Phoenix: A Refactored I/O Stack for GPU Direct Storage without Phony Buffers

Jianqin Yan*
Xiamen University
Xiamen, China
jqyan@stu.xmu.edu.cn

Shi Qiu*
Xiamen University
Xiamen, China
qiushijsxs@stu.xmu.edu.cn

Yina Lv
Xiamen University
Xiamen, China
elainelv95@gmail.com

Yifan Hu
Xiamen University
Xiamen, China
yifanh@stu.xmu.edu.cn

Hao Chen
Xiamen University
Xiamen, China
chenhao2@stu.xmu.edu.cn

Zhirong Shen
Xiamen University
Xiamen, China
shenzr@xmu.edu.cn

Xin Yao
Huawei Theory Lab
Hong Kong, China
yao.xin1@huawei.com

Renhai Chen
Huawei Theory Lab
Hong Kong, China
chenrenhai@huawei.com

Jiwu Shu
Xiamen University
Xiamen, China
jwshu@xmu.edu.cn

Gong Zhang
Huawei Theory Lab
Shenzhen, China
nicholas.zhang@huawei.com

Yiming Zhang†
Xiamen University
Xiamen, China
Shanghai Jiao Tong University
Shanghai, China
sdiris@gmail.com

Abstract

GPU Direct Storage (GDS) plays a vital role in GPU-based training and inference systems, leveraging Peer-to-Peer Direct Memory Access (P2P-DMA) to establish a direct data transfer path between the GPU and the storage device. The direct I/O path reduces GPU storage access latency and CPU overhead, thus improving the efficiency of data transfer. Currently, however, GDS employs a *phony buffer* in the host memory to interact with the Linux kernel, which results in suboptimal I/O performance, extra resource consumption, and high deployment complexity.

In this paper, we propose *Phoenix*, a refactored I/O stack for GDS without phony buffers. Phoenix employs the memory mapping service of ZONE_DEVICE, a new feature supported by Linux kernel since version 4.3, to map GPU memory into the page table at system startup. The kernel module of Phoenix stores the returned address information, allocates user-space virtual memory, and establishes a mapping with the designated GPU memory. We have implemented Phoenix for recently-released models of GPUs and NPUs. Evaluation shows that compared to the state-of-the-art GDS I/O stack, Phoenix reduces the average software processing overhead along the critical I/O path by 70.3%, and improves the small-size I/O performance (e.g., for KV-cache) and large file loading performance (e.g., for checkpoints) by 2.29× and 4.11×, respectively.

CCS Concepts

• **Computer systems organization → Parallel computing**; **Storage**.

Keywords

GPU Direct Storage (GDS), P2P-DMA, phony buffers, I/O stack, GPU memory

ACM Reference Format:
Jianqin Yan, Shi Qiu, Yina Lv, Yifan Hu, Hao Chen, Zhirong Shen, Xin Yao, Renhai Chen, Jiwu Shu, Gong Zhang, and Yiming Zhang. 2025. Phoenix: A Refactored I/O Stack for GPU Direct Storage without Phony Buffers. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3712285.3759862

1 Introduction

GPU-accelerated applications, such as large language models (LLMs) [10, 25, 59, 65], are experiencing growing demand for storage capacity due to the continuous increase in model and data sizes. Consequently, they typically involve accessing numerous files that can reach up to tens of terabytes (TB) in size and are expected to continuously grow. Data transfer between the GPU and storage devices

*Jianqin Yan and Shi Qiu are co-primary authors.
†Yiming Zhang is the corresponding author.

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/2025/11
https://doi.org/10.1145/3712285.3759862

must be sufficiently fast to prevent storage access from becoming a bottleneck slowing down GPU computation [48, 49].

Traditionally, GPU storage access [8, 19, 37, 50], such as the DataLoader of PyTorch [50], requires a host bounce buffer. Data is first read from the storage device to the bounce buffer using the host file system interface, and then copied to the GPU memory by invoking the device driver. Unfortunately, the extra memory copies (to/from the bounce buffer) prolong the data path and severely lower the I/O performance.

To bypass the bounce buffer, NVIDIA has proposed *GPU Direct Storage* (GDS) [33] that enables a direct data transfer path between GPU memory and NVMe devices. Currently, NVIDIA GDS is the only practical solution for direct data transfer between GPUs and storage devices. GDS is based on Peer-to-Peer Direct Memory Access (P2P-DMA) [1, 22, 31], a feature of PCIe. Unlike traditional DMA methods, the addresses of P2P-DMA point to the memory of a peripheral device. Data transfer between PCIe devices is orchestrated by the PCIe switch or the PCIe root complex, eliminating the need of using the inefficient host bounce buffer.

The direct I/O path significantly reduces storage access latency and CPU overhead, thereby improving end-to-end data transfer efficiency. GDS is of particular importance for GPU-accelerated applications that involve heavy I/O workloads such as KV-cache offloading and model checkpoint loading/saving, which impose stringent performance requirements on both I/O bandwidth and latency [14–16, 28, 55]. As a result, GDS has been extensively integrated into various distributed file systems [18, 32, 56, 58, 61, 63] and LLM serving systems [51].

However, the current design of GDS I/O stack is inefficient in realizing direct data transfer, preventing it from fully harnessing the advantage of P2P-DMA. Specifically, during P2P-DMA I/O, the kernel needs to take a reference to each relevant *struct page*, in order to ensure that these pages will not be freed [21]. To achieve this for GPU memory pages, GDS uses a *phony buffer* [7] in the host memory to represent the GPU buffer in interactions with the kernel. When the NVMe device driver receives an I/O request, it will replace the DMA address in the phony buffer with that in the GPU buffer to realize P2P-DMA.

The problems caused by the phony buffer mainly include:

- **Suboptimal performance**: To adopt the phony buffer, GDS has to incorporate complex and time-consuming logic. Furthermore, the overhead associated with allocating and releasing phony buffers is non-negligible, increasing the overhead on the critical I/O path of GDS and consequently leading to a degradation in the overall I/O performance.
- **Excessive resource consumption**: The phony buffer has the same size as the GPU buffer, leading to extra host memory consumption. In addition, the logic for I/O efficiency (such as asynchronous I/O and batch I/O) and for phony buffer management and state maintenance, impose substantial CPU cycle costs.
- **Poor compatibility**: To achieve DMA address replacement, GDS requires customized device drivers. Furthermore, the reference count of the phony buffer must be incremented to prevent accidental release during the I/O process. Consequently, I/O requests can only be submitted through the non-POSIX interface provided by GDS, thus increasing the complexity of programming and the difficulty of implementing asynchronous I/O.

To avoid the problems caused by phony buffers, in this paper we refactor GDS and propose *Phoenix*, a novel GDS I/O stack that realizes GPU direct storage without phony buffers. Phoenix employs the memory mapping service provided by ZONE_DEVICE, a new feature supported by Linux kernel since version 4.3 [26], to map GPU memory directly into the page table at system startup. The kernel module of Phoenix stores the returned address information, allocates user-space virtual memory, and establishes a mapping with the designated GPU memory. Therefore, the application can perform P2P-DMA I/O directly on GPU memory by invoking the standard file system interface, using the CPU virtual address as the interface parameter.

This paper makes the following contributions.

- We have refactored the GDS I/O stack, allowing the kernel storage software stack to directly access GPU memory via the page table. This eliminates the need for phony buffers, thus improving the compatibility of GDS I/O with the existing Linux kernel software stack and simplifying the architecture of GDS.
- We have redesigned the mapping process for GPU memory, realizing a lightweight kernel module (called *phoenix.ko*) for memory registration and deregistration. This reduces the software processing overhead along the critical path of GDS I/O, thus enhancing overall I/O performance.
- We offer a user-friendly library (called *libPhoenix*) to lower the complexity of using GDS, which utilizes the existing infrastructure of the kernel software stack. We have also integrated io_uring [20] to provide high-performance batch I/O and asynchronous I/O.

