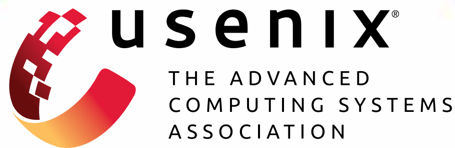


GeminiFS: A Companion File System for GPUs

Shi Qiu, Weinan Liu, Yifan Hu, Jianqin Yan, and Zhirong Shen, *NICE Lab, Xiamen University;* Xin Yao, Renhai Chen, and Gong Zhang, *Huawei Theory Lab;* Yiming Zhang, *NICE Lab, Xiamen University and Shanghai Jiao Tong University*

https://www.usenix.org/conference/fast25/presentation/qiu

This paper is included in the Proceedings of the
23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

Open access to the Proceedings
of the 23rd USENIX Conference on
File and Storage Technologies
is sponsored by

GeminiFS: A Companion File System for GPUs

Shi Qiu[1], Weinan Liu[1], Yifan Hu[1], Jianqin Yan[1], Zhirong Shen[1], Xin Yao[3], Renhai Chen[3]
Gong Zhang[3], Yiming Zhang[1,2]

[1]*NICE Lab, XMU,* [2]*SJTU,* [3]*Huawei Theory Lab*

Abstract

GPU-centric storage solutions enable direct access from the GPU to the storage device via NVMe queues, completely bypassing the CPU. These solutions alleviate the problems of previous CPU-centric solutions that relied on the host CPU to initiate data storage access, such as high CPU-GPU synchronization overheads, I/O traffic amplification, and high CPU processing latency. However, the state-of-the-art GPU-centric solutions have no file abstraction or management functionalities (e.g., fine-grained isolation and access control) of traditional host file systems, and cannot satisfy the needs of GPU-accelerated machine learning (ML) applications like GNN and LLM which require fast file access and data sharing. Therefore, existing GPU-centric storage solutions are inefficient and inconvenient when being applied in practical ML scenarios.

This paper presents a companion file system (called *GeminiFS*) for GPUs. GeminiFS offers a file system interface to GPU programs that enables direct file-based access to NVMe storage, which is managed by the host file system. GeminiFS realizes metadata synchronization between the host and GPU file systems by embedding the metadata directly into the files. We extend the existing NVMe driver to allow the CPU and the GPU to set up their control planes in parallel for the storage device. Moreover, GeminiFS provides a GPU-friendly, software-defined page cache to fully utilize the internal bandwidth of the GPU. We further offer a convenient library (libGemini) tailored for GPU programmers, which abstracts away various underlying complexities thereby reducing programming complexity. Extensive evaluation shows that GeminiFS significantly outperforms the state-of-the-art storage solutions for large-scale ML workloads.

1 Introduction

GPU-accelerated machine learning (ML) applications, such as graph neural networks (GNN) [13, 19, 20] and large language models (LLM) [6, 24, 49, 58], have massive datasets and weights. This typically involves accessing numerous files. Their sizes can reach up to tens of terabytes (TBs) and are expected to continuously grow. As a result, although the capacity of GPU memory has substantially increased in the last decade [1, 23], the gap between GPU memory capacity and application demand is widening.

To mitigate the capacity-demand gap of GPU memory, traditional *memory-based* expansion solutions use host memory (DRAM) [29] or pool together multiple GPUs' memories [10] to accommodate the massive datasets and weights, which is quite expensive in the foreseeable future. In contrast, *storage-based* GPU memory expansion solutions [4, 11, 22, 38, 40] extend GPUs' reach into storage, allowing GPUs to directly access the NVMe storage devices. With the development of high-performance flash technologies, storage-based solutions are more cost-effective than memory-based ones with little performance degradation [4, 39].

Most storage-based GPU memory expansion solutions like Dragon [25] and GDS [35] are *CPU-centric* [39] (a.k.a. CPU-orchestrated [7]). They rely on the CPU to initiate access to the storage either explicitly (via CPU user/OS code to manage data transfer) or implicitly (via CPU page fault handler activated by GPU page faults of memory-mapped files). Unfortunately, CPU-orchestrated approaches are inefficient and cannot satisfy the GPU's throughput demands [39], as the CPU as well as CPU-GPU synchronization becomes a bottleneck when hundreds or even thousands of GPU threads read/write data on the storage device.

To address the inefficiency of CPU orchestration, BaM (Big accelerator Memory) [39] proposes a GPU-centric approach that transfers data directly between GPU memory and NVMe storage devices. BaM implements the control plane completely on the GPU by allocating NVMe queues in GPU memory, through which GPU threads can directly send NVMe I/O commands to the NVMe device without involving the host CPU. On the downside, however, the state-of-the-art GPU-orchestrated approaches do not support file abstraction and management, thus suffering from the lack of data and metadata integrity, crash consistency, durability, and

the ability to manage in-storage resources across GPU devices [17]. When accessing files managed by a traditional file system, they still require memory copies between the CPU and GPU, preventing NVMe storage devices from being efficiently utilized by GPU processes. Consequently, the existing GPU-orchestrated approach cannot meet the high parallelism and data sharing requirements of common training/inference scenarios such as accessing input data [4, 56] and sharing KV-cache [11, 38]. However, building a general GPU file system is prohibitively difficult, since GPUs are naturally unsuitable for running storage software that requires complex metadata maintenance [12].

To tackle these challenges, we introduce a lightweight GPU file system called *Companion File System*, which *coexists* with the host file system. On the host, we use the host file system for file management (e.g., created, moved, and deleted) and integrate necessary metadata into the files [9, 59, 60], so that the file system metadata can be managed on the CPU and shared with the GPU. On the GPU, we retrieve the metadata into the GPU memory, based on which we can provide file system abstractions.

However, designing a companion file system for GPUs faces unique technical challenges including (i) metadata synchronization, (ii) device driver limitations, (iii) GPU page cache efficiency, and (iv) GPU programming complexity.

First, metadata synchronization between the host and GPU file systems is vital for GPU I/O performance [21, 60]. GeminiFS needs to retrieve the metadata integrated into the file efficiently and support a set of file operations specific to GPU programs' storage access demands with high parallelism. Further, crash consistency and concurrency control need to be carefully considered in metadata synchronization.

Second, currently, the NVMe driver only allows the host file system to *exclusively* control the NVMe storage device, and cannot support concurrent control planes for both the host and the GPU [54]. Therefore, even though the GPU can retrieve the metadata shared by the host, it still cannot directly submit NVMe commands to the NVMe storage device.

Third, page cache is essential for improving the performance of not only the host file system but also the GPU file system. However, the traditional page cache design of the host file system cannot satisfy the shareable access and high parallelism requirements of GPU storage access patterns.

Fourth, accelerating GPU storage access necessitates interactions between the GPU programs, the coexisting host/GPU file systems, as well as the NVMe driver. The substantial differences between CPUs and GPUs lead to high programming complexity [41], involving various underlying details.

Fortunately, our analysis shows that GPU-accelerated applications (like GNN and LLM) exhibit two useful characteristics. First, GPU storage I/O workloads have certain predictability, allowing the storage access information to be obtained beforehand based on the model settings. Second, most on-disk data is read-only during its lifetime, and data is written to the storage only through appends. This implies that the metadata of these data is also predictable and remains stable. We exploit these characteristics to simplify the design (e.g., the metadata and index structures synchronization) of the companion file system for GPUs.

In this paper, we present a design of a companion file system (called *GeminiFS*) for GPUs, which provides a set of simplified file system interfaces to GPU programs allowing file-based direct storage access. To the best of our knowledge, GeminiFS is the first GPU-centric file system that unlocks the GPU's view of the host file system and enables GPUs to create on-demand file accesses directly to data on the disk, without relying on the CPU to initiate or trigger.

This paper makes the following contributions.

- We propose a file format (called GVDK) that integrates the indispensable file system metadata into the file, including the file size/type/offset and the mapping of file logical blocks to NVMe physical blocks. Based on GVDK, we realize efficient metadata synchronization between the CPU/GPU file systems and support specific file operations required by GPU programs to access NVMe storage.
- We extend the existing NVMe driver to allow the CPU and the GPU to set up control planes in parallel for the storage device. The shared driver supports I/O queues on both the CPU and the GPU, enabling the host/GPU file systems to concurrently submit NVMe requests to the storage device.
- We design a GPU-friendly, software-defined page cache architecture, which provides a flexible API to exploit locality and control data placement for GPUs' predictable data access. The page cache can be shared by multiple GPU processes to reduce GPU memory footprint.
- We offer an efficient library (called libGemini), which provides simple but powerful abstractions to lower the complexity of using GeminiFS. libGemini hides the details of realizing metadata retrieval, synchronization, control of NVMe I/O queues, as well as host-side initialization.

