created using `new_H_tree()` before the inode is unlocked.

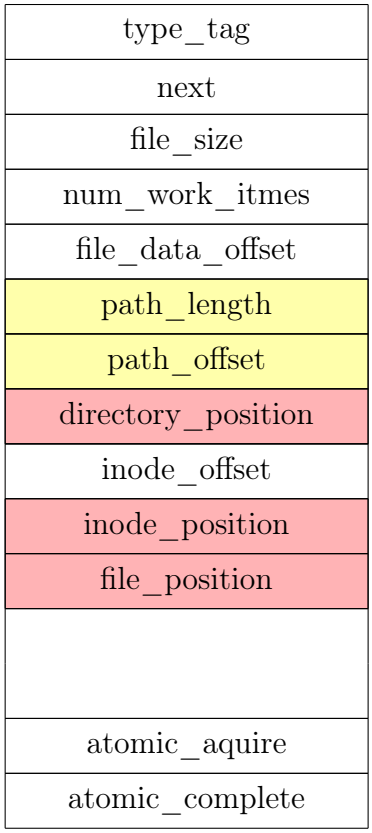

Figure 12: The GPU4FS [17] file writing command descriptor. In addition, to the default command descriptor fields, it contains several offsets into the command buffer to configure which data is copied from DRAM to NVM. `path_length` and `path_offset` were previously limited to the file name but are now used for the file path (highlighted in yellow). `directory_position`, `inode_position`, and `file_position` are no longer needed (highlighted in red).

5.1.3 The File Lookup Command

The new file lookup command, as seen in Figure 13, can be used to let the GPU resolve a given path. If the file exists, the inode offset will be written to the provided result offset; otherwise, an error code will be written to the response code offset. The file lookup command is just a wrapper around the `resolve_path()` function, which can be seen in Listing 1 and Listing 2. Each field in the descriptor has the following purpose, where the index shows the offset inside the descriptor in multiples of 8 B:

2. Reserved for the number of SIMD lanes, but currently unused.

3. The offset of the path in the command buffer that should be resolved.

4. The length of the path.

5. The offset in the command buffer where the response code is written to.

6. The offset in the command buffer where the inode position is written to.

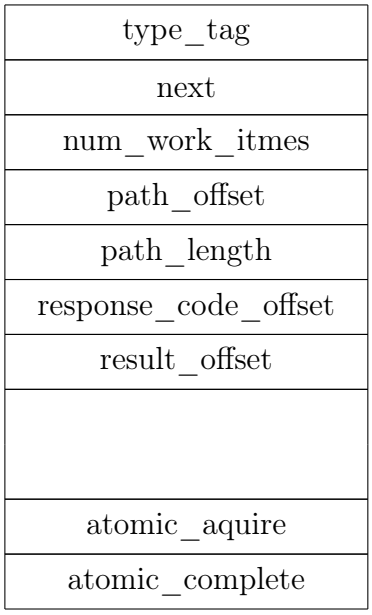

Figure 13: The new path lookup command descriptor. It contains offsets for the path which should be resolved and offsets specifying where to write the result and response code.

5.1.4 The Directory Create Command

The new directory create command, as seen in Figure 14, can be used to create directories using the GPU. Both commands `mkdir()` and `mkdir_all()` use the same command descriptor; the only difference is the `type_tag`. `mkdir()` tries to create the directory at the provided path and fails if one or more intermediate directories do not exist. `mkdir_all()`, on the other hand, creates the directory at the provided path and also any nonexisting intermediate directories. Each field in the descriptor has the following purpose, where the index shows the offset inside the descriptor in multiples of 8 B:

2. Reserved for the number of SIMD lanes, but currently unused.

3. The offset of the path in the command buffer specifying where the directory should be created.

4. The length of the path.

5. The offset in the command buffer where the response code is written to.

Main Functionality: The path is resolved, and the directory is inserted the same way it is done in the file create command using the functions `resolve_path()` and `dir_insert_file()`, which can be seen in Listing 1, 2, 3, and 4. `mkdir()` will execute `resolve_path()` and `dir_insert_file()` a single time and fail if any intermediate directories do not exits. In the case of `mkdir_all(path)`, all missing intermediate directories must also be created. This is done using `resolve_path()`. Every time a missing directory is encountered, the lookup will stop, and all information needed to create the missing directory and resume the lookup is returned. Then the directory is created using `dir_insert_file()`. After that, the lookup is resumed.

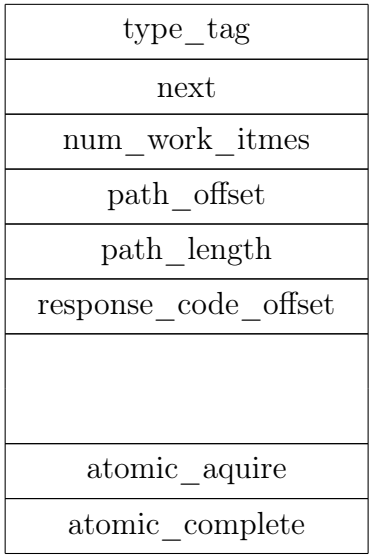

Figure 14: The new directory creation command descriptor. It is for both commands `mkdir` and `mkdir_all`. It contains an offset for the path and an offset specifying where to write the response code.

5.2 Problems

The workgroup size specifies the number of SIMD lanes that should run on each SIMD processor. The SIMD lanes act like threads with their own program counter and variables. They normally communicate through shared variables inside the local memory, but they can also communicate via the storage medium or variables inside the VRAM. The order in which the SIMD lanes are executed is undefined [14].

Therefore, barriers must be used to synchronize execution and ensure that values modified by one SIMD lane are visible to another [14]. However, we noticed that changes to shared variables and the storage medium are sometimes not visible to some SIMD lanes.

Visibility inside workgroups: We tested this on two GPUs: an NVIDIA GTX 1070 and an AMD Radeon RX 6600. On the AMD GPU 64 lanes, and on the NVIDIA GPU, only 32 lanes were able to sync correctly. We implemented our own barrier, which can be seen in Listing 5, to work around this when calculating the minimum of values. However, we could not solve the inconsistencies regarding the storage medium. Thus, we only use 32 and 64 lanes for our evaluation.

Visibility between workgroups: We also noticed inconsistencies between workgroups. Our locks were able to synchronize the workgroups correctly. Thus not more than one workgroup can write to the same directory simultaneously. However, sometimes the workgroups did not see some of the changes made by other workgroups. Despite the use of `memoryBarrier()`, which should guarantee the visibility of prior writes [14].

