Full-Scale File System Acceleration on GPU

Peter Maucher
Karlsruhe Institute of Technology
Germany

Lennard Kittner
Karlsruhe Institute of Technology
Germany

Nico Rath
Karlsruhe Institute of Technology
Germany

Gregor Lucka
Karlsruhe Institute of Technology
Germany

Lukas Werling
Karlsruhe Institute of Technology
Germany

Yussuf Khalil
Karlsruhe Institute of Technology
Germany

Thorsten Gröninger
Karlsruhe Institute of Technology
Germany

Frank Bellosa
Karlsruhe Institute of Technology
Germany

ABSTRACT

Modern HPC and AI Computing solutions regularly use GPUs as their main source of computational power. This creates a significant imbalance for storage operations for GPU applications, as every such storage operation has to be signalled to and handled by the CPU. In GPU4FS, we propose a radical solution to this imbalance: Move the file system implementation to the application, and run the complete file system on the GPU. This requires multiple changes to the complete file system stack, from the actual storage layout up to the file system interface. Additionally, this approach frees the CPU from file system management tasks, which allows for more meaningful usage of the CPU. In our preliminary implementation, we show that a fully-featured file system running on GPU with minimal CPU interaction is possible, and even bandwidth-competitive depending on the underlying storage medium.

KEYWORDS

File System, GPU, Direct Storage Access, GPU-Acceleration, GPU-Offloading

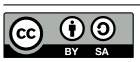
1 INTRODUCTION

Graphics processing units (GPUs) offer massive parallelism for data-parallel applications. This data needs to be loaded into GPU memory in some way. Currently, the data management is handled mostly on CPU, with the GPU only signalling progress to the CPU. Assuming this data comes from storage, the request has to go through the interconnect to the CPU. After the CPU actually starts working on the request, the CPU will have to go through the file system (FS) implementation until it hits the storage device, and after receiving the response, has to signal completion to the GPU. Some of this latency is unavoidable, especially the storage latency, but a lot of the latency can be avoided if GPU hits storage directly. Even with CPU file systems, Volos et al. [26] argue that some overhead is added by old FS interfaces originally designed for HDDs.

In this paper, we present GPU4FS, a modern, fully-featured GPU-side FS designed to solve both the latency induced by the FS interface as well as by the trip through the interconnect. GPU4FS runs on the GPU, in parallel to the actual GPU-side application, which allows for low latency access and shared memory communication between application and the FS. Given the lack of a system call instruction on GPUs, this communication is done via shared video memory (VRAM), utilizing a parallel work queue implementation. We also place work queues in DRAM, which enables fast, parallel, user space access for CPU-side applications to the FS.

We designed a file system with a feature set closely following modern file systems, as the file system interface is widely used and well understood by programmers. Instead of having to port each and every application, we can completely hide the implementation details to unaware applications, but enable an opt-in for CPU-side applications to benefit from the changed semantics. On the GPU, the application needs to be modified for any kind of storage access. Using GPUfs, Silberstein et al. [24] demonstrate that offering a library interface to CPU FS eases access to storage for GPU programmers, but GPUfs only calls a CPU-side file system. GPU4FS offers a similar interface to GPUfs, but runs the file system on the GPU.

In our preliminary implementation, we demonstrate GPU access to Intel Optane Persistent Memory (PMem) [19], and



 DOI: https://doi.org/10.18420/fgbs2024f-03. FGBS '24, March 14-15, 2024, Bochum, Germany.

show that we are write-bandwidth competitive to contemporary Optane file systems. We also demonstrate an implementation for GPU-optimized folder structures and a RAID implementation. We argue that this shows that a fully-featured GPU-side file system is both useful and can work well.

2 METHODOLOGY

Given the novel idea to attempt to move the implementation from the CPU to the GPU, GPU4FS shows major differences to contemporary file systems in the implementation. To benefit from the compatibility with preexisting CPU-side applications and also with the familiarity for new GPU-side applications, these differences need to be hidden by the interface.

2.1 Interface

The most interesting aspect for compatibility is the file system interface. In POSIX [17], accesses are commonly handled using syscalls, which is not feasible in GPU4FS for several reasons: A classical system call instruction is not available on GPUs, and even with such an instruction, it would most certainly not be usable from the CPU. Instead, we elected the common approach (e.g., FUSE [6]) to use a shared-memory buffer for communication. A requesting process will insert a command for every intended file system operation, while the GPU4FS process can then parse the information and handle the request.