We have implemented GeminiFS for recently released GPUs. Extensive evaluation shows that GeminiFS significantly outperforms the state-of-the-art GPU storage solutions. We have open-sourced the key components of GeminiFS at https://github.com/nicexlab/GeminiFS.

2 Background and Motivation

2.1 Storage Access of GPU Workloads

Various GPU-accelerated ML workloads, including DNN (deep neural networks) [4, 56], LLM [11, 14, 38, 55, 61] and GNN [3, 15, 16, 30, 36, 53] require efficient storage access, which has been extensively studied in the literature.

In DNN [4, 56] and LLM [14, 55, 61] training, existing studies focus on offloading intermediate data, including intern

Table 1: Summary of storage access characteristics of GPU-accelerated ML workloads.

Application	Data Types	Access Mode	Data Size	Retention
DNN [4, 56]	Training-inputs	Read only	10^{-1} TB~10^{3}TB	Years
	Intermediate weights / activations	Read & Write	10^{1} TB~10^{2}TB	Minutes
	Model weights	Read & Append only seq.write	10^{-1} GB~10^{3}GB	Years
GNN [3, 15, 16, 30, 36, 53]	Adjacency Matrix	Read only	10^{2} GB~10^{1}TB	Years
	Feature vectors	Read & Append only	10^{3} GB~10^{1}TB	Years
	Intermediate weights/ activations/ feature vectors	Read & Write	10^{3} GB~10^{2}TB	Minutes
	Model weights	Read & Append only seq.write	10^{2} GB~10^{3}GB	Years
LLM [2, 11, 14, 38, 43, 55, 61]	Training-inputs	Read only	10^{3} TB~	Years
	Intermediate weights / activations	Read & Write	10^{1} TB~10^{3}TB	Minutes
	KV-Cache	Read & Append only	10^{3} TB~10^{1}PB	Years
	Model weights	Read & Append only seq.write	10^{2} GB~10^{1}TB	Years

weights and activation generated during forward propagation. In GNN training, some works [16, 30, 36, 53] utilize high-volume SSDs to store hundreds of terabytes of adjacency matrices, feature vectors, and intermediate data generated during the training process, which are referred to as short-term data. The retention of short-term data typically does not exceed several minutes. Once the subsequent training iteration starts, the short-term data will become invalid [4]. Currently, storage devices mainly serve as the secondary cache for GPU memory. Training workloads also periodically write model weights to storage as checkpoints, which can be used to resume training in case of failures. The size of checkpoint is basically the same as that of model weight.

In LLM inference applications, some efforts [2, 43] utilize SSDs to store model weights, which can reach hundreds of GBs to TBs. Recent works [11, 38] store KV-cache in SSDs and reuse the KV-cache across multi-turn conversations, to reduce the repetitive computation overhead and improve the inference performance. The model weights and KV-cache in LLM inference, as well as the training input data, require long-term retention. During the lifetime, the long-term data undergoes an initial phase of append-only sequential writes as data is generated. Subsequently, it remains unchanged and will be accessed in a read-only manner.

Long-term data is often shared and read by multiple GPU processes. For instance, model weights are shared across multiple GPU processes when performing parallel model training. KV-cache is widely used in prefix caching to enable their reuse across multiple requests for reducing computational overhead in LLM inference.

Most data access exhibits certain predictability. For instance, the process of DNN training and data features are determined by the model design and the number of iterations. Therefore, the data access pattern and data size of DNN training can be statically analyzed and predicted [56].

Table 1 summarizes the data types, access modes, data sizes, and retentions of storage access of common GPU workloads, which have the following characteristics:

- Short-term data needs no persistence.
- Long-term data is append-only.
- Most storage access has predictable patterns.
- Data needs to be shared across multiple GPU processes.

2.2 Extending GPU Reach to Storage

As illustrated in Fig. 1, existing GPU storage access approaches can be classified into two categories [7, 39], CPU-centric and GPU-centric, depending on whether they are initiated by the CPU or by the GPU.

2.2.1 CPU-Centric Storage Access

CPU-centric approaches rely on the CPU to initiate storage access requests. For instance, GPUfs [44] and syscalls for GPUs [50] allow GPUs to request file data through the host CPU. ActivePointers [42] adds a memory-map abstraction on top of GPUfs, allowing GPU threads to access file data as if they were accessing memory. Dragon [25] incorporates storage access to the UVM [32] page fault mechanism.

These efforts provide a POSIX-like interface for GPU programs and rely on CPU user/OS code to orchestrate data transfer between storage and GPU memory. They utilize the host file system to simplify the programming complexity of GPU storage software. However, the complex control logic between the CPU and the GPU prolongs the storage access path. Furthermore, using low-parallelism CPUs to serve the data demands of high-parallelism GPUs is inefficient [7, 39].

To address this inefficiency, GDS [35] establishes a direct data plane between GPU memory and storage by exploiting their DMA capabilities. Unfortunately, GDS can only provide a non-POSIX interface, which greatly increases the complexity of programming. Moreover, GDS still relies on the CPU to initiate I/O to the GPU, thus being a bottleneck for GPU storage access. For instance, the *cuFileBatchIOSubmit* operation in GDS is limited to handling at most 128 operations

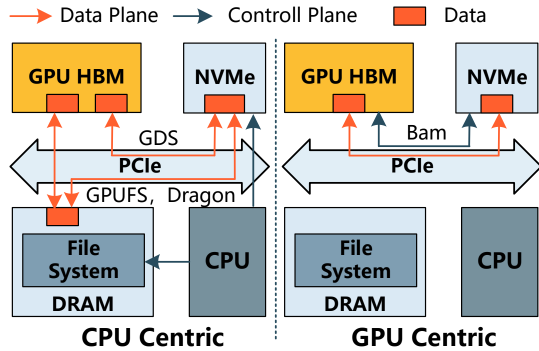


Figure 1: CPU-centric and GPU-centric storage architectures.

per batch [31], which falls short of meeting the demands of GPUs.

We compare the read performance of two representative CPU-centric approaches, namely, GPUfs [44] and GDS [35], with different parallelism levels. The experiment is run on a server equipped with 64 CPU cores and an NVIDIA GPU with 80GB memory. The L1 cache of the GPU is disabled. We use two NVMe storage devices, Zhiti TiPro 7000 (with R/W Latency of 15 μs [27]) and Intel Optane P5800X (with R/W Latency of 4 μs [8]), both of which can achieve a bandwidth of up to 7 GB/sec. The results are shown in Fig. 2.

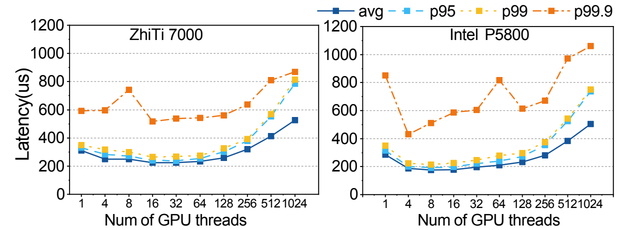
As illustrated in Fig. 2(a), for GPUfs, when the number of GPU threads is low, the I/O latency is higher than 190 μs on both storage devices, implying that the software stack overhead accounts for over 90% of the overall I/O latency. Further, when the number of GPU threads exceeds the number of CPU cores, both the average and tail latencies surge dramatically, e.g., increasing by about 250% for 1024 GPU threads. Fig. 2(b) shows that with small batch sizes, the average and tail latencies of GDS are even higher than those of GPUfs. As the batch size increases, the average and tail latencies decrease but still remain relatively high (around 160 μs). The software stack overhead of GDS is still non-ignorable.

2.2.2 GPU-Centric Storage Access

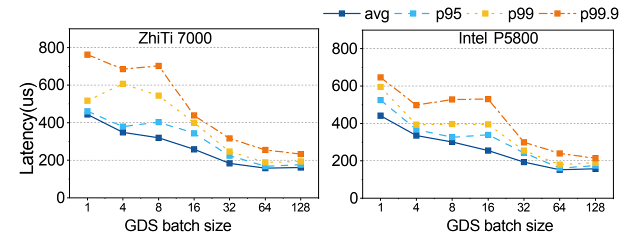
To avoid the inefficiency of the CPU-centric approaches. BaM [39] enables GPUs to orchestrate high-throughput, fine-grained accesses to storage without CPU orchestration overhead. GMT [7] further extends BaM's two-tier hierarchy (GPU memory and storage) to a three-tier one, adding host memory between the GPU memory and the storage device.