Synchronization: When a SIMD lane reaches a `barrier()`, its execution should pause until all other lanes reach the same `barrier()` [14]. Therefore, the code in Listing 6 should result in a deadlock when it is executed with more than three SIMD lanes because the first three lanes should never leave the if statement. However, the shader will finish execution without a timeout, and the from buffer will contain an entry from every lane, even from the first three. This is unexpected and may also cause the visibility issues.

```
1    void main() {
2      uint global = gl_GlobalInvocationID.x;
3
4      if (global < 3) {
5          memoryBarrier();
6          barrier();
7      }
8      file.from[global] = to_64bit_literal(0, global);
9      return;
10   }
```

Listing 5: This GLSL code should result in a deadlock. The variable `gl_GlobalInvocationID.x` assigns each lane a unique id. It starts at zero and counts up to the total number of SIMD lanes. Therefore, only the first three lanes should enter and never leave the if statement.

```
#define sync_minima(_minima) \
  uint _local = gl_LocalInvocationID.x;\
  while (_minima[3] != 1 || _minima[2] != 1) {\
    if (_local < 32) atomicMin(_minima[0], _minima[1]);\
    if (_local >= 32) atomicMin(_minima[1], _minima[0]);\
    if (_minima[0] == _minima[1]) {\
        if (_local < 32) _minima[2] = 1;\
        if (_local >= 32) _minima[3] = 1;\
    }\
    barrier();\
  }\
```

Listing 6: The goal of the GLSL macro is to calculate a minimum and synchronize the SIMD lanes so all see the same value. The code is not inside a function because we need a direct reference to the shared array `_minima`, and GLSL has no pointers [14]. Before this macro is called, `_minima[0]` should contain a value considered by the first 32 SIMD lanes to be the minimum, and `_minima[1]` should contain a value considered by the second 32 SIMD lanes to be the minimum. `_minima[2]` and `_minima[3]` are used to check that both SIMD lane groups saw the other group's value and have updated their value accordingly. The SIMD lanes will loop until both groups agree on the smallest value.

6 Evaluation

This thesis aims to evaluate the extra latency of file lookups introduced by using a GPU as a file system accelerator. Therefore, we evaluate the latency of the GPU itself, path lookups, and directory creations. We also evaluate the new directories by measuring the time it takes to create directories and the time it takes to perform lookups in large directories. We then compare our results with Ext4 [16], a conventional Linux file system.

6.1 Test System

The system used for the evaluation has the following specifications:

- Processors: Two Intel Xeon Silver 4215, operating at 2.5 GHz
- Memory: Eight DIMMs of DDR4 at 2400 MT/s with a combined capacity of 128 GB: Each CPU has four 16 GB DIMMs.

- NVM: Four DIMMs of DDR4-socket-compatible Intel Optane memory at 2400 MT/s with a combined capacity of 512 GB; Each individual CPU is connected to two 128 GB DIMMs.
- GPU: AMD Radeon RX 6800 with 16 GB of VRAM and connected via 16 PCIe Gen3 lanes
- Storage: One Samsung NVMe SSD and three Micron NVMe SSDs

6.2 Limitations

Various factors limit the number of tests we can perform:

Number of SIMD lanes: The shader requires at least 32 SIMD lanes to perform string comparisons. Furthermore, the number of SIMD lanes should be a multiple of the physical vector size of the GPU. Although our implementation can handle any number of SIMD lanes greater than 32, it is better to use a power of two because the directory data structures also often use powers of two. However, due to the problems described in Section 5.2, we can use 64 SIMD lanes at most. Therefore, only two workgroup sizes, 64 and 32 SIMD lanes can meaningfully be tested.

Timeout: The kernel will terminate any GPU program after approximately 10 s. This problem was already identified by Maucher [17] but could not be solved so far. Therefore, all test cases have to take less than 10 s; this limits the possible number of directory lookups or creations.

Number of workgroups: As described in Section 5.2 there are visibility issues between workgroups. However, this only affects writing data with multiple workgroups, not reading data. Thus, the lookup benchmarks are unaffected and use up to 512 workgroups, but the directory creation benchmark is limited to at most two workgroups. Wa also wanted to include a benchmark where each workgroup creates directories inside its own subdirectory. However, this benchmark was also not possible due to the visibility issues between workgroups.

6.3 Latency

Every time we execute a command on the GPU, the GPU must first be initialized, and then the CPU has to send the commands via the PCIe bus to the GPU. This latency is present whenever we execute a command on the GPU. We assume that the 10 s timeout limitation of GPU tasks will be solved in the future. This would allow the GPU shader to run constantly; thus, no GPU initialization is required when executing a new command. Therefore, we excluded the base latency from all other benchmarks.

We measured the base latency by sending a command to the GPU that immediately signals completion and exits. Figure 15 shows that the base latency is around 0.012 s. It is also worth noting that the number of workgroups does not influence the latency. We also noticed that our base latency is between five to ten times higher than the latency measured by Maucher [17]. Our test system and the system used by Maucher are very similar, but one difference is the GPU; he used an AMD Radeon RX 6600 compared to our AMD Radeon RX 6800. We also tested the latency on another system with an AMD Radeon RX 6600 and also measured around 0.012 s base latency. Therefore, we think that the GPU is not the cause of the lower latency, but we are unsure why his latency is much lower than ours.

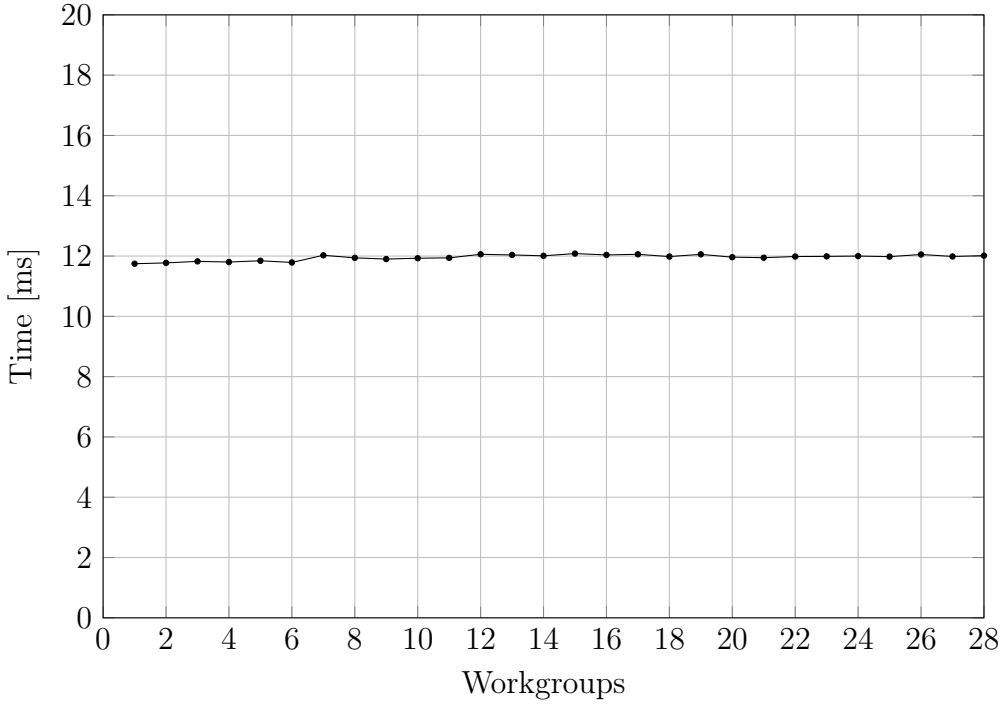

Figure 15: The latency of the GPU for different numbers of workgroups. We measured the base latency by sending a command to the GPU that immediately signals completion and exits. It shows that the number of workgroups does not influence the latency.