We have implemented Phoenix for both recently-released models of NVIDIA GPUs and an upcoming model of NPUs from an anonymous company. Extensive evaluation shows that Phoenix significantly outperforms the state-of-the-art GDS I/O stack. Phoenix achieves an average reduction of 70.3% in software processing latency along the critical I/O path. Its small-size I/O performance (e.g., for KV-cache) is 2.23× that of GDS, and the large file I/O performance (e.g., for checkpoints) is 4.11× that of GDS.

2 Background and Motivation

2.1 ZONE_DEVICE

To facilitate effective management of system physical memory, the Linux kernel traditionally partitions physical memory into multiple zones, such as *ZONE_DMA* and *ZONE_NORMAL*, to better accommodate differences in memory access patterns and usage requirements. However, these zones exclusively handle conventional system RAM, lacking the flexibility needed for effective device memory management. To address this, since Linux 4.3, the kernel introduced *ZONE_DEVICE*, a dedicated memory zone designed specifically to provide unified and flexible management of device memory.

Specifically, the facility of ZONE_DEVICE builds upon *SPARSEMEM* model to associate *struct page* structures with device-specific physical address ranges. This approach integrates device memory seamlessly into the unified memory management framework of the kernel, allowing user-space applications direct memory access

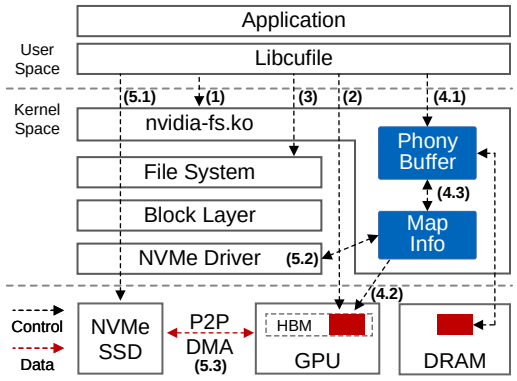


Figure 1: NVIDIA GDS architecture and I/O flow.

Operation	Latency (μs)
(1) cuFileDriverOpen	107,961.7
(2) cudaMalloc	3.2
(3) cuFileHandleRegister	307.7
(4) cuFileBufRegister	73.8
(5) cuFileRead	28.4
(6) cuFileBufDeregister	60.5
(7) cuFileHandleDeregister	0.9
(8) cudaFree	3.4
(9) cuFileDriverClose	2,033,194.3

Table 1: The workflow of a synchronous GDS operation with the corresponding latency. The GDS software stack requires six additional steps(1)(3)(4)(6)(7)(9) to setup platform and contributes to 82.5% of the total latency on the I/O critical path(4)(5)(6).

through standard interfaces such as *mmap*. As a result, it eliminates intermediate bounce buffers and redundant data copies, substantially reducing access overhead and enabling efficient data transfers such as direct I/O operations to persistent memory and P2P-DMA transactions. Therefore, we leverage the feature of ZONE_DEVICE to provide foundational GPU memory mapping services for Phoenix, addressing the struct page requirement for P2P-DMA I/O operations.

2.2 GPU Direct Storage

GDS enables a direct path between GPU memory and local or remote storage devices, such as NVMe or NIC for data transfers. Bypassing the traditional data path and the CPU when moving data can significantly reduce latency and CPU overhead in data-intensive applications.

Fig. 1 illustrates the architecture of NVIDIA GDS, which consists of three parts: a user-space library named *libcufile* [38], a kernel module named *nvidia-fs* and a customized NVMe driver. libcufile provides a set of APIs that enable applications and frameworks to leverage the capabilities of GDS. It supports asynchronous and batched I/O operations to improve data transfer efficiency, and internally manages interactions with the nvidia-fs, shielding applications from the complexity of kernel-level operations. nvidia-fs is the core of the GDS software stack, responsible for maintaining memory mappings and coordinating I/O requests issued by the user-space library. The customized device drivers integrate GPU DMA address translation logic and, through interfaces provided by nvidia-fs, replace the phony buffer with the corresponding GPU DMA address during I/O submission.

Specifically, as shown in Fig. 1, a typical GDS I/O workflow proceeds as follows:

(1) At the beginning, *cuFileDriverOpen* is invoked to initialize the custom driver, which opens the session with the GDS kernel driver to enable communication between userspace and the kernel and sets up the resources required to support GDS I/O operations. It will also check whether the current platform supports GDS and initialize the cuFile library.

(2) Using *cudaMalloc* to allocate GPU memory.

(3) *cuFileHandleRegister* is invoked to register the file descriptor to evaluate the suitability of the file state for GDS.

(4) Registering the specified GPU address with *cuFileBufRegister*. The interface (4.1) allocates a corresponding phony buffer of the same size in the host memory, (4.2) pins the pages of the GPU buffer using the device driver, and (4.3) establishes a mapping between them.

(5) The application (5.1) issues I/O operations using *cuFileRead* and *cuFileWrite*. The customized NVMe driver intercepts the I/O operations, (5.2) replaces the phony buffer address with the GPU memory address, and (5.3) performs DMA transfer between the GPU memory and the storage device.

(6) After completing the I/O operations, the registered GPU buffers are released with *cuFileBufDeregister*, which unmaps the buffer, puts the pages, and frees the phony buffer.

Finally, the file descriptor, GPU buffer, and GDS session state will be released by invoking (7) *cuFileHandleDeregister*, (8) *cudaFree*, and (9) *cuFileDriverClose*, respectively.

Due to the existence of phony buffers, GDS has to adopt customized device drivers. To facilitate compatibility checks and ensure kernel security, GDS has incorporated a substantial amount of extra complex logic (1)(3)(7)(9).

Moreover, allocating a phony buffer not only requires additional host memory space, but the kernel's allocation and deallocation of memory is also time-consuming, leading to a non-negligible overhead during GPU memory mapping (4)(6), and consequently increase I/O latency.

To prevent kernel errors caused by the release of the phony buffer during the I/O process, the GDS kernel is required to increase the reference number of phony buffer when submitting an I/O request and decrease it upon completion of the I/O. Consequently, user-space applications are forced to use non-POSIX interfaces (5), resulting in poor compatibility.

2.3 Analysis

To better understand the impact of phony buffers on I/O performance, we have conducted an experiment to evaluate the latency of each operation in the GDS workflow. We use an Intel Optane P5800X [13] as the underlying storage device, which achieves 25μs read latency for 64KB operations.

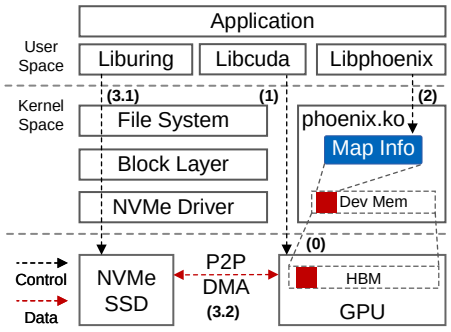


Figure 2: The overview of the Phoenix architecture.

Table 1 shows the latency of each operation in the GDS workflow. The result shows that the overhead of the GDS software stack is significant. For a 64KB synchronous read operation, GDS requires six additional steps(1)(3)(4)(6)(7)(9) to complete a single I/O request. We categorize the above time-consuming operations into three parts: driver management (1) and (9), file management (3) and (7), and buffer management (4) and (6).

These three management operations are critical for ensuring the correct functioning of GDS but introduce additional overhead. For driver and file management operations, they are typically performed only once at the start and end of the application, and not on the I/O critical path. However, for scenarios where the environment needs to be frequently initialized, such as in serverless computing and RPC, these driver and file management operations can degrade the system's end-to-end performance. For buffer management operations, particularly in buffer registration, they must be performed for every GPU buffer used in I/O operations. These operations introduce unavoidable overhead on the critical path, even surpassing the time taken by the actual I/O operations. From table 1, we can find that the latency of the GDS software stack accounts for 82.5% of the total latency on the I/O critical path when performing 64KB read operations. In the following, we focus on simplifying this GDS software stack to improve performance.