BaM allocates NVMe queues in GPU memory and maps them via the GPU driver, making them visible to other devices on the PCIe bus. Further, BaM integrates an NVMe driver into the GPU, enabling GPU threads to directly send NVMe I/O commands, which are then executed by the SSD controllers.

By analogy, it can be seen that BaM adopts a similar approach to the well-known Storage Performance Development Kit (SPDK) [57]: both provide a full block stack as a user-level library to enable direct storage access, respectively for



(a) Average and tail latencies of 4KB read I/O for GPUfs under different numbers of threads. As the number of threads increases, the average and tail latencies significantly increase due to contention among CPU cores.



(b) Average and tail latencies of 4KB read I/O for GDS under different batch sizes. In GDS, the software overhead accounts for more than 90% of the memory access latency. As the batch size increases, the overhead gradually decreases but remains at a high level.

Figure 2: Average and tail latencies of 4K read I/O for GPUfs and GDS under different parallelism levels.

GPU and CPU programs. Therefore, BaM naturally encounters the same problem [62] as SPDK: each GPU/CPU process is granted direct access to the storage device as a block device, losing the appropriate abstraction provided by the traditional file system in the host kernel. As a result, the process needs to implement a user-level file system (e.g. BlobFS provided by SPDK [47]) that manages all data and metadata to ensure system integrity, crash consistency, and durability. Moreover, metadata isolation makes it difficult to share data both among different GPU processes and between GPU and CPU processes [17]. When loading files from the host file system, BaM needs to read in-file data from the NVMe device to the host memory, and then copy it to the GPU memory. This is inefficient in various AI computation scenarios, such as model loading/saving, checkpointing, and sharing [39].

2.3 Challenges

To satisfy the stringent performance demand of GPU-accelerated applications, the goal of this paper is to design a GPU-centric approach for GPU storage access, which can submit I/O requests directly to the storage device completely bypassing the CPU. Meanwhile, the GPU-centric solution needs to support essential management and provide a set of POSIX-like file system interfaces to GPU programs.

These two objectives make it necessary to design a new file system for GPUs. Typically, a file system needs to implement a minimum set of functionalities, including: (i) maintaining metadata for files and directories (including the inode informa-

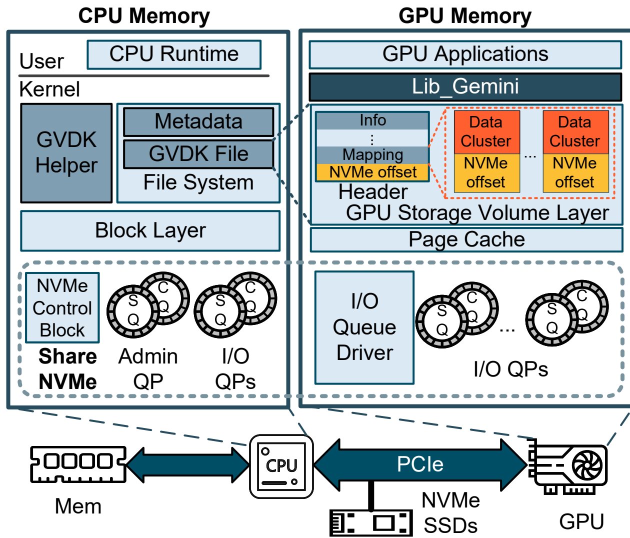


Figure 3: Overview of GeminiFS.

tion and transaction processing for consistency), (ii) mapping logical offset to physical data blocks for I/O requests, (iii) offering a unified interface for upper-layer applications, and (iv) caching recently accessed data blocks for performance purpose. Unfortunately, building such a general GPU file system is prohibitively difficult, since GPUs are naturally unsuitable for running a stateful software such as a file system that requires complex metadata maintenance [12].

Considering all GPU-accessed files can be managed (e.g., created, moved, and deleted) by the CPU, a straightforward solution is to introduce a lightweight GPU file system that coexists with the host file system, so that the file system metadata can be managed on the CPU and shared with the GPU. However, applying this method to GPU scenarios poses the following challenges.

(1) **Metadata synchronization.** The file system metadata needs to be synchronized efficiently and safely between the host and the GPU. First, it is difficult to fully utilize the high parallelism of the GPU in metadata synchronization [28], which may cause a degradation in I/O performance. Second, in traditional host file systems like EXT4, the metadata is exclusively managed in the host kernel. When both the host and the GPU have their own file systems, it is challenging to ensure metadata safety, as GPU file operations can be closely intertwined with metadata operations. It would potentially undermine the benefit of bypassing CPU and results in suboptimal performance due to communication overhead [5].

(2) **Limitations of the NVMe device driver.** Traditionally, the NVMe driver in the kernel manages the NVMe host part of the protocol through the native driver available in the operating system. The driver adheres to the NVMe protocol specification to carry out the initialization procedures, which includes implementing the Admin queue pair and I/O queue pairs used to submit the admin and I/O commands to the NVMe device. Currently, the NVMe driver in the kernel does not support the simultaneous establishment of NVMe queue pairs on both the host and the GPU. As a result, the GPU is unable to directly submit NVMe commands to NVMe devices.

(3) **Inefficiency of GPU page cache.** Establishing a page cache on a GPU file system can fully leverage the GPU's internal memory bandwidth, significantly surpassing PCIe bandwidth. However, modern GPUs do not support a public documented privileged mode [45], meaning that page cache can only be constructed within the GPU process, making it difficult to share across GPUs. This results in memory redundancy and increases synchronization issues. Moreover, the page cache incorporates intricate control logic to ensure data consistency. Implementing this same logic within GPUs may lead to a reduction in GPU parallelism, compromising system performance. Therefore, a GPU-specific system design is crucial to maintaining optimal system performance.

(4) **GPU programming complexity.** To implement a GPU-specific file system capable of sharing metadata with the host file system, GPU programs must efficiently coordinate with the host file system and NVMe device drivers on the host kernel. To provide a POSIX-like interface for host programmers, it is essential to abstract the differences among underlying system processing units and ensure compatibility with the existing well-established GPU programming models, which needs developers to explicitly distinguish between host and GPU code, as well as manage the movement of data between the CPU memory and the GPU memory [28, 46]. This approach demands a considerable programming effort.

3 GeminiFS

To address these challenges, we present GeminiFS, a companion File System for GPUs, which provides GPU programs with direct access to disk space managed by the host file system through file interfaces. The architecture overview of GeminiFS is shown in Fig. 3.

3.1 CPU-Bypassing via Metadata Embedding

File systems achieve a wealth of functionalities based on metadata. Typically, metadata comprises the following components: inode, superblock, directories, index structure, journal, etc. GeminiFS embeds the metadata of the host file system to facilitate sharing this metadata between the CPU and GPU, thereby enabling file system functionality within the GPU.

3.1.1 Selective Embedding of Metadata

Embedding all file system metadata directly within the files and constructing a complete file system on the GPU are prohibitively costly.

First, the kernel file systems on the host have exclusive control over metadata management, primarily to uphold the

security and integrity of the file system. Performing file operations on GPUs is inherently intertwined with metadata manipulations. For instance, appending data during writes requires the allocation of index structures, whereas opening and closing files involves updating access and modification timestamps. To maintain coherence, metadata needs to be synchronized between the CPU and GPU file systems. This runs counter to our initial objective of bypassing the CPU. Fortunately, our analysis in Section 2.1 reveals that GPU storage I/O workloads possess a degree of predictability, permitting us to obtain storage access information beforehand, guided by the model settings. This advantageous feature allows us to preemptively allocate a fixed-size file on the host for GPU-accelerated applications, integrating only *existing metadata* (e.g., access mode, file size, and index structure). We do not need to allocate new metadata within the GPU's file system and synchronize it with the CPU's file system.

Second, specific metadata related to file system management is neither suitable nor necessary for implementation on the GPU side. For instance, directories in a file system are used to organize files. However, implementing them on the GPU significantly increases programming complexity. Furthermore, the primary programming languages for GPUs, such as CUDA, involve applications where the source code is a mixture of traditional C++ host code and GPU device functions. From a developer's perspective, it is straightforward to obtain host file system information and directory details using traditional C++. Therefore, we do not embed such metadata for file system management; instead, we only embed *private metadata specific to each file*.