6.4 NVM Data Initialization

At the moment, inode data pointer indirection is limited to one level. However, all allocations related to directories are only 4 kB in size. We, therefore, had to increase the size of the indirection block from 128 B to 4 kB so enough pointers

could be stored in one indirection level. However, every indirection block has to be initialized with zero. Thus, 4 kB of additional zeros must be written whenever a new directory is created. Given the high write costs of NVM, we expected this would impact our measurements significantly [31]. To test our hypnosis, we ran our subdirectory creation benchmark two times. The first time every byte inside the NVM was set to zero; therefore, we deactivated the GPU code that writes zeros to the indirection block. The second time every byte was set to 0xDC; therefore, the indirection block had to be set to zero explicitly by the GPU. Figure 16 shows that the difference between the two runs is minimal. We suspect that the large write queues of the GPU help hide the additional write time. We decided to use 0xDC as the initial NVM value for all other benchmarks and set the indirection block explicitly to zero because this way, we do not make any assumptions about the initial value of the NVM and keep our implementation more robust and flexible.

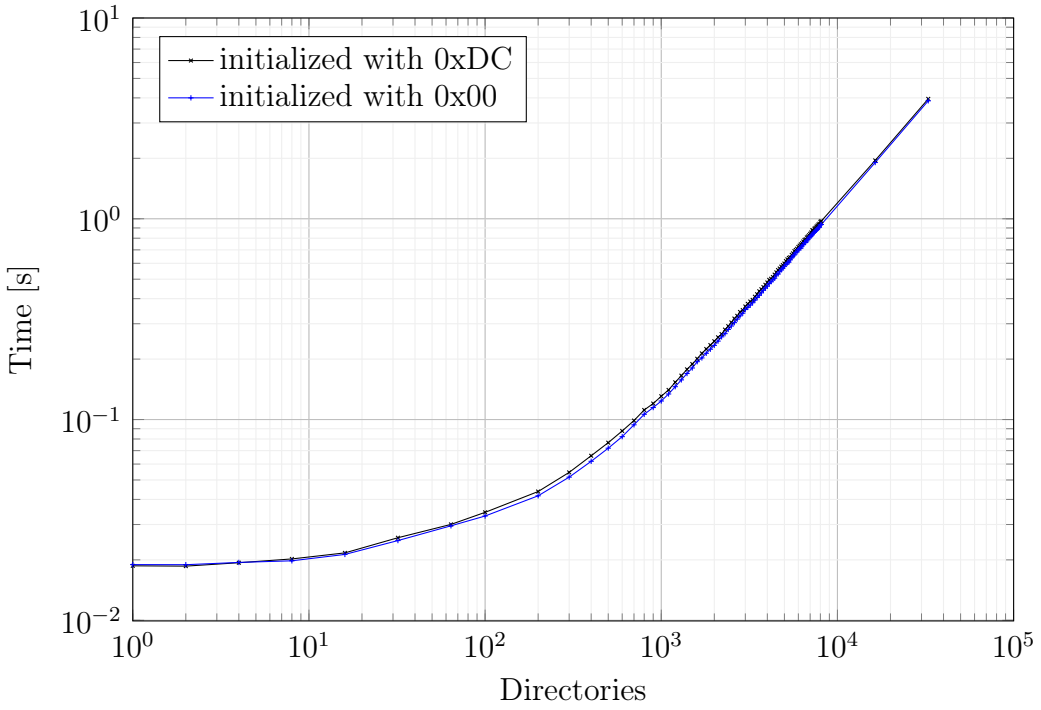

Figure 16: A comparison between different NVM initial values. We test if different NVM initial values affect the time it takes to create and add subdirectories. The data shows that if every byte on the NVM is set to zero, the directory creation is slightly faster than if every byte is set to 0xDC.

6.5 Deep Directory Creation

In Figure 17, we measured the time it takes to create a deep directory structure, i.e., `a/b/c/d/...` We compare the GPU directories with one workgroup and different numbers of SIMD lanes to two different Ext4 directory creation APIs. We wanted to use the same number of directories for both creation and lookup benchmarks. However, the lookup benchmarks encountered timeouts with more than 4000 directories, and thus we limited the depth to 4000 directories.

First, we will discuss the performance differences between the two workgroup sizes on the GPU and then compare them to EXT4. We test one workgroup with two sizes, 64 and 32 SIMD lanes. This benchmark does not benefit from the new directory H-tree structure because each directory only holds one entry. Therefore, the file system overhead is small, and the time is dominated by the time it takes to write the data. This can also be seen by the fact that the results are similar to the write times measured by Maucher [17]. It is also worth noting that 64 SIMD lanes are consistently faster than 32. When creating 4000 directories, 64 SIMD lanes are 1.63 times or 0.0695 s faster than the GPU directories with 32 SIMD lanes. This is unexpected because, without an H-tree, we only use the extra 32 SIMD lanes when setting the indirection block zero, and Figure 16 showed that the additional time required to set the indirection block zero is negligible.

We test two different approaches to create the directories one uses the C++ [3] API `std::filesystem::create_directories` and the other the C system call `mkdirat`. Using `std::filesystem::create_directories` limits the directory depth to 1000. Linux has no system call to create a directory and all missing parent directories. Therefore the `mkdirat` has to use `mkdirat` and `openat` on every directory in the chain. The base overhead of the Ext4 file creation using `mkdirat` is 865 times or 0.0165 s lower than the GPU directories with 64 SIMD lanes, but the relative time difference shrinks with an increasing number of directories. Thus, at a directory depth of 4000 directories, `mkdirat` is 1.84 times or 0.0507 s faster than the GPU directories with 64 SIMD lanes. It is also worth noting that `mkdirat` fluctuates more than the GPU directories. Using `std::filesystem::create_directories` is consistently slower than using `mkdirat`. After 800 directories, the GPU directories with 64 SIMD lanes and after 1000 directories, the GPU directories with 32 SIMD lanes are faster than `std::filesystem::create_directories`. These measurements exclude the base latency as measured in Figure 15. With the base latency included, the GPU directories with 64 SIMD lanes are still faster when creating 1000 directories.

We would expect that creating a single directory takes a constant amount of time. Thus, creating one directory should be around 4000 times faster than creating 4000 directories. This can be seen using Ext4; creating one directory is

3144 times faster than creating 4000 directories. However, creating one directory using the GPU directories is only 6.7 times faster than creating 4000 directories. This indicates some hardware or software bottlenecks.

