CAM: Asynchronous GPU-Initiated, CPU-Managed SSD Management for Batching Storage Access

Ziyu Song[1], Jie Zhang[1], Jie Sun[1], Mo Sun[1], Zihan Yang[1], Zheng Zhang[2], Xuzheng Chen[1], Fei Wu[1,3], Huajin Tang[1], Zeke Wang[1]
[1]College of Computer Science and Technology, Zhejiang University
[2]Purdue University [3]Shanghai Institute for Advanced Study of Zhejiang University

2025 IEEE 41st International Conference on Data Engineering (ICDE) | 979-8-3315-3603-9/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICDE65448.2025.00175

***Abstract*—With the wide adoption of GPU and the explosion in data volumes, existing accelerator-centric systems require massive storage access. They adopt high-performance storage devices like NVMe SSDs to scale up single-node systems cost-effectively and leverage the CPU to manage these SSDs. However, they suffer from performance bottlenecks because of the high CPU OS kernel overhead and the CPU memory intermediated data transfer. To address this issue, GPU-initiated and GPU-managed SSD management is proposed to allow the GPU to fully manipulate SSDs: 1) direct data transfer from SSD to GPU memory (data plane) and 2) GPU-managed SSD control (control plane). This can potentially enable these GPU systems to fully leverage the SSD bandwidth. However, we still identify two severe issues. First, the GPU-management SSD control leads to low GPU Streaming Multiprocessor utilization. Second, it leads to the serial execution of SSD accesses with GPU computation, which slows down the overall computing task. To this end, we propose CAM, the first asynchronous GPU-initialized, CPU-managed SSD management for batching storage access. It 1) offloads the SSD control plane from GPU to CPU, thus maximizing GPU streaming multiprocessor utilization, and 2) adopts asynchronous user-friendly APIs that allow programmers to easily overlap GPU computation and SSD I/O operations while keeping a synchronous programming experience. As such, CAM enables us to achieve the best of two worlds: high performance and high programmability. The experimental results show that CAM can perform GNN model training, mergesort, and GEMM up to 1.84×, 1.5×, and 1.84× faster, compared to the existing state-of-the-art GPU systems, while keeping high programmability.**

I. Introduction

With the advancement of GPUs, many cutting-edge applications, such as neural network models [1], [21] and GPU-based database systems [6], [50], are turning into GPU-centric systems, which can benefit from GPU's massive parallel computing power. In particular, the NVIDIA A100 GPU delivers 312 TeraFLOPS (TFLOPS) of computing capability, while the AMD Threadripper 3995WX CPU has fewer than 3 TFLOPS.

Together with the increasing requirement of computing power, the problem size of an application also increases faster. For GNN, the graph can contain billions of vertices and tens of billions of edges [35], [59], [63], which needs several terabytes of storage space. For DLRM, the memory capacity of embedding tables has increased dramatically from tens of GBs to TBs throughout the industry [33], [61], [62]. Therefore, many researches [43], [46], [48], [58] leverage SSDs to break the GPU memory and server memory boundaries so as to enable out-of-core computation on massive data volume for a broad range of applications. Storing data in SSDs not only addresses the data volume limitation but also decreases the system cost for the same amount of data.

However, the benefits of offloading data from GPU to SSDs come at the cost of potential performance degradation if the corresponding GPU-powered system is not well optimized, especially for its I/O side. Essentially, un-optimized GPU-powered systems could serialize SSD accesses and GPU computation. Because SSD accesses suffer from long access latency and low throughput, these GPU-powered systems that need to access data from SSDs could be easily bottlenecked by slow SSD I/O accesses. In order to better analyze their properties, we classify these systems into two categories according to how they manage SSDs.

1. Traditional CPU-OS-Managed SSD Management. Many GPU computing systems leverage CPUs to manage SSDs, such as Ginex [44], MariusGNN [55], and ZeRO-Infinity [46]. These systems use `libaio` [46] or POSIX I/O such as `pread` [44], [55] to manage SSDs. The CPU is responsible for 1) sending read/write commands to SSDs, 2) polling the completion information, and 3) notifying the GPU to continue the following computation. However, these systems fail to fully utilize the SSD bandwidth, especially in a multi-SSD setting. On the one hand, the overhead of invoking OS kernel significantly degrades the achievable SSD bandwidth. On the other hand, these kernel functions can only move data between SSDs and CPU memory. As such, the data movement between GPU memory and SSDs must use CPU memory as an intermedium, thus degrading the overall bandwidth and increasing the I/O latency.