Private metadata specific to each file encompasses the file type, file size, I/O block size, data blocks, index structure, and block bitmap. By leveraging the metadata, we can perform operations like translating file virtual offsets to physical disk offsets, managing file offsets, and assessing read/write ranges, which pave the way for offering a *read()* and *write()* interface to GPU programs. In addition, the metadata also includes a dirty bitmap, which GeminiFS utilizes to support crash consistency. The dirty bitmap NVMe offset in the metadata points to a contiguous storage page, where each bit records whether the corresponding file page has been written. After the data is written, the dirty bitmap flag is updated, and the bitmap will be flushed later.

3.1.2 Embedded Block Map

In current file systems, the index structure is typically a complex data structure. For instance, in EXT4 [26], an extent tree is employed to record the block mapping between logical and physical blocks. Implementing the translation logic involves intricate control logic and branch jumps. Nevertheless, the control unit of a GPU is relatively simple and cannot effectively handle advanced features, such as branch prediction and out-of-order execution in complex programs [28]. Implementing address translation on a GPU would require lots of effort and would decrease the efficiency of GPU I/O. Therefore, we implemented a kernel module, GVDK helper, on the host. During file creation, this module obtains the physical block offset corresponding to each logical block from the host kernel and embeds it into the file. Although this results in approximately a 0.2% capacity overhead (as a 4KB physical block requires 8B to store the NVMe offset), it can enhance GPU I/O performance.

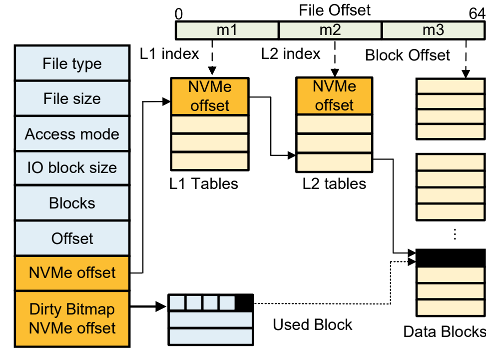


Figure 4: File with metadata embedding.

3.1.3 File Organization

Based on the aforementioned design approaches, we propose a file format tailored for GPU usage, named the GPU virtual disk format (GVDK). Fig. 4 illustrates the organization of the file format.

The GVDK is organized in units of blocks, each of which is equivalent to the block size of the host file system (for instance, 4K in the case of EXT4). A block is a unit in which all allocations are done both for actual data and metadata. The private metadata specific to each file is embedded within the first block of the file. When the file is opened, these data will be read out and cached in GPU memory.

GeminiFS uses a two-level structure to map file offset to NVMe offset. They are called first-level mapping table (i.e., L1 table) and second-level mapping table(i.e., L2 table). The L1 table has a variable size (stored in the header) and may use multiple blocks. However, it must be contiguous in the file. Each entry in the L1 table records an NVMe offset that points to an L2 table, with each L2 table being exactly one block in size. Similarly, each entry in the L2 table records an NVMe offset that points to a data block. A file offset of GVDK m is split into three parts, i.e., $m = (m1, m2, m3)$, where $m1$ is the index to the L1 table, which is used to locate the corresponding entry in the L2 table; $m2$ is the index to the L2 table, which is used to locate the NVMe offset of the corresponding data block; and $m3$ is the offset in the data cluster. To accelerate the address lookup, when invoking an *open()* operation, this mapping table will be cached into the GPU memory. For system security, GeminiFS can adopt a simple method (similar to CUDA lib) to provide users with only pre-compiled

static/dynamic libraries and header files to conceal the details of files. Furthermore, GeminiFS can perform integrity checks on the metadata region to prevent malicious tampering of the embedded metadata, thus protecting against unauthorized access to other files on the disk.

3.2 CPU/GPU Shared NVMe Driver

GeminiFS requires the establishment of a storage device's control plane and data plane on both the CPU and GPU simultaneously. However, current operating system kernels or device drivers do not possess such capabilities.

The NVMe protocol [54] establishes the commands for communication between the host and SSD and how these commands are executed. On a typical server, the CPU allocates *submission queues* (SQ) and *completion queues* (CQ) in host memory to transmit NVMe commands. The establishment of the NVMe device controller in the Linux kernel includes the following processes: First, exactly one pair of an admin submission queue (SQ) and its associated completion queue (CQ) is created, which is used to manage the NVMe controller (e.g., creation and deletion of I/O submission and completion queues, aborting commands, etc.). Next, the NVMe driver submits the admin commands to get the capabilities and settings of the controller data structure. It also submits the admin commands to indicate capabilities and settings specific to a particular namespace. Finally, the controller allocates an appropriate number of I/O queue pairs and registers them using the admin command.

After this process, the NVMe driver registers the namespace with the block layer as host-managed block devices, upon which the operating system can establish a file system.

Our insight is that the GPU does not need to manage the entire NVMe device space; it only needs to correctly read and write it on demand. Therefore, the GPU does not need to implement the entire complex NVMe driver, which increases programming complexity and makes the NVMe device exclusive to the process. Establishing I/O queue pairs on the GPU is only necessary, and providing a relatively simple I/O QP driver can establish the control plane for NVMe I/O.

Thus, we propose the CPU/GPU Shared NVMe Driver (SNVMe), as illustrated in the lower half of Fig 3, enclosed within the blue dashed box. Compared to the standard NVMe host driver, SNVMe makes two major changes. First, we have added a GPU buffer management module to the NVMe driver. The main role of this module is to record the existing memory allocation on GPUs, which are used to construct I/O queues. This module also cooperates with the GPU driver, using the *nvidia_p2p_get_pages_persistent* API to pin the I/O queue memory pages allocated on the GPU and make the pages underlying a range of GPU virtual memory accessible to a third-party device. Then, it uses *nvidia_p2p_dma_map_pages* to convert GPU virtual memory into DMA addresses, making it visible to the NVMe device. Secondly, we have revised the first and third steps of the initialization process for the standard NVMe subsystem, as defined by the NVMe specification mentioned earlier. The corresponding changes for each stage are detailed below:

(1) Before the first stage, we initially allocate memory for the I/O queue in the GPU memory. The GPU buffer management module then translates the GPU virtual address into a DMA address, making it visible to the NVMe device. By leveraging the flexibility of the NVMe standard, we preset the number of I/O queues and their depth, recording this information in the GPU buffer management module. After completing these steps, we start the first step of the standard process.

(2) In the third step, the controller also registers the I/O queues that have been allocated in the GPU memory. These queues do not require the use of host interrupts to complete I/O operations; instead, they utilize GPU thread polling. In GPU memory, we utilize the high-throughput I/O queues driver proposed by BaM [39] to achieve efficient SQ command submission and CQ polling for GPU threads with high parallelism.

3.3 GPU-Specific Page Cache

Compared with page cache on CPUs, constructing a page cache on GPUs exhibits two primary distinctions:

Firstly, because GeminiFS can only run in non-privileged mode on the GPU, page caches can only be allocated within GPU processes' memory space. This results in the redundancy of page caches and subsequent wastage of GPU memory when multiple GPU processes open the same file. To address this issue, we have designed a page cache management module within the SNVMe on the host. This module functions as follows: whenever a program opens a file and establishes a page cache on the GPU, it checks whether the corresponding page cache exists within the management module. If not, it allocates the requisite memory on the GPU and employs the host driver to establish a persistent mapping. Subsequently, it retrieves an inter-process memory handle for the existing device memory allocation and stores it within the management module. When another process attempts to open the same file, a pointer to the corresponding page cache can be obtained via this memory handle. Current GPU drivers, such as CUDA, already support this process; for instance, *cuIpcGetMemHandle* exports existing device memory for use in another process, and *nvidia_p2p_get_pages_persistent* can pin GPU memory persistently.

Secondly, locks are essential to ensure mutually exclusive access to the cache when modifying page mappings in the page cache during file page swapping. Due to the GPU's higher level of parallelism, lock contention becomes more severe than that on the CPU, which can lead to a significant degradation in page cache performance. Thus, GeminiFS employs the following two methods to mitigate this issue.

GeminiFS acquires pages at the warp level rather than the

thread level, which helps reduce lock contention to a manageable level. From a micro-architectural perspective, each Streaming Multiprocessor (SM), which is the fundamental component of GPUs, executes one instruction from a warp at a time, and the number of SMs and warp schedulers determines the maximum number of warps that can execute concurrently, which in turn affects the intensity of lock contention. For example, in the Ampere architecture [34], there are 108 SMs, each with 4 warp schedulers. This means that even if there are thousands of warps trying to acquire different pages, at most 4×108 control flows will be competing for the page cache lock at any given moment. Compared to the thread contention at the level of thousands, this level of contention is more reasonable, albeit it still exceeds the contention typically observed in the CPU page cache.