3 Design and Implementation

The analysis in Section 2.3 reveals that phony buffers have hindered GDS from achieving high performance and CPU utilization. To fully utilize the advantages of P2P-DMA to directly transfer data between storage devices and GPU memory, this paper designs a novel GDS software stack (called Phoenix) with the characteristics of high compatibility, high performance, and low resource consumption. One key challenge is how to establish the mapping relationship between the user-space virtual address and the GPU memory address. Since Linux 4.3 [26], ZONE_DEVICE is tied into the memory allocator's zone system, providing *struct page* services for driver-recognized device memory, such as persistent memory and GPU memory. Therefore, Phoenix can leverage ZONE_DEVICE to eliminate the need for phony buffers and simplify the deployment of GDS.

Fig. 2 illustrates the architecture of Phoenix, which consists of two parts: a kernel module named *phoenix.ko* (3.1) that provides the capability to manage the GPU buffer and a programming mode named *libPhoenix* (3.2) that simplifies the interaction between applications and *phoenix.ko*.

```
int phxfs_open(int device_id);
int phxfs_close(int device_id);
int phxfs_regmem(int device_id,
            const void *addr, size_t len,
            void **target_addr);
int phxfs_deregmem(int device_id,
            const void *addr, size_t len);
```

Listing 1: The interfaces of the libPhoenix.

3.1 Buffer Management

The phoenix.ko kernel module utilizes the struct page to abstract driver-recognized device memory, which allows the kernel storage software stack to directly access GPU memory through the page table instead of relying on phony buffers as proxies for GPU memory. Specifically, phoenix.ko consists of the following two parts: memory initialization and buffer registration.

3.1.1 Memory Initialization. Phoenix employs the memory mapping services of ZONE_DEVICE to register the physical memory of the GPU device into the page tables of the kernel space during the initialization of the kernel module. This enables GPU physical memory to be directly mapped through standard memory management interfaces (e.g., mmap).

Specifically, when loading the phoenix.ko, it enumerates all GPU devices in PCIe address order and assigns each device's index in this order as its device identifier (i.e., device ID). For each device, phoenix.ko invokes the kernel-provided *devm_memremap_pages* interface to establish the mapping of ZONE_DEVICE and specialize the remapped GPU memory as P2P-DMA memory. This operation creates *struct page* objects to allow peer devices in a PCI/-E topology to coordinate direct-DMA operations and then returns the corresponding virtual address of the remapped GPU memory. phoenix.ko stores the returned address information and maintains the corresponding metadata structures to support subsequent GPU buffer registration requests. Notably, this remapping method does not interfere with the original access to GPU memory by device drivers, as it simply sets the memory attributes of the GPU's physical pages in advance and maps them into kernel space.

3.1.2 Buffer Registration. Similar to the GDS kernel module, the phoenix.ko kernel module interacts with user space through a character device driver to register GPU buffers. During initialization, phoenix.ko creates a character device for each GPU and assigns an ID corresponding to the GPU index. It also provides driver support for these character devices, allowing applications to interact with the GDS software stack through POSIX interfaces. Specifically, the driver provides the following interfaces:

Open verifies whether the GPU memory has been successfully remapped and updates the *private_data* field of the driver device file with the GPU metadata saved during initialization.

Mmap creates a virtual address for applications. Unlike conventional mmap, the address returned at this time is merely a segment

```
//  Using POSIX-interface to issue GDS I/O
//  file must be open with O_DIRECT flag,
//  i.e., fd = open(file_path, O_DIRECT);
//  device id is PCIe address order index of GPU
void phxfs_xfer(int fd, void *dev_ptr, int device_id,
                size_t nbytes, off_t offset){
    void *host_ptr;

    // initialize Phoenix driver
    phxfs_open(device_id);

    // register GPU Buffer with Phoenix
    phxfs_regmem(device_id, dev_ptr, nbytes, &host_ptr);

    // issue GDS I/O request using pread
    pread(fd, host_ptr, nbytes, offset);

    // deregister GPU buffer
    phxfs_deregmem(device_id, dev_ptr, nbytes);

    // close Phoenix driver
    phxfs_close(device_id);
}

// register a RDMA memory region with libPhoenix
void phxfs_init_ctx(){
    ... // initialize other resources
    phxfs_regmem(device_id, dev_ptr, size, &host_ptr);
    struct ibv_mr *mr = ibv_reg_mr(pd, host_ptr,
                                   size, access_flags);
}
```

Listing 2: Sample pseudo-code of using libPhoenix

of a virtual address in the process's address space, and it does not directly map to actual physical pages. Instead, it is later mapped to GPU physical pages. phoenix.ko first initializes this virtual address and sets its flag to *VM_MIXEDMAP*, indicating that the addresses map to non-standard pages, such as device memory or I/O memory. Subsequently, phoenix.ko creates a metadata structure describing the mapping information, including the starting segment and size of the segment as well as the associated GPU metadata. To ensure the security of the mappings, phoenix.ko uses a hash table to store these structures and maintains reference counts to guarantee the validity of the mappings. Finally, this structure is stored in the private data of the virtual memory area, allowing it to be used later when establishing the GPU buffer mapping.

Ioctl is primarily used in conjunction with mmap operations to implement GPU buffer registration and deregistration. When registering a GPU buffer, it first receives the GPU buffer information passed from user space, including the virtual address of the GPU buffer, as well as the address and size of the virtual memory area. Subsequently, it uses the metadata stored in the private data field to verify this information, ensuring the legality of the virtual address.

To prevent the GPU pages corresponding to GPU buffers from being released during DMA operations, phoenix.ko pins these pages by invoking driver-specific interfaces and gets the list of GPU physical addresses. The implementation of this interface depends on the device driver, such as the *nvidia_p2p_get_pages* interface provided

```
struct phxfs_data{
    int fd, op;
    void *buf; // host ptr for registed GPU buffer
    size_t nbytes;
    off_t offset;
    // pointer to the number of bytes that were read.
    size_t *bytes_done;
};

void CUDART_CB phxfs_callback(void *user_data){
    auto *data = static_cast<phxfs_data *>(user_data);
    data->bytes_done = 0;
    if (data->op == 0) { // read
        *data->bytes_done = pread(data->fd, data->buf,
                                  data->nbytes, data->offset);
    } else { // write
        *data->bytes_done = pwrite(data->fd, data->buf,
                                   data->nbytes, data->offset);
    }
    delete data;
}

cudaError_t phxfs_async(int fd, int op,
                        void* buf,
                        size_t nbytes, off_t offset,
                        size_t* bytes_done,
                        CUstream stream){
    struct phxfs_data *data = new phxfs_data {
        .fd = fd, .op = op, .buf = buf,
        .nbytes = nbytes, .offset = offset,
        .bytes_done = bytes_done,
    };
    return cudaLaunchHostFunc(stream, phxfs_callback, data);
}
```

Listing 3: The pseudo-code of Phoenix integration with CUDA stream.

for NVIDIA GPUs [39], and the developer can implement this interface to support different GPU devices. Finally, it inserts the pages corresponding to the GPU physical addresses into the virtual memory area, establishing the mapping between the virtual memory area and the GPU physical pages. For GPU buffer deregistration, it performs the reverse operation.

3.2 Programming Model

In the subsection, we introduce *libPhoenix*, a programming model for Phoenix, which simplifies the interaction between applications and phoenix.ko. As shown in Listing 1, libPhoenix only provides four interfaces for applications to support GDS I/O operations, where their names are self-explanatory, directly reflecting their respective functionalities.

3.2.1 Integrating with GPU Direct Storage. The *phxfs_xfer* function in Listing 2 presents a sample pseudo-code of issuing a GDS I/O request using libPhoenix, with the following steps:

(1) When an application initiates a GDS request, it first calls the *phxfs_open* interface to initialize the Phoenix driver. This interface opens the character device corresponding to the provided

device ID and initializes the associated metadata. The device ID refers to the index of a GPU within the PCIe address ordering. In most cases, this index same with the GPU ID reported by `nvidia-smi` [46].