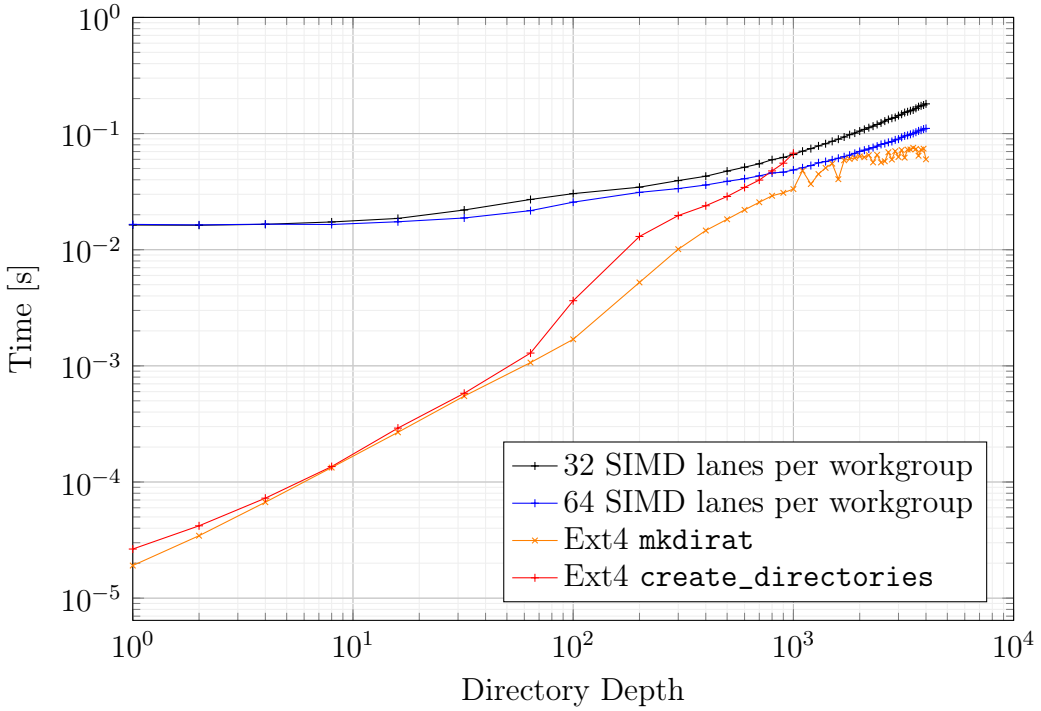

Figure 17: The time it takes to create a deep chain of directories. We exclude the base latency of 0.012 s from the GPU directory results.
We use two different methods to create directories in Ext4 `mkdirat` (x) and `std::filesystem::create_directories` (+). Using `std::filesystem::create_directories` limits the directory depth to 1000. The writes to NVM dominate the total directory creation time. Notably, the workgroup with 64 lanes is consistently faster than the one with 32. Ext4 has a lower base latency, but the relative time difference decreases as the directory depth increases. The GPU directories with 64 SIMD lanes outperform the `std::filesystem::create_directories` implementation after 800 directories.

6.6 Deep Directory Lookups

We performed two different benchmarks related to lookups in a deep directory structure. Both benchmarks measure the time it takes to look up the deepest directory 256 times. In Figure 18, we compare the GPU directories with one workgroup and different numbers of SIMD lanes to Ext4.

First, we will discuss the performance differences between the two workgroup sizes on the GPU and then compare them to EXT4. We test two workgroup sizes, 64 and 32 SIMD lanes. After 3200 directories, the workgroup with 32 SIMD lanes had a timeout. We, therefore, only test 64 SIMD lanes from 3200 to 4000 directories. After around 100 directories, the graph is very linear, which is to be expected because it takes a linear amount of time to walk the directory chain. Most of the time using 64 SIMD lanes is around 1.5 times faster than using 32. This is unexpected because without using the H-tree, no calculation or reads should take advantage of more than 32 SIMD lanes.

Performing 256 lookups with only one directory in the chain is 62 times or 0.0318 s faster when using Ext4 compared to one workgroup with 64 SIMD lanes. However, unlike the deep directory creation benchmark, the time difference between the GPU directories and Ext4 grows as the directories chain gets longer. Thus, looking up a directory at depth 4000 Ext4 is 88 times or 7.994 s faster. These comparisons exclude the base latency measured in Figure 15. We suspect this is because Ext4 uses some lookup cache; our lookup cache only contains 64 entries and is, therefore, too small for the long directory chains.

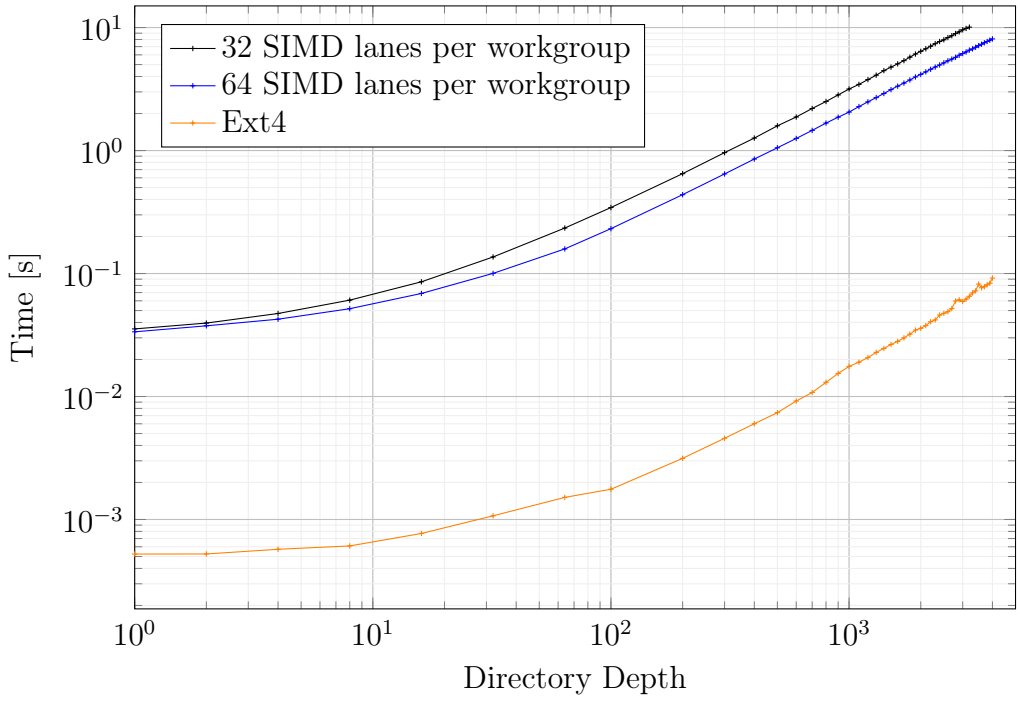

Figure 18: The time it takes Ext4 and the GPU directories using different numbers of SIMD lanes to perform 256 file lookups in a deep file structure. We exclude the base latency of 0.012 s from the GPU directory results. Notably, the number of SIMD lanes does affect the time it takes to perform a lookup, as 64 SIMD lanes are consistently faster than 32. Ext4 has a lower base latency and requires less time per directory.

In Figure 19, we compare the GPU directories with different numbers of workgroups, each with 64 SIMD lanes, to Ext4. First, we will show the results for the GPU directories and then how they compare to Ext4. The workgroups can perform lookups simultaneously and independently from each other. Furthermore, the time it takes to look up the deepest directory increases linearly with the number of directories in the chain. The linear growth becomes less visible as the number of workgroups increases. We also notice that a doubling in workgroups also roughly doubles the lookup speed. There is only a tiny difference between the 128, 256, and 512 workgroups when performing lookups in a directory chain with more than 32 directories. We expected that 512 workgroups would not perform better because only 256 lookups were performed. The number of compute units probably limits the 256 workgroups. The RX 6800 only has 60 compute units; therefore, only 60 workgroups can run simultaneously [2]. We think there are two possible reasons why 128 workgroups are faster than 64: more read commands can be bundled and sent to the NVM controller, and while one workgroup waits for a response from the NVM, another can resume execution.

