Karlsruhe Institute of Technology

File Deletion and File Movement for GPU4Fs

Bachelor's Thesis
submitted by

cand. inform. Leoluca Tinzmann

to the KIT Department of Informatics

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

24. Juni 2024 – 24. October 2024

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

Karlsruhe, October 24, 2024

Abstract

Persistent memory, like Intel Optane, is a step to narrow the performance gap between non-volatile Solid State Drives (SSDs) and volatile Dynamic Random Access Memory (DRAM). However, file systems backed by Intel Optane cause disproportionate CPU usage when compared to file systems backed by SSDs. GPU4FS intends to solve that problem by offloading all file system operations to the GPU. We extend the limited feature set of GPU4FS to include file deletions and moves. In our evaluation, we compare our implementation to the CPU file system ext4. We conclude that GPU file deletions can at least be four times faster than its CPU counterpart, depending on the number of files and structure of directories that contain the files. However, GPU file moves are slower than CPU file moves due to the transfer time of file paths from DRAM to VRAM.

Contents

Abstract **v**

Contents **1**

1 Introduction **5**

2 Background **7**
2.1 File Systems 7
2.1.1 Files and Directories 7
2.1.2 File Moves and Deletions 8
2.1.3 Crash Consistency and Journaling 9
2.1.4 Garbage Collection 9
2.2 H-Tree of GPU4FS 9
2.2.1 Structure 10
2.2.2 H-Tree Operations 11
2.3 Persistent Storage 12
2.3.1 Memory Hierarchy and Memory Wall 12
2.3.2 Intel Optane 13
2.4 GPU Architecture and Programming Models 14
2.4.1 Heterogeneous-Compute Interface for Portability 14
2.4.2 NVIDIA GPU Architecture 14
2.4.3 CUDA 15

3 Related Work **17**
3.1 GPU4FS 17
3.1.1 The original GPU4FS 17
3.1.2 Previous Work on GPU4FS 19
3.2 Parallel File Systems 20
3.3 GPUs and File Systems 22
3.4 Load Balancing Algorithms 22

4 Design **25**
4.1 File Deletions and File Moves 25
4.1.1 Unlinking a File from an H-Tree 25
4.1.2 Parallelizing File Deletion and File Movement 26
4.1.3 Characteristics of Consumer-Producer-Pattern for File Deletions 27
4.1.4 Characteristics of the Shared Buffer 28
4.1.5 Shared Buffer as Collection of Arrays 29
4.1.6 Alternative Data Structures 30
4.1.7 Summary of File Deletions and File Moves 30
4.2 Repairing the H-Tree 31
4.2.1 Repairing from Leafs to Root 31
4.2.2 Repairing by Reconstructing 33
4.2.3 Summary 34

5 Implementation **35**
5.1 Locking 35
5.1.1 Locking for Unlinking 35
5.1.2 Locking Elements of Shared Buffers 36
5.2 Command Descriptors 36
5.3 Load Balancing 38
5.4 Start Index for Search in Shared Buffer 38
5.5 Summary 38

6 Evaluation **41**
6.1 Configuration and Methodology 41
6.1.1 Testbed Configuration 41
6.1.2 Measurement Tools 41
6.2 Measurements for Rebuilding 44
6.2.1 Testing Scenarios 45
6.2.2 Considerations regarding File Deletions and File Moves . 45
6.3 Evaluation of File Deletion 46
6.3.1 Testing Scenarios 46
6.3.2 Measurements 46
6.3.3 Discussion 51
6.4 Evaluation of File Moves 54
6.4.1 Testing Scenaros 54
6.4.2 Measurements 54
6.4.3 Discussion 56
6.5 Summary 56

7 Future Work **57**
7.1 Separate Research Topics 57
7.2 Further Implementation 58

8 Conclusion **59**

Bibliography **61**

Chapter 1

Introduction

High-Performance Computing (HPC) requires fast access to memory and ever greater memory capacity [11]. Dynamic Random Access Memory (DRAM) is a volatile storage medium, capable of a lower latency per memory access than Solid State Drives (SSDs) or Hard Drive Disks (HDDs) [14]. However, DRAM's memory capacity is constrained by power consumption or limited storage board area [30]. In contrast, SSDs are a non-volatile storage medium with a larger capacity than DRAM but also higher latency [14].

Intel Optane is a persistent storage medium and an attempt to combine the low latency of DRAM with the non-volatility of SSDs [19].

However, according to Werling et al. [40], random writes by multiple threads to an Optane-backed ext4 file system (FS) cause disproportionate Central Processing Unit (CPU) utilization compared to ext4 file systems backed by HDDs or SSDs.

To decrease the workload of the CPU when interacting with Intel Optane, Maucher et al. [25] proposed a new file system called GPU4FS. In GPU4FS, a process accesses memory on Intel Optane by making a request to the GPU using a shared buffer. The GPU then processes the request and accesses the memory on Intel Optane. In case of a read request, the GPU deposits the requested data in the shared buffer.

Currently, GPU4FS is not a fully functional file system. Instead, GPU4FS is a testbed for certain file system operations. Our goal is to extend the limited feature set of GPU4FS by adding file moves and file deletions to it, thereby further reducing the load on the CPU. By doing so, we also examine the feasibility and efficacy of parallelizing file moves and file deletions in an Intel Optane-backed and GPU-accelerated file system.

To examine the feasibility and efficacy of parallelizing file deletions and moves, we start in Chapter 2 by introducing information about file systems, file system operations and data structures, and storage media. Chapter 3 then explores cur-

rent work in the field of (parallel) file systems, load balancing algorithms, and General Purpose Computation on Graphics Processing Units (GPGPU). Chapters 4 and 5 contain the design and implementation of file deletions and file moves in GPU4FS. In chapter 6 we evaluate our design before we draw conclusions in chapters 7 and 8.

Chapter 2

Background

This chapter is a summary of background information necessary to understanding file systems (FS), Intel Optane, and Graphics Processing Units (GPUs). Section 2.1 is a quick summary of file systems. In section 2.2, we introduce the H-Tree of GPU4FS, an important data structure to accelerate file lookups in directories. We continue by explaining in section 2.3 why persistent storage media like Intel Optane are desirable for file systems. We conclude this chapter in section 2.4 by breaking down the inner workings of Graphics Processing Units (GPUs).

2.1 File Systems

A file system manages accesses by processes to one or more storage media by abstracting the flat physical address space of a storage medium into a directed mostly acyclic graph through the abstraction mechanisms of files and directories. The only circles in a file system structure are the references of directories to themselves and their parents. These references in Linux are represented by the "." and ".." entries, respectively [36].

"Mostly treelike structure" is an equally valid term for the structure of a file system. The term highlights the tendency of the file system structure to grow "wide" rather than "tall" which is important when we present the design of file deletions in chapter 4. Going forth, we will use "mostly treelike structure" instead of "directed mostly acyclic graph structure".

2.1.1 Files and Directories

A central task of file systems is to allocate memory space on a storage medium to file names [36]. ext4 and GPU4FS use a data structure called an "inode" (index-node) to do this [25, 4]. We will got into more detail about the design of GPU4FS

in chapter 3.1. This section introduces common concepts of file systems.

Inodes, Files & Directories

An inode is a collection of metadata as it contains pointers to memory regions of fixed size on a storage medium that store the data of a file. Inodes are used for both directories and files [36]. Going forth, the term "file" will be used for both regular files and directories, except when the difference matters.

GPU4FS and ext4 implement directories as special files that each hold a list of file name inode ID pairs that map file names to inodes. ext4 and GPU4FS allow for multiple file names to map to the same inode, if that inode represents a regular file and not a directory, as referencing directories by more than one file name can create unwanted ambiguities. An exception to this are the "." and ".." entries, which we explained in the introduction of section 2.1. The number of references to an inode is called the "hard link count" and it is stored in the inode's hard link counter [25, 4].

Beyond directories and regular files

In Linux, files can not only be used to store data but also to communicate with other IO devices like directly connected printers or special storage media like Intel Optane. Writing data to or reading data from such a device file causes the kernel to use the device drivers associated with the file instead of regular file system operations [39].

2.1.2 File Moves and Deletions

In this section we describe how to move and delete files.

Moving a File

Moving a file from directory A to directory B is done by removing the directory entry for the file from A and adding it to B. The process of removing a file entry is called "unlinking". Equally, the process of adding a file entry to a directory is called "linking" [36].

Deleting a File

To delete a file A contained in directory B, A must be unlinked from B. The following behavior now depends on whether the file is a regular file or a directory. If A is a regular file, then the hard link count is decreased by 1. Should the hard link count reach 0, then the memory areas occupied by the file can be reused by

giving them to the garbage collector. The garbage collector will be explained in section 2.1.4. If A is a directory, then all files in the subtree of the file system structure which has the directory as its root must be deleted as well. We call that "recursive hard link counter decrementation". Unlinking can be skipped, as the root is already unlinked and therefore inaccessible through the file system, as a directory can only ever have one file name mapped to it. Again, the memory areas occupied by all directories in the subtree and all regular files whose hard link count reaches 0 can be given to the garbage collector [36].

2.1.3 Crash Consistency and Journaling

Interrupting a file system operation may lead to an inconsistent file system state. A possible solution is "Journaling", which allows the file system to repair the inconsistent state by re-executing the interrupted operation [36].

A file system implementing Journaling first writes down what operation will be performed before then executing the operation. The entry for the operation is removed from the journal after the execution of the operation ends. In case of a crash, the file system can read the operation from the journal and perform it again. Important for this kind of file system is that all steps of an operation are "idempotent", meaning each step can be repeated endlessly without changing the result, compared to performing the step only once. Otherwise, re-executing the operation may lead to even more inconsistencies [36].

We will not cover Journaling as it is outside the scope of this thesis, and we will also not consider it when designing file deletions and moves in chapter 4.

2.1.4 Garbage Collection

"Garbage Collection" is the process of making previously used memory areas available again to the file system allocator [6]. For example, the garbage collector allows the file system to reuse the memory areas of deleted files.

Implementing garbage collection lies outside the scope of this thesis. We will only implement a stub function, which we use to indicate that a memory area can be processed by the garbage collection.

2.2 H-Tree of GPU4FS

The H-Tree was originally developed by Phillips [31] for Ext2 to speedup file lookups in directories. Kittner [20] transferred the concept of the H-Tree from Ext2 to GPU4FS. This section is dedicated to the H-Tree, as both file deletions and file moves modify the parent directory and therefore its H-Tree.

2.2.1 Structure

The core idea of the H-Tree is to add a balanced tree using hash values as keys to the linked list of directory entries to decrease the lookup time in directories. In GPU4FS, H-Trees are added to directories that have more than 255 entries. Directories with less than or equal to 255 entries effectively use a simple linked list instead [20].

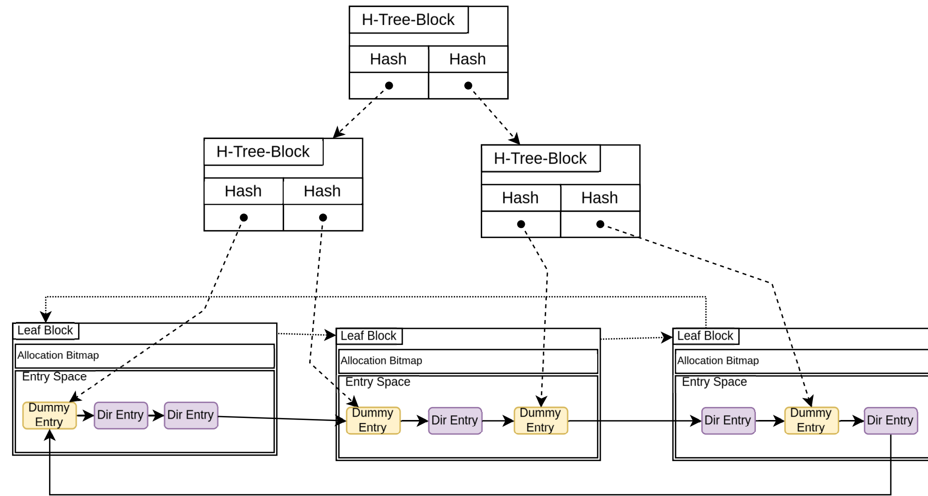


Figure 2.1: The structure of an H-Tree. To keep the illustration comprehensible, an H-Tree-Block can only hold up to 2 pointers in this illustration. The leaf level contains leaf blocks that hold the linked list of directory entries. A leaf block may contain more than one dummy entry. The first entry of the first leaf block is always a dummy entry. H-Tree-Blocks in the first level point to dummy entries in the leaf level. The root block points to the H-Tree-Blocks of the first level. It is noteworthy, that an H-Tree may also have 2 levels instead of 3. In that case, the root block would be located on the first level and points directly to dummy entries.

Figure 2.1 shows the structure of an H-Tree. The leaf level contains the linked list of directory entries, which is split into different hash regions. Each hash region begins with a “dummy entry”, which holds the lowest hash value of the hashed file names of the region. The hash regions are sorted from lowest to highest value in the linked list of directory entries, therefore making it split-ordered. Hash regions can hold up to 255 directory entries [20].

All levels, except the leaf level, contain so-called “H-Tree-Blocks”. An H-Tree-Block holds an array of pointers, each pointing to either a dummy entry or an H-Tree-Block. A hash value in an H-Tree block entry is either the hash value of a hash region or the lowest hash value of the array of another H-Tree-Block.

Currently, an H-Tree can only have up to three levels. Each H-Tree-Block can only hold up to 255 hash values[20].

Originally, Kittner [20] proposed $255 * 255 * 256 = 16646400$ as the maximum number of directory entries with the hash region size being 256 and not 255. This seems to have changed as the development of GPU4FS progressed. In the version of GPU4FS we use, the hash region size is 255. Therefore, a directory can store up to $255^3 = 16581375$ directory entries.