(2) The application calls the *phxfs_regmem* interface, which utilizes the *mmap* and *ioctl* system calls to register the allocated GPU buffer with the GDS software stack through the corresponding character device. The virtual address mapped to the GPU physical memory is subsequently returned to the application through *target_addr*.
(3) The applications use this virtual address to issue GDS I/O requests through the POSIX interfaces, such as *pread/pwrite*.
(4) After the GDS request is completed, the application calls the *phxfs_deregmem* interface, which is the reverse of the GPU buffer registration process. This interface deregisters the GPU buffer from the GDS software stack and frees the associated resources.
(5) The application calls the *phxfs_close* interface to close the driver and release all mapped resources.

Compared to cuFile APIs, libPhoenix simplifies the interaction between the application and the GDS software stack, offering a more user-friendly programming interface with only four interfaces. All I/O operations are initiated through standard file system interfaces, enabling the application to deploy GDS with minimal modifications.

3.2.2 Integrating with GPU Direct RDMA. Similar to GDS, NVIDIA also provides GPU Direct RDMA (GDR) to support direct data transfers between the HBM/DRAM of different nodes over RDMA networks. To enable P2P-DMA transfers between GPU memory and RDMA-capable NICs, NVIDIA offers the *nvidia-peermem* kernel module, which collaborates with customized NIC drivers to pin the pages associated with GPU virtual addresses and expose their DMA addresses to the NIC. This memory registration process is conceptually similar to that of GDS, where the system must establish a mapping between GPU memory and the device-accessible address space to facilitate direct access.

Phoenix leverages the ZONE_DEVICE feature to associate struct page structures with GPU memory and provides a similar memory registration mechanism. Therefore, as shown in the *phxfs_init_ctx* function in Listing 2, applications can directly use GPU memory registered via phxfs_regmem to initiate RDMA memory region registration with the NIC driver. The NIC driver treats the memory transparently as conventional host memory, tries to pin the corresponding pages (already done during driver initialization), and retrieves their DMA addresses without requiring any additional kernel module support.

3.3 Integration with the CUDA Stream

CUDA Stream [34] enables concurrent execution within applications by allowing operations to be issued to independent execution queues. The CUDA runtime guarantees that operations within the same stream are executed sequentially in submission order. By associating I/O operations with a designated stream, non-blocking submission can be achieved, allowing precise coordination between data transfers and GPU kernel execution without the need for explicit synchronization.

System	Operation	Latency (µs)
GDS	cuFileDriverOpen	107,961.7
	cuFileDriverClose	2,033,194.3
Phoenix	phxfs_open	2.3
	phxfs_close	0.5

Table 2: The latency of driver management operations for GDS and Phoenix. Phoenix incurs negligible overhead for these operations.

However, using only POSIX-compatible interfaces decouples I/O operations from GPU kernel launches, preventing their scheduling within the same CUDA stream. As a result, operations with data dependencies must be explicitly synchronized to ensure correct execution. This leads to interruptions in the computational flow, thus increasing latency and reducing the overall efficiency of the system.

To overcome this limitation, Phoenix utilizes the cudaLaunchHostFunc [35] interface provided by CUDA, which allows host callback functions to be attached to a CUDA stream. This callback is executed after all preceding GPU operations in the stream have completed, and subsequent operations are issued only after the callback finishes. As demonstrated in the pseudo-code shown in Listing 3, Phoenix integrates with the CUDA stream. Specifically, when initiating asynchronous GDS I/O requests, Phoenix passes the I/O metadata as parameters to the CUDA callback function, which uses POSIX interfaces to issue the GDS I/O request and stores the result to the *bytes_done* pointer. It is noteworthy that, to ensure the integrity of the data within the stream, I/O operations must be entirely contained within the callback, including both initiation and result checking. Therefore, Phoenix directly initiates GDS I/O requests synchronously to avoid the performance degradation caused by asynchronous interfaces.

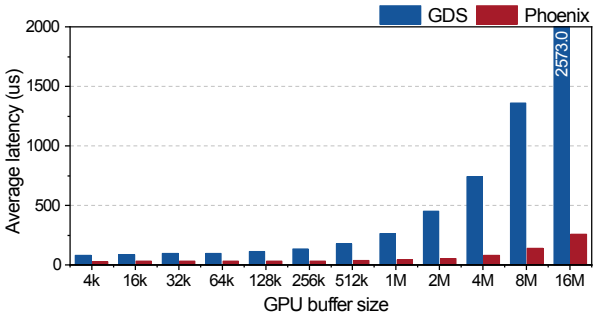
4 Evaluation

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

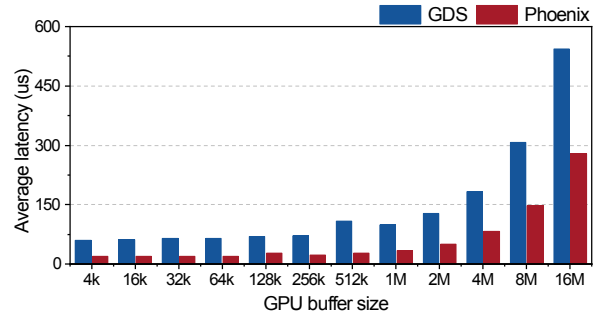
- What is the reduction in GDS software stack overhead achieved by Phoenix (§4.1)?
- What is the advantages of Phoenix in terms of I/O latency, throughput and end-to-end performance compared to GDS (§4.2)?
- Can Phoenix improve storage access performance in typical LLM workloads such as KV-cache and checkpoint loading (§4.3)?

System settings. All experiments are conducted on a 128-core Intel Xeon 6530 server equipped with 512 GB of memory, running Ubuntu 22.04 with Linux kernel 6.1.0. The server is equipped with a GPU with 48GB VRAM [44], utilizing CUDA 12.4, and nvidia-fs 2.19. It also has a Mellanox ConnectX-5 100G network card, which provides RDMA support for both NVMe-over-Fabrics (NVMe-oF) and NFS storage. The NVMe devices are Intel Optane P5800X [13], which serves as the underlying storage. All devices are co-located under the same PCIe switch to enable P2P-DMA transfers.

Baselines. We compare Phoenix with the existing NVIDIA GDS software stack. Following the example code provided in the official

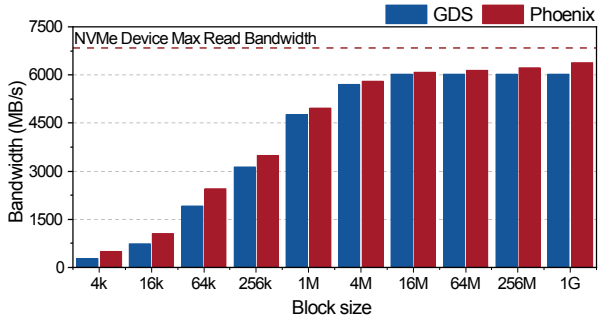


(a) Registration

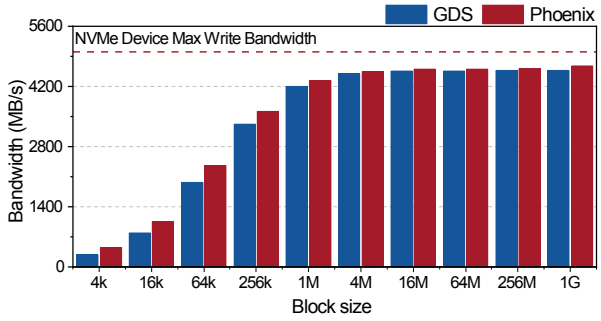


(b) Deregistration

Figure 3: The performance comparison of (a) registration and (b) deregistration operations with various GPU buffer sizes. Phoenix reduces the average software processing overhead along the critical I/O path by 70.3%.



(a) Read operations



(b) Write operations

Figure 4: Comparison of bandwidth of synchronous (a) read and (b) I/O operations across various I/O sizes. Phoenix exhibits similar performance improvements for both read and write operations.

repository [42], we implement the GDS baseline using the interfaces exposed by cuFile.