GPU4FS is intended to offer two sets of commands: One closely aligned to the POSIX interface, and a high-level interface intended to reduce the number of requests and thus decrease latency and increase bandwidth. Volos et al. [26] show that major improvements can be achieved by reducing the number of interactions: To read a single file into memory using POSIX interfaces requires five operating system interactions: A call to `open()` for the file descriptor, a call to `fstatat()` to get the length of the file, followed by `malloc()` for memory allocation, then one or more calls to `read()` to fetch the actual data and `close()` to release the file descriptor. This does not meaningfully improve with the usage of `mmap()`, as the only call to be left out is the call to `malloc()`. In both cases, `read()` and `mmap()`, the process is inherently racy, as other tasks might change both file content as well as metadata like length at any time. Instead, we offer an interface to load a complete file, with the allocation of memory for the loaded file handled inside the command handler. The resulting data is written into the shared buffer, and a pointer/length pair is returned. This way, the operation can be handled atomically, and only one request is needed. We expect a mixture of both interfaces to be used in GPU applications: For parallel reads of a folder, the POSIX interface can be used so that each item is only loaded once, but each individual file can be loaded using the atomic request. Nevertheless, the POSIX interface is required for compatibility.

One important consideration for the compatible applications is supporting `mmap()`. In Linux, one can either use a private or a shared mapping, which governs two independent features: A shared mapping implies that modified memory is written back to disk, but also allows for shared memory communication with other processes. We assume that most applications use a shared mapping to persist their changes, rather than for shared memory communication. In a CPU-only file system with a DRAM page cache, implementing both is relatively easy, but mixing in multiple devices with their own memory adds complications, and forces the data to be held in DRAM to allow access from all devices. To allow for high-bandwidth VRAM-side mmap, we add an additional write-back mode that only flushes data to disk when unmapped. Additionally, this lets us avoid costly page table changes. We expect this mode to be widely used in GPU-side applications that are inherently incompatible with any other file system.

2.2 GPU Implementation

As a broad overview, a system running GPU4FS has at least two corresponding processes running: the main GPU-side GPU4FS process handling the complete file system, and a CPU-side process responsible to set the GPU process and to establish communication. Additionally, there may be multiple additional processes running on both CPU and GPU that interface with the two GPU4FS processes. In this section, we present the GPU-side implementation, which is responsible for communicating to requesting processes using the shared command buffers as described above and communicating to the storage backend. FS management tasks like garbage collection and defragmentation are also controlled on and run by the GPU.

A request to the file system will always occur through one of the command buffers, either through a buffer in VRAM issued by another GPU-side process or through a buffer in DRAM for any other request. Such a request usually causes accesses to the disk or PMem, either to load or to store information. Additionally, even a data load request can cause data changes on the storage medium, as metadata needs to be updated.

Read Request. First, we describe a read request without any data changes: After the request is parsed, the GPU tries to load data from one of its caches, either in VRAM or in DRAM. We employ both caching levels, as VRAM is usually a lot smaller and DRAM is still a lot faster than contemporary storage devices like NVMe and PMem, and an early paper suggests the same behavior for Compute Express Link (CXL) implementations [25]. If the requested data is not cached,

the data is instead fetched from storage. The main goal of GPU4FS is to bypass the CPU for storage access: PMem is mapped directly to the GPU, and NVMe memory is accessed via Peer to Peer-DMA (P2PDMA) as suggested by Qureshi et al. [21] for their BaM system architecture. After the data is fetched, it is possibly cached depending on application hints. We add caching hints as some applications only require a file once, thus freeing up cache capacity for repeated accesses. Except in a shared mmap mapping, we allocate space for the result in the shared memory area, and store the fetched data there. To signal completion, we store the data pointer and length into the initial command structure, and conclude the interaction with an atomic store to a completion flag.