A key difference to the original H-Tree by Phillips is that the blocks of the leaf level in GPU4FS are mere storage containers for the linked list of directory entries. A leaf block contains an allocator to allocate space inside the block for directory entries and dummy entries. Within the original H-Tree by Phillips, H-Tree-Blocks point directly toward leaf block which contain an array to hold directory entries only [20, 31].

2.2.2 H-Tree Operations

This section covers how to find a directory entry in an H-Tree and how to add an entry to the H-Tree.

File Lookup

The algorithm to find a file F in a directory is as follows:

1. Find the largest entry in the root block of the H-tree that still has a lower hash value than the hash value of the file name of F. Follow the pointer $p1$ of that entry to the first-level block I.

2. Repeat the previous step for I until you get to the leaf level.

3. Search the linked list of the hash region to find the directory entry for F [20].

The parallel nature of GPUs speeds up the search in the root block and first-level blocks as the entries are of fixed size and stored in an array. There is no such speed up for the search in the linked list of entries as it is a list and not an array [20].

Entry Insertion

The algorithm to insert a file F into a directory D is as follows:

1. Perform a file lookup to determine the hash region H for F.

2. Check if a file entry can be added to one of the leaf blocks that contain the hash region H. If there is enough space in the leaf blocks, add the entry to them. Otherwise, add a new leaf block to the linked list of leaf blocks and add the entry to that leaf block.

3. Check if the size of the hash region exceeds 255. If so, then split the hash region by adding another dummy entry. The entries of the hash region are divided equally among the two new hash regions. A new entry for the new hash region is added to the H-tree. Depending on how much space there is left in the H-tree, either a new level is added if the H-tree does not have a depth of three or a block is split [20].

We do not cover what happens when there are more files than the maximum number of files in an H-Tree directory, and consider that subject to be future work.

2.3 Persistent Storage

This section motivates the use of Intel Optane as a storage medium and presents the performance characteristics of Intel Optane. In section 2.3.1, we begin by explaining the problem of the "memory wall" and its solution, the "memory hierarchy". We continue in section 2.3.2 with placing Intel Optane in the memory hierarchy. We then analyze the performance characteristics of Intel Optane that might be of interest to our implementation of file deletions and file moves.

2.3.1 Memory Hierarchy and Memory Wall

Since the 1980s, the gap between the speed at which processors can request memory and the speed of main memory to cope with such requests increases [14]. This problem is called the "memory wall" [41].

To alleviate this issue, computers utilize different kinds of memory which form a memory hierarchy. Solving the problem of the memory wall with a memory hierarchy is most suited for programs which access memory with temporal or spacial locality. At the top of the memory hierarchy are registers of processors. Next in the hierarchy are caches, which are further divided into L1, L2, and L3 caches. Latency and storage capacity increase from L1 to L3. Caches are followed by main memory in the hierarchy. Further down the hierarchy are storage devices like SSDs, HDDs or magnetic tapes. Latency and memory capacity increase from the top to the bottom of the hierarchy. The differences in these characteristics are the result of different construction methods and components [14].

Despite the memory hierarchy, it would be ideal if computers had vast amounts of memory with low latency and high bandwidth [14].

2.3.2 Intel Optane

Intel Optane is an attempt at merging or at least narrowing the gap between main memory and storage media like an SSD in the memory hierarchy. Effectively, Optane tried to combine the low latency of DRAM with the large storage capacity and non-volatility of SSDs [17].

Flavors and Modes of Intel Optane

Intel Optane is available as an SSD which can be connected to the computer via Peripheral Component Interconnect Express (PCIe) or as Intel Optane Persistent Memory Modules (Intel Optane DC PMMs), which, like DRAM, can be connected to the computer via a motherboard's memory channels. We only use Intel Optane DC PMMs [17].

Intel Optane DC PMM can be used as the system's main memory in the "memory mode", optionally with DRAM as a cache, or as a separate storage device in the "App Direct Mode" which can either be used by filesystems or mapped into an address space using a device file. Intel Optane only stores data persistently in the App Direct Mode and not in the memory mode [19].

We use both options of the App Direct Mode as we map Intel Optane into the GPU4FS address space using a device file and make it visible to the GPU, so that the GPU can then access it. We also install ext4 on Intel Optane to compare the runtime of our file deletions and file moves commands with their ext4 counterparts. ext4 and GPU4FS are strictly separated on Intel Optane.

Performance Characteristics of Intel Optane

Yang et al. [19] analyze the performance of Intel Optane and compare it with DRAM. This section is a brief summary of their conclusions that may be relevant to the implementation of file deletions and file moves in chapter 4. Knowing the performance characteristics and quirks of Intel Optane is important when we evaluate file deletions and file moves of GPU4FS.

The authors of [19] assume that Intel Optane merges adjacent memory accesses into single 256 byte accesses. Reading Intel Optane memory sequentially has an up to two times lower latency than reading it randomly and an up to four times higher bandwidth. Yeng et al. also note that it requires more threads to saturate Intel Optane read bandwidth than write bandwidth (17 vs. 4 for 6 Intel Optane DC PMMs) and that accesses of 256 Bytes seem to be the best size to fully saturate the bandwidth.

2.4 GPU Architecture and Programming Models

The final section of the chapter is dedicated to GPU architecture and GPU programming models. Understanding both is critical to creating efficient and fast file system operations in chapter 4.

Unfortunately, there is no standardized terminology for GPU terms yet. NVIDIA, Advanced Micro Devices (AMD) and OpenCL each have their own terms for the same GPU components. A fourth terminology was proposed by Patterson and Hennessy [14]. We will use NVIDIA's terminology, as we use NVIDIA's programming model Compute Unified Device Architecture (CUDA).

2.4.1 Heterogeneous-Compute Interface for Portability

In addition to CUDA, we also use an API called Heterogeneous-Compute Interface for Portability (HIP) [38], which facilitates the porting of code from NVIDIA to AMD GPUs and back. HIP achieves this by presenting a unified interface for both NVIDIA and AMD GPUs to the programmer, which is linked during compilation to the corresponding libraries for the GPU. The portability of the code comes at the cost of missing features, as not all features of CUDA, like for example dynamic parallelism, are supported by HIP [3].

2.4.2 NVIDIA GPU Architecture

The remaining sections of this chapter are a summary of Hennessy's and Patterson's "Computer Architecture: A Quantitative Approach" [14] about GPUs.

The information presented in this section represents a greatly simplified view on GPUs as it leaves out many features such as Tensor Cores, Streaming Multiprocessor (SM) partitions or datapaths [28]. While such features can greatly improve the performance of programs running on the GPU, they are not necessary to understanding the common performance pitfalls when programming GPUs.

In essence, a GPU is a collection of so-called "Streaming Multiprocessors" (SM), which work in parallel and are similar to vector processors. Each Single Instruction Multiple Data (SIMD) lane of every SM has its own Load store unit and complement of registers. SIMD lanes can access "local memory", which is unique per SM, and Video Random Access Memory (VRAM), which is accessible by all lanes of a GPU [14].

Conditional Branching and Coalescent Memory Access

SMs, like vector processors, handle conditional branching through the use of mask registers, although there are significant differences in how the mask registers are

managed. We will not go into detail on how mask registers are managed. The important takeaway is that if only some, not all, SIMD lanes of an SM execute a branch, then all SIMD lanes of the SM execute both branches with those not participating in a branch turned off. Split execution of conditional branches by SIMD lanes degrades performance and should be avoided [14].

A significant difference to vector processors is how SMs access VRAM. SMs exclusively use gather/scatter instructions to access memory. An “address coalescing unit” then tries to unify the individual memory accesses. Unifying memory accesses can be done if the memory accesses try to access neighboring memory. The result is fewer memory accesses which can be processed faster. For this reason, the programmer should always try to access memory in such a way that memory accesses can be unified [14].

2.4.3 CUDA

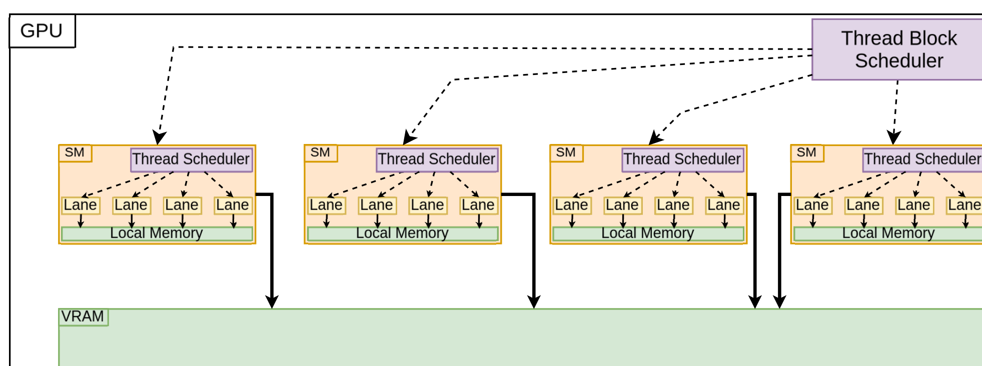


Figure 2.2: Simplified structure of a GPU. The Thread Block Scheduler schedules Thread Blocks for the different Streaming Multiprocessors. Each Thread Scheduler schedules Warps for the SIMD lanes of a SM. All lanes within an SM have access to local memory. All lanes of all SMs have access to Video Random Access Memory (VRAM). [14]

Figure 2.2 summarizes the interaction of GPU architecture and CUDA. CUDA for the most part hides the structure of the GPU. The smallest programmable unit in CUDA is the CUDA Thread. 32 consecutive CUDA Threads, called a “Warp”,‘ are mapped to the SIMD lanes of an SM by the Thread Scheduler. Up to a 1024 CUDA Threads form a “Thread Block”, which is assigned by the Thread Block Scheduler to an SM. The number of CUDA Threads per Thread Block can exceed the number of SIMD lanes per SM. To execute all CUDA Threads, GPUs multiplex the execution of CUDA Threads on SIMD lanes. This has the added benefit that branch divergence does not affect performance if the divergence is between

Warps. A GPU program in execution is called a "Grid" and is made up of all Thread Blocks.

CUDA Threads have access to several kinds of memory, which abstract VRAM, local memory, and registers of a GPU. "Global memory" is accessible by all CUDA Threads. "Shared Memory" is unique to every Thread Block and can only be accessed by the CUDA Threads that make up the block. Every CUDA Thread also has its own local variables. CUDA does not map these abstractions directly to hardware components, as for example shared memory can consist of both registers and local memory [28]. In general, we can say that shared memory is faster but also smaller than global memory [13].

Chapter 3

Related Work

This chapter explores current work in the field of GPU4FS, parallel file systems, General Purpose Computation on Graphics Processing Units (GPGPU), and load balancing. All works related to GPU4FS [25, 23, 32, 20], share similarities to file systems, parallel file systems, and file system accelerators in particular. As we will see in chapter 4, the algorithm used there to delete files and repair H-Trees is inspired by parallel algorithms to traverse graphs and rebalancing algorithms.

3.1 GPU4FS

3.1.1 The original GPU4FS

GPU4FS aims to be a POSIX-compliant file system demonstrator which tries to accelerate file system operations using a GPU. GPU4FS was developed with Intel Optane as the storage medium in mind [25].

Motivation for GPU4FS

Werling et al. [40] showed that Intel Optane causes disproportionate Central Processing Unit (CPU) usage when used as the storage medium for a file system. A possible cause could be synchronous accesses of CPU cores to Intel Optane. These cores stall in the time it takes Intel Optane to execute the requested operations, wasting valuable CPU time. Werling et al.s' solution is to use a Direct Memory Access (DMA) engine like Intel IO Acceleration Technology (Intel IOAT) for low demand memory accesses and only use the CPU for high demand accesses. While this approach looks promising, Werling et al. suggests further research with other DMA devices like GPUs leading to GPU4FS.

Figure 3.1 describes the principle architecture of GPU4FS. A process wishing to access data on Intel Optane makes a request via a shared buffer with the GPU.

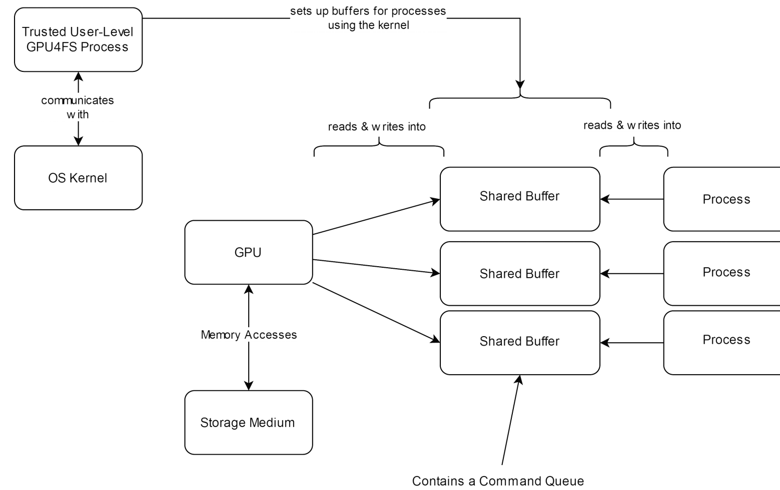


Figure 3.1: The principle architecture of GPU4FS - Processes communicate with the GPU via shared buffers which are setup by a trusted user level process of GPU4FS using the operating system kernel. The GPU communicates with the processes using these buffers. Every buffer contains a command queue for commands to the file system. Only the GPU has access to the storage medium.