2. GPU-Managed SSD Management. To relieve the severe CPU-managed overhead, the state-of-the-art GPU-managed SSD management approach BaM [45] enables GPU to directly access SSDs without the involment of CPU. The state-of-the-art approach BaM [45] allows GPU thread blocks to directly submit their read/write commands to Submission Queues (SQs) of SSDs using a synchronous API and then allows these GPU threads to poll on the corresponding entries in the Completion Queues (CQs), while the data is directly transferred between GPU memory and SSDs, without involving CPU memory as an intermedium. We observe that letting GPU directly manage SSDs greatly improves the achieved GPU-SSD bandwidth and decreases the I/O latency. However, the benefits come at the cost of low GPU resource utilization for compute kernels. BaM needs to launch a large number of GPU

2375-026X/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDE65448.2025.00175

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.

thread blocks to submit enough in-flight I/O requests and then poll their completions. SSDs experience a high I/O latency (tens of μs), and these threads are all waiting idly most of the time. In particular, an A100 GPU needs to use all its SMs (Streaming Multiprocessor) to fully exploit the SSD bandwidth when the number of SSDs exceeds five.

To this end, we propose CAM, the first GPU-initialized, CPU-managed SSD management for batching SSD access GPU applications. We offload the SSD control plane from GPU to CPU, thus maximizing GPU streaming multiprocessor utilization for compute kernels during SSD I/O processes. However, it is non-trivial to achieve. We propose two key designs to solve the corresponding challenges. As such, CAM enables us to achieve the best of two worlds: high performance and high programmability.

First, we propose a dynamic adjustment method to change the number of cores for CPU control SSD to address the challenge of managing SSDs with as few CPU cores as possible without sacrificing performance **(Challenge 1)**. Assigning multiple CPU cores for each SSD achieves high bandwidth. However, this consumes a large number of CPU cores, especially considering the large number of SSDs that a server may be equipped with. For instance, controlling 12 SSDs would require 13 cores for reading and an additional 13 cores for writing, resulting in a total of 26 cores. As the number of SSDs increases, this becomes increasingly unsustainable. To save CPU cores, CAM can dynamically adjust the number of cores for CPU control SSD. In an environment with N SSDs, CAM can use N/4 to N/2 cores dynamically according to the relative time of computation and I/O.

Second, we propose asynchronous user-friendly APIs that allow programmers to easily overlap GPU computation and SSD I/O operations while keeping synchronous programming experience to tackle the problem that an asynchronous interface comes with low programmability **(Challenge 2)**. Asynchronous APIs, though powerful, are often difficult to work with. To simplify usage, we want to design a set of user-friendly synchronous APIs without sacrificing performance.

We evaluate CAM on the sort and GEMM workloads, three popular GNN models (GCN, GAT, and GRAPHSAGE), and real-world datasets (Paper100M and IGB-Full) on a GPU server with A100 80G GPU and 12 SSDs. Experimental results show that our system can fully utilize the I/O bandwidth. In the GNN application, our approach has consistently outperformed state-of-the-art implementations across various models and datasets, achieving up to 1.84× training speed. CAM also outperforms up to 1.50× and 1.84× compared to baselines in sort and GEMM workloads.

Overall, our contributions are as follows:

1) We identify the concrete SSD access issues of existing GPU-powered computing systems.
2) We propose CAM, the first GPU-initialized, CPU-managed SSD management that enables parallel execution of SSD I/O and GPU computation. The compute kernel can use all GPU streaming multiprocessors so as to maximize the GPU utilization for computation;

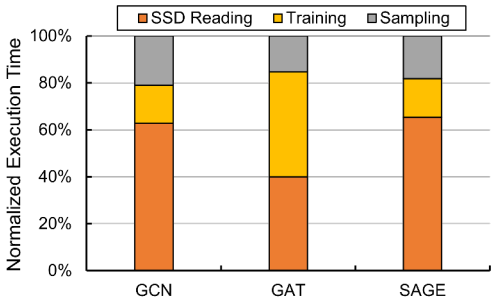


Fig. 1: GNN training time breakdown for the baseline BAM-based GIDS on the Paper100M [19] graph dataset and different models (GCN [26], GAT [53], GRAPHSAGE [16]). The node feature data is stored in 12 SSDs, while the graph structure data is stored in the CPU memory.

3) We design the first asynchronous APIs that enable a synchronous programming experience while keeping high performance for asynchronous GPU-Initiated, CPU-Managed SSD Management.