4.1 Breakdown of Operations

To evaluate the reduction in software overhead achieved by Phoenix, we repeat the experiments described in Section 2.3, collecting the operation latencies of both GDS and Phoenix by varying the GPU buffer sizes from 4KB to 16MB. Notably, GDS limits the maximum registered GPU memory size to 16MB to balance memory consumption and efficiency [41]. As a result, when the registered memory size is 16MB or larger, GDS registers a fixed 16MB phony buffer and internally partitions the I/O request into multiple 16MB segments, which are then submitted sequentially. This internal segmentation introduces additional overhead and results in degraded performance for large-granularity I/O operations, which will be demonstrated in Section 4.2.

Driver management. Table 2 shows the operation latencies of GDS and Phoenix for driver management, which are independent of GPU buffer size and are typically executed once. Compared to GDS, the latency of Phoenix for driver open and close operations is negligible. This is because the GDS driver needs to verify system compatibility during initialization and then register customized driver functions. Similarly, during close operations, the GDS driver performs cleanup processes, such as releasing allocated phony buffers and unregistering driver functions. Different from GDS, Phoenix avoids these steps entirely.

In addition to the overhead associated with driver management operations, we also evaluate the cost of invoking the ZONE_DEVICE memory mapping service during the initialization of the kernel module. Although the GPU provides only 48 GB of on-board VRAM, it exposes 64 GB of addressable memory over PCIe, including additional host-mapped memory regions [43]. Constructing the corresponding `struct page` for this address space requires approximately 150 ms.

Buffer management. Fig. 3a illustrates the latency of GPU buffer registration and deregistration operations. For registration, Phoenix achieves a latency reduction of (63% - 90%) under different buffer sizes compared to GDS, since it eliminates the need to allocate the phony buffer. For deregistration, Phoenix achieves 54%-75% reduction in latency by eliminating the need to release memory on the host.

4.2 Comparison with NVIDIA GDS

Phoenix allows using POSIX interfaces to submit I/O and using Linux *io_uring* to realize asynchronous and batch I/O. Thus, we compare the performance of Phoenix and GDS under different I/O modes, including synchronous I/O, asynchronous I/O, and batch I/O. In addition, for asynchronous I/Os, we further evaluate the

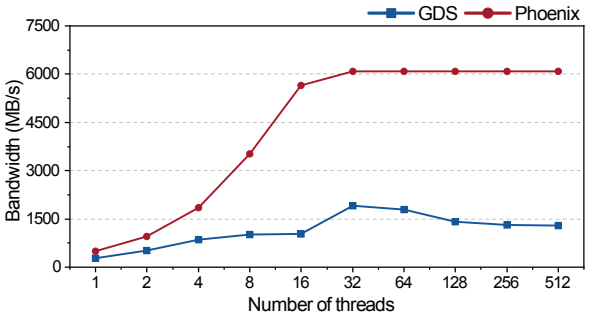


Figure 5: Comparison of 4K synchronous read I/O bandwidth at various levels of parallelism.

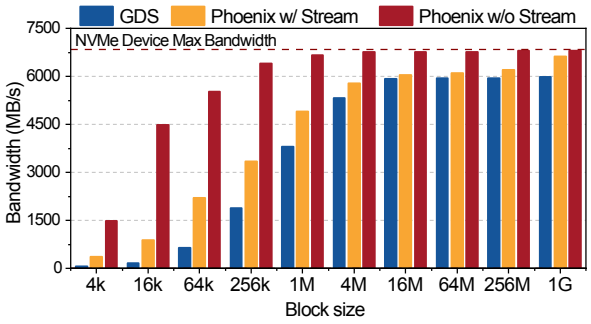


Figure 6: Comparison of bandwidth of batched asynchronous read I/O operations across various I/O sizes.

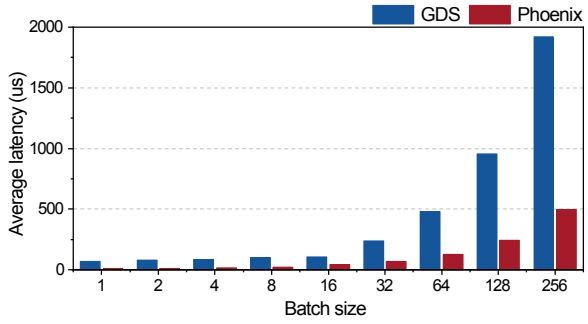


Figure 7: Comparison of read latency for different batch sizes.

performance under integration with CUDA streams. To eliminate the impact of GPU buffer registration, we pre-register the GPU buffers in all experiments unless otherwise specified. The total I/O size is configured to at least 40 GB.

4.2.1 Local NVMe Storage. **Synchronous I/O.** Fig. 4a illustrates the bandwidth comparison of synchronous read I/O operations across different I/O sizes. Compared to GDS, Phoenix achieves a 1% to 76% improvement in bandwidth. This improvement can be attributed to the elimination of overhead related to managing I/O states and querying DMA address mappings within the device driver. However, as the I/O size increases, the impact of this overhead on I/O operations diminishes, leading to similar performance between them.

Fig. 4b shows the bandwidth comparison of synchronous write I/O operations across different I/O sizes. Compared to GDS, Phoenix achieves up to a 65% improvement and exhibits a performance trend consistent with that of read operations. This is because GDS introduces similar software overhead for both reads and writes such as I/O management and DMA address mapping, and Phoenix mitigates this overhead. Therefore, in the subsequent experiments, we only focus on the performance of the read operations.

In particular, once the block size exceeds 16MB, the performance of GDS ceases to scale with I/O size. This limitation arises because GDS employs a fixed 16MB phony buffer to facilitate address mapping, segmenting the I/O request and submitting it sequentially.

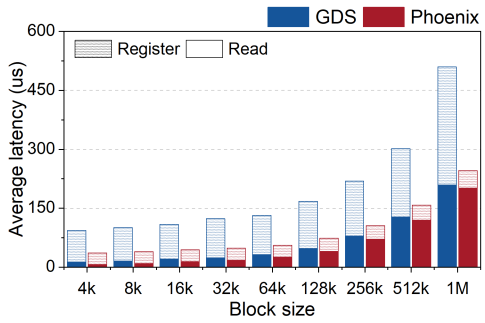
Fig. 5 illustrates the bandwidth comparison under 4KB I/O granularity as the number of concurrent threads increases. Compared to GDS, Phoenix demonstrates better scalability and achieves up to a 442% improvement in bandwidth. While Phoenix achieves saturation at 32 threads, GDS exhibits degraded performance when exceeding this threshold, primarily due to the overhead of internal metadata management and address translation within its software stack.

Asynchronous I/O. Fig. 6 illustrates the latency of asynchronous read operations at a batch size of 16 across different I/O granularities. We further evaluate the performance of Phoenix with and without integration with CUDA streams. Compared to GDS, Phoenix achieves a bandwidth improvement ranging from 2% to 467% when integrated with CUDA streams, with larger gains observed at smaller block sizes. This is because GDS relies on CUDA streams to implement asynchronous operations, which are ultimately executed synchronously by the GDS kernel module, introducing additional control overhead. These overheads become more pronounced under small I/O granularities, where the relative cost of control operations dominates overall performance.

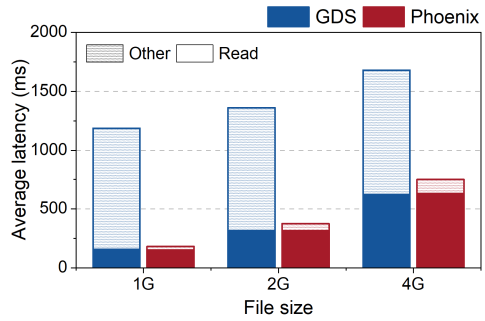
However, when compared to directly submitted asynchronous operations in Phoenix, integrating with CUDA streams results in a 3%–80% performance degradation, with the impact diminishing as the I/O size increases. This is because, to ensure data correctness within the stream, I/O operations must be fully executed inside the stream context. This requirement effectively serializes all operations, preventing the system from exploiting the benefits of asynchronous I/O.

Batch I/O. Fig. 7 shows the average latency of 4KB read operations under different batch sizes. Compared to GDS, Phoenix achieves up to a 76% reduction in latency. This is because the batch submission in GDS is implemented using a synchronous interface, where it sequentially submits I/O requests through the VFS interface in the kernel module and waits for each request to complete, significantly increasing I/O operation latency.