Write Request. In the opposite direction, when data is to be written, a few more steps are required to guarantee data consistency and to allocate space for new files. Even though latency reduction for GPU-side application is a major goal for GPU4FS, the overall latency for CPU-side applications is expected to grow. This increases the amount of data in-flight, and therefore the pressure on the consistency system. We intend to solve this issue using a combination of log-structuring [23] and journaling [20]: Data writes are done in a log-structured way, while metadata updates are journaled. The journal therefore is a major point of contention. To solve this issue, we reserve a big chunk of persistent storage to the journal, and assign each independent GPU workgroup one smaller subarea of the large chunk. To avoid multiple conflicting journal entries for the same object, we limit every file system object to at most one workgroup at runtime. A journal entry is fixed-size, and allocated using a single atomic store to a flag variable. Another atomic store marks the completion of the journal entry setup. That way, should a crash occur, the whole journal area can be scanned and operation completion can be verified. To commit a data-write, the inode structure needs to be modified to add the pointers to the new file system blocks. As compared to a file system on a block device, PMem only allows for smaller granularity of data persistence, which means that we cannot atomically change the inode in-place. Instead, the journal allows for recovery of either the old or the new state even though we use in-place modifications.

GPU4FS is designed from the ground up for modern storage with good random access performance, and for PMem direct mapping capabilities. WineFS [11] shows the advantages of using hugepages for direct mapping, as compared to only 4 kB pages. GPU4FS incorporates this into its design, combined with a novel take on extents, a common feature in multiple file systems [7, 22] . We use 64 bit pointers where two bits are used for a flag bit to discern between aligned 4 KiB, 2 MiB and 1 GiB pages as well as 256 byte inodes. Not only does this allow for direct mapping, but it also decreases the number of block pointers. On GPU, a decrease in block pointers has the additional advantage of having less pointer divergence, which means the GPUs memory management unit can coalesce more accesses. Similar to WineFS, the allocator maintains free-lists of aligned pages for each of the sizes which are split up into smaller blocks as needed. GPU4FS assumes a flat address space, and does not reserve special areas for inodes, directories, or indirect blocks filled with block pointers. This means that gradual fragmentation is a major concern, which we combat mainly by reusing garbage-collected space and partial defragmentation if needed.

Additional Features. Additionally, GPU4FS implements features found in modern file systems like BTRFS [22] and ZFS [2], namely checksums, deduplication and storage distribution over multiple drives (RAID) [18]. We can use checksums to notify the application if data has been modified, which avoids propagating erroneous data into the application. On top of that, checksums enable deduplication to increase the usable storage area if identical blocks are stored, and improve mapping write-back performance. A RAID implementation lets us combine the storage capacity of multiple drives for larger data sets. Additionally, in combination with checksums, we can use the redundant information in the RAID to recover from checksum errors. Both checksums and RAID require parallel computations, which is well-suited for a GPU.

For the checksum functionality, we select BLAKE3 [15] as the hash function as it is designed for parallel computations, but is still secure with 128 bit collision resistance. To further increase the exploitable parallelism, we checksum each block individually. To differentiate between changes in the checksum as compared to the data, we also calculate the hash for every block storing checksums. We further utilize them to avoid storing the same data twice. At runtime, checksums help with the mmap implementation: If we use mmap in VRAM, the buffer is mapped as read/write to avoid page table modifications, but then, no pagefaults occurs on modification. To detect changes when writing back to disk, instead of having to compare every byte and potentially having to fetch them first, we can simply compare the checksums, thus increasing performance and decreasing storage latency.

The other important feature we implement is software RAID: GPU4FS takes heavy inspiration from the BTRFS volumes, and similarly offers runtime-configurable RAID levels using volumes. The design goal of supporting direct mapping is leads to difficulties with parity information, so direct mapping is enabled on a per-file basis. We tag each block pointer to be either physical, or virtual, with physical pointers encoding a disk and an offset, while virtual pointers are translated through a BTRFS-inspired volume tree to blocks on disk, potentially multiple. This flexibility even allows different RAID levels for different parts of the file.

2.3 CPU Implementation

In GPU4FS, the GPU is mainly responsible for the file system. Nonetheless, the CPU-side GPU4FS implementation plays a major role in establishing communication to the FS for both GPU-side and CPU-side clients, and CPU-side application are able to access the FS on the GPU.