II. Motivation

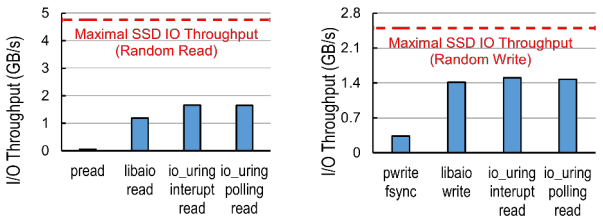
Compared with storing data in the server memory, SSDs' large volume comes at the cost of performance degradation. The main reason for the performance degradation is the high data transfer overhead between GPUs and SSDs. To illustrate this, we make GNN training, a typical big data application involving GPU and SSDs, as an example. We profile a GNN training system GIDS [43] on the Paper100M [19] dataset using 12 SSDs. As depicted in Figure 1, GIDS spends 40%-65% of the overall training time on extracting node features, which mainly involves reading data from SSDs to GPUs.

However, the poor performance can not be blamed on SSD's abilities. Modern SSDs (solid-state drives) offer significantly higher performance than traditional HDDs (hard disk drives). A typical enterprise SSD provides random read bandwidth up to 4.8 GB/s, and the throughput of NVMe-based SSDs scales linearly with the number of SSDs used [11], [15]. GIDS [43] uses 12 SSDs when training on the IGB [25] dataset. It only achieves 15 GB/s SSD bandwidth. However, the examined CPU-SSD bandwidth of using 12 SSDs can be up to 20 GB/s.

Applications such as storage-offloaded LLM training and DLRM training present similar conclusions. For example, the DLRM training system TorchRec [37] spends 75% of each iteration time on the embedding access, which mainly reads the embedding table from SSD with only the ~64% SSD bandwidth utilization [2]. LLM training system Zero-infinity [21], [46] spends more than 80% of time on the update phase that mainly consists of SSD accesses with only ~70% SSD bandwidth utilization [32].

In the following, we categorize existing GPU-powered systems into two types according to the devices for managing SSDs and identify the concrete issues that prevent them from fully exploiting SSD abilities.

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.



(a) 4KB Random Read (b) 4KB Random Write

Fig. 2: 4KB random read and write I/O throughput of software I/O stacks

A. GPU-Powered systems with CPU-Managed SSDs.

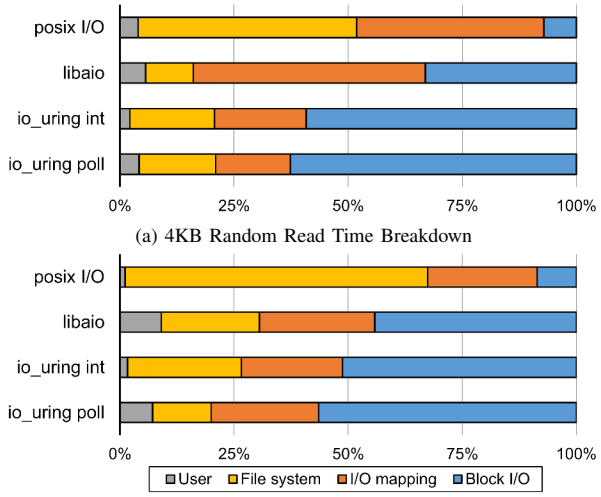
Many GPU-powered computing systems leverage CPUs to manage SSDs, such as Ginex [44], MariusGNN [55], and ZeRO-Infinity [46]: 1) For the control path, these systems use `libaio` or POSIX I/O primitives `pread` and `pwrite` to manage SSDs. In particular, the CPU sends the read/write commands to SSDs, polls for completion information, and notifies the GPU to continue the following computation. 2) For the data path, these systems use CPU memory as an inter-medium between GPU memory and SSDs. In particular, these systems only support transferring data between SSDs and CPU memory and then use `cudaMemcpy` to transfer data between CPU memory and GPU memory.

The above systems experience two severe issues that prevent the system from fully exploiting SSD performance.

Issue 1: I/O Stack Overhead Due to Heavy OS Kernel. Current CPU-managed systems adopt I/O stacks that require OS kernel functions to perform I/O, such as POSIX I/O, `libaio`, and `io_uring`. We found that these I/O stacks incur heavy overhead for GPU-SSD transfer. To show this, we measure the maximum throughput of various I/O stacks, including POSIX I/O, `libaio`, `io_uring` in interrupt mode (io_uring int) and `io_uring` in polling mode (io_uring poll) when manipulating a single Intel P5510 SSD. Figure 2 shows the random read and write throughput with the 4KB access granularity, where the dashed line indicates the maximum I/O throughput the SSD provides. The result shows that all these software I/O stacks' performance is far below SSD's throughput, indicating the severe overhead.