To further reduce lock contention, a constant-time container for insertion, deletion, and lookup operations can be employed. First, a hash table is used to track the mapping between file pages and memory pages, allowing for constant-time lookup to determine whether a page is a cache hit. If cache misses, a combination of a doubly linked list and a hash table is used to manage pages with zero reference. This combination ensures that insertion, deletion, and lookup operations are all constant-time. If cache hits, it is first queried the page if it is in the zero-reference set and then removed from this set in constant time. If a page is released and becomes a zero-reference page again, it is added to the end of the doubly linked list. By doing so, pages that have not been referenced for a long time will be gradually moved to the head of the doubly linked list, becoming the coldest pages. This approach minimizes the duration of critical sections when manipulating the page cache, thus reducing lock contention.

Based on the above design, GeminiFS offers a GPU page cache with a large tuning space. The page cache size and page size can be flexibly set to adjust the contention intensity of page cache locks. When the data locality of the warp remains unchanged, increasing the page size can reduce the number of pages involved by the warp, thereby decreasing the frequency with which the warp accesses the page cache, and consequently reducing page cache lock contention. The page cache also includes a prefetching feature, which allows for adjusting the number of prefetched pages. This enables a single warp to issue a large number of NVMe commands to the NVMe I/O queue while acquiring pages at warp granularity, thereby maximizing NVMe bandwidth. The GPU page cache with a large tuning space can help users achieve optimal performance across different GPU models. Note that page cache is part of the GPU user-space program, and thus leaving it up to a setting will not introduce new security vulnerabilities.

3.4 GPU Programming Model

In this section, we present libGemini, a GPU-oriented programming model within GeminiFS, which offers abstractions for both the underlying architecture and file system. From the perspective of GPU programming, developers utilize a simple interface to initialize GeminiFS, remaining oblivious to the complexity of the underlying system components throughout the system initialization process. There is no discernible difference for developers when initiating file I/O operations within the GPU compared to those on the host, and it can be accessed via a subset of POSIX-like file I/O APIs outlined in Table 2. These APIs streamline the process of integrating GeminiFS into existing frameworks, such as PyTorch's DataLoader [37]. By merely setup GeminiFS at the initial stage and replacing the host file system interface with GeminiFS's interface, the need for a host bounce buffer is eliminated.

Table 2: CPU-side and GPU-side APIs of GeminiFS.

Type	GeminiFS Interface
host	int Geminifs_init(char *dev_path, char * GPU_ids, int Q_num)
	dev_fd G_open(char *path, uint16 flag, uint64_t cache_capacity, int page_size)
	int G_close(dev_fd fd)
device	int G_read(dev_fd fd, void *buf, uint64_t offset, size_t nbyte)
	int G_write(dev_fd fd, void *buf, uint64_t offset, size_t nbyte)
	int G_sync(dev_fd fd)

libGemini does not provide the full set of POSIX semantics, such as crash consistency. We leave the option of crash consistency to the applications by providing the sync interface to ensure that data and metadata are flushed to the disk consistently. This is because of the following reasons. First, as analyzed in Section 2.1, data is mostly read-only. Intermediate data offloaded to disk during training does not require consistency guarantees. Second, all metadata within files is managed by the application itself, and GPU read/write operations do not affect the metadata of the host file system.

In addition, libGemini does not implement a comprehensive suite of POSIX I/O operations. We concur with the viewpoint expressed in some works [41] that being fully POSIX-compliant is not only costly but also unnecessary for most use cases. The current read/write interfaces are sufficient to meet the storage needs of GPU-accelerated applications. In this section, We will present the GPU programming mode in the context of system startup and the read/write process. Fig. 5 illustrates the corresponding diagram.

SNVMe Init: As illustrated in the first diagram of Fig. 5, the developer initializes the system by calling *Geminifs_init*, which serves as a CPU-side interface. This interface comprises three parameters: (i) *dev_path*, which represents the NVMe device to be used (e.g., /dev/nvme0n1); (ii) *GPU_ids*, which signifies an array of GPU device IDs for which I/O queues need to be established; and (iii) the number of I/O queue pairs to be created on each GPU device. After the de-

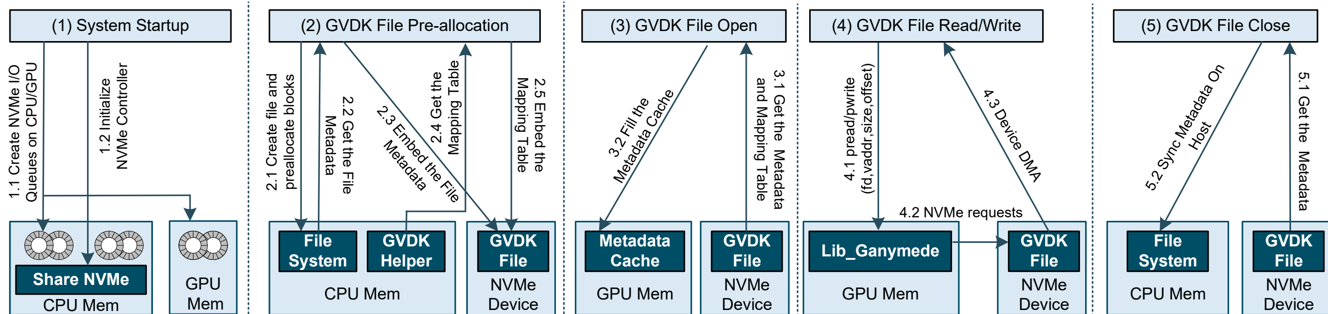


Figure 5: The system diagram of performing file Read/Write on GPU using GeminiFS.

veloper calls *Geminifs_init*, the NVMe I/O queues will be created on the GPUs, and subsequently, the system initializes SNVMe per the method described in Section 3.2.

File Open: The GeminiFS offers a CPU-side interface called *G_Open*. It returns a POSIX-like file descriptor that GPU-side programs can utilize. Compared to the standard *open* interface, this interface abstracts the complex logic required by GeminiFS, such as creating specifically formatted files and establishing page caches. The process is illustrated in Fig. 5(2)(3).

This interface uses four input parameters, namely, *path*, *flag*, *cache_capacity*, and *page_size*. The *path* parameter denotes the address of a specific path on the host. It is noteworthy that *G_open* is solely capable of opening GeminiFS files. The *flag* parameter, similar to its counterpart in the standard *open* function, must include one of the access modes listed below: O_RDONLY, O_WRONLY, or O_RDWR. If the specified *path* does not exist, the *flag* must additionally include O_CREAT, prompting *G_Open* to generate the file in accordance with the format detailed in Section 3.1. In addition, the *flag* should encompass O_DIRECT to indicate whether the page cache should be employed. After the completion of the aforementioned procedures, GeminiFS proceeds to store the pertinent metadata of the file in the GPU's memory and provides the address of this metadata cache. This address is abstracted as *dev_fd*. The subsequent read/write operations can leverage *dev_fd* to access the necessary metadata.

File Read/Write: The GeminiFS offers two GPU-side interfaces for data transfer, namely *G_read* and *G_write*. Both of these functions encompass four parameters identical to the standard *pread*/*pwrite*: *dev_fd*, *buf*, *offset*, and *nbyte*. Initially, the interfaces locate the metadata of the file to be read from or written to based on *dev_fd*. They begin by assessing the access mode, checking whether the *offset* and *nbytes* fall within the file's boundaries. If the request is valid, they convert the virtual address (*vaddr*) using the transformation method described in Section 3.1, transforming the file offset into an NVMe offset. Subsequently, they invoke the NVMe I/O queue driver on the GPU side to generate an NVMe request, which is then submitted to the CQ. Following this, they determine the completion status of the read/write request based on the results obtained by polling the CQ and return the number of bytes transferred.

File Sync: Based on the analysis in Section 2.1, during the generation of Model weights and KV-cache in GPU workloads, write operations occur. These involve modifications to both internal file metadata and data blocks. To mitigate the risk of data loss or file corruption, GeminiFS provides a GPU-side *G_sync* mechanism to ensure that the modified file system metadata and cached file data are written to the file, guaranteeing data persistence.

File Close: To ensure the encapsulation of operations, GeminiFS offers a CPU-side *G_sync* interface function. This function locates the metadata and the page cache of the file within the GPU based on *dev_fd*. After ensuring that the modified data has been written to disk, it releases these GPU memory resources.