The connection is always established on the CPU, as some parts of the procedure require page table changes and thus kernel privileges. Hence, if a GPU-side client desires to interface to GPU4FS, its CPU-side process managing the GPU-side client is responsible for initializing the communication. The requesting CPU-side process starts by sending an inter-process communication (IPC) message to the GPU4FS CPU-side process, which includes the desired size for the GPU4FS communication buffer. The CPU-side GPU4FS process allocates a part of VRAM for GPU4FS, which is used for shared memory buffers, caches, and internal data, with the remainder of VRAM being used by actual applications. Similarly, the CPU-side GPU4FS process controls shared memory buffers in DRAM for CPU-side FS clients. After the request, GPU4FS allocates some space in its VRAM area or in DRAM for communication, and maps it as a buffer in the requesting process. Here, the page tables need to be modified, so this part needs to run on the CPU. Given that the GPU controls the file system, the GPU-side GPU4FS is notified before communication can be fully established. The CPU-side and GPU-side GPU4FS processes communicate via the same shared memory interface as the other processes, the only difference is the existence of a few higher-privilege commands. One of these higher-privilege commands is used here by the CPU-side process to inform the GPU of the new connection and the newly allocated buffer. The GPU then initializes the buffer and further data structures needed for communication, and signals completion to the GPU4FS CPU-side process. With this completion, the CPU-side process sends a response with the address of the new buffer to the requesting CPU-side process, which can then finally start using GPU4FS. The only difference between a CPU-side GPU4FS client and a GPU-side GPU4FS client is that in the GPU case, two buffers, one in DRAM and one in VRAM, are initialized, and the CPU-side requesting process hands the buffer address to its GPU process.

As mentioned above, during the communication setup, certain changes to the page table of the requesting process need to be made. Additionally, this operation is needed for the shared mmap case. This operation is usually forbidden in user space, and can only be implemented in the kernel. Following the design of Aerie [26] we add a minor kernel modification that allows the CPU-side implementation to map pages in other processes. In normal operation, FS access completely bypasses the kernel, which makes GPU4FS a user space file system, and opens the possibility for performance gains. Figure 1 shows the complete setup with the command flow, including the kernel doing page table modifications for shared mmap.

CPU	2× Xeon Silver 4215, operating at 2.5 GHz
DRAM	8× DDR4 at 2400 MT/s, 16 GiB
PMem	4× DDR-T at 2400 MT/s, 128 GiB
GPU	AMD Radeon RX 6800, VRAM 16 GiB, 16 PCIe Gen3 lanes

Table 1: Test Platform

3 PRELIMINARY RESULTS

We implement a demonstrator to build the case for a full-blown GPU4FS implementation. The main concerns are bandwidth and latency of the file system as compared to a simple GPU access to storage without the file system overhead. Our demonstrator uses Vulkan [9] as the programming interface, running on the RADV [5] driver on Linux. Our implementation adds a simple write path to Optane PMem, and we evaluate the directory creation using an EXT-inspired H-Tree approach [7] and RAID address translation overhead. Due to the Vulkan implementation, we have to restart the file system for every test, which incorporates a startup latency of 12 ms. All the tests run using Intel Optane as the storage medium, the exact specification can be found in Table 1. We make sure that the GPU accesses Optane without going through the inter-processor interconnect.

We achieve a bandwidth maximum of $1.5\,\mathrm{GB\,s^{-1}}$ to one Optane DIMM using our configuration. Even though the CPU can write with up to $2\,\mathrm{GB\,s^{-1}}$, we use the GPU bandwidth as the baseline for our evaluation. Similarly, we also have to wait about 12 ms for the GPU to respond to our command, even if nothing in the file system happens. We assume this is because we have to reinitialize large buffers, which seems to be uncommon in the video game applications the driver is optimized for. In our complete implementation, we expect to reach bandwidth equality to the CPU with all storage media, and we expect the latency to reduce to a few microseconds if GPU4FS is already running instead of having to be started.

The first file system test inserts a single file into a folder, to verify that given a large enough file size, the bandwidth limit mentioned above is reached. In the plot given in Figure 2, we reach the maximum possible bandwidth at a file size of 128 MiB with $1.49\,\mathrm{GB\,s^{-1}}$. This means that the file system does not add an inherent overhead to every copy operation. This plot includes the 12 ms startup latency, to show that we can be bandwidth-competitive even including the higher latency.