The reason for the overhead is due to heavy OS kernel processing. In the kernel mode, they all need the logical block address retrieval, io_map, and Block I/O (the OS I/O abstraction) to handle a single request. In the following, we take the most commonly used `pread` primitive from POSIX I/O as an example, while POSIX `pwrite` and other I/O stacks are alike to `pread`. We divide the I/O procedure into four layers, namely User, file system, I/O mapping, and Block I/O (the OS I/O abstraction), and describe concrete the I/O procedure and the layer that handles each step:

- User: The GPU calculates the file's data address and transfers it to the CPU.



(a) 4KB Random Read Time Breakdown

(b) 4KB Random Write Time Breakdown

Fig. 3: Read/write I/O time breakdown of software I/O stacks

- User: The CPU application issues I/O requests through `pread` calls, with the parameter file descriptor, offset, and the destination buffer. Each `pread` call reads a sequential chunk in the file.
- File system: The file system retrieves the page's logical block address (LBA) mapped to the file request according to the file ID and the offset.
- I/O mapping: The I/O mapping module calls I/O map-related functions to pin kernel pages and add them to the Block I/O.
- Block I/O: The block I/O module assigns the requests to SSD's request queue and communicates with SSDs for I/O transaction notification.
- I/O mapping: Upon the SSD I/O completion, the data has been transferred to the CPU memory by SSDs, and the CPU unpins the pinned kernel pages. The `pread` procedure is completed, and the CPU exits to user mode. After that, the CPU transfers data to the GPU.

To illustrate the OS kernel overhead, we break down the I/O procedure into time spent in the four layers. We evaluate on both 4KB random read and random write workloads, and via different kernel I/O stacks including POSIX I/O, `libaio`, `io_uring` in interrupt mode (io_uring int) and `io_uring` in polling mode (io_uring poll). Figure 3 shows the I/O time breakdown. We observe that significant amount of time is spent on the io_map and logical block address retrieval procedure (more than 34%). According to the SSD characteristics, the I/O stack should issue many NVMe commands concurrently to maximize the SSD's throughput. However, too much time spent on the file system and I/O mapping layers limits the number of concurrent NVMe commands sent to SSDs. In contrast, the smaller fraction of time spent in the two layers results in more throughput.

Opportunity for Improvement. Since I/O spends a large

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.

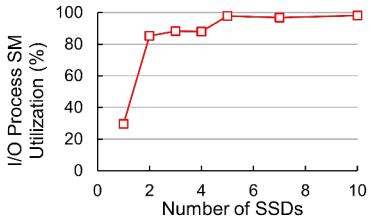


Fig. 4: Single A100 SM utilization (%) in BAM system with the number of SSDs to saturate SSDs.

fraction of time on end-to-end performance, reducing the CPU OS kernel software overheads is essential. We observe that overhead caused by the file system and I/O mapping layers can be eliminated. The underlying reason that these I/O stacks need to map and unmap the buffer is that these management handle requests one by one. They don't know the total request size ahead of time, so they can't map once in a single batching access. Besides, traditional file systems like EXT4 require logical block address retrieval design because the file is not always mapped to continuous blocks. The file system must look up the LBA from the file and offset.

In contrast, the SSD data access granularity in the scenario of GPU-powered systems is often 512 B or 4 KB, and the batch size is usually large. The file size is usually fixed or varies regularly. As such, the LBA can be mapped using a simpler method, such as direct mapping. Meanwhile, the total batch number can be determined before the first request is handled. As such, the buffer only needs to be mapped once before batching access is made and only needs to be unmapped after the whole batch accesses.

Issue 2: Redundant Memory Copy. Instead of directly transferring data between SSDs and GPU memory, system calls like `pread` can only take CPU memory addresses as parameters. As such, these systems use CPU memory as an inter-medium between GPU memory and SSDs. When the access granularity is small, the `cudaMemcpyAsync` function needs to be called multiple times. The smaller the access granularity, the greater the impact on performance. A 4KB granularity can only yield 1.3GB/s SSD bandwidth, which is only 6% of PCIe peak bandwidth. When we evaluate the ANNS workload that mainly involves 4 KB SSD accesses, `cudaMemcpyAsync` costs 78% of the total time. Such a large proportion can not be overlapped by computation.

B. GPU-Powered Systems with GPU-Managed SSDs

In order to address the issues of OS kernel-managed SSDs, recent works [45] intend to offload control and data planes onto GPUs. BaM (Big Accelerator Memory) [45] is the state-of-the-art GPU-initiated GPU-managed SSD management. BaM allows GPU thread blocks to submit their read/write commands to Submission Queues (SQs) to SSD using a synchronous API and allows these GPU threads to poll on