4 Evaluation

In this section, we seek to answer the following questions:

- What is the performance advantage of GeminiFS compared to the state-of-the-art solutions (§4.1)?
- How to maximize the performance of GeminiFS on GPU Architectures (§4.2)?
- How does GeminiFS benefit real-world applications (§4.3)?

System settings. We have implemented GeminiFS for recently released GPUs, where about 2000 line-of-code (LoC) is for the shared NVMe kernel module, and about 3000 LoC for LibGemini. All experiments are conducted on a 64-core Intel Xeon 5416S server equipped with 512 GB of memory, running Ubuntu 20.04 and Linux kernel 5.15.0. The server was equipped with a GPU with 80GB HBM [33]. The GPU's HBM bandwidth peaks at 1,935GB/s, and it interconnected with the host via PCIe Gen4 x16, offering a bandwidth of 64GB/s. The NVMe device is an Intel Optane 5800X [8] with an EXT4 file system mounted on it. The NVMe controller of this device supports the establishment of up to 135 pairs of I/O QPs. We allocate 64 QPs on the host and 32 QPs on the GPU. The allocation of 32 QPs is sufficient for GPU to maximize the bandwidth of the disk [39]. The block size of

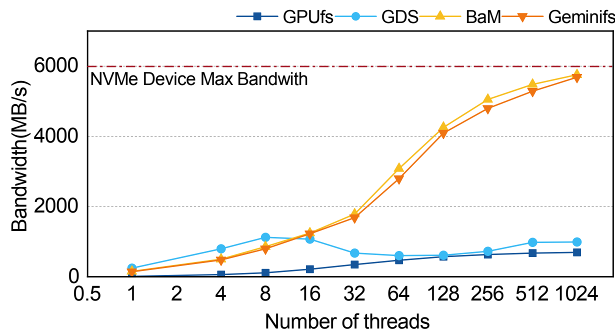


Figure 6: Comparison of 4K read I/O bandwidth of GeminiFS with GPUfs, GDS and BaM at various levels of parallelism.

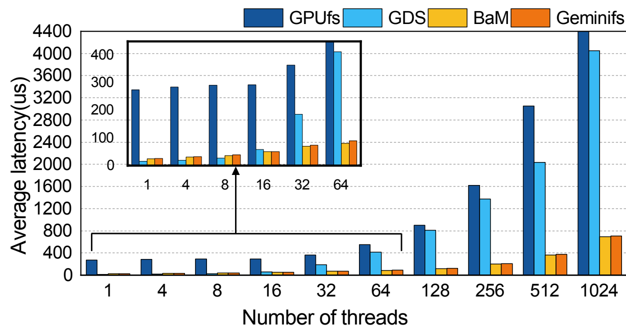


Figure 7: Comparison of 4K read I/O average latency of GeminiFS with GPUfs, GDS and BaM at various levels of parallelism.

the EXT4 file system on the host and that of GeminiFS on the GPU are both 4k. Note that EXT4 cannot handle cases where the block size exceeds the system page size.

Baselines. We compare GeminiFS with state-of-the-art GPU storage solutions. (a): GPUfs [44], CPU-centric solution using the accelerator-centric model. For GPUfs, we allocate four CPU threads to handle incoming requests from the GPU, a configuration capable of saturating the physical disk's capacity [57]. (b): NVIDIA GDS [35], CPU-centric solution using CPU code for data orchestration. For GDS, we use the cufile API provided by NVIDIA to create CPU threads for data transfer. (c): BaM [39] is a GPU-centric solution without file system support. We employ a non-file-system interface to transfer data from NVMe raw devices to GPU memory.

4.1 Comparison with the SOTA Solutions

To investigate the architectural advantages of GeminiFS, we initially compare its read performance, operating with a 4K granularity, across various levels of parallelism, with that of GPUfs, GDS, and BaM. To eliminate the impact of caching on performance, our tests in this section specifically bypassed the caching mechanism of GeminiFS. For GPUfs, we reduced its GPU cache size to 256MB and opened host files using *O_direct* mode.

Bandwidth. Fig. 6 shows the bandwidth performance of various solutions employing a 4KB granularity across diverse levels of parallelism. Compared to GPUfs, GeminiFS exhibits higher bandwidth across all numbers of GPU threads, averaging 7.33× the bandwidth of GPUfs. When the thread number reaches 1,024, GeminiFS reaches the NVMe bandwidth peak. Although GPUfs can also initiate I/O requests within the GPU program, exploiting the high parallelism within the GPU to enhance request concurrency, the limited number of cores on the host CPU leads to performance degradation due to contention when processing GPU I/O requests. GeminiFS bypasses the CPU, allowing the GPU kernel to directly submit requests to NVMe I/O queues. Additionally, GeminiFS incorporates the high-throughput I/O queues design on the GPU side, proposed by BaM [39], which enhances the parallelism of GPU thread request submissions and reduces contention for I/O queues, resulting in improved overall performance. GDS demonstrates a bandwidth that is approximately 57% higher than that of the GeminiFS system only when the thread count ranges from 1 to 16. This advantage stems from GDS's utilization of the CPU to orchestrate data transfers. With its CPU core frequency reaching 4GHz, which surpasses the 1GHz of the GPU core (as mentioned in [48]), GDS ensures superior performance for its software stack. However, as the GPU thread count increases to 128 to 512, the bandwidth of the GeminiFS system reaches 6.2 times that of GDS. This is attributed to the GPU's ultra-high physical parallelism, which allows the file system to fully leverage the maximum bandwidth of the disk even at a small 4K granularity. In contrast, GDS fails to deliver sufficient parallelism at lower thread counts to fully saturate the NVMe read bandwidth. Conversely, higher thread counts lead to thread contention, resulting in a decline in performance [51]. Compared to BaM, GeminiFS exhibits a slightly lower bandwidth by 4.6% across various levels of parallelism. This is because, unlike BaM, which utilizes NVMe devices in a raw device manner, GeminiFS accesses files through its read/write interfaces. These interfaces incur overhead due to metadata parsing and address translation.

Latency. Fig. 7 shows the average latency of 4K sequential read. Compared to GPUfs, GeminiFS achieves a latency reduction of (79.6% - - 90.9%) under different number of threads. This is because GPUfs demands significant memory on both the GPU and CPU for I/O request transmission. This also prolongs both the control plane and data plane and leads to a high I/O latency. GeminiFS eliminates the overhead incurred by GPU-to-GPU communication and the software stack cost on the host. Compared to GDS, when the number of threads ranges from 1 to 8, the latency of GeminiFS is 57.2% higher than that of GDS. However, as the number of threads increases, the latency of GeminiFS does not rise as rapidly as that of GDS. Specifically, when the thread count reaches 1,024, the latency of GeminiFS is only 17% of GDS. This advantage is attributed to the high parallelism within

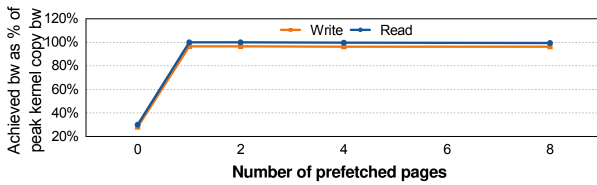


Figure 8: Contribution of different numbers of prefetched pages to the page cache performance. Prefetch is needed.

the GPU, which prevents performance degradation due to thread contention for CPU cores, unlike in the case of GDS. Finally, compared to BaM, the latency of GeminiFS has only increased by approximately 4.8%.

4.2 Performance of Page Cache

GeminiFS provides a configurable page cache to leverage the GPU's internal bandwidth fully. We use the simple microbenchmark below to examine the essential performance of the page cache and its sensitivity to several key configuration parameters. The microbenchmark sequentially reads data from a single 20 GB file in the GPU kernel via GeminiFS. To prevent NVMe bandwidth from becoming a bottleneck in the system, we conducted tests using memory replication as a substitute for accessing NVMe devices.

Prefetch. Firstly, We evaluated the impact of prefetch on page cache performance. In this experiment, we set the page size to 64KB and the page cache size to 1GB. The prefetch strategy employed is straightforward: the page cache would prefetch multiple pages into memory upon each cache miss. This strategy caters to the demands of sequential read and write workloads. Fig.8 compares page cache performance with and without prefetch enabled. Before enabling prefetch, the read and write bandwidth of the page cache were only 30.2% and 28% of the theoretical bandwidth, respectively. After enabling prefetch, regardless of the number of memory pages prefetched, the read and write performance of the page cache approximately improved by 2.4× and 2.34×, respectively, nearly reaching the maximum bandwidth.