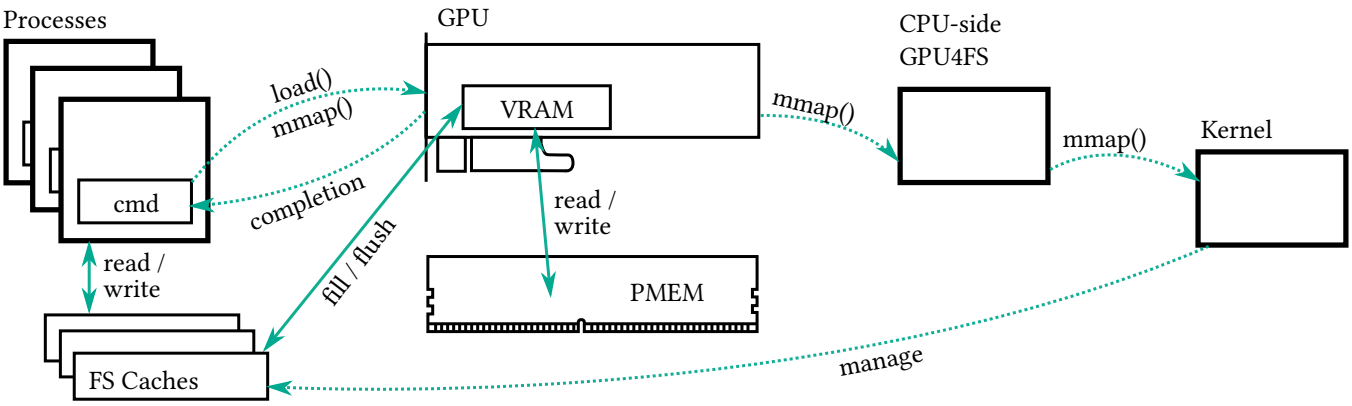


Figure 1: Full-Scale GPU4FS design. Processes either in DRAM or VRAM communicate their requests to the GPU, which accesses storage. Optionally, shared mmap is handed to the kernel for page table modifications.

We also evaluate metadata operations in isolation, but subtract the latency here as the runtime of the metadata operation is similar to the startup latency. As the metadata operation, we create a deep directory chain, similar to a call to `mkdir -p a/b/c/d/...`. On the GPU, we issue a single command, which also shows the benefit of our more general interface, whereas the CPU has to issue repeated `mkdirat()` calls, which incurs repeated syscall overhead. We compare to EXT4, as GPU4FS uses EXT4-inspired H-trees. The results can be seen in Figure 3. GPU4FS is slightly slower than the CPU for large enough requests, but we expect metadata operations to be few and the latency to be hidden by other operations. Also, we expect to reduce the additional latency in the future. The main takeaway is that even dependent metadata operations, though rare, can be efficiently executed on the GPU.

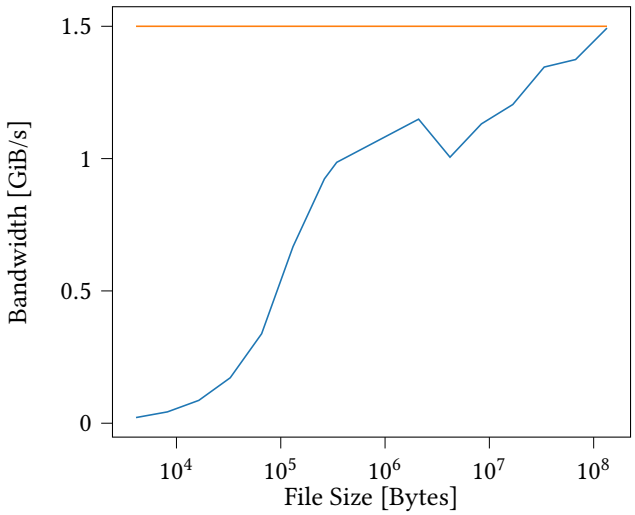


Figure 2: GPU write bandwidth to Intel Optane, for one file with different sizes. The bandwidth increases to the measured max of 1.5 GB s^{-1}.

3.1 Discussion

Our current implementation is not fully optimized, as can be seen by the sub-optimal bandwidth and latency. Currently, we use the RADV driver, which is designed for video games, not for compute, which might contribute to especially the latency issues. In the future, we intend to use AMD's ROCm stack [1], an actual compute-focused API, and to optimize our code to the used hardware based on that new implementation. This will also enable GPU4FS to operate as a daemon instead of having to start the system and incurring other slowdowns, such as TLB misses or cache misses. We expect the latency to further drop when requests are issued from the GPU, and when the file system runs as a daemon instead of having to setup the GPU for every run. We also expect the bandwidth to increase with further optimizations. Nonetheless, there are definitely cases in which the GPU-side comes close to CPU-side implementations, even when tested using CPU access to the GPU, and when incurring additional latency. The results show the potential for a full implementation.