The GPU then executes the request and deposits any requested data or operation result in the shared buffer [25]. Currently, GPU4FS does not fully implement this architecture. It is instead a testbed for executing file system operations on the GPU. There is only a user-level process and no buffers that are shared with other processes. Instead, commands are scheduled using a CUDA feature called "streams".

Data Structures of GPU4FS

As is mentioned in chapter 2.1.1, GPU4FS uses inodes to organize data [4, 25]. Figure 3.2 shows the structure of a GPU4FS inode. It is noteworthy, that 2 Bytes after "mode" are unused. Allocating memory in GPU4FS is done by allocating "pages" of sizes of 256 B, 4 kB, 2 MB, and 1 GB similarly to how memory is allocated in DRAM. This is due to the byte addressable nature of Intel Optane in contrast to the block addressable of SSDs and HDDs [25]. The pointer size of GPU4FS is 64 bits, 57 bits are used to point to a location in memory. Three bits are unused. The remaining four bits are used for tagging. Going forth, we will call GPU4FS pointers "block pointers". In GPU4FS, a block pointer is in practice an offset from the storage medium's base [25]. The reason for using offsets as block pointers is that Intel Optane is mapped to a random point in the virtual address space of the process. Therefore, the pointer of the virtual address space to Intel

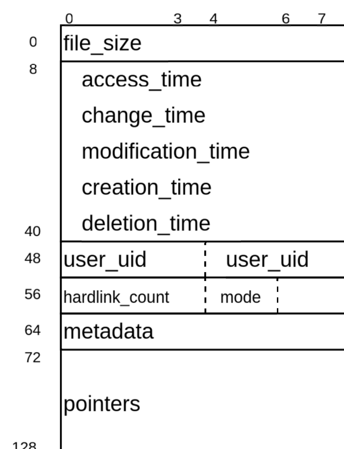


Figure 3.2: This figure shows the structure of a GPU4FS inode. Data structure size is measured in bytes. 2 bytes after “mode” are unused.

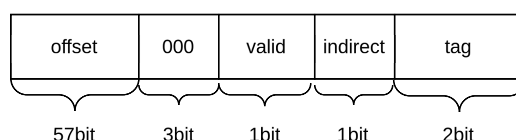


Figure 3.3: Bit distribution of a block pointer - 57 bits are used as an offset to point to a location on the storage medium. 3 bits are unused. The remaining bits are used for tagging (specifying the size of the block the block pointer points to) and to indicate if the block pointer is valid and / or indirect.

Optane is only constant during the lifetime of the process. Figure 3.3 shows the structure of a block pointer of GPU4FS [25].

3.1.2 Previous Work on GPU4FS

The original version of GPU4FS was written in Vulkan [16] and only supported three file system operations: Setting bytes to a specific value, copying bytes and creating files [24]. Since then, GPU4FS has been rewritten using Heterogeneous-Compute Interface for Portability (HIP) [38] and been extended with directories, Redundant Array of Inexpensive Disks (RAID), checksumming, and deduplication. We describe directories by Kittner [20] in chapter 2.2 as we actively use Kittner’s implementation of the H-Tree to modify directories. The remaining features are described in this section, as they are important to GPU4FS, but not file deletions and file moves.

Redundant Array of Inexpensive Disks

Redundant Array of Inexpensive Disks (RAID) describes a system of storage media linked together to increase read and write speeds and make disks more reliable. RAID comes in different flavors which increase either read and write speeds or reliability of data or both [1].

Lucka [23] adds RAID to GPU4FS. To support RAID on GPU4FS, Lucka has to modify the block pointer and superblock of GPU4FS. In his evaluation, Lucka comes to the conclusion that RAID on GPU4FS outperforms competitors like ext4 and BTRFS in certain configurations of RAID, while simultaneously greatly decreasing CPU usage.

We will not use Luckas modified version of GPU4FS, as that version is based on OpenGL Shading Language (GLSL) and not Heterogeneous-Compute Interface for Portability (HIP) [23].

Checksumming and Deduplication

Checksum functions take an input of arbitrary length and output a bit pattern of fixed size [1]. Checksums are for example used in BTRFS to verify the integrity of data [33].

Deduplication is a file system feature used to save storage space by detecting redundant data and replacing it with a link to a single instance of that data [26].

Rath [32] implements both checksumming and deduplication in GPU4FS. Rath's implementation is independent of the choice of the checksumming algorithm. Building upon checksumming, Rath implements inline deduplication. Rath comes to the conclusion that checksumming and deduplication on the GPU is best suited for many large pages. Fewer and smaller pages should be done by the CPU.

Again, we will not use the modified version of GPU4FS by Rath as his version of GPU4FS was written in Vulkan [32].

3.2 Parallel File Systems

In a parallel file system, multiple processors can try to access data in parallel, which may be spread across multiple locations. The interactions of the processors are managed by the file system. Figure 3.4 shows the common structure of such a file system with processors dedicated to IO tasks or computational tasks [7]. While GPU4FS is no cluster or distributed file system, as the examples in the following paragraph, the relationship of CPU and GPU Streaming Multiprocessors (SMs) shares some similarities with parallel file systems. The CPU can be seen as a computation processor that communicates with the GPU as a multicore IO processor. The communication medium are the shared buffers each process

shares with the GPU in GPU4FS. Notable parallel file systems are Galley [27], Vesta [7], Lustre [29] and Parallel Virtual File System (PVFS) [12].

Galley [27] is an attempt at optimizing a parallel file system for access patterns specific to scientific computing. Galley divides processors strictly into IO processors and calculation processors. This is similar to the relationship between CPU and GPU in GPU4FS. However, GPU4FS is not optimized for specific access patterns.

In Vesta [7] calculation processors do not communicate directly with IO processors but instead over an abstraction mechanism called "cells". Cells represent IO tasks which are distributed to different IO processors. Another feature of Vesta is that a single file can be processed by multiple calculation processors working in parallel to achieve a single goal. This is exemplified in a sorting algorithm "FastMeshSort" based on bitonic sort. While in GPU4FS, the different SMs can work together on one file, there is no abstraction mechanism like in Vesta to coordinate the different processors.

In contrast to Galley and Vesta, PVFS [12] and Lustre [29] both aim to be commercially usable parallel file systems. In PVFS, processors can be both IO processors and calculation processors. Lustre aims to be usable by thousands of nodes of a cluster. GPU4FS was developed for scientific purposes and does not operate on the same scale as PVFS or Lustre.

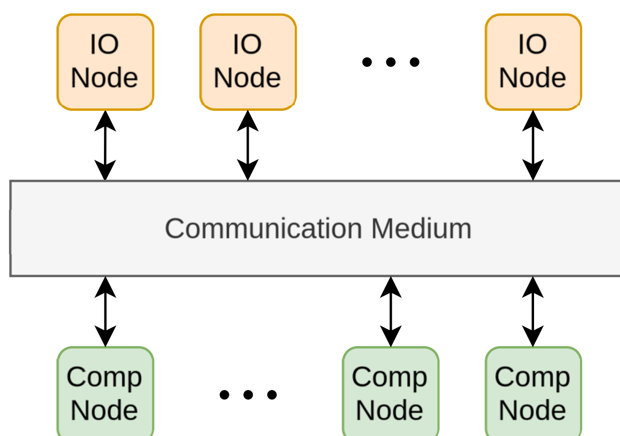


Figure 3.4: Generic parallel file system Structure - n IO processors handle IO requests by m computational processors. Depending on the implementation, it is possible for a processor to be both an IO processor and a computational processor. The figure is inspired by [7].

3.3 GPUs and File Systems

GPU4FS is not the only work that combines file systems and the GPU in some manner.

GPUfs by Silberstein et al. [35] is a POSIX like API which exposes the host's file system to the GPU. The most significant difference between GPUfs and GPU4FS is that GPUfs makes the host's file system available to the GPU, while GPU4FS is an entirely new file system for both the host and the GPU.

In 2015 Zhang et al. [42] made Solid State Drives (SSDs) accessible to the GPU. NVIDIA has incorporated the ability to access storage media into CUDA with Heterogeneous Memory Management [15]. GPU4FS uses Heterogeneous Memory Management to access Intel Optane.

3.4 Load Balancing Algorithms

In chapter 4, we will describe the deletion and rebuilding algorithms, which must traverse parts of the mostly treelike structure of the file system. The irregularity of the file system structure necessitates the use of load balancing algorithms.

Load balancing algorithms can be divided into static load balancing, which divides the load at compile time, and dynamic load balancing, which divides the load at runtime [34]. For the irregular tree-like structure of file systems, we believe dynamic load balancing to be more effective. Due to a lack of time, we implement only a basic static load balancing algorithm, which has to be replaced in future work with dynamic load balancing. That is why we explore the field of dynamic load balancing in this section. Dynamic load balancing is widely used in parallel calculations with irregular spaces. For example, Bian et al. [2] use load balancing to optimize the calculation of matrix-vector product of sparse matrices.

Kohring [21] uses the CPU time that has elapsed since the last load balancing step as a measure to determine if a processor should execute a load balancing algorithm. This is not applicable to the GPU, as time measurement on the GPU is difficult. We further discuss the problem in chapter 6.1.2.

Tomboulian [37] implements load balancing for Single Instruction Multiple Data (SIMD). We considered using a SIMD approach to implement file deletion, but ultimately rejected the idea because of the cost of synchronizing the Streaming Multiprocessors (SMs) and Warps. We go into more detail about our decision not use SIMD in chapter 4.1.3.

Lin et al. [22] present a dynamic load balancing algorithm that redistributes load only among neighboring processors. While the algorithm is easy to implement and effective at distributing the load, it also requires global synchronization, making it unsuitable for file deletion.

Particularly interesting for GPU4FS is [5] by Chen et al. who implement a task queue and load balancing for single and multi GPU systems. We did not use [5] as the approach is too comprehensive for our purposes. However, future implementations of GPU4FS might want to adopt the design in some manner, as the task queue paradigm in [5] is similar to how GPU4FS in [25] is designed to process requests of user processes.

Chapter 4

Design

In this chapter, we present our design of file deletions and file moves for GPU4FS. Sections 4.1.1 and 4.1.2 are about how to parallelize file deletions and file moves. We argue that a producer-consumer pattern is necessary to parallelize file deletions but is not required to parallelize file moves. In sections 4.1.3, 4.1.4, 4.1.5, and 4.1.6, we describe desirable properties for the shared buffer of the producer-consumer pattern and how to realize them. File deletions and file moves damage the H-Tree of the parent directory. Therefore, section 4.2 describes how to repair the H-Tree, again using a producer-consumer pattern.

4.1 File Deletions and File Moves

This section describes how we design our file deletion and file move algorithms. We will first explain how to unlink files from an H-Tree before we analyze how to parallelize file deletion and file moves.

4.1.1 Unlinking a File from an H-Tree

Chapter 2.2.2 explains how to insert entries into an H-Tree, thereby linking a file name to an inode. Unlinking a file is similar to linking a file in that first, a file lookup is performed to determine the hash region a directory entry is located in. Then, the algorithm removes the directory entry for the file from the linked list of directory entries.

Removing a directory entry like that does not update the allocation table of the affected leaf block, thereby creating a memory leak. More memory space is wasted by not merging hash regions or reducing the size of the tree if possible. We will go into more depth about how to repair an H-Tree in chapter 4.2.

4.1.2 Parallelizing File Deletion and File Movement

As explained in Chapter 2.1.2, the building blocks of file deletions and moves are unlinking and linking files, and recursive decrementation of hard link counters.

Unlinking and linking files consist of file lookups and operations to add or remove entries from an H-Tree. A file lookup is a mostly sequential operation. The only aspect of file lookups that can be effectively parallelized is the calculation of hash values for the search for directory entries in H-Trees [20]. Therefore, linking and unlinking are operations that can only be effectively parallelized if there are more than one operation.

Recursive decrementation of hard link counters can be parallelized to an extent as to decrease the hard link counter of a file, an algorithm needs to know the position of the file's inode in memory. Directories store the position of their children's inodes in the directory's linked list of directory entries. Therefore, once the content of a directory is read from memory, the hard link counters of the directory children can be decremented in parallel.

The Need for Producer-Consumer-Pattern for File Deletion

To avoid synchronization, parallel algorithms usually divide resources in time or space. This can only be done effectively if the amount of resources is known in advance.

The nature of recursive decrementation means we do not know how many hard link counters will have to be processed when we delete a file, unless we spend additional processing time or memory space. However, there are several problems with such an approach.

Storing Directory Entry Count reduces Parallelism

One way to determine how many hard link count decrementations a file deletion would entail is to store a count of how many files a directory and all its children contain. Storing the directory entry count for a directory and all its children would be inefficient. A change in the number of directory entries in a child directory would also lead to a change in the number of entries in the parent directory. Ultimately, storing such a count would greatly increase the number of accesses to the storage medium.

Prior knowledge about Directory State reduces Parallelism

Another issue is, the file system's state might change in the time between reading the state of the file system and the beginning of the execution. As a result, a file deletion algorithm that is supposed to delete a directory might have an inaccurate

hard link count if, in the meantime, another worker adds a file to the directory. A solution would be to sequentialize the access to the file system. Sequential access to the file system is not desirable for performance reasons.

Consumer-Producer-Pattern for File Deletions

Since storing the number of files a directory and all its children have is not feasible, a parallel algorithm to delete files must be able to handle the unknown number of child files a directory might contain. A solution to the problem of not knowing how many hard link count decrementations a file deletion entails is the producer-consumer pattern with an unbounded shared buffer.

A task in this buffer represents a hard link counter decrementation of a regular file or the deletion of a directory, as a directory, unlike a regular file, can only ever have one file name mapped to it. The deletion of a directory spawns new tasks, as the content of the directory is added to the shared buffer.

4.1.3 Characteristics of Consumer-Producer-Pattern for File Deletions

In this section, we explain how tasks are processed, what a worker in the context of file deletions is, and how we implement the shared buffer of the producer-consumer pattern.