In addition, we found that GDS imposes strict constraints on batched operations in practice: within a single batch, the size of each individual I/O submission must not exceed 16MB, and the total size of multiple I/O requests targeting the same GPU buffer is limited to 1MB. These restrictions significantly increase the complexity of using the batch interface for GDS.



(a) Small-size I/O



(b) Large file loading

Figure 8: Comparison of end-to-end latency for (a) small-size I/O and (b) large file loading.

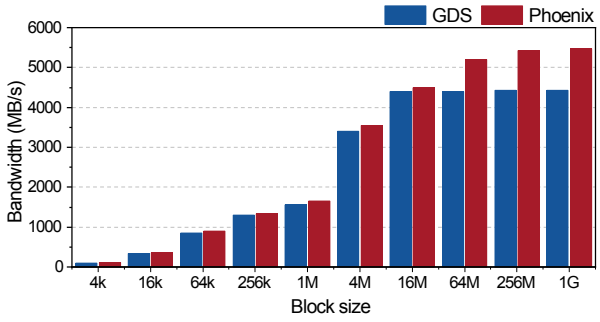


Figure 9: Comparison of bandwidth of synchronous read I/O operations across various I/O sizes on NFS storage.

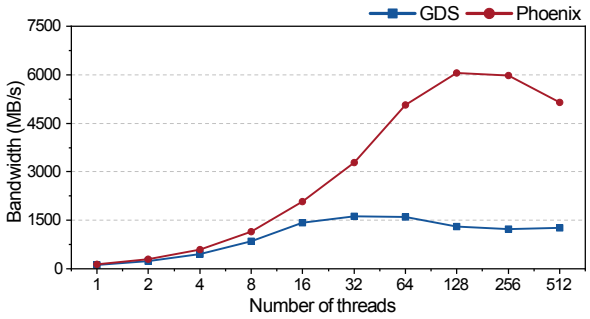


Figure 10: Comparison of 4K synchronous read I/O bandwidth under different levels of parallelism on NVMe-oF storage.

End-to-end performance. To quantify the overall impact introduced by the software stack, we evaluate the end-to-end performance of two representative GPU-to-storage access scenarios: one is small-size I/O, such as dataset and KV cache loading, where we measure the latency of GPU buffer registration and read I/O for each request; the other is large file loading, such as checkpoint loading, where we measure the end-to-end latency of each operation, including driver management, file management, and memory registration. In addition, we compare the proportion of overhead introduced by software beyond the actual data transfer.

Fig. 8(a) illustrates the performance of small I/O across different I/O sizes. Compared to GDS, Phoenix reduces end-to-end latency by 47.4% to 60.8% for small-size I/O, since it can reduce both memory registration overhead and I/O latency. Note that, under small-size I/O, the overhead of memory registration for GDS consistently exceeds the cost of data transfer, ranging from 1.3× to 5.71×. In contrast, for Phoenix, the cost of data transfer surpasses the overhead of memory registration when the I/O size reaches 128KB.

For large file loading, as shown in Fig. 8(b), Phoenix delivers an average performance improvement of 4.11× over GDS, while achieving comparable data transfer efficiency and reducing the overhead of non-transfer operations by 93.3% on average. This is attributed to the reduction in driver management, file management, and buffer management overhead, as demonstrated in the previous experiments.

4.2.2 Remote Network Storage. By eliminating the reliance on phony buffers for GDS, Phoenix also removes the need for customized driver support, enabling broader compatibility with existing storage systems. To validate this compatibility, we conduct experiments on NFS and NVMe-oF network storage.

NFS storage. Fig. 9 illustrates the bandwidth comparison between Phoenix and GDS on NFS network storage across different I/O sizes. Compared to GDS, Phoenix achieves a 2%–23% improvement in bandwidth by eliminating the need for additional DMA address mapping queries and I/O state management. Similar to synchronous I/O on local storage, GDS performance saturates when the I/O size exceeds 16MB, as it internally limits each request to a maximum of 16MB and submits them sequentially.

NVMe-oF storage. Fig. 10 shows the bandwidth comparison under 4KB I/O granularity at different levels of parallelism, using NVMe-oF as the storage backend. Phoenix at least a 10% bandwidth improvement over GDS and demonstrates better scalability as the number of threads increases. In particular, Phoenix continues to scale effectively up to 128 threads, while GDS saturates at 32 threads and shows limited performance improvement beyond that point.

4.3 Storage Performance in LLM

To evaluate the performance of storage in LLM inference, we consider two representative I/O patterns: offloaded KV cache loading

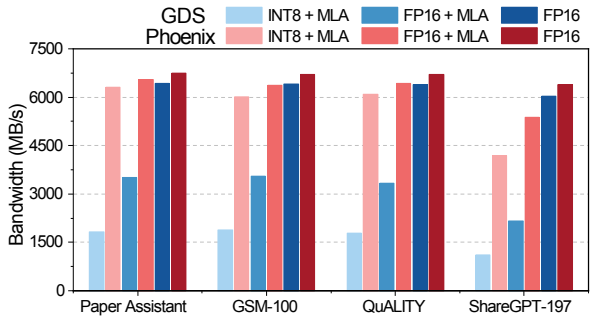


Figure 11: Comparison of KV cache loading bandwidth under different traces, quantization formats, and MLA optimization. The legend corresponds to KV cache block sizes of 8KB (INT8 + MLA), 16KB (FP16 + MLA), and 64KB (FP16).

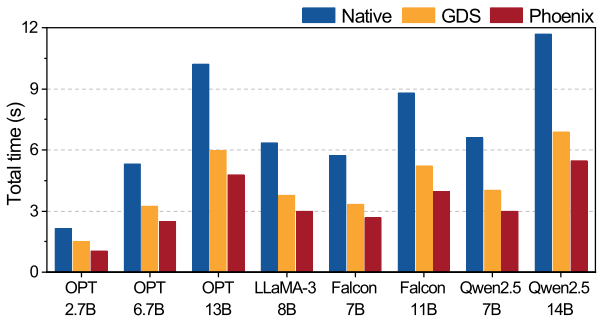


Figure 12: Comparison of model loading speeds across different models, where native refers to data transfer via mmap and cudaMemcpy.

and model loading. The former involves frequent, fine-grained reads, while the latter reflects large, sequential data transfers.

Offloaded KV Cache loading. LLM inference systems [51, 60] typically adopt PagedAttention [23] to optimize KV cache memory management. Specifically, the KV cache for all tokens is organized at the block granularity and scattered within a preallocated large contiguous memory region. Therefore, we follow this organization strategy and employ four different traces to obtain the KV cache management information from real conversations. Table 3 presents detailed statistics of these traces. To evaluate KV cache loading performance, we use a large file to simulate the offloaded KV cache, and access it with layer granularity. We further consider different model configurations, including quantization formats and the use of Multi-Head Latent Attention (MLA) [24] optimization, which lead to variations in KV cache size. For simplicity, We assume that MLA can reduce the KV cache size by 75% [24].

Trace	#Sequences	#Blocks
Paper Assistant [4]	23	15,708
GSM-100 [12]	100	33,600
QuALITY [47]	15	6,428
ShareGPT-197 [54]	197	20195

Table 3: Statistics of traces.

Fig. 11 presents the bandwidth of offloaded KV cache loading across four traces under various model configurations. The legend reflects the resulting KV cache sizes determined by different quantization and attention settings: 8KB (INT8 with MLA), 16KB (FP16 with MLA), and 64KB (FP16 without MLA). To compare the performance of Phoenix and GDS under fragmented and mixed-granularity access patterns, we measure only the data transfer overhead. We use the batch I/O interface to retrieve all blocks associated with a sequence in a single operation whenever possible, and merge physically contiguous blocks into a larger I/O request. Due to interface limitations in GDS (i.e., a maximum of 256 requests per batch submission), some sequences may require multiple batch submissions to complete the read operation. In contrast, Phoenix imposes no such restriction, as it eliminates the reliance on phony buffers and integrates io_uring to submit batch I/O operations.