Single-threaded Ext4 is 23.8 times or 0.0125 s faster than the GPU directories with 128 workgroups when the directory chain only contains one directory. However, without the base latency measured in Figure 15, the GPU directories are 1.12 times or 54.58 µs faster than Ext4. After that, Ext4 is consistently faster than the GPU directories. The performance gap fluctuates, and Ext4 is at most 11 times faster than the GPU directories. At 4000 directories, Ext4 is 1.31 times or 0.0281 s faster than the GPU directories; this excludes the base latency. We expected that the performance gap would be smaller than in Figure 18 because, unlike our single-threaded Ext4 test, the workgroups can perform lookups simultaneously. We can assume that in the real world, more workgroups can work simultaneously on the GPU than threads on the CPU.

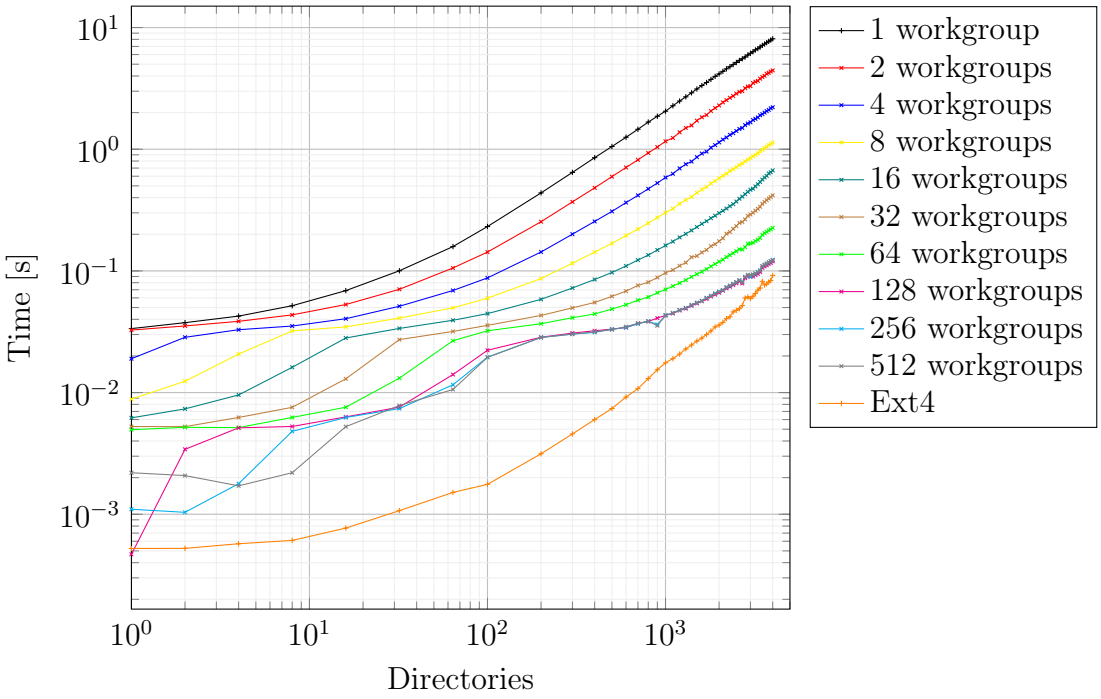

Figure 19: The time it takes Ext4 and the GPU directories using different numbers of workgroups to perform 256 file lookups in a deep file structure. We exclude the base latency of 0.012 s from the GPU directory results. Notably, the more workgroups are dispatched, the faster the lookups are completed. Ext4 has a lower base latency but requires more time per directory.

6.7 Wide Directory Lookups

We performed two benchmarks related to lookups in a directory with many subdirectories. Both benchmarks measure the time it takes to perform 512 file lookups to random subdirectories in a directory with an increasing number of direct subdirectories.

In Figure 20, we compare the GPU directories with different numbers of SIMD lanes to single-threaded Ext4. We test two workgroup sizes, 64 and 32 SIMD lanes. First, we will discuss the performance differences between the two workgroup sizes on the GPU and then compare them to EXT4. The base latency dominates from one to 32 files, so the graph only grows slowly. Then, the time increases faster because the linear search time dominates. After that, the time fluctuates but stays consistent at around 0.067 s, which is to be expected because lookups using the H-tree only require a constant amount of time regardless of the number of subdirectories as long as the H-tree is not full.

After 200 subdirectories, the performance of the different workgroup sizes is

almost identical. We think this is because the lookup can be split into two phases: H-tree and linear. The H-tree phase profits from more SIMD lanes but does not require many accesses to the NVM. The linear phase, on the other hand, can perform up to 256 sequential accesses to the NVM that do not benefit from more SIMD lanes. Therefore the linear time dominates the total time a single lookup takes. However, this contradicts the fact that from 0 to 200 subdirectories, the workgroup with 64 SIMD lanes performs better than the workgroup with 32 SIMD lanes because directories with 200 or fewer subdirectories are not managed by an H-tree. This is unexpected but similar to the results from the deep directory lookup, where the workgroup with 64 SIMD lanes also outperformed the one with 32.

Directory lookups in Ext4 are around 70 times or 0,066 s faster than lookups using the GPU directories with one workgroup and 64 SIMD lanes; this excludes the base latency measured in Figure 15. Furthermore, the difference stays roughly the same. This is similar to the deep directory lookup benchmark, and we suspect this is also because of a lookup cache used in Ext4 or more efficient code.

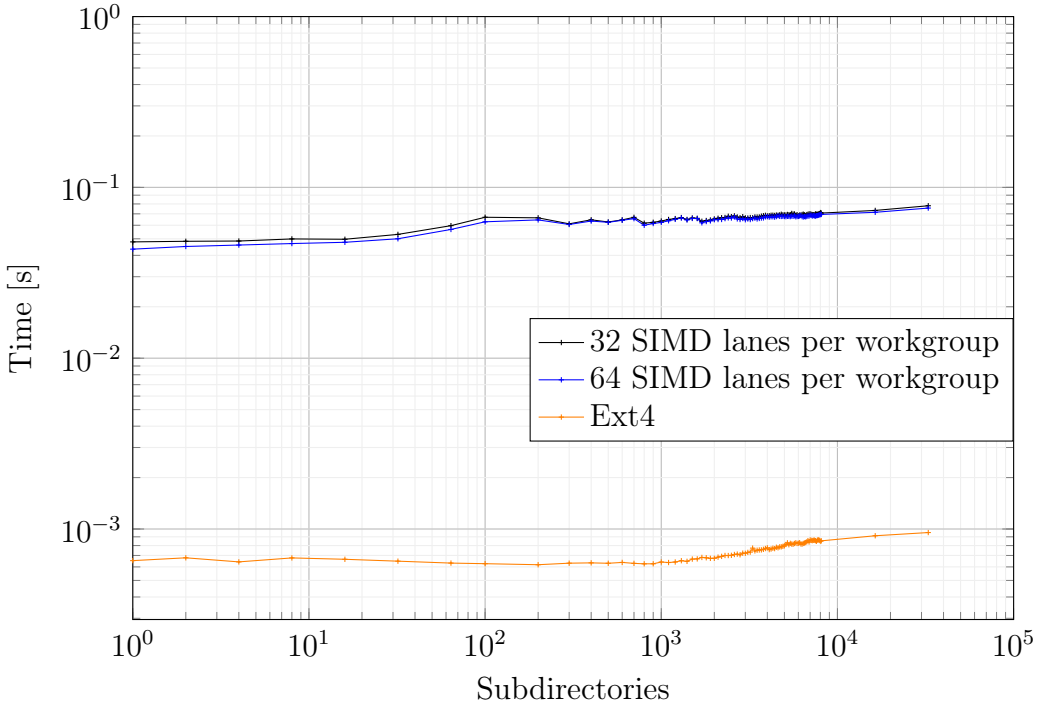

Figure 20: The time it takes Ext4 and the GPU directories using different numbers of SIMD lanes to perform 512 file lookups in a directory with an increasing number of subdirectories. We exclude the base latency of 0.012 s from the GPU directory results. Notably, the lookup time is remarkably constant for large directories. Ext4 is consistently around 0,066 s faster than the GPU directories. The log scale may cause the impression that Ext4 lookup times grow faster than the GPU directory lookup time, but this is not the case. 512 lookups in a directory with only one subdirectory are 1.74 times faster than 512 lookups in a directory with 32768 subdirectories using the GPU directories compared to a factor of 1.46 when using Ext4.