Number of Warps. GeminiFS acquires the pages in page cache at the warp level and employs a constant-time container for insertion, deletion, and lookup operations to reduce lock contention. To validate the effectiveness of this strategy, we utilized the *G_Read*/*G_Write* interfaces for reading and writing page cache, assigning 32 threads to each warp. Then, we adjusted the number of warps to evaluate the performance of the page cache under various conditions. The results are illustrated in Fig. 9. As depicted in the figure, the write bandwidth steadily increases from approximately 1.7 GB/s to roughly 641.2 GB/s, while the read bandwidth rises from about 2.3 GB/s to nearly 658.1 GB/s. The bandwidth growth follows a multiplicative trend, ultimately peaking at around 650 GB/s.

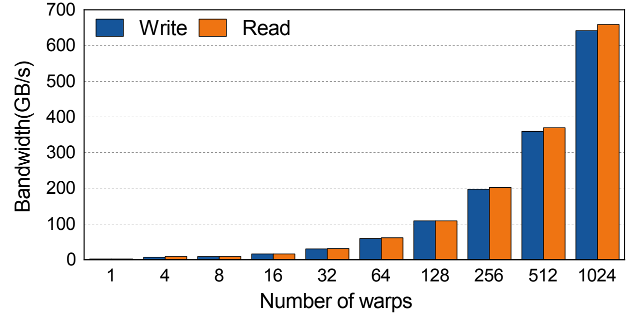


Figure 9: Contribution of different number of warps to the page cahe performance. The peak bandwidth provided by the page cache in GeminiFS exceeds 640 GBps.

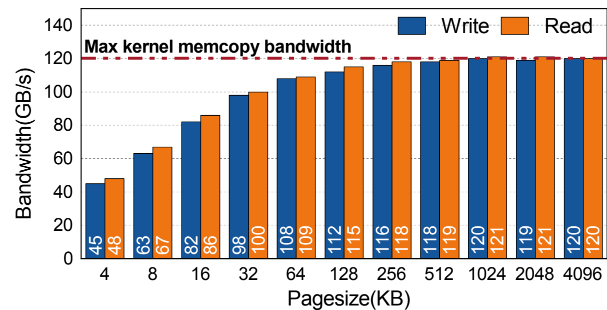


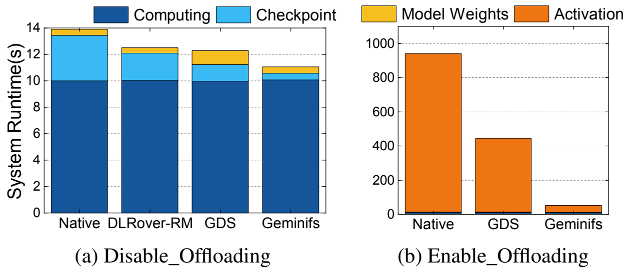
Figure 10: Contribution of different number of page sizes to the page cahe performance.

This observed trend demonstrates the effectiveness of our strategy in reducing lock contention within the page cache. Currently, a single NVMe device can provide a maximum bandwidth of 7GB/s. This implies that it would require close to 100 NVMe drives to saturate the bandwidth of the page cache, making it highly unlikely for the page cache to become a bottleneck in system performance.

Page size. We also conducted evaluations to assess the performance of the page cache for various page sizes. With a warp count of 128, based on the experimental results from the previous section, the maximum memory bandwidth for kernel memcopy is approximately 120GB/s. The page cache comprises 4,096 pages, and prefetching is enabled to enhance performance further. Fig. 10 shows read bandwidth for different page sizes. As the page size increased from 4KB to 1,024KB, the write bandwidth notably rose from approximately 45.8 GB/s to 120.1 GB/s, while the read bandwidth also climbed from around 48.4 GB/s to 121.4 GB/s. With 1,024KB page size, both the read and write bandwidths approached the theoretical limits. Based on the experimental results, it is evident that larger page sizes are more effective in achieving the peak bandwidth of the page cache.

Type	File Size	Access Mode
Model Weights	238MB	Read & Write
Checkpoint	713MB/Step	Append-only seq. write
Activation	57.96GB	Read & Write

Table 3: Storage access in GPT2-124M training.



(a) Disable_Offloading (b) Enable_Offloading

Figure 11: Comparison of GPT2-124M training performance between GeminiFS, GDS, DLRover-RM and a native way (memcopy + read/write). (a) Maintain the activation on HBM; (b) Offload the activation.

4.3 Performance Benefit for LLM training

We compared the performance of several existing GPU storage solutions, including native (memcopy + read/write), DLRover-RM [52], GDS, and GeminiFS, in offloading data generated during LLM training. DLRover-RM provides a fast checkpoint mechanism that accelerates the process by performing asynchronous data persistence. The size of the GeminiFS page cache is set to 2GB. We conducted training using the GPT2-124M model [18], which is a large transformer-based language model with 1.5 billion parameters. During our training process, we used a batch size of 64. We run the training in only three steps, with each step including one forward pass and one backward pass. The model weights were also updated, and a checkpoint was recorded after each step. This is sufficient for studying storage access optimization, as the computation and storage access patterns remain almost consistent in each step of LLM training. The characteristic of storage access is shown in Table 3.

We compared two training models, one that keeps activations in memory during the training process and another that offloads activations to storage during training, to evaluate the benefits of the system in different scenarios. Fig. 11(a) presents the system runtime using different schemes in the first training mode and a thorough time-slicing analysis. The system runtime of GeminiFS has decreased by 25%, 12% and 10% compared to native, DLRover-RM and GDS, respectively. In this training mode, computation time accounts for the majority of the total system runtime, and the main contribution of optimization comes from GeminiFS's optimization of checkpoint I/O. Specifically, the checkpoint write time has been reduced by 85%, 75% and 59% compared to others. This is because, on one hand, GeminiFS completely bypasses the CPU to reduce communication overhead. On the other hand, the ultra-high bandwidth of page cache also improves system throughput.

Fig. 11(b) presents the experimental results under the condition of unloading activations. GeminiFS significantly reduces the training time, decreasing 94.5% and 91% compared to native and GDS, respectively. Under this workload, the unloading of activations accounts for the majority of the training time, including a large amount of small I/Os, preventing the effective utilization of system bandwidth. In CPU-centric solutions, GeminiFS mitigates the frequent communication overhead between the GPU and the CPU. Furthermore, the high-bandwidth page cache enables full utilization of system bandwidth. Moreover, compared to keeping all activations in GPU memory, the training time only increases by about 4×. Theoretically, by integrating multiple NVMe devices into the system, similar performance to a DRAM-only setup can be achieved.

5 Conclusion and Future Work

This paper presents a companion file system (GeminiFS) for GPUs. GeminiFS realizes metadata synchronization between the host and GPU file systems by embedding metadata into the file and extends the existing NVMe driver to allow the CPU and the GPU to set up control planes in parallel. These enable GPU-centric storage solutions to direct access to disk space managed by the host file system through a file interface. GeminiFS shortens the control and data planes of the GPU storage and improves file system performance by leveraging the architectural advantages of the GPU. Its coexistence with the host file system better satisfies the storage access demands of ML applications.

GeminiFS will completely support multi-GPUs. Our roadmap is as follows. We will first enable parallel reads and writes by splitting files logically. Next, we aggregate multiple NVMe devices using RAID to meet the bandwidth requirements of multiple GPUs. For unpredictable workloads, GeminiFS will adopt file pre-allocation [21] that allocates pre-allocated file slots before running the system, and batch-allocate files to fill these slots based on actual usage during runtime. Besides, We will integrate GeminiFS into the PyTorch framework so that state-of-the-art solutions, such as vLLM, will benefit from it.

Acknowledgment

We thank our shepherd, Prof. Sudarsun Kannan, and the anonymous reviewers for their valuable feedback and suggestions. The work is supported by the National Key Research and Development Program of China (grant no. 2022YFB4500302) and the National Natural Science Foundation of China (grant no. 62441220). Yiming Zhang is the corresponding author.

References

[1] Nvidia hopper h100. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet?ncid=no-ncid, 2024.

[2] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. *arXiv preprint arXiv:2312.11514*, 2023.

[3] Jiyoung An, Esmerald Aliaj, and Sang-Woo Jun. Baraddur: Near-storage accelerator for training large graph neural networks. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 225–237. IEEE, 2023.