MIMD and SIMD for Consumer-Producer-Pattern

The character of the producer-consumer pattern naturally lends itself to a MIMD approach, as multiple workers (processors) process different tasks (data). Still, GPUs can also work in a SIMD fashion, which is why we also considered processing the content of the shared buffer in a such a way. The following paragraph describes how tasks might be processed in a SIMD fashion.

Suppose we have a non-empty buffer of the producer-consumer-pattern. The first step would be to divide all tasks equally among all workers. The second step would be to process the tasks. Newly spawned tasks are stored in a private buffer of each worker. Once all workers have added all newly spawned tasks to their private buffers, they are transferred from the private buffers to the shared buffer in parallel. The processing would then begin anew.

The main problem with this approach would be that the time required to process a task fluctuates because directories can have varying amounts of children. The synchronization between the different steps of the algorithm means that workers would idle until every worker would be done processing. We therefore choose a MIMD approach to the producer-consumer pattern of file deletions.

Definition of a Worker

A worker can be implemented on the GPU by grouping CUDA Threads together. We considered two possible group sizes: 32 CUDA Threads, which constitutes a single Warp, or up to 1024 CUDA Threads, which form an entire Thread Block. More CUDA Threads per worker mean more parallel processing power per task, but this comes at the cost of fewer workers. We selected a group size of 32 CUDA Threads, as the amount of parallelism in tasks is low.

In the future, the group size may change when encryption is added to the feature set of the current version of GPU4FS. Part of the reason we choose a group size of 32 CUDA Threads is to evaluate if there is a significant performance gain when executing tasks using Warps and not Thread Blocks. Should there indeed be a significant performance gain, then it might be better to implement encryption on a Warp level rather than a Thread Block level.

4.1.4 Characteristics of the Shared Buffer

In this section, we explain what properties are desirable for the shared buffer of the producer-consumer pattern.

Quick Task Insertion

The nature of file systems as mostly tree-like structures leads to a specific access pattern to the shared buffer of the producer-consumer pattern: Workers will only ever retrieve one task at a time from the shared buffer, as a worker can only effectively process one task at once. However, a worker might add several new tasks to the buffer after processing a single task, leading to a sharp rise in the fill level of the shared buffer. Therefore, the shared buffer should be optimized for quickly adding new tasks.

Unbounded Array

During the execution of the file deletion algorithm, the buffer might run out of space, as we do not know how many tasks have to be processed. The algorithm would then have to allocate additional memory, either in VRAM or on Intel Optane. Going forth, we assume that the shared buffer has enough memory. A single entry in the shared buffer only occupies 128 bits of memory. GPUs usually have several GBs of VRAM. The GPU we use, an RTX 4070, has 12 GBs of VRAM. That means that we can store about 93 million entries. We will not exceed that number during our tests in the evaluation. Therefore, we consider how to handle a buffer overflow of the shared buffer in VRAM to be future work.

Low Amount of Lock Contention

As the shared buffer is used by all workers simultaneously, hence the name *shared* buffer, access to individual tasks of the buffer must be locked. To avoid time-consuming lock contention, the shared buffer should implement some kind of policy.

4.1.5 Shared Buffer as Collection of Arrays

We choose to implement the shared buffer as an array in VRAM and several smaller buffers in shared memory.

Quick Task Insertion

To facilitate the quick insertion of new tasks, we complement the shared buffer in VRAM with shared buffers located in CUDA's shared memory, as accesses to shared memory are faster than accesses to global memory. Unlike, the buffer in VRAM, we assume that the buffers in shared memory can overflow, as they are much smaller. Another issue is that tasks might be unevenly distributed among the shared buffers, which is why we need to implement load balancing.

We cover a range of load-balancing algorithms in chapter 3.4. Due to a lack of time, we only implement a rudimentary load-balancing algorithm, that, once there are a certain amount of tasks in a shared buffer in shared memory, transfers all new tasks to the shared buffer in VRAM. If a worker cannot acquire a task in shared memory, the worker will try to transfer tasks from the shared buffer in VRAM to the buffer in shared memory. The exact number of tasks will be described in chapter 5.3, as they are experimentally determined.

Low Amount of Lock Contention

To reduce the amount of lock contention in a shared buffer, we try to keep workers apart from one another. Our first idea was to randomly determine the start position of a worker with equal distribution in an interval of indices which are guaranteed to contain all tasks and some free space. The complement of the interval would only contain free space. The interval's boundaries would be updated after each insertion. We partially abandoned the idea as randomness on a GPU comes with a relatively large overhead for the developer, and therefore a lot of time fixing bugs, compared to CPU randomness.

Instead, we modified the approach by not choosing starting positions at random, but simply dividing the interval into equal pieces that serve as starting posi-

tions. A worker wishing to acquire a task calculates a start index with

$$\frac{warp_id * (upper - lower)}{num_workers}$$

A worker wishing to acquire a free entry in the buffer calculates an index using

$$\frac{warp_id * free_space_size}{num_workers}$$

That index is then mapped to the interval(s) of free space, which may be discontinuous, depending on whether a boundary of the marked interval is the same as a boundary of the shared buffer.

Maintaining the invariant of the interval comes at the cost of sequential access to the lower and upper bounds. In chapter 5.4, we will show runtime measurements which indicate that the use of boundaries can improve the runtime.

An alternative to using an interval would be to use a histogram, which would represent the task distribution inside the shared buffer more accurately and could guide the search in it. The intervals would have to be large enough to avoid lock contention. The starting positions inside an interval would be equally distributed for all workers. We did not implement this approach due to a lack of time. Future work might reveal if a histogram performs better than the solution we implemented.

4.1.6 Alternative Data Structures

The design as presented in the previous section does not enforce a specific search pattern in the file system structure. This reduces access time to the buffer. Still, a depth-first-search (DFS) for example might reduce the maximum number of tasks in the shared buffer due to the mostly tree-like structure of file systems. A DFS needs a last-in-first-out (LIFO) queue. One might also implement a breadth-first-search using a first-in-first-out queue. However, doing so would likely increase the number of tasks in the shared buffer, which would only be advantageous if workers are task starved.

As limited memory space is no concern to us, we do not implement a LIFO queue.

4.1.7 Summary of File Deletions and File Moves

We start by analyzing how file deletions and file moves can be parallelized. We come to the conclusion that file moves can only be parallelized by performing multiple moves in parallel. A file deletion can be partially parallelized if the file is

a directory with at least two entries, as the children of a directory are independent of each other. File deletions must be able to handle an unknown number of files, as we do not know how many files are stored in a directory. We achieve this by using a producer-consumer pattern. We implement a worker in that pattern as a Warp of the GPU. Every worker should be able to quickly add new tasks to the buffer, which we bring about by complementing the buffer in the comparatively slow global memory with smaller buffers in fast shared memory. There should also be a low amount of lock contention, which we try to realize by using a policy for the calculation of start indices in the shared buffer, that tries to keep workers apart. We reduce the impact of that policy on the search times in the shared buffer by maintaining an interval of indices that highlight areas of interest.

4.2 Repairing the H-Tree

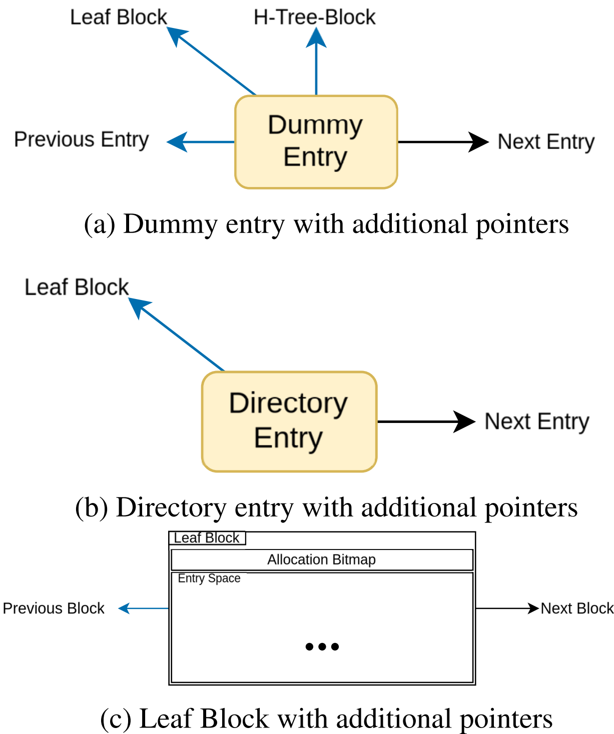
Chapter 4.1.1 describes how to unlink a file from a directory with an H-Tree.

In this chapter, we present two feasible approaches to repairing the H-Tree. The first approach allows us to quickly repair the H-Tree after an unlinking operation, but this comes at the cost of additional pointers that have to be added to the H-Tree. That is why we choose the second option, which has no need of additional pointers. Instead, the H-Tree is reconstructed. The cost of reconstructing is amortized over successive unlinking operations.

4.2.1 Repairing from Leafs to Root

To repair the H-Tree, an algorithm first has to update the allocation table of the leaf block that contained the directory entry. Then, the algorithm has to potentially remove the dummy entry of the hash region and the H-Tree-Block entry from the corresponding H-Tree-Block if the hash region has become empty. There is also the possibility that the leaf block no longer contains any directory entries or dummy entries, making it advantageous to give it to the garbage collector.

The H-Tree needs additional pointers to implement the algorithm of the previous paragraph, that allow it to move through the H-Tree from leafs to the root. The additional pointers can be seen in figures 4.1b and 4.1c. Every dummy entry is given a pointer to its previous directory entry, the leaf block that contains it and the root block that points to it. Every directory entry is given a pointer to the leaf block that contains it. All leaf blocks receive a pointer to the previous leaf block.



(a) Dummy entry with additional pointers

(b) Directory entry with additional pointers

(c) Leaf Block with additional pointers

Figure 4.1: Data structures that have to be modified to allow repairing the H-Tree from leafs to the root. Blue arrows indicate pointers we add. Black arrows indicate pointers that are present in the unmodified version by Kittner [20].

Future Considerations

If the maximum number of levels increases in the future, then every H-Tree-Block will also need a pointer that points to the H-Tree-Block that contains a pointer to it, except for the root block. With only three levels, every H-Tree-Block that is not the root block must automatically be an H-Tree-Block of the first level.

Algorithm to Repair H-Tree

An algorithm to repair the H-Tree therefore looks as follows:

Let d be the directory entry for the file we want to unlink. After unlinking, d is no longer part of the linked list of directory entries. Follow the newly added pointer of d to its leaf block and update the bitmap of the leaf block. Then check if the hash region of d is now empty. If so, then also remove the dummy entry for the hash region from the linked list of directory entries. Now check if the leaf blocks that contained d and if applicable the dummy entry are now empty,

again using newly added pointers. Should that be the case, then remove the leaf blocks from the linked list of leaf blocks. If a dummy entry was removed, then also remove the H-Tree-Block entry that contained it. Again, dummy entries have a newly added pointer to access the H-Tree-Block that points to them. The H-Tree-Block should then be merged with another H-Tree-Block of the same level, if it has sufficiently few elements. Blocks of the same level can be accessed by accessing the H-Tree-Block entries of the "parent" H-Tree-Block that points to the H-Tree-Block. Similarly, the depth of the H-Tree can be reduced if there are sufficiently few H-Tree-Block entries.

4.2.2 Repairing by Reconstructing

The previous algorithm repairs the H-Tree after each unlinking operation. This comes at the cost of additional pointers. An alternative is to reconstruct the H-Tree. Reconstructing the H-Tree would be inefficient if it was done after every unlinking operation. That is why we perform reconstruction only after a certain amount of unlinking operations to amortize the costs.

Storing the Number of Unlinking Operations

This requires that we store the number of unlinking operations that have been performed on a directory somewhere in the inode or the H-Tree. As seen in figure 3.2 of chapter 3.1, the inode currently has two bytes that are unused. $2^{2 \cdot 8} = 65536$ should be large enough to store the number of unlinking operations. Figure 4.2 shows the modified inode.

Producer-Consumer Pattern for reading Directory Entries

To reconstruct an H-Tree, all directory entries of the linked list of directory entries must be read and added to a new H-Tree. Reading entries in parallel from a linked list is not possible, as knowledge of the location of an entry is dependent on knowledge about the location of the previous entry. We make use of the H-Tree to speed up the process of reading the directory entries.

We again use the producer-consumer pattern to process the tree-like structure of the H-Tree. A task in the shared buffer is an H-Tree-Block entry which either points to an H-Tree-Block or a dummy entry. If an H-Tree-Block entry points to an H-Tree-Block, then all H-Tree-Block entries of that H-Tree-Block are added as new tasks. If the H-Tree-Block entry instead points to a dummy entry, then all directory entries of the corresponding hash region are added to the new H-Tree.

We also modify the concept of a worker for this buffer to be a Thread Block with at 256 CUDA Threads. This is because H-Tree-Blocks contain at most 255

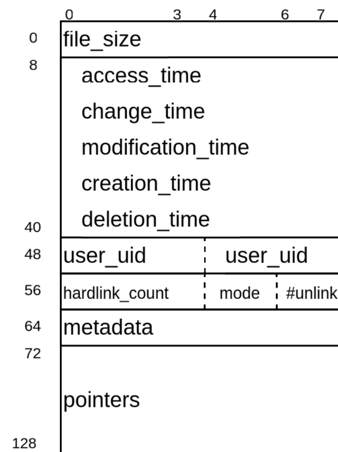


Figure 4.2: Modified inode - Two bytes after mode are now used to store the number of unlink operations that have been performed on a directory without rebuilding the H-Tree

H-Tree-Block entries, and 256 is a multiple of 32. These entries are stored in an array, making it possible to access them in parallel. Adding elements to the shared buffer is still done by individual Warps. This is due to the fact that we only have an upper bound for the number of H-Tree-Block entries in an H-Tree-Block. So there might be less than 256 H-Tree-Block entries in an H-Tree-Block. The unknown number of H-Tree-Block entries in an H-Tree-Block leads to a situation in which we either add elements to the shared buffer only at intervals of 255 elements, or every Warp search for an empty space for itself to not waste memory space.