4 RELATED WORK

In this section, we compare GPU4FS to other file systems and GPU projects.

4.1 File Systems

Kernel File Systems. File systems have been a major component of operating systems that they are part of the Portable Operating System Interface (POSIX) [17]. POSIX suggests a file system implementation inside the kernel, thus enforcing system calls for most operations. In Linux, the system calls

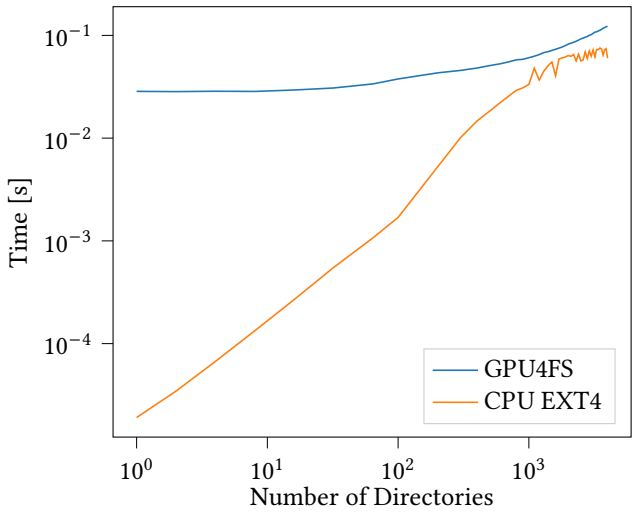


Figure 3: Directory creation time per directory tree depth. At about 1000 directories deep, the GPU and CPU come close.

are the defined interface. POSIX also lists metadata which can be requested for file system objects. Examples of mostly POSIX-compliant file system are the EXT family of file systems [7] for a simple file system, and more feature rich file systems with RAID, full data checksums, deduplication and encryption like ZFS [2] and BTRFS [22]. To reach our goal of POSIX compliance for legacy applications, we take inspiration of the aforementioned file systems in both features and data layout.

A recent comparison are PMem file systems: In NOVA [27], the file system uses direct pointers instead of indices into data structures to cater to modern storage media, and uses different logging strategies for data and metadata, similar to GPU4FS. In comparison to NOVA, GPU4FS changes the metadata log to a classical journal for ease of access and to move the point of parallelization from the file system object to the workgroup. WineFS [11] inspired our tiered allocator for the pages, but we also use the pages as a lightweight implementation of extents, thus keeping the indirection tree small.

User Space File Systems. As compared to file systems completely or mostly implemented in the kernel, Aerie [26] shows the benefit of file systems in user space, especially for PMem storage. To support POSIX, they add a small kernel module to implement the required semantics. Strata [13] and EvFS [28] show the use of tiered storage and of multiple layers of caching inside the user space file systems. However, these file systems all run on the CPU. A GPU is much less optimized for random jumps and pointer chasing, a common feature in these file systems. In comparison, we optimize to extract as much parallelism from the file system as possible, while keeping the benefits of a user space file system.

An interesting hybrid between kernel space and user space file systems is the "Filesystem in Userspace" (FUSE) [6]. FUSE allows the implementation of file systems in user space while preserving the kernel interface, by performing an upcall from the kernel into the user space FUSE driver whenever the kernel is called by an application. FUSE is also implemented using shared memory buffers between user and kernel, similar to GPU4FS, but these buffers are in the DRAM, instead of VRAM or even the memory of a different device.

4.2 GPUs

Applications running on GPU include graphics [4], high performance computing applications [14] and artificial intelligence applications [8], which all require frequent data access or are currently limited by VRAM capacity. Each of these applications can profit from GPU4FS.

GPU-side File Systems. The main comparison point in GPU file systems is GPUfs [24], which demonstrates the use of a file system interface on the GPU by allowing the GPU to access the CPU-side file system. In GPU4FS, we run the complete file system on GPU, and let the CPU access the file system. [21] and [16] show the validity of direct storage access to the GPU, but they never use it for full file systems.