[4] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W Lee. Flashneuron: Ssd-enabledlarge-batch training of very deep neural networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 387–401, 2021.

[5] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 56–61, 2017.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[7] Chia-Hao Chang, Jihoon Han, Anand Sivasubramaniam, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-Mei Hwu. Gmt: Gpu orchestrated memory tiering for the big data era. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 464–478, 2024.

[8] Intel Corporation. Intel optane dc ssd series 400 gb. https://www.intel.cn/.

[9] Deepset. Embedding metadata for improved retrieval. https://haystack.deepset.ai/tutorials/39_embedding_metadata_for_improved_retrieval, 2024.

[10] Jianbo Dong, Hao Qi, Tianjing Xu, Xiaoli Liu, Chen Wei, Rongyao Wang, Xiaoyi Lu, Zheng Cao, Yunfei Du, and Fu BinZhang. Kspeed: Beating i/o bottlenecks of data provisioning for rdma training clusters. In *IEEE ICNP*, 2024.

[11] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Attentionstore: Cost-effective attention reuse across multi-turn conversations in large language model serving. *arXiv preprint arXiv:2403.19708*, 2024.

[12] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. Gpgpu processing in cuda architecture. *arXiv preprint arXiv:1202.4347*, 2012.

[13] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430*, 2021.

[14] Hongsun Jang, Jaeyong Song, Jaewon Jung, Jaeyoung Park, Youngsok Kim, and Jinho Lee. Smart-infinity: Fast large language model training using near-storage processing on a real system. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 345–360. IEEE, 2024.

[15] Qisheng Jiang, Lei Jia, and Chundong Wang. Gnndrive: Reducing memory contention and i/o congestion for disk-based gnn training. In *Proceedings of the 53rd International Conference on Parallel Processing*, ICPP '24, page 650–659, New York, NY, USA, 2024. Association for Computing Machinery.

[16] Qisheng Jiang, Lei Jia, and Chundong Wang. Reducing memory contention and i/o congestion for disk-based gnn training. *arXiv preprint arXiv:2406.13984*, 2024.

[17] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true {Direct-Access} file system with {DevFS}. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 241–256, 2018.

[18] Andrej karpathy. Gpt2-124m. https://github.com/karpathy/llm.c.git.

[19] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[20] Cangyuan Li, Ying Wang, Cheng Liu, Shengwen Liang, Huawei Li, and Xiaowei Li. Glist: Towards in-storage graph learning. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 225–238, 2021.

[21] Haoyu Li, Sheng Jiang, Chen Chen, Ashwini Raina, Xingyu Zhu, Changxu Luo, and Asaf Cidon. Rubbledb: Cpu-efficient replication with nvme-of. In *2023 USENIX*

Annual Technical Conference (USENIX ATC 23), pages 689–703, 2023.

[22] Changyue Liao, Mo Sun, Zihan Yang, Kaiqi Chen, Binhang Yuan, Fei Wu, and Zeke Wang. Adding nvme ssds to enable and accelerate 100b model fine-tuning on a single gpu. *arXiv preprint arXiv:2403.06504*, 2024.

[23] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[24] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. Deja vu: Contextual sparsity for efficient LLMs at inference time. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 22137–22176. PMLR, 23–29 Jul 2023.

[25] Pak Markthub, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Satoshi Matsuoka. Dragon: breaking gpu memory capacity limits with direct nvm access. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 414–426. IEEE, 2018.

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

[27] YANGTZE MEMORY. Tipro7000 nvme ssd. https://www.ymtc.com/cn/products/34.html?cat=44/.

[28] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. Solros: a data-centric operating system architecture for heterogeneous computing. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[29] Seungwon Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-Mei Hwu. EMOGI: efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 14(2):114–127, 2020.

[30] Fuping Niu, Jianhui Yue, Jiangqiu Shen, Xiaofei Liao, and Hai Jin. Flashgnn: An in-ssd accelerator for gnn training. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 361–378. IEEE, 2024.

[31] NVIDIA. Gpudirect storage parameters. https://docs.nvidia.com/gpudirect-storage/configuration-guide/topics/gds-parameters.html.

[32] Nvidia. Unified memory for cuda beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners, 2017.

[33] Nvidia. Nvidia a100 tensor core gpu architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[34] Nvidia. Nvidia ampere ga102 gpu architecture. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, 2020.

[35] Nvidia. Nvidia gpudirect storage. https://docs.nvidia.com/gpudirect-storage/index.html, 2024.

[36] Jeongmin Brian Park, Kun Wu, Vikram Sharma Mailthody, Zaid Quresh, Scott Mahlke, and Wen-mei Hwu. Lsm-gnn: Large-scale storage-based multi-gpu gnn training by optimizing data transfer scheme. *arXiv preprint arXiv:2407.15264*, 2024.

[37] Pytorch. Pytorch documentation. https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader.

[38] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Kimi's kvcache-centric architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

[39] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, Chris J Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. 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*, pages 325–339, 2023.

[40] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–14, 2021.

[41] Zhenyuan Ruan, Tong He, and Jason Cong. Insider: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, 2019.

[42] Sagi Shahar, Shai Bergman, and Mark Silberstein. Activepointers: a case for software address translation on gpus. *ACM SIGARCH Computer Architecture News*, 44(3):596–608, 2016.

[43] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Re, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR, 23–29 Jul 2023.

[44] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: Integrating a file system with gpus. *ACM Trans. Comput. Syst.*, 32(1):1:1–1:31, 2014.

[45] Mark Silberstein, Bryan Ford, and Emmett Witchel. Gpufs: The case for operating system services on gpus. *Communications of the ACM*, 57(12):68–79, 2014.

[46] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. Gpunet: Networking abstractions for gpu programs. *ACM Transactions on Computer Systems (TOCS)*, 34(3):1–31, 2016.

[47] SPDK. Blobfs. https://spdk.io/doc/blobfs.html.

[48] Techpower. Nvidia a100 pcie 80 gb. https://www.techpowerup.com/gpu-specs/a100-pcie-80-gb.c3821, 2021.

[49] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[50] Ján Veselỳ, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H Loh, Mark Oskin, and Steven K Reinhardt. Generic system calls for gpus. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 843–856. IEEE, 2018.

[51] Devavret Makkar Vukasin Milovanovic and Gregory Kimball. Boosting data ingest throughput with gpudirect storage and rapids cudf. https://developer.nvidia.com/zh-cn/blog/boosting-data-ingest-throughput-with-gpudirect-storage-and-rapids-cudf/, 2022.

[52] Qinlong Wang, Tingfeng Lan, Yinghao Tang, Bo Sang, Ziling Huang, Yiheng Du, Haitao Zhang, Jian Sha, Hui Lu, Yuanchun Zhou, et al. Dlrover-rm: Resource optimization for deep recommendation models training in the cloud. *Proceedings of the VLDB Endowment*, 17(12):4130–4144, 2024.

[53] Yuyue Wang, Xiurui Pan, Yuda An, Jie Zhang, and Glenn Reinman. Beacongnn: Large-scale gnn acceleration with out-of-order streaming in-storage computing. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 330–344. IEEE, 2024.

[54] NVM Express Workgroup. Nvm express base specification revision 2.0c. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified.pdf, 2022.

[55] Kun Wu, Jeongmin Brian Park, Xiaofan Zhang, Mert Hidayetoğlu, Vikram Sharma Mailthody, Sitao Huang, Steven Sam Lumetta, and Wen-mei Hwu. Tba: Faster large language model training using ssd-based activation offloading. *arXiv preprint arXiv:2408.10013*, 2024.

[56] Chunhua Xiao, Shi Qiu, and Dandan Xu. Pasm: Parallelism aware space management strategy for hybrid ssd towards in-storage dnn training acceleration. *Journal of Systems Architecture*, 128:102565, 2022.

[57] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[58] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[59] Yiming Zhang, Li Wang, Shengyun Liu, Shun Gai, Haonan Wang, Xin Yao, Meiling Wang, Kai Chen, Dongsheng Li, and Jiwu Shu. Cheetah: Metadata aggregation for fast object storage without distributed ordering. In *ACM EuroSys 2025*, 2025.

[60] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. Madfs:per-file virtualization for userspace persistent memory filesystems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 265–280, 2023.

[61] Tianle Zhong, Jiechen Zhao, Xindi Guo, Qiang Su, and Geoffrey Fox. Optimizing data i/o for llm datasets on remote storage. 2024.

[62] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. Xrp:in-kernel storage functions with ebpf. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.