Due to a lack of time, we do not complement the shared buffer in VRAM with shared buffers in shared memory, and instead just use the shared buffer in VRAM.

4.2.3 Summary

To conclude, we choose to reconstruct H-Trees after a certain amount of unlinking operations rather than adding new pointers to the H-Tree to repair the data structure after each unlinking operation. The cost of reconstructing the H-Tree is amortized over successive unlinking operations. We read the directory entries of a directory by traversing the tree-like structure of the H-Tree using the producer-consumer pattern. A worker in this instance of the producer-consumer pattern is a Thread Block not a Warp, though Warps are still used to insert elements into the shared buffer.

Chapter 5

Implementation

In this chapter, we present details of our implementation that can have a large impact on the evaluation, but are too specific to be included in chapter 4. Section 5.1 describes how we implement locking for H-Trees and elements of the shared buffers of the producer-consumer patterns. Section 5.2 contains the command descriptors for the file deletions, moves, and rebuild H-Tree commands. In section 5.3, we present our choice of parameters for the load balancing mechanism for file deletions. We conclude this chapter in section 5.4 by showcasing how we optimize the start index calculation by allowing the boundaries to be cached.

5.1 Locking

We use locking during unlinking to manage access to the H-Tree and in the file deletions algorithm to manage access to elements of the shared buffers.

5.1.1 Locking for Unlinking

During unlinking, the algorithm modifies the linked list of directory entries and possibly the H-Tree of the parent directory. To maintain the invariants of the linked list of directory entries and the H-Tree, access to the directory is locked. Kittner [20] implements locking of directories by locking the first block of the directory if the directory does not use an H-Tree, or by locking the root block of the H-Tree, if the directory has one. Determining if a directory has an H-Tree can only be done reliably after locking the first block of the directory. Should it turn out that the directory uses an H-Tree, then the first block is unlocked, and the algorithm tries to lock the root block. A modified version of Kittners directories is part of the GPU4FS we use, which locks the first dummy entry instead of the first block. The H-Tree is still locked using the root.

Locking of the entire parent directory can have a negative impact on the performance if many files in a shared parent directory are unlinked in parallel, as access to the parent directory is sequentialized. Due to a lack of time, we do not improve the locking mechanism to reduce the impact on performance. However, in our evaluation we will present a test scenario in which each file has its own parent directory, thereby reducing the impact of locking on runtime and showing how fast file moves could be with improved locking.

5.1.2 Locking Elements of Shared Buffers

Structure of a Lock

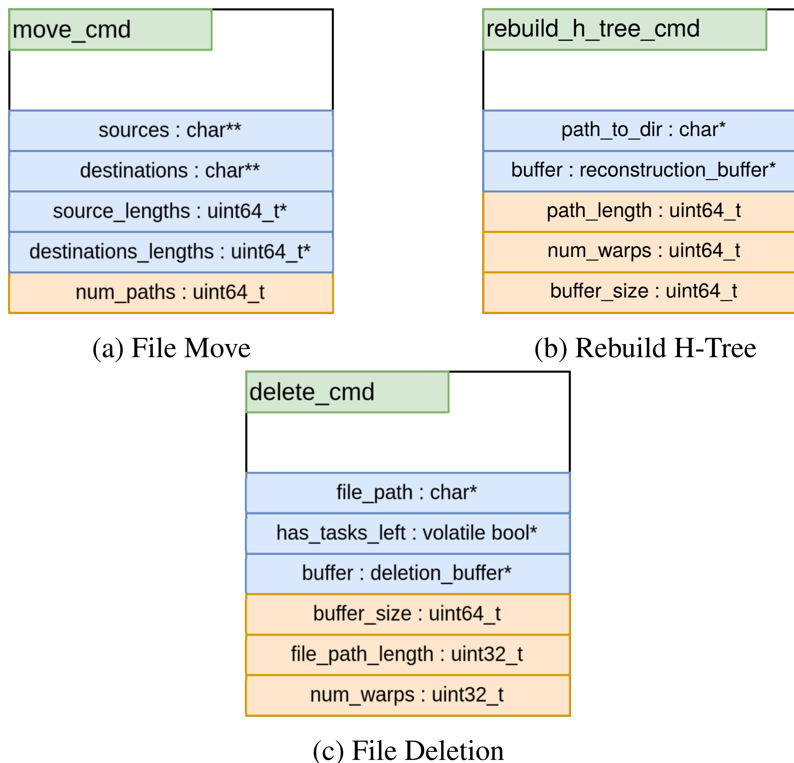
We implement locks of the shared buffers as mutexes rather than read-write locks because every worker intends to change the state of an entry, either to add a task to the entry or process the task contained in the entry. We implement the mutexes using an atomic compare-and-switch operating on a 64 bit variable. The lower 32 bits are either 0, unlocked, or 1, locked. We initially considered using a read-write lock, which is why we only use 32 of the 64 bits, but quickly abandoned the idea because of the stated reason.

Locks can be used by both Warps and Thread Blocks

Our locks are designed to be accessible by both Warps and Thread Blocks. This is necessary because the file deletions and the rebuilding commands for H-Trees use Warps and Thread Blocks as workers, respectively. The atomic compare-and-switch operation is executed by the first CUDA Thread of the Warp or Thread Block. The result of that operation is then shared with the rest of the CUDA Threads. In case of a Thread Block, this is done by storing the result in shared memory. Warps on the other hand use the Warp function `shfl_sync()` to share the result. Thread Blocks synchronize after sharing the result using `__syncthreads()`. Warps automatically synchronize using `shfl_sync()`.

5.2 Command Descriptors

Figure 5.1 shows the command descriptors for deleting and moving files, and rebuilding H-Trees. Variables which are declared as `volatile` are not cached, as preventing caching is important for synchronization. The move command receives two arrays of char pointers, which point to the source and destination paths for the files to be moved, and two pointers to two arrays containing the lengths of those paths. Finally, the move command is given the number of files to move. The delete command receives a pointer to the path that leads to the file that is to



(a) File Move

(b) Rebuild H-Tree

(c) File Deletion

Figure 5.1: Command Descriptors - Pointers and uint64_t are of size 64 bits. uint32_t is made up of 32 bits. The move command receives the source and destination paths, their lengths, and the number of paths. The delete command receives a path to the file to be deleted, its length, a pointer to the shared buffer in global memory, values to initialize the buffer, and a pointer to a boolean array used to communicate if a worker has tasks. The rebuild command receives a path to the directory it is supposed to repair, its length, a pointer to the shared buffer, and values to initialize it.

be deleted, the length of the path, the size of the buffer and the number of Warps, which it uses to initialize the shared buffer in global memory. Additionally, the command also has a pointer to an array, whose entries indicate if the individual workers still have tasks or not. If all entries in that array are false, then the command finishes. The rebuild H-Tree command is given a path to the directory and its length that the command it is supposed to repair. Like with the deletion command, the rebuild command also receives the number of Warps and the buffer size to initialize the shared buffer.

5.3 Load Balancing

In chapter 3.4, we explain that we would like to use dynamic load balancing but choose static load balancing instead due to a lack of time. We implement static load balancing by storing tasks in the buffer in global memory once the buffer in shared memory is half full, which happens at 500 elements. If a worker notices, that the buffer in shared memory is empty, then the worker tries to transfer up to 50 tasks from the buffer in global memory to the buffer in shared memory. We determine these values experimentally with the goal of minimizing the runtime for the test scenarios “Mix” and “Wide”, which contain 1000 and 10000 tasks respectively, in the evaluation in chapter 6. These values should not be assumed to be universally valid for every possible scenario, which is why in future work, the static load balancing should be replaced by dynamic load balancing.

Transferring a task from one buffer to the other is done by locking the task and searching for an empty entry in the destination buffer. Once found, the worker locks the empty entry and inserts the task. After that, the worker first releases the lock in the destination buffer and then the lock in the origin buffer.

5.4 Start Index for Search in Shared Buffer

In chapter 4.1.5, we explain, how workers calculate start indices for their searches in shared buffers by using an interval that marks the region in a shared buffer that contains all tasks. That approach seems to be about 7 ms faster than hard-coding the start index to be 0 in a test scenario in which a directory containing 100 directories containing 100 files each is deleted. We will later call that scenario “Mix”. The runtime value is the average of ten repetitions. Caching the boundaries again improves the runtime by 3 ms, although caching weakens the invariant of the interval, that it contains all tasks in the shared buffer. This is likely because the algorithm saves more time by performing less atomic operations, even though it has to search longer.

5.5 Summary

To summarize, we use locking to manage access to directories and access to elements of the shared buffers of the producer-consumer patterns of the file deletions and rebuilding H-Tree commands. Parallel access to a directory is sequentialized through locking, which can have a negative impact on performance. We do not improve the locking due to a lack of time and consider improvement to directory locking to be future work. The locks of the elements of the shared buffers are

mutexes, as every worker seeks to change the state of an element in the buffer by either removing or adding a task. These locks can be used by both Warps and Thread Blocks.

The commands receive data, which is partly directly in their algorithms and partly to initialize helper data structures.

We experimentally determine that transferring tasks from the buffer in shared memory to the buffer in global memory for deleting a directory with 100 directories containing 100 files should be done after the buffer in shared memory contains 500 tasks. Similarly, we determine the number of tasks that should be transferred by a single worker from the global buffer to the buffer in shared memory is 50. These values are fine-tuned for our evaluation, which is why this static load balancing should in the future be replaced by dynamic load balancing.

Finally, we show that using an interval, as described in chapter 4.1.5, improves the runtime of file deletions by about 7 ms. The runtime value can be further improved by 3 ms if caching of the boundaries is enabled.

Chapter 6

Evaluation

In this chapter, we evaluate our implementation in terms of speedup compared to the CPU file system ext4. Section 6.1 presents the configuration of our testing machine and how we measure the runtime of our implementation. In sections 6.2, 6.3, and 6.4 we evaluate the rebuilding H-Tree, file deletions, and file moves commands, respectively.

6.1 Configuration and Methodology

In this section, we look at the specifications of our testing machine and how we measure on it.

6.1.1 Testbed Configuration

Our machine uses an Intel Xeon Silver 4215 CPU @ 2,5 GHz [18] with two sockets, 8 cores per socket and two threads per core together with 128 GB of DDR4 DRAM in total, one 128 GB Intel Optane DC PMM and an NVIDIA RTX 4070. An NVIDIA RTX 4070 has 46 Streaming Multiprocessors (SM), a base clock of 1.92 GHz, a boost clock of 2.48 GHz, and 12 GBs of VRAM [10]. We use Fedora Linux 40 Server Edition as our operating system and Linux 6.10.7-200.fc40.x86_64 as the kernel.

6.1.2 Measurement Tools

CUDA offers cudaEvents, `clock()`, and `clock64()` functions to measure how much time passes during an execution of a Grid.

CUDA Events

CUDA Events can be inserted before and after a Grid launch by the CPU into the CUDA stream that contains the Grid. The host function `cudaEventElapsedTime()` then calculates how much time in milliseconds passed between the two events. This function is useful for measuring how much time one or more Grids need to execute.

Clock Cycle Count

A different method of measurement are the device functions `clock()` and `clock64()`, which return the clock cycle count of the Streaming Multiprocessor that executes them. They differ merely in which format that data is returned. Therefore, we will only use `clock64()`. The difference between two samples of `clock64()` is an indicator of how many cycles passed in the time between [8]. A drawback of measuring with `clock64()` is that clock cycle counts cannot reliably be converted to milliseconds because the clock speed of the GPU changes dynamically during execution. However, `clock64()` can be used to get an estimate of how long individual components of a Grid take to execute compared to all other components, as long as the components contain enough instructions. We came to this conclusion early on during our evaluation as we tried to measure how much time a Grid spends on accessing Intel Optane. We inserted `clock64()` instructions directly before and after memory instructions that accessed Intel Optane. The measured differences were erratic and nonsensical, as they were sometimes larger than the overall clock cycle count. We believe, this is caused by instruction reordering. Therefore, taking the difference between two `clock64()` instructions is only useful when there are enough instructions in between the two `clock64()`s to mitigate the effects of instruction reordering and hidden memory latency.

We further examined the problem in a dedicated GPU program which was executed on an NVIDIA GTX 1070 [9] to determine how many instructions have to be in between two `clock64()`s for their sample to be usable. In that program, each CUDA Thread calculates a certain number of iterations of the Fibonacci series, and writes the value of each iteration into global memory. The code can be seen in listing 1. We compiled the program using NVCC with optimization disabled and launched the Grid using a single Warp. On the one hand, the grid uses `clock64()` instructions to store the aggregated clock cycle count differences of x iterations each. On the other hand, the grid measures how many clock cycles it needs in total to execute. Figure 6.1 shows how the difference between these two values shrinks with an increasing number of iterations between two measurements. The two graphs converge after about 200 to 300 iterations of the Fibonacci

series. The graph for the clock cycle difference between 10000 iterations is not flat. This is because the smaller the number of iterations between measurements, the more frequently measurements are taken, leading to more time spent on measuring. Even so, the difference in precision between the two measurements is clearly visible. We repeated the experiment, but this time stored the results of the Fibonacci series in a local variable. The shape of the graphs remained the same. Only the convergence clock cycle count changed from $1.39403 \cdot 10^8$ to $1.37181 \cdot 10^8$ which indicates that the GPU amortizes the costs of accessing global memory. Furthermore, it means that 200 to 300 instructions seem to be enough to effectively use `clock64()`. We recommend those values for anyone who wants to measure using `clock64()`.