In Figure 21, we compare the GPU directories with different numbers of workgroups, each with 64 SIMD lanes, to Ext4. First, we will show the results for the GPU directories and then how they compare to Ext4. The workgroups can perform lookups simultaneously and independently from each other. Therefore increasing the number of workgroups reduces the time it takes to perform the lookups. However, the speed gains shrink with an increasing number of workgroups. The performance gains after 64 workgroups are small, but using 512 gives the best results. Using 512 workgroups is only 1.38 times faster than 64. This has two possible reasons. The first is that a single dir lookup is short, so the GPU can not take full advantage of all workgroups. The second is that the RX 6800 has

only 60 compute units; therefore, only 60 workgroups can run simultaneously [2].

Single-threaded Ext4 performs the 512 directory lookups around 6.74 times or 0.0049 s faster than the GPU directories with 512 workgroups; this excludes the base latency measured in Figure 15. We can assume that in the real world, more workgroups can work simultaneously on the GPU than threads on the CPU.

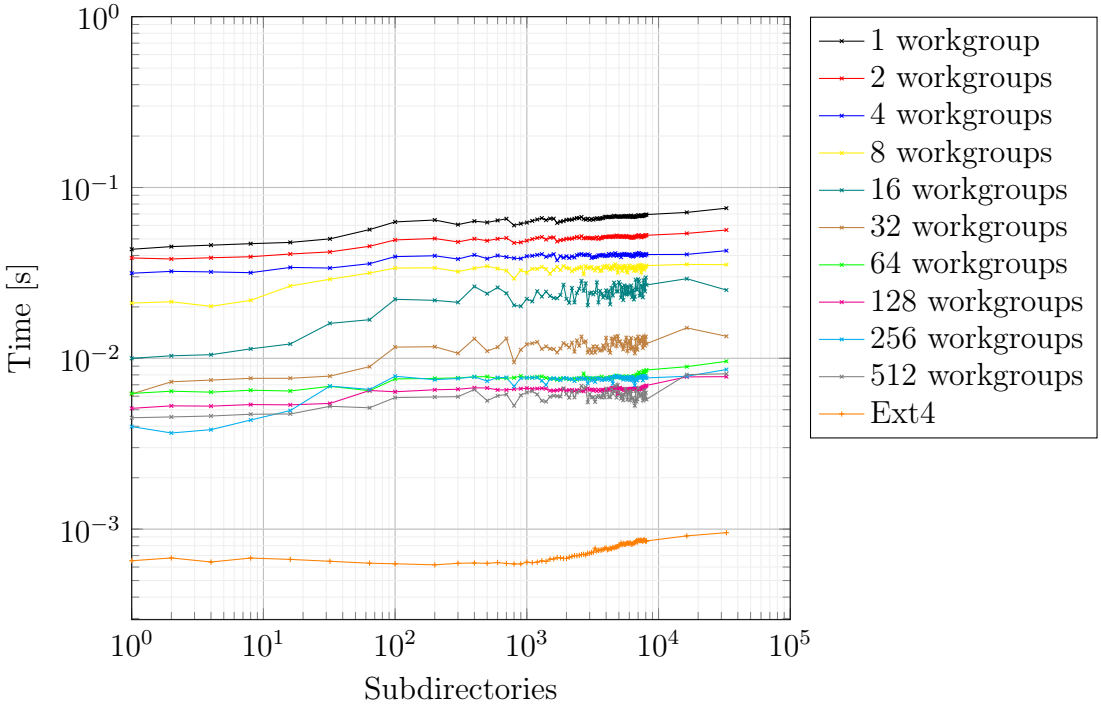

Figure 21: The time it takes Ext4 and the GPU directories using different numbers of workgroups to perform 512 file lookups in a directory with an increasing number of subdirectories. We exclude the base latency of 0.012 s from the GPU directory results. Notably, increasing the number of workgroups decreases the time it takes to perform the lookups, and the lookup time is very constant for large directories. The log scale may cause the impression that Ext4 lookup times grow faster than the GPU directory lookup time, but this is not the case. 512 lookups in a directory with only one subdirectory are 1.74 times faster than 512 lookups in a directory with 32768 subdirectories using the GPU directories compared to a factor of 1.46 when using Ext4.

6.8 Wide Directory Creation

In Figure 22, we measured the time it takes to create a directory with many subdirectories. We test Ext4 and compare it to the GPU directories using two

workgroup sizes, 64 and 32 SIMD lanes, with one and two workgroups per size.

First, we show the results of the GPU directories and then how they compare to Ext4. After creating 600 and 1000 directories, errors started appearing when using more than one workgroup, so we limited tests to one workgroup per size after that. This benchmark benefits from the new directory H-tree structure because all directories are located in the same directory.

After 200 subdirectories, the graphs are linear, which is to be expected, because the subdirectory creation can be split into two phases: lookup and write to NVM. The lookup is constant in large directories, as seen in Figure 20, and the amount of data that has to be written is also constant.

It is also worth noting that more total SIMD lanes reduce the time it takes to create the subdirectories. Therefore, one workgroup with 32 SIMD lanes is the slowest, two workgroups with 32 SIMD lanes (64 SIMD lanes in total) are very similar to one workgroup with 64 SIMD lanes, and the two workgroups with 64 SIMD lanes (128 SIMD lanes in total) are the fastest. The difference between one and two workgroups is small because the directory's inode is locked; thus, only one workgroup at a time can write to the NVM.

The following comparisons exclude the base latency measured in Figure 15. Similar to other benchmarks, the base latency of Ext4 is much lower than the GPU directories; in this case, Ext4 is 239 times or 0.0068 s faster when creating a single subdirectory. However, with increasing subdirectories, the relative performance difference between the GPU directories and Ext4 shrinks. The relative difference plateaus at 1200 subdirectories and stays roughly the same. After 1200 subdirectories, Ext4 is consistently around 4.7 times faster. This indicates that directory creation in Ext4 is faster than when using the GPU directories, regardless of base latency.

As we would expect, creating one directory using Ext4 is 28752 times faster than creating 32768 directories. However, with the GPU directories creating one directory is only 590 times faster than creating 32768 directories. This again indicates some hardware or software bottlenecks.