Compared to GDS, Phoenix achieves bandwidth improvements of 246%, 101%, and 5% under three different configurations. As the KV cache size increases, the performance gains delivered by Phoenix progressively diminish. This is primarily attributed to the limited effectiveness of Phoenix in optimizing I/O transfer performance under large-granularity I/O scenarios, which aligns with the observations presented in Section 4.2.

Model loading. To perform the process of loading model parameters, we employ an open-source C++ library [57] for parsing the checkpoints in the safetensors format. The headers of the safetensors are first loaded into host memory, and each tensor is subsequently transferred to GPU memory based on the associated metadata. Along with GDS and Phoenix, we also evaluate a native approach based on mmap and cudaMemcpy [36] for comparison.

Fig. 12 shows the end-to-end performance of model checkpoint loading across different models and parameter sizes. For both GDS and Phoenix, we measure the overhead associated with data transfer and buffer management. Compared to the native approach, GDS and Phoenix reduce model loading latency by 39% and 54%, respectively. Additionally, Phoenix has achieved a 23% reduction in model checkpoint loading compared to GDS. This is because Phoenix reduces both buffer management and data transfer overhead, as demonstrated in the previous experiments.

5 Discussion

Limitations. GDS I/O is fundamentally a form of direct I/O. Consequently, although GDS offers substantial improvements in data transfer efficiency, both NVIDIA GDS and Phoenix remain constrained by the fundamental limitations inherent to direct I/O. Specifically, GDS I/O operations require that the GPU buffer address, I/O size, and file offset be strictly aligned to a specific granularity (e.g., 4KB); otherwise, the I/O request will fail.

Under these constraints, applications must explicitly ensure that memory allocation and access patterns satisfy the required alignment, which may introduce additional memory overhead and increase programming complexity. For instance, when the data size is not aligned to the GPU page granularity (e.g., 64KB), applications

must still required to allocate memory in page-aligned units, resulting in suboptimal utilization of GPU memory. Moreover, the padding or data copying required to meet alignment constraints may introduce additional memory transfers, thereby degrading overall I/O efficiency.

In addition, Phoenix establishes the mapping between the virtual address of the application and GPU remapped pages via the *mmap* system call. However, this mapping process is constrained by a limitation of *mmap*, which restricts each invocation to mapping at most 1GB of memory (i.e., when using huge page). To register GPU buffers exceeding this size, the application must explicitly manage multiple mapping segments, typically by providing a point array to the *phxfs_regmem* interface to store the returned addresses. In contrast, as discussed in Section 4.1, NVIDIA GDS imposes a 16MB limit on each buffer registration, which means that it does not suffer from this limitation.

Unified P2P access. NVIDIA Magnum IO [42] is an I/O subsystem designed for GPU-accelerated data centers, integrating technologies such as GPU Direct Storage (GDS) and GPU Direct RDMA (GDR). These technologies typically require the installation of additional kernel modules, such as nvidia-fs for GDS and nvidia-peermem [45] for GDR. In contrast, Phoenix offers a more unified GPU peer-to-peer access mechanism.

With Phoenix, users can register a memory region once and utilize it for both GDS data access and GDR memory region registration, eliminating redundant module dependencies. More aggressively, users can even use memory semantics to directly access the Phoenix-registered memory region, enabling direct data transfers between the host memory and GPU memory (potentially leading to consistency issues).

6 Related Work

6.1 Peer-to-Peer Direct Storage Access

P2P-DMA transfers are widely employed in the GPU storage stack[5, 7, 9, 11, 16, 17, 29, 30, 48, 53, 64] to bypass host memory and improve the efficiency of data transfer between storage systems and GPUs. SPIN [7] is the first work that uses phony buffers as a substitute for GPU buffers to interact with the direct disk I/O interface. Although it provides seamless OS integration, it faces the limitations associated with using phony buffers, as discussed in Section 1.

BaM [53] is a more radical P2P-DMA approach that maps NVMe device queues directly into GPU memory to facilitate the transmission of control plane (NVMe commands) through P2P-DMA, empowering the GPU to fully oversee the NVMe device and entirely bypass the CPU. While this approach excels in I/O performance, its SPDK-like [62] method abandons the file system abstraction, making it difficult to meet the high parallelism and data sharing requirements of typical training and inference scenarios.

To address this limitation, GeminiFS [52] introduces a companion file system abstraction to GPU programs by extending the NVMe driver to support shared control planes between the CPU and GPU. By embedding file system metadata directly into the header of files, GeminiFS enables efficient metadata synchronization between the host and GPU file systems. However, compared to Phoenix or GDS, the access scope of both BaM and GeminiFS is limited to local NVMe devices and cannot be extended to network storage devices.

In addition, FlashNeuron [6] and hcache [16] present an alternative approach to realize P2P-DMA between GPUs and storage devices: they employ GDRCopy [40] to expose GPU memory and retrieve its DMA address, and then use SPDK's user-space block I/O interface to initiate P2P data transfers between storage and GPU memory. However, this SPDK-based approach also lacks file system semantics, facing the same limitations as BaM.

Compared to SPIN or GDS, which utilize phony buffers, Phoenix offers a simplified GPU storage stack, higher I/O performance, and better compatibility with the Linux kernel. Currently, these GPU-centric storage solutions only provide superior performance in very limited scenarios, whereas Phoenix surpasses these solutions in terms of storage access range and ease of use.

6.2 Direct Data Transfer of GPUs

Currently, NVIDIA GDS is the only practical solution for direct data transfers between GPUs and storage devices, while other similar-purpose techniques have limitations. AMD's DirectGMA [2] maps GPU memory into PCIe BAR space, allowing direct data transfers between PCI device memories. However, it does not support GPU direct storage, as COTS (commercial off-the-shelf) storage devices cannot initiate or coordinate such transfers. In addition, AMD also provides the peer memory client [3] to enable direct data transfers between GPUs and RDMA-capable NICs, but it similarly offers no support for GPU access to storage devices. Microsoft has introduced the DirectStorage [27] API to accelerate loading game files by allowing the GPU to read data directly from NVMe SSDs. However, it only supports one-way transfers from storage to the GPU.

7 Conclusion

This paper presents a novel GDS software stack (called Phoenix), which eliminates the adoption of phony buffers to achieve high performance. Phoenix leverages the ZONE_DEVICE feature supported by Linux kernel since version 4.3 to map GPU memory directly into the page table during system initialization. Furthermore, Phoenix redesigns the mapping process, which not only reduces software overhead along the critical I/O path, but also minimizes resource consumption, thereby improving overall efficiency. Experiments have demonstrated that, in scenarios involving small granularity I/O, multi-threaded I/O, end-to-end testing, and real workloads for LLM, the Phoenix exhibits significantly higher storage performance compared to the current GDS. Additionally, it incurs lower software stack overhead and offers better compatibility.

In our future work, we will integrate Phoenix into existing frameworks, such as PyTorch and Tensorflow, to optimize real-world GPU applications, including large language models and graph neural networks. We have open-sourced Phoenix at https://github.com/nicexlab/phoenix.

Acknowledgments

We thank 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] AMD. 2022. Xilinx Runtime library (XRT). https://xilinx.github.io/XRT/master/html/p2p.html.

[2] AMD. 2025. DirectGMA. https://www.bitflow.com/technology/directgma/.

[3] AMD. 2025. RoCm. https://github.com/RoCm/ROCK-Kernel-Driver/blob/master/drivers/gpu/drm/amd/amdkfd/kfd_peerdirect.c.

[4] Chenxin An, Shansan Gong, Ming Zhong, Xingjian Zhao, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. 2023. L-eval: Instituting standardized evaluation for long context language models. *arXiv preprint arXiv:2307.11088* (2023), 26 pages.