Prior work has implemented several parts of the file system on accelerators, e.g., RAID [3, 12] or checksums [15] or encryption [10], but these parts have never been integrated into one file system.

5 CONCLUSION

In conclusion, in this paper we propose the design of GPU4FS, a novel GPU-side file system with interfaces to be used from the CPU as well as the GPU. We also build the case for a complete realization using our preliminary implementation, which demonstrates that the bandwidth limits are not imposed by the file system implementation, but instead by the storage bandwidth.

With this result, we intent to first finish the port to the new ROCm [1] platform, and use it to increase performance. The next step is to implement the consistency mode, as the complexity is high and it is a central part of the design. Going from there, we will focus on allocation and garbage collection, and integrate the remaining features like RAID, checksumming as well as kernel communication on top.

REFERENCES

[1] Advanced Micro Devices, Inc. 2021. *AMD ROCm™ documentation.* https://rocmdocs.amd.com/en/latest/index.html

[2] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2002. *The Zettabyte File System.*

https://www.cs.hmc.edu/~rhodes/courses/cs134/fa20/readings/The%20Zettabyte%20File%20System.pdf

[3] Matthew L. Curry, H. Lee Ward, Anthony Skjellum, and Ron Brightwell. 2010. A Lightweight, GPU-Based Software RAID System. In *2010 39th International Conference on Parallel Processing*. 565–572. https://doi.org/10.1109/ICPP.2010.64

[4] Blender Developers. 2022. *blender*. https://www.blender.org/

[5] Freedesktop Developers. 2022. *RADV RADV is a Vulkan driver for AMD GCN/RDNA GPUs*. https://docs.mesa3d.org/drivers/radv.html

[6] FUSE Developers. May 06, 2022. *Filesystem in Userspace*. https://github.com/libfuse/libfuse

[7] Linux Kernel Developers. September 20, 2016. *EXT4 Linux kernel wiki*. https://ext4.wiki.kernel.org/index.php/Main_Page

[8] TensorFlow Developers. 2022. TensorFlow. *Zenodo* (2022).

[9] Khronos® Group. 2022. *Khronos Vulkan Registry*. https://registry.khronos.org/vulkan/

[10] Keisuke Iwai, Naoki Nishikawa, and Takakazu Kurokawa. 2012. Acceleration of AES encryption on CUDA GPU. *International Journal of Networking and Computing* 2, 1 (2012), 131–145.

[11] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 804–818. https://doi.org/10.1145/3477132.3483567

[12] Aleksandr Khasymski, M. Mustafa Rafique, Ali R. Butt, Sudharshan S. Vazhkudai, and Dimitrios S. Nikolopoulos. 2012. On the Use of GPUs in Realizing Cost-Effective Distributed RAID. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 469–478. https://doi.org/10.1109/MASCOTS.2012.59

[13] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

[14] Christoph A Niedermeier, Christian F Janßen, and Thomas Indinger. 2018. Massively-parallel multi-GPU simulations for fast and accurate automotive aerodynamics. In *Proceedings of the 7th European Conference on Computational Fluid Dynamics, Glasgow, Scotland, UK*, Vol. 6. 2018.

[15] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. 2021. *BLAKE3 - One function, fast everywhere*. https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf

[16] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2022. GPM: leveraging persistent memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 142–156. https://doi.org/10.1145/3503222.3507758

[17] PASC. 2018. *The Open Group Base Specifications Issue 7, 2018 edition*. https://pubs.opengroup.org/onlinepubs/9699919799/

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

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

[20] Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2005. Analysis and Evolution of Journaling File Systems.. In *USENIX Annual Technical Conference, General Track*, Vol. 194. 196–215.

[21] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 325–339. https://doi.org/10.1145/3575693.3575748

[22] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage* 9, 3, Article 9 (aug 2013), 32 pages. https://doi.org/10.1145/2501620.2501623

[23] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.* 10, 1 (feb 1992), 26–52. https://doi.org/10.1145/146941.146943

[24] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. *SIGARCH Comput. Archit. News* 41, 1 (mar 2013), 485–498. https://doi.org/10.1145/2490301.2451169

[25] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. arXiv:2303.15375 [cs.PF] https://doi.org/10.48550/arXiv.2303.15375

[26] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-System Interfaces to Storage-Class Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 14, 14 pages. https://doi.org/10.1145/2592798.2592810

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

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