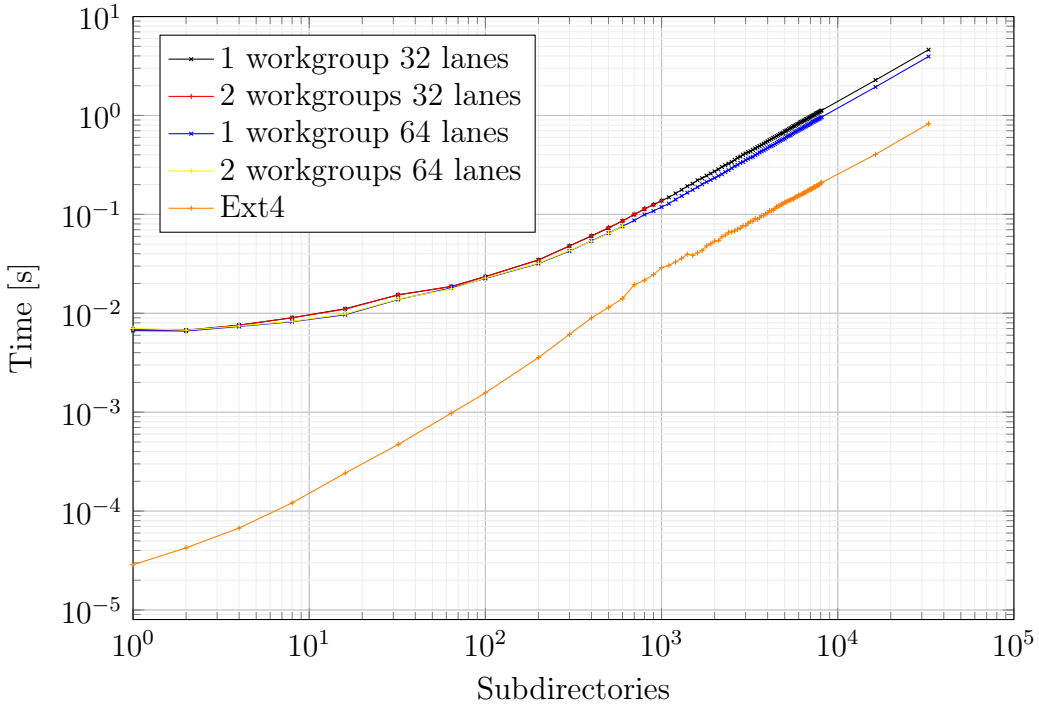

Figure 22: The time it takes Ext4 and the GPU directories to create and add subdirectories to the same directory. We exclude the base latency of 0.012 s from the GPU directory results. After 200 subdirectories, the graphs are linear. It is also worth noting that more workgroups and SIMD lanes only slightly decrease the directory creation time. Ext4 is faster than the GPU directories. After 1200 subdirectories, Ext4 is consistently around 4.7 times faster.

6.9 Discussion

This section will discuss our results, what we have learned, how the GPU directories can be improved, and the broader context of the results.

The base latency represents a significant difference between the GPU directories and Ext4 across all benchmarks. It is, therefore, crucial to reduce this latency. We see two possible ways to reduce the latency. First, remove the GPU task 10 s timeout. This would allow the GPU shader to run permanently, removing GPU initialization latency every 10 s and only leaving transmission latency over the PCIe bus. The second way is to determine why Maucher [17] measured a lower latency than we did; this may allow us to reduce latency even further.

Sometimes the first command executed by the GPU takes one to five minutes to complete. This is substantially longer than the latency measured in Figure 15. This has to do with shader compilation. The GLSL [14] shader is first compiled to SPIR-

V [8], an intermediate language, and then compiled again to the binary running on the GPU. The first compilation step is not part of our measurements, but the second is done right before the shader execution and thus causes a substantially longer latency. We were able to verify this using gdb [5]. Before the shader is executed, the AMD compiler (ACO) [1] is executed. ACO compiles an intermediate language to AMD RDNA-specific binary code.

One advantage of the GPU is parallelism. In our context, workgroups on the GPU are comparable to threads on the CPU. Like CPU threads, workgroups can independently and synchronously create or look up files and directories. Especially in the directory lookup tasks, more workgroups improved performance drastically. We expect that directory creation performance can also be improved by using more workgroups if the locking described in Section 4.1 is implemented because the current locking in the directory inode is too restrictive.

The results show that even if the shader code sometimes does not directly take advantage of more SIMD lanes, in most cases, workgroups with 64 SIMD lanes outperformed workgroups with 32 SIMD lanes. However, all 64 SIMD lanes execute the shader even if the shader code does not directly take advantage of all 64 SIMD lanes. We also explained in Section 5.2 that 64 SIMD lanes are the highest number of SIMD lanes that synchronize correctly. This could have something to do with the physical vector size of the GPU. Therefore, we think the AMD Radeon RX 6800 is better suited to operate with 64 SIMD lanes. However, we would also like to see if more than 64 SIMD lanes improve performance even further.

In the Deep directory creation benchmark, the GPU directories beat Ext4 when `std::filesystem::create_directories` is used for directory creation. However, other benchmarks, e.g., wide directory creation and single deep directory lookup, showed that the GPU directories are slower than Ext4 per operation. This can have multiple reasons:

- Our code needs more optimization, and the Ext4 code is very optimized.
- The caches are currently too small for the number of directories we used for benchmarking.
- The GPU is slower than the CPU when calculating, e.g., hashes.
- The PCIe bus latency slows down sequential reads e.g. when following a linked list.

The first two reasons can be addressed in the future, and the latter two can not be solved but measured and evaluated. Currently, we use the same hash algorithm as Ext4, which is only single-threaded; a multithreaded algorithm could improve GPU performance.

All benchmarks we evaluated are synthetic and therefore differ from typical file system usage. Typical file system use cases do not only comprise of hundreds of directory lookups or thousands of directory creations. They also involve file creation and read or write access to existing files. In his thesis, Maucher [17] shows file creation and file write performance of 1.4 - 1.9 Gbit/s, close to Intel Optane's maximum write bandwidth of 2 Gbit/s. However, we do not have measurements related to file read bandwidth on the GPU but suspect that it is similar to write bandwidth, thus around 70 - 95% of Intel Optane's maximum read bandwidth. Considering this, we can assume that the file reads and writes in typical file system use cases will hide the performance difference between the GPU directories and Ext4. Therefore, we think that GPU accelerated file systems are a feasible solution.

7 Future Work

There are still some areas where the current directory implementation could be improved, including file deletion, tree depth, better locking, caching in VRAM, size of workgroups, number of workgroups, reducing linear search, and reducing latency.

File deletion: The directory entry list and leaf block allocator are designed with possible file deletion in mind. However, file deletion would require tree balancing, which has not been implemented so far. The tree can only manage a limited number of hash regions. Therefore tree balancing is necessary to avoid unevenly filled hash regions and thus increase the number of files the tree can handle.

More levels: More levels would allow the H-tree to access more than 16.646.400 files per directory. Currently, the block split algorithm can only handle trees of depth three, but it could be made more recursive to support more.

Better locking: Currently, the whole inode of a directory is locked whenever a file is inserted into it. This simplifies keeping the directories consistent but reduces parallelism. Theoretically, locking is only required on the allocation bitmap and when a hash region splits.