[5] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2021. FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 387–401. https://www.usenix.org/conference/fast21/presentation/bae

[6] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2021. FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 387–401. https://www.usenix.org/conference/fast21/presentation/bae

[7] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. 2017. {SPIN}: Seamless Operating System Integration of {Peer-to-Peer} {DMA} Between {SSDs} and {GPUs}. 167–179. https://www.usenix.org/conference/atc17/technical-sessions/presentation/bergman

[8] BlazingSQL. 2025. BlazingSQL: A GPU-accelerated SQL engine for data science on rapids.ai. https://github.com/BlazingDB/blazingsql.

[9] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. 2019. {GAIA}: An {OS} page cache for heterogeneous systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 661–674.

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

[11] Chia-Hao Chang, Jihoon Han, Anand Sivasubramaniam, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-Mei Hwu. 2024. 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*. 464–478.

[12] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

[13] Intel Corporation. 2020. Intel Optane DC SSD Series 400 GB. https://www.intel.com/content/www/us/en/products/sku/201860/intel-optane-ssd-dc-p5800x-series-800gb-2-5in-pcie-x4-3d-xpoint/specifications.html.

[14] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. {Check-N-Run}: A checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 929–943.

[15] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.

[16] Shiwei Gao, Youmin Chen, and Jiwu Shu. 2025. Fast state restoration in LLM serving with hcache. In *Proceedings of the Twentieth European Conference on Computer Systems*. 128–143.

[17] Yuchen Huang, Xiaopeng Fan, Song Yan, and Chuliang Weng. 2024. Neos: A NVMe-GPUs Direct Vector Service Buffer in User Space. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3767–3781. https://doi.org/10.1109/ICDE60146.2024.00289

[18] IBM. 2024. IBM Storage Scale. https://www.ibm.com/docs/en/storage-scale/5.2.1?topic=installing-gpudirect-storage-storage-scale.

[19] Keras. 2025. Keras. https://keras.io/.

[20] The Linux Kernel. 2025. io-uring. https://man7.org/linux/man-pages/man7/io_uring.7.html.

[21] The Linux Kernel. 2025. Linux Filesystems API summary. https://www.kernel.org/doc/html/latest/filesystems/api-summary.html#c.bio_iov_iter_get_pages.

[22] The Linux Kernel. 2025. PCI Peer-to-Peer DMA support. https://docs.kernel.org/driver-api/pci/p2pdma.html.

[23] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[24] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434* (2024).

[25] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. 2023. Deja Vu: Contextual Sparsity for Efficient LLMs at Inference Time. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 22137–22176. https://proceedings.mlr.press/v202/liu23am.html

[26] The Linux Kernel Memory. 2025. Physical Memory Model. https://docs.kernel.org/mm/memory-model.html.

[27] Microsoft. 2025. DirectStorage. https://github.com/microsoft/DirectStorage.

[28] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. {CheckFreq}: Frequent,{Fine-Grained}{DNN} Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 203–216.

[29] Harini Muthukrishnan, Daniel Lustig, Oreste Villa, Thomas Wenisch, and David Nellans. 2023. FinePack: Transparently Improving the Efficiency of Fine-Grained Transfers in Multi-GPU Systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 516–529. https://doi.org/10.1109/HPCA56546.2023.10070949

[30] Harini Muthukrishnan, David Nellans, Daniel Lustig, Jeffrey A. Fessler, and Thomas F. Wenisch. 2021. Efficient Multi-GPU Shared Memory via Automatic Optimization of Fine-Grained Transfers. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 139–152. https://doi.org/10.1109/ISCA52012.2021.00020

[31] Ryo Nakamura, Yohei Kuga, and Kunio Akashi. 2020. How beneficial is peer-to-peer DMA?. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 25–32.

[32] NetApp. 2024. NetApp ONTAP. https://www.netapp.com/data-storage/ontap/.

[33] NVIDIA. 2024. nvidia-fs. https://github.com/NVIDIA/gds-nvidia-fs.

[34] NVIDIA. 2025. CUDA Stream. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html.

[35] NVIDIA. 2025. cudaLaunchHostFunc. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EXECUTION.html#group__CUDART__EXECUTION_1g05841eaa5f90f27124241baafb3e856f.

[36] NVIDIA. 2025. cudaMemcpy. https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/html/group__CUDART__MEMORY_g48efa06b81cc031b2aa6fdc2e9930741.html.

[37] NVIDIA. 2025. cuDF - GPU DataFrames. https://github.com/rapidsai/cudf.

[38] NVIDIA. 2025. cuFile API Reference Guide. https://docs.nvidia.com/gpudirect-storage/api-reference-guide/index.html.

[39] NVIDIA. 2025. Developing a Linux Kernel Module using GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html#kernel-api.

[40] NVIDIA. 2025. GDRCopy. https://github.com/NVIDIA/gdrcopy.

[41] NVIDIA. 2025. GPUDirect Storage Configuration Parameters. https://docs.nvidia.com/gpudirect-storage/configuration-guide/index.html#gds-configuration-parameters.

[42] NVIDIA. 2025. Magnum IO. https://github.com/NVIDIA/MagnumIO.

[43] NVIDIA. 2025. NVIDIA GPU PCI Bar Size. https://docs.nvidia.com/cuda/gpudirect-rdma/#pci-bar-sizes.

[44] NVIDIA. 2025. NVIDIA L40S. https://www.nvidia.com/en-sg/data-center/l40s/.

[45] NVIDIA. 2025. nvidia-peermem. https://github.com/Mellanox/nv_peer_memory.

[46] NVIDIA. 2025. System Management Interface SMI. https://developer.nvidia.com/system-management-interface.

[47] Richard Yuanzhe Pang, Alicia Parrish, Nitish Joshi, Nikita Nangia, Jason Phang, Angelica Chen, Vishakh Padmakumar, Johnny Ma, Jana Thompson, He He, et al. 2021. QuALITY: Question answering with long input texts, yes! *arXiv preprint arXiv:2112.08608* (2021).

[48] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. 2024. Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses. *Proc. VLDB Endow.* 17, 6 (May 2024), 1227–1240. https://doi.org/10.14778/3648160.3648166

[49] Jeongmin Brian Park, Kun Wu, Vikram Sharma Mailthody, Zaid Qureshi, Scott Mahlke, and Wen-mei Hwu. 2024. LSM-GNN: Large-scale Storage-based Multi-GPU GNN Training by Optimizing Data Transfer Scheme. *arXiv preprint arXiv:2407.15264* (2024).

[50] Pytorch. 2025. PyTorch documentation. https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader.

[51] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/conference/fast25/presentation/qin

[52] Shi Qiu, Weinan Liu, Yifan Hu, Jianqin Yan, Zhirong Shen, Xin Yao, Renhai Chen, Gong Zhang, and Yiming Zhang. 2025. {GeminiFS}: A Companion File System for {GPUs}. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 221–236.

[53] 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* (New York, NY, USA, 2023-01-30) *(ASPLOS 2023)*. Association for Computing Machinery, 325–339. https://doi.org/10.1145/3575693.3575748

[54] ShareGPT. 2025. ShareGPT. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered.

[55] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[56] Pure Storage. 2024. Pure Storage Delivers All-Flash for Every Storage Need. https://www.purestorage.com/company/newsroom/press-releases/pure-delivers-all-flash-for-every-storage-need.html.

[57] syoyo. 2025. safetensors-cpp. https://github.com/syoyo/safetensors-cpp.git.

[58] ThinkParQ. 2022. BeeGFS. https://doc.beegfs.io/7.3.0/advanced_topics/gds_support.html/.

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

[60] vllm project. 2025. vLLM. https://github.com/vllm-project/vllm.

[61] WekaFS. 2020. WekaFS Architecture White Paper. https://www.weka.io/wp-content/uploads/files/2020/08/weka-architecture-white-paper.pdf.

[62] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 154–161. https://doi.org/10.1109/CloudCom.2017.14

[63] YRCloudFile. 2024. YRCloudFile Delivers Unmatched Performance, Scalability, Simplicity and Flexibility for AI and HPC Workloads Without Compromise. https://www.yanrongyun.com/.

[64] Jie Zhang, David Donofrio, John Shalf, Mahmut T Kandemir, and Myoungsoo Jung. 2015. Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 13–24.

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