We use cudaEvents for the measurements in the following sections, as we are mainly interested in how long the commands take compared to their CPU counterparts.

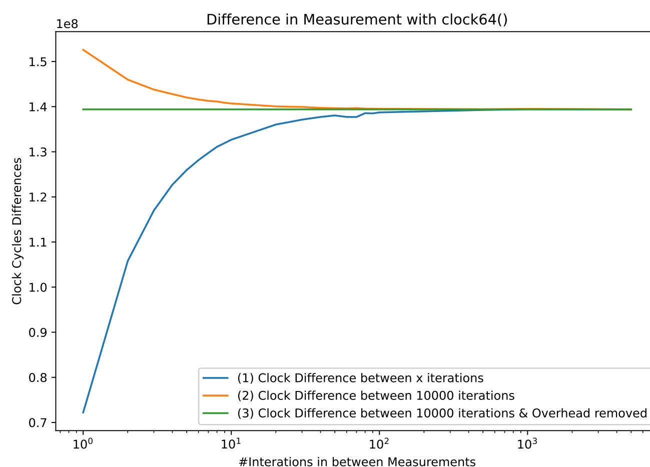


Figure 6.1: Measuring the time to calculate the Fibonacci series with 10000 elements. Line 1 is the result of aggregating the clock cycle differences after x iterations of the Fibonacci series over 10000 iterations of the Fibonacci series. Line 2 is the clock cycle differences after 10000 iterations. The curvature of the graph is the result of the overhead necessary to storing the clock cycle differences. The overhead is larger the more often the measurements are taken. Line 3 represents the number of clock cycles necessary to calculating 10000 iterations of the Fibonacci series with the overhead removed.

```
__global__ void test_kernel(
    uint64_t num_iterations, volatile uint64_t* data,
    uint64_t* overall_time, uint64_t* fine_measured_time,
    uint64_t interval) {

    auto thread_id = blockDim.x * blockIdx.x + threadIdx.x;
    uint64_t c0 = 0;
    uint64_t c1 = 1;
    uint64_t r;

    auto overall_clock = get_clock();
    uint64_t fine_clock = 0;
    uint64_t temp;
    for (auto i = 0; i < num_iterations; i++) {
        if (i % interval == 0) temp = get_clock();
        r = (c0 + c1);
        c0 = c1;
        c1 = r;
        data[i] = r;
        if ((i % interval) == (interval - 1)) {
            fine_clock += get_clock() - temp;
        }
    }
    __syncthreads();
    overall_time[thread_id] = get_clock() - overall_clock;
    fine_measured_time[thread_id] = fine_clock;
}
```

Listing 1: Code to measure the differences between clock64()s between x iterations of the Fibonacci series and 10000 iterations

6.2 Measurements for Rebuilding

In chapter 4, we start with file deletions and file moves, before we explain our design for rebuilding H-Trees. In this chapter, we evaluate the commands in the reverse order, as the results of the evaluation of the rebuilding H-Tree command are a deciding factor for our conclusions about file deletions and file moves. Unfortunately, we are not able to properly evaluate our implementation of the rebuilding command, as we encounter an illegal memory access, when more than one worker tries to modify an H-Tree. The algorithm seems to work with just one worker. We believe the illegal memory access is caused by a race condition, as it does not occur, when we use `printf()` statements to print the values of variables during file lookups. We only describe how we think the rebuilding command would perform and how that might differ from how it actually performs, as we lack the time

to measure the rebuilding command with just one worker.

6.2.1 Testing Scenarios

With the testing scenarios in the following sections, we aim to determine optimal configurations of the number of Thread Blocks and Warps per Thread Block and to measure the runtimes of the commands to compare them to their CPU counterparts and decide whether to use their CPU or GPU versions. The scenarios are defined by the number of file originally present in the directory and the number of file unlinked from it. The former is identified by an "F" in the scenario name, the latter by a "U". For example, in scenario F100U10, we unlink 10 files from a directory of 100. Our scenarios are F100U10, F100U50, F100U100, F1000U100, F1000U500, and F1000U1000.

For scenarios F100U10, F100U50, and F100U100, we expect to see the runtime linearly decreasing with the number of files unlinked, as there should only be one hash region which can only be linearly searched. We expect that the runtime decreases exponentially and not linearly for scenarios F1000U100, F1000U500, and F1000U1000, as a directory with 1000 files uses an H-Tree and not just a linked list.

Regarding the number of Thread Blocks and Warps per Thread Block, we expect the runtime to decrease with an increasing number of Warps in scenarios F1000U100, F1000U500, and F1000U1000 until the number of Warps reach 8. 8 Warps consist of 256 CUDA Threads in total, which is one CUDA Thread more than the maximum number of H-Tree block entries in an H-Tree block and is therefore the maximum number of tasks that can be processed in parallel.

There should be no speedup with an increasing number of Thread Blocks and Warps per Thread Block in scenarios F100U10, F100U50, and F100U100, as all entries are stored in a linked list which can only be read linearly.

6.2.2 Considerations regarding File Deletions and File Moves

The performance behavior of the rebuilding command plays a critical role when deciding whether file deletions and file moves are best done using the CPU or GPU. We consider the GPU implementation to be superior if the amortized costs for the GPU operation are smaller than for the CPU operation.

Therefore, the cost for an individual file deletion or file move must be smaller than their CPU counterpart. As we will see in the following sections, that can be true for file deletions, but is never true for our implementation of file moves.

6.3 Evaluation of File Deletion

In this section, we evaluate our implementation of file deletions. We begin by introducing the different testing scenarios we use. We then present the measurements and draw conclusions from them.

6.3.1 Testing Scenarios

In scenario “Deep” we nest 1000 directories inside each other, effectively creating a chain of child directories. In scenario “Wide” we store 1000 directories in a single parent directory. Scenario “Mix” is a combination of scenarios “Deep” and “Wide”, as 100 directories are nested in 100 parent directories each. Scenario “Single File” is a base case scenario, where we only delete a single file.

6.3.2 Measurements

We execute file deletion with up to 46 Thread Blocks and 19 Warps. An RTX 4070 only has 46 Streaming Multiprocessors. Therefore, we can only use 46 Thread Blocks in parallel. We only use up to 19 Warps, because any more Warps cause a compilation error. We suspect that the compiler runs out of registers it can use.

Deep

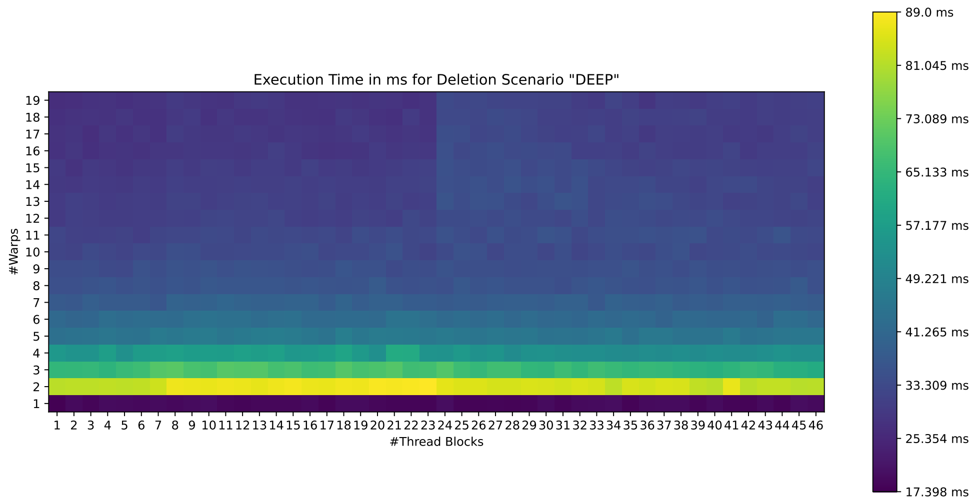


Figure 6.2: Runtime of the file deletion algorithm for scenario “Deep” - The minimal runtime of 17.39772 ms is achieved with 1 Thread Blocks and 1 Warp. The runtime decreases with an increasing number of Warps per Thread Block starting at 2 Warps per Thread Block. There is also an increase in the runtime from 23 to 24 Thread Blocks, and an increase by a factor of 5 from 1 Warp per Thread Block to 2 Warps.

Figure 6.2 shows the runtime measurements for the scenario “Deep”. The file deletion algorithm takes the least amount of time, 17.39772 ms, when executed with 1 Thread Block and 1 Warp per Thread Block compared to 21.873 ms in ext4 on the CPU. This is a speedup of about 1.26. Launching the Grid with 2 Warps per Thread Block increases the runtime by a factor of up to 5 when compared to 1 Warp per Thread Block. There is also an increase in the runtime from 23 Thread Blocks to 24 Thread Blocks. The runtime of the file deletion algorithm is primarily influenced by the number of Warps per Thread Blocks and not the number of Thread Blocks: The maximum difference between runtimes with the same number of Warps per Thread Block is 10.045 ms. In contrast, the maximum difference between runtimes with the same number of Thread Blocks is 70.788 ms.

Wide

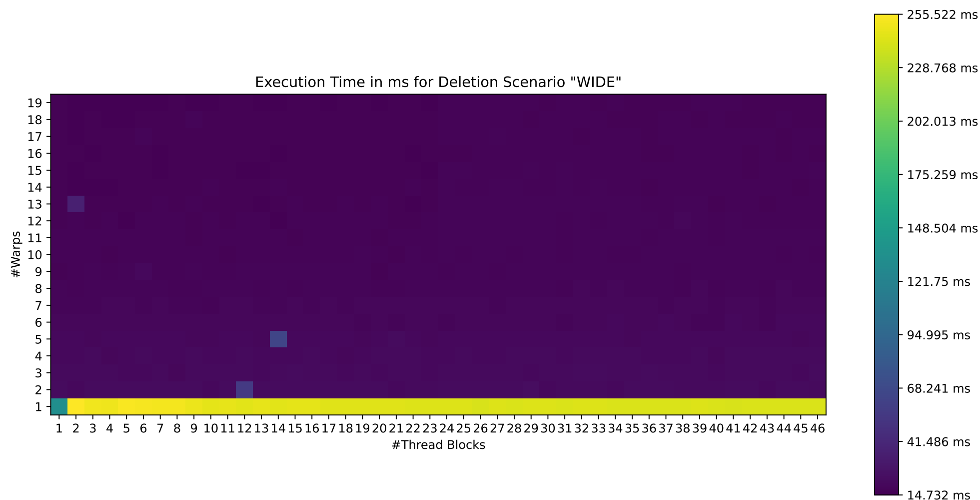


Figure 6.3: Runtime of file deletion algorithm for scenario “Wide” - The minimum of 14.73195 ms is at 17 Thread Blocks and 19 Warps. The runtimes at 1 Warp per Thread Block are larger by a factor of 2.7 to 18 than the runtimes at 2 Warps per Thread Block. The runtimes at 12 Thread Blocks and 2 Warps, 14 Thread Blocks and 5 Warps, and 2 Thread Blocks and 13 Warps are larger by a factor of 3 to 8.6.

Figures 6.3 and 6.4 show the runtime measurements for the scenario “Wide”. The minimum of 14.73195 ms is achieved at 17 Thread Blocks and 19 Warps. This is about as fast as ext4, with a runtime of 15.104 ms. Launching the Grid with only one Warp per Thread Block causes an up to 17 times higher runtime compared to the minimum. Figure 6.4 also shows that the runtime increases from 23 Thread Blocks to 24 Thread Blocks, like in scenario “Deep”. The runtimes seem to be influenced more by the number of Warps per Thread Block rather than the number of Thread Blocks, as the maximum difference in runtime between differing numbers of Warps is 240.355 ms, whereas for Thread Blocks that number is 122.419 ms.

Figure 6.4: Runtime of file deletion algorithm for scenario "Wide" starting at 2 Warps per Thread Block clipped at 30ms - The runtime decreases with an increasing number of Warps per Thread Block. Like in scenario "Deep", the runtime increases from 23 Thread Blocks to 24 Thread Blocks.

Mix

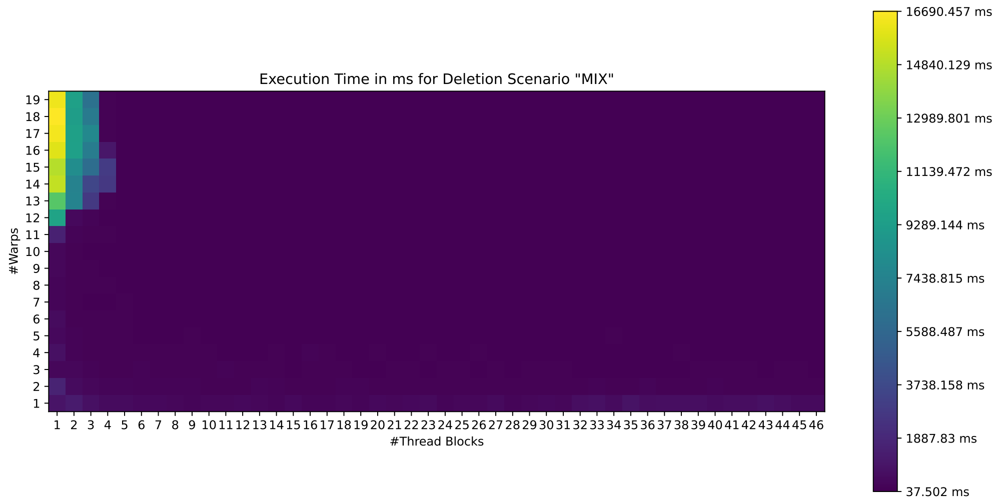


Figure 6.5: Runtime of file deletion algorithm for scenario "Mix" - The maximum of 16676.9 ms at one Thread Block and 18 Warps is 445 times greater than the minimum of 37.5015 at 45 Thread Blocks and 12 Warps.