Caching in VRAM: At the moment, the cache is located inside the local memory of each workgroup. Caching in the VRAM would reduce the number of cache misses because it would be bigger and shared by all workgroups. However, this would also require locking, which is unnecessary if every workgroup has its own cache.

More SIMD lanes: Due to synchronization issues, our implementation is limited to 32 or 64 SIMD lanes depending on the hardware. A solution could be to implement new barriers and other synchronization primitives like we already did in Listing 5. Increasing the number of SIMD lanes per workgroup could improve performance and use the GPU more effectively because each workgroup would be

able to process more data in parallel.

More workgroups: Due to visibility issues between workgroups, creating directories with multiple workgroups can lead to inconsistencies. Solving this problem is essential because using multiple workgroups to create and lookup files or directories simultaneously is a significant performance benefit.

A new level to avoid linear search: Another level could be added between the tree blocks and leaf blocks. Every hash region would have its own block that maps a file name hash to the entry inside the leaf blocks. Therefore the GPU could locate a file inside a hash region by checking the block instead of using a linear search inside the leaf blocks. This will increase the lookup speed because multiple entries can be checked in parallel.

Inode indirection: Currently, inode data pointer indirection is limited to one level. However, all allocations related to directories are only 4 kB in size. We, therefore, had to increase the indirection block size from 128 B to 4 kB so enough pointers could be stored in one indirection level. This is not ideal because most of the indirection block is unused in small directories, and the block has to be initialized with zero, which means a lot of potentially unnecessary writes to NVM. More indirection levels with 128 B blocks would alleviate these issues.

latency: In Chapter 6, we measured a five to ten times higher base latency than Maucher [17] on a very similar test system. We were not able to find the reason for the difference in latency. However, reducing base latency is crucial for a file system. Thus, further research is required.

8 Conclusion

This thesis aimed to extend the implementation of GPU4FS by adding new H-tree-based directories and to determine if a GPU-accelerated file system is feasible. GPU4FS is a file system demonstrator that offloads file system management to the GPU to reduce CPU usage.

In this thesis, we described the design and implementation of our H-tree-based directories. We evaluated our directory implementation with a number of synthetic benchmarks and compared the performance to Ext4, a conventional Linux file system.

Our evaluation tested latency, directory creation, and directory lookups in both long directory chains and directories with many direct subdirectories. The results showed that commands executed on the GPU have a base latency of 0.012 s. Furthermore, Ext4 outperformed the GPU directories in most benchmarks. The base latency heavily influenced short benchmarks; Ext4 was between 1000 and 22 times faster than the GPU directories. In longer benchmarks, this performance gap shrunk, and in most cases, Ext4 was between 119 and 2 times faster. One

exception is the benchmark that tested long directory chain creation, where the GPU directories managed to be faster than the C++ file system library. However, we only evaluated synthetic benchmarks that do not necessarily reflect real-world file system use cases.

Prior work by Maucher [17] lets us assume that real-world file system use cases that also include reading and writing files will hide the performance gap between the GPU directories and Ext4. Thus, we conclude that GPU-accelerated file systems are feasible, and more research should be conducted in this area.

We also present possibilities for future work to improve the GPU directories' latency and performance or to add a new feature necessary in a file system.

References

[1] Amd compiler (aco). https://gitlab.freedesktop.org/mesa/mesa/-/tree/ffebe480133479be58eb6057f08bec893cd999f8/src/amd/compiler. Accessed: 30.03.2023.

[2] Amd radeon rx 6800. https://www.amd.com/en/products/graphics/amd-radeon-rx-6800. Accessed: 24.03.2023.

[3] C++. https://isocpp.org/. Accessed: 2.03.2023.

[4] G++. https://gcc.gnu.org/. Accessed: 2.03.2023.

[5] The gnu project debugger (gdb). https://sourceware.org/gdb/. Accessed: 30.03.2023.

[6] Daniel Bittman, Matthew Gray, Justin Raizes, Sinjoni Mukhopadhyay, Matt Bryson, Peter Alvaro, Darrell D.E. Long, and Ethan L. Miller. Designing data structures to minimize bit flips on nvm. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 85–90, 2018.

[7] Google. glslc. https://github.com/google/shaderc. Accessed: 22.02.2023.

[8] The Khronos® Group. Spir-v. https://www.khronos.org/spir/. Accessed: 2.03.2023.

[9] The Khronos® Group. Vulkan. https://www.vulkan.org/. Accessed: 22.02.2023.

[10] The Khronos® Group. Opencl overview, 2022. https://www.khronos.org/opencl/.

[11] Christoph Hellwig. Xfs: the big storage file system for linux. *; login:: the magazine of USENIX & SAGE*, 34(5):10–18, 2009.

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

[13] Linux kernel. The hash function used by ext4. linux/fs/ext4/hash.c https://github.com/torvalds/linux/blob/094226ad94f471a9f19e8f8e7140a09c2625abaa/fs/ext4/hash.c. Accessed: 14.11.2022.

[14] John Kessenich, Dave Baldwin, and Randi Rost. The opengl®shading language, 2014.

[15] Hyunjun Kim, Joonwook Ahn, Sungtae Ryu, Jungsik Choi, and Hwansoo Han. In-memory file system for non-volatile memory. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, page 479–484, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2513228.2513325.

[16] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[17] Peter Maucher. Gpu4fs: A graphics processor-accelerated file system, 2022.

[18] NVIDIA. Cuda zone. https://developer.nvidia.com/cuda-zone. Accessed: 27.01.2023.

[19] 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*, pages 304–315, 2019.

[20] Daniel Phillips. A directory index for EXT2. In *5th Annual Linux Showcase & Conference (ALS 01)*, Oakland, CA, 2001. USENIX Association. https://www.usenix.org/conference/als-01/directory-index-ext2.

[21] Gianlucca O. Puglia, Avelino Francisco Zorzo, César A. F. De Rose, Taciano Perez, and Dejan Milojicic. Non-volatile memory file systems: A survey. *IEEE Access*, 7:25836–25871, 2019.

[22] Daniel A Reed and Jack Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[23] Card R'emy, Ts'o Theodore, and Tweedie Stephen. The second extended filesystem (ext2), 1993.

[24] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[25] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. Nvc-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, pages 1–8, 2015.

[26] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: Integrating a file system with gpus. *ACM Trans. Comput. Syst.*, 32(1), feb 2014. https://doi.org/10.1145/2553081.

[27] Peter Snyder. tmpfs: A virtual memory file system. In *Proceedings of the autumn 1990 EUUG Conference*, pages 241–248, 1990.

[28] Junji Tominaga. Topological memory using phase-change materials. *MRS Bulletin*, 43(5):347–351, 2018.

[29] Lukas Werling, Christian Schwarz, and Feank Bellosa. Towards less cpu-intensive pmem file systems. Talk presented at Fachgruppentreffen Betriebssysteme 2021, Trondheim. https://www.betriebssysteme.org/wp-content/uploads/2021/09/FGBS_Herbst2021_Folien_Werling.pdf, 2021.

[30] 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)*, pages 323–338, 2016.

[31] Chun Jason Xue, Youtao Zhang, Yiran Chen, Guangyu Sun, J. Jianhua Yang, and Hai Li. Emerging non-volatile memories: Opportunities and challenges. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, pages 325–334, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/2039370.2039420.