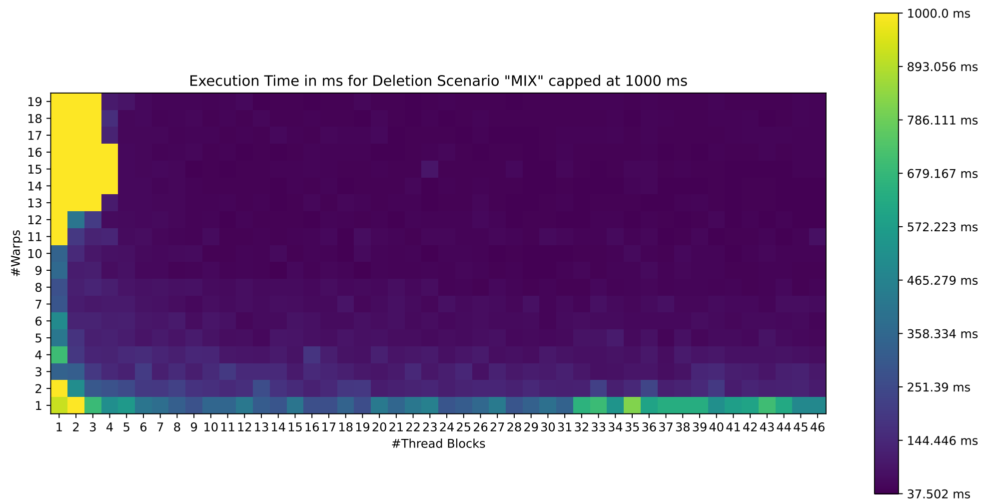


Figure 6.6: Runtime of file deletion algorithm for scenario “Mix” capped at 1000 ms - The minimum of 37.5015 ms is achieved at 45 Thread Blocks and 12 Warps. The runtime decreases with an increasing number of Thread Blocks and Warps. The runtimes for low numbers of Warps per Thread Block or Thread Blocks are up to 27 times greater than the minimum.

The runtime measurements for scenario “Mix” can be seen in figures 6.5 and 6.6. The minimum of 37.5015 ms can be found at 45 Thread Blocks and 12 Warps per Thread Block. This is about 4 times better than ext4. For comparison, Ext4 needs 149.413 ms. The runtime of our implementation of file deletion decreases with an increasing number of Thread Blocks and Warps. The runtime is about 445 times greater than the minimum when the scenario is executed with low numbers of Thread Blocks and high of Warps.

Single File

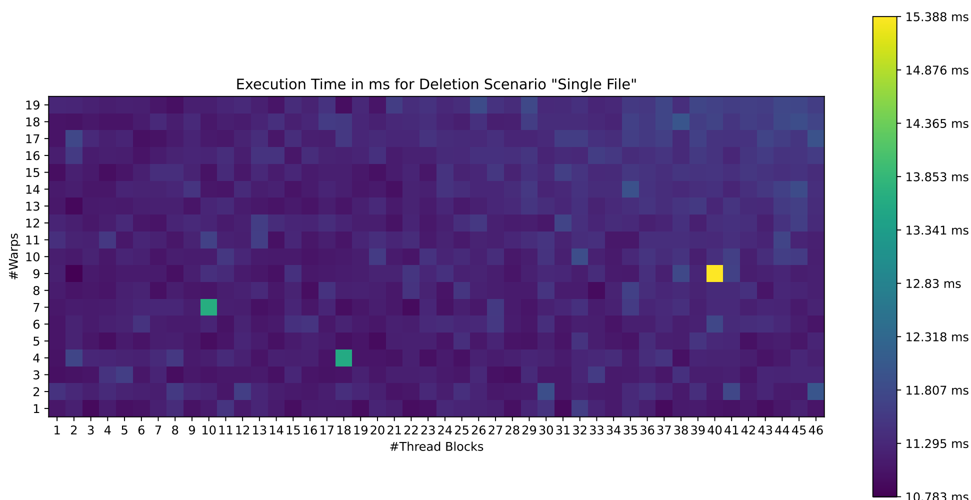


Figure 6.7: Runtime of File Deletion Algorithm for Scenario "Single File" - The minimum of 10.783224 ms is at 2 Thread Block and 9 Warps. The runtime increases with the number of Thread Blocks and Warps and is up to 1.4 times greater at 10 Thread Blocks and 7 Warps, 18 Thread Blocks and 4 Warps, and 40 Thread Blocks and 9 Warps than the minimum.

The runtime measurements for deleting a single file can be seen in figure 6.7. Deleting a single file takes the least amount of time, with 2 Thread Blocks and 9 Warps (10.783224 ms). This is about 513.5 times slower than ext4, with 0.021 ms. The runtime of our implementation increases with the number of Thread Blocks and Warps seems to be equally influenced by both. The runtime is up to 1.4 times higher in 10 Thread Blocks and 7 Warps, 18 Thread Blocks and 4 Warps, and 40 Thread Blocks and 9 Warps than the minimum.

6.3.3 Discussion

The central question of the evaluation is whether parallelizing file deletions and file moves using the GPU is sensible and faster than their CPU counterparts.

Scenario "Single File" shows that deleting few files should be done using the CPU and not the GPU, as the GPU implementation is 514 times slower than the CPU implementation and fastest when executed with few workers. The runtime grows with an increasing number of Thread Blocks and Warps, possibly because the workers require more and more time to realize that no worker has any tasks anymore. The runtimes in scenario "Single File" at 10 Thread Blocks and 7

Warps, 18 Thread Blocks and 4 Warps, and 40 Thread Blocks and 9 Warps are due to increased transfer times. We do not know why the transfer time increases, as the same amount of data is transferred as in every other configuration. We observe similar behavior in scenario “Wide” at 12 Thread Blocks and 2 Warps, 14 Thread Blocks and 5 Warps, and 2 Thread Blocks and 13 Warps. We suspect these increases to be random. Extrapolating from scenario “Single File” to scenario “Wide” supports the hypothesis that GPU file deletion can be faster than CPU file deletion, if there are many files to delete. That hypothesis is supported by the scenario “Mix” and, curiously, also by the scenario “Deep”. The results of scenario “Deep” are surprising to us, because, while there are many files to delete, the amount of parallelism inherent in that scenario is low. Nonetheless, the file deletion algorithm in scenario “Deep” is faster than its CPU counterpart. The algorithm is even then still faster if the Grid is launched with only a single worker, which takes 17.39772 ms to execute. We do not know why scenario “Deep” is faster on the GPU than on the CPU. One possible reason is that directories in that scenario are stored in neighboring addresses on Intel Optane in GPU4FS. That makes it possible to load several directories at once. If that is the case, then modifying the block allocator to spread blocks apart in Intel Optane would greatly increase the runtime of the file deletion algorithm in scenario “Deep”. ext4 possibly arranges directories differently on Intel Optane which would explain the speedup.

Another oddity is the increase of the runtime in scenarios “Deep” and “Wide” from 23 Thread Blocks to 24 Thread Blocks. We have no explanation for this. However, as 23 is exactly half the number of Streaming Multiprocessors an RTX 4070 is equipped with, we assume there to be a hardware related explanation.

Another observation in scenario “Deep” is the increase from one Warp per Thread Block to two Warps, which can be explained by the low amount of parallelism in that scenario and the increased amount of lock contention compared to one Warp per Thread Block. The runtime decreases again with an increasing number of Warps, likely because more workers trying to acquire tasks mitigate the effects of lock contention by just two Warps.

Finally, the maximum runtime of 16 seconds in scenario “Mix” may be explained by many disparate small memory accesses of a single Streaming Multiprocessor (SM) as the result of the directory structure of the scenario.

Choosing an Optimal Grid Configuration

As can be seen in figure 6.8, the optimal configurations of the number of Thread Blocks and number of Warps per Thread Block vary from scenario to scenario. Determining an optimal configuration for all scenarios is difficult, as determining one that suits all scenarios is an optimization task with many goal functions, one for each possible scenario. One possible solution is to define a new goal function

as the weighted sum of all goal functions, and then select the configuration that is lowest for that goal function. Given the data of the four scenarios and equally distributed weights, the optimal configuration would be 22 Thread Blocks and 19 Warps per Thread Block. With that configuration, the runtimes in the different scenarios are only up to 2.3 times worse than their respective minima. The runtimes for scenarios “Mix”, “Wide”, and “Deep” for this configuration are still lower than their CPU counterparts. Deleting a single file using the GPU is still about 514 times slower than its CPU counterpart.

	Single File	Mix	Wide	Deep
Optimal Number of Thread Block	2	45	17	1
Optimal Number of Warps per Thread Blocks	9	12	19	1
Runtime in ms	10.783224	37.5015	14.73195	17.39772

Figure 6.8: Optimal configurations of number of Thread Blocks and Warps per Thread Blocks for the different scenarios

Conclusion

The scenarios showcase that GPU file deletion is only then faster than CPU file deletion, when there are enough files to delete, even with the optimal configuration of 22 Thread Blocks and 19 Warps per Thread Block of the previous section. For that reason, we recommend that GPU file deletion does not replace CPU file deletion, but instead supplements it to be used when there are guarantees by the user that a directory contains enough files. For an RTX 4070, 1000 is a lower bound for the number of files.

6.4 Evaluation of File Moves

In this section, we evaluate our implementation of file moves. Again, we use different test scenarios.

6.4.1 Testing Scenaros

The scenarios differ in how many files are moved and whether these file share a common parent directory or have their own parent directories. The latter property should have a large impact on the runtime, as many unlinking and linking operations have to be performed on the same directory. Access to a directory is locked. Therefore, having a shared parent directory should increase the runtime of the algorithm compared to each file having its own unique parent directory. The numbers of files that are moved are 10, 50, 100, 500, and 1000 files.

6.4.2 Measurements

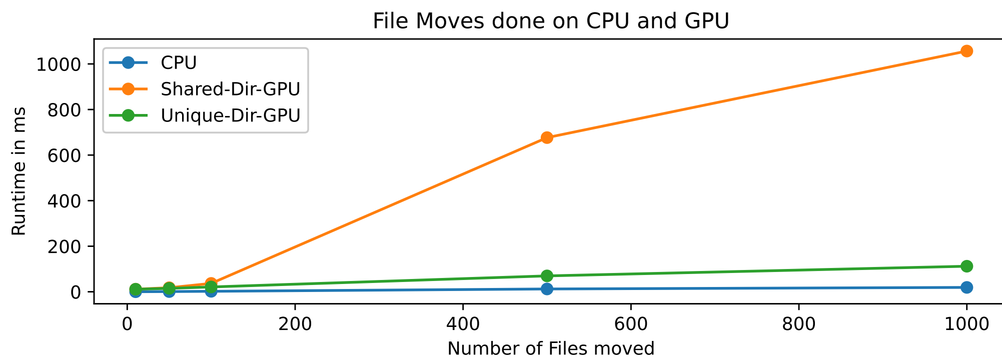
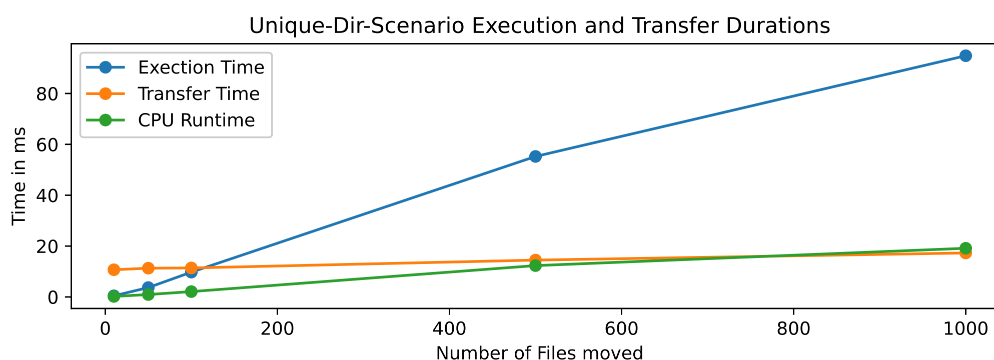
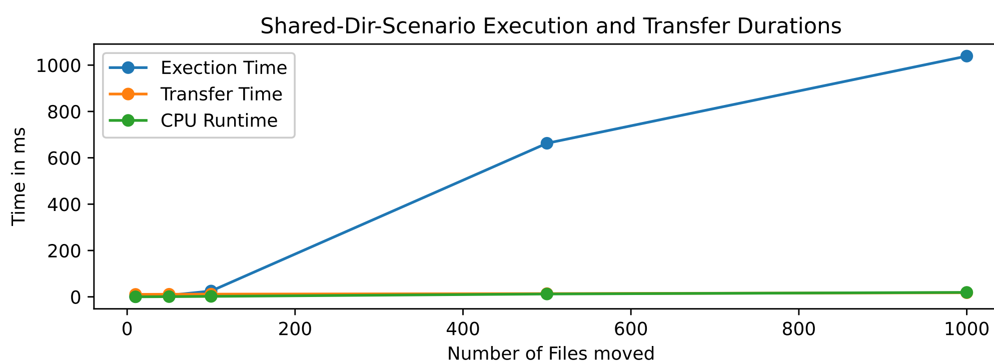


Figure 6.9: File Moves on CPU and GPU - The Shared-Dir-GPU graph represents measurements for directories that share a parent directory. In the Unique-Dir-GPU graph, every file has its own unique parent directory.

Figure 6.9 shows the runtime measurements for file moves. The three graphs are the result of measuring the runtimes of file moves on the CPU, on the GPU with a shared parent directory, and on the GPU with each file having its own parent directory. The figure shows that CPU file moves are consistently faster than their GPU counterparts. Also, file moves of files with a shared parent directories are slower than file moves with their own parent directories.



(a) Measurements for Unique-Dir Scenario



(b) Measurements for Shared-Dir Scenario

Figure 6.10: Execution and transfer time of file moves compared to the runtime of CPU.

While we expect the CPU implementation to be faster when moving only dozens of files, we are surprised that it is also the case with larger numbers of files. In fact, the margin in runtime between CPU and GPU implementation grows with an increasing number of files. One reason for that seems to be that an increasing number of file paths have to be copied from DRAM to VRAM. Figure 6.10 shows the time it takes to allocate memory for the file paths and to transfer the file paths. The figure show, the time to allocate memory and transfer the file paths grows with an increasing number of files. The transfer time is greater than the time the CPU takes to move the files for up to about 750 files, where the CPU time outgrows the transfer time.

Another reason is, that the time it takes to execute the Grid also grows with an increasing number of files, as can be seen in figure 6.10.

6.4.3 Discussion

File moves on the GPU are not faster than on the CPU, as the durations to transfer the file paths from DRAM to VRAM and execute the algorithm are larger than the duration of the corresponding CPU algorithm. GPU file moves must be improved to be faster than CPU file moves. One way, that may improve GPU file moves, is to change the worker size from a Thread Block to a Warp, which could increase the throughput of the algorithm. This hypothesis is supported by the evaluation result for file deletions, which are faster with an increasing number of Warps. We do not change the size of a worker due to a lack of time.

6.5 Summary

File deletions, in general, should not be done using the GPU. Instead, GPU file deletions can supplement CPU file deletions to be used when the user guarantees that there are enough files to delete. The lower bound for the number of files for an RTX 4070 is 1000 files. GPU file moves in its current implementation should not be used under any circumstances, as it is slower than its CPU counterpart. That may change, if the worker size is reduced from a Thread Block to a Warp. We do not change the worker size due to a lack of time.

Chapter 7

Future Work

This chapter lists aspects of the implementation we did not finish due to a lack of time, unanswered questions, and new questions that arise from our evaluation.

7.1 Separate Research Topics

In chapter 6.3, we observe that scenario "Deep" is faster on the GPU than on the CPU, even though the amount of parallelism is low in that scenario. We suspect the cause of this behavior to be the different organization of directories in GPU4FS and ext4. Future research has to verify that suspicion. Furthermore, future research may include how to optimize the arrangement of regular files and directories in GPU4FS to minimize runtimes.

In scenarios "Deep" and "Wide", we see an increase in the runtime from 23 Thread Blocks to 24 Thread Blocks. We have no explanation for this behavior. However, 23 Thread Blocks is exactly half the number of Streaming Multiprocessors (SM) an RTX 4070 contains. Therefore, we believe the reason to be hardware related. Future work has to find explanations for this behavior, which may help to avoid performance pitfalls when implementing producer-consumer patterns on the GPU, due to the increased work load generated by multiple computational processors.

As we mentioned in chapter 3.4, GPU4FS might want to adopt the design of [5], especially if GPU4FS is extended to use multiple GPUs and implements a proper command buffer. Furthermore, developing GPU4FS into a truly parallel file system in the sense of chapter 3.2 might be help to avoid situations in which GPU4FS is slower than CPU based file systems like ext4.

7.2 Further Implementation

In chapters 2.2 and 4.1.4 both, we assume that directories of GPU4FS and the shared buffer in global memory do not overflow. Future research has to address how to handle these situations, possibly by extending the buffers into Intel Optane.

Due to a lack of time, we do not implement dynamic load balancing in chapter 3.4 and do not fix the illegal memory access in the rebuild H-Tree command. Both problems must be solved to determine precisely how many files are necessary for GPU file deletions to be faster than CPU file deletions.

Future research has to examine how to optimize GPU file moves to be faster than CPU file moves, once the rebuild H-Tree command is fixed, so that GPU file moves can be properly evaluated. Part of such research would be to improve the locking mechanism of directories, possibly by replacing it with locks locking smaller parts of the tree, to allow multiple workers to use it simultaneously.

Chapter 8

Conclusion

Our goal was to determine the feasibility and efficacy of parallelizing file deletions and file moves. We implemented file deletions as a producer-consumer pattern. We optimized the buffer of the pattern to handle multiple rapid task insertions by complementing the buffer in global memory with smaller buffers in shared memory. Furthermore, we reduced lock contention by trying to keep worker apart. File moves do not need a producer-consumer pattern, as the amount of files is known in advance. Both file deletions and file moves modify the H-Tree of directories, which is why the tree must be repaired afterward. We repair the H-Tree by reconstructing it and amortize the costs over successive file deletions and file moves.

In our evaluation, we tested file deletions and file moves in scenarios, which differ in the number of files deleted or moved and the structure of the directories that contain the files. We were not able to properly test the command to rebuild an H-Tree, as the command crashes due to an illegal memory access. We conclude that file deletions can be faster than their CPU counterpart if there are enough files to delete. For an RTX 4070, 1000 is a lower bound for that number. The current implementation of file moves is slower in every possible scenario than its CPU counterpart. That may change if the worker size is reduced from a Thread Block to a Warp, which may increase the throughput of files moved.

Future work includes fixing the command to rebuild an H-Tree and changing the worker size of file moves, to analyze precisely under what conditions GPU file deletions and GPU file moves are faster than their CPU counterparts. Furthermore, examining the likely hardware related quirks of file deletion scenarios “Wide” and “Deep” may yield valuable insight for implementing producer-consumer patterns on the GPU. Finally, implementing a multi-GPU command buffer like Chen et al. [5], would expand GPU4FS to multi-GPU systems and replace the limiting cudaStreams which currently serve as command buffers.

Bibliography

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2018. ISBN: 198508659X.

[2] Haodong Bian et al. "ALBUS: A method for efficiently processing SpMV using SIMD and Load balancing." In: *Future Generation Computer Systems* 116 (2021), pp. 371–392. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2020.10.036. URL: https://www.sciencedirect.com/science/article/pii/S0167739X2033020X.

[3] *C++ language extensions*. URL: https://rocm.docs.amd.com/projects/HIP/en/latest/reference/cpp_language_extensions.html (visited on 10/21/2024).

[4] Rémy Card, Theodore Ts'o, and Stephen Tweedie. *Design and Implementation of the Second Extended Filesystem*. URL: https://web.stanford.edu/class/archive/cs/cs240/cs240.1236/old//sp2014/readings/ext2_design.pdf (visited on 09/29/2024).

[5] Long Chen et al. "Dynamic load balancing on single- and multi-GPU systems." In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470413.

[6] Jacques Cohen. "Garbage Collection of Linked Data Structures." In: *ACM Comput. Surv.* 13.3 (Sept. 1981), pp. 341–367. ISSN: 0360-0300. DOI: 10.1145/356850.356854. URL: https://doi.org/10.1145/356850.356854.

[7] Peter F. Corbett and Dror G. Feitelson. "The Vesta parallel file system." In: *ACM Trans. Comput. Syst.* 14.3 (Aug. 1996), pp. 225–264. ISSN: 0734-2071. DOI: 10.1145/233557.233558. URL: https://doi.org/10.1145/233557.233558.

[8] NVIDIA Corporation. *Cuda C Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 10/20/2024).

[9] NVIDIA Corporation. *Gefroce GTX 1070-Series*. URL: https://www.nvidia.com/de-de/geforce/10-series/ (visited on 10/24/2024).

[10] NVIDIA Corporation. *Gefroce RTX 4070-familie*. URL: https://www.nvidia.com/de-de/geforce/graphics-cards/40-series/rtx-4070-family/ (visited on 10/20/2024).

[11] Jack Dongarra et al. "The International Exascale Software Project roadmap." In: *The International Journal of High Performance Computing Applications* 25.1 (2011), pp. 3–60. DOI: 10.1177/1094342010391989. eprint: https://doi.org/10.1177/1094342010391989. URL: https://doi.org/10.1177/1094342010391989.

[12] Ibrahim F. Haddad. "PVFS: A Parallel Virtual File System for Linux Clusters." In: *Linux J.* 2000.80es (Nov. 2000), 5–es. ISSN: 1075-3583.

[13] Mark Harris. *Using Shared Memory in CUDA C/C++*. URL: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/ (visited on 05/22/2024).

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Amsterdam: Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[15] John Hubbard et al. *Simplifying GPU Application Development with Heterogeneous Memory Management*. June 10, 2024. URL: https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/ (visited on 06/10/2024).

[16] The Khronos® Group Inc. *Vulkan*. URL: https://www.vulkan.org/ (visited on 10/13/2024).

[17] Intel. *Intel Optane Technology: Memory or Storage? Both.* URL: https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/what-is-optane-technology-brief.pdf (visited on 08/10/2024).

[18] Intel. *Intel® Xeon® Silver Prozessor 4215*. URL: https://www.intel.de/content/www/de/de/products/sku/193389/intel-xeon-silver-4215-processor-11m-cache-2-50-ghz/specifications.html (visited on 10/20/2024).

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

[20] Lennard Kittner. "Directories for GPU4FS." English. Bachelor's Thesis. Karlsruher Institut für Technologie, Apr. 4, 2023. URL: https://os.itec.kit.edu/downloads/2023_BA_Kittner_Dirs_for_GPU4FS.pdf (visited on 05/27/2024).

[21] G.A. Kohring. "Dynamic load balancing for parallelized particle simulations on MIMD computers." In: *Parallel Computing* 21.4 (1995), pp. 683–693. ISSN: 0167-8191. DOI: https://doi.org/10.1016/0167-8191(94)00112-N. URL: https://www.sciencedirect.com/science/article/pii/016781919400112N.

[22] F.C.H. Lin and R.M. Keller. "The Gradient Model Load Balancing Method." In: *IEEE Transactions on Software Engineering* SE-13.1 (1987), pp. 32–38. DOI: 10.1109/TSE.1987.232563.

[23] Gregor Lucka. "RAID on a File System Level for GPU4FS." Bachelor's Thesis. Karlsruher Institut für Technologie, Sept. 18, 2023.

[24] Peter Maucher. "GPU4FS: A Graphics Processor-Accelerated File System." Master Thesis. Karlsruher Institut für Technologie, Oct. 12, 2021.

[25] Peter Maucher et al. *Full-Scale File System Acceleration on GPU*. Tagungsband des FG-BS Fruehjahrstreffens 2024. 2024. DOI: 10.18420/fgbs2024f-03.

[26] Dutch T. Meyer and William J. Bolosky. "A study of practical deduplication." In: *ACM Trans. Storage* 7.4 (Feb. 2012). ISSN: 1553-3077. DOI: 10.1145/2078861.2078864. URL: https://doi.org/10.1145/2078861.2078864.

[27] Nils Nieuwejaar and David Kotz. "The galley parallel file system." In: *Proceedings of the 10th International Conference on Supercomputing*. ICS '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 374–381. ISBN: 0897918037. DOI: 10.1145/237578.237639. URL: https://doi.org/10.1145/237578.237639.

[28] NVIDIA. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf (visited on 10/13/2024).

[29] OpenSFS and EOSF. *Lustre*. URL: https://www.lustre.org/ (visited on 09/23/2024).

[30] Milan Pavlović. "Memory architectures for exaflop computing systems." PhD thesis. UPC, Departament d'Arquitectura de Computadors, Feb. 2016. DOI: 10.5821/dissertation-2117-96269. URL: http://hdl.handle.net/2117/96269.

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

[32] Nico Rath. "Extending GPU4FS with advanced File System Functionalities." Bachelor's Thesis. Karlsruher Institut für Technologie, Aug. 29, 2023.

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

[34] Jaimeel M Shah et al. "Load balancing in cloud computing: Methodological survey on different types of algorithm." In: *2017 International Conference on Trends in Electronics and Informatics (ICEI)*. 2017, pp. 100–107. DOI: 10.1109/ICOEI.2017.8300865.

[35] Mark Silberstein et al. "GPUfs: Integrating a file system with GPUs." In: *ACM Trans. Comput. Syst.* 32.1 (Feb. 2014). ISSN: 0734-2071. DOI: 10.1145/2553081. URL: https://doi.org/10.1145/2553081.

[36] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. 3., aktualisierte Auflage. Pearson Studium, 2009. ISBN: 3827373425.

[37] S. Tomboulian and M. Pappas. *Indirect addressing and load balancing for faster solution to Mandelbrot set on SIMD architectures*. 1990. DOI: 10.1109/FMPC.1990.89495.

[38] *Understanding the HIP programming model*. URL: https://rocm.docs.amd.com/projects/HIP/en/latest/understand/programming_model.html (visited on 05/20/2024).

[39] Brian Ward. *How Linux Works. What every superuser should know*. *How Linux Works - What every superuser should know*. English. 3. edition. No Starch Press, Sept. 19, 2024, pp. 48–19. ISBN: 978-1-7185-0040-2.

[40] Lukas Werling, Christian Schwarz, and Frank Bellosa. *Towards Less CPU-Intensive PMEM File Systems*. Karlsruhe Institut für Technologie. Sept. 21, 2021. URL: https://www.betriebssysteme.org/wp-content/uploads/2021/09/FGBS_Herbst2021_Folien_Werling.pdf.

[41] Wm Wulf and Sally McKee. “Hitting the Memory Wall: Implications of the Obvious.” In: *Computer Architecture News* 23 (Jan. 1996).

[42] Jie Zhang et al. “NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures.” In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 2015, pp. 13–24. DOI: `10.1109/PACT.2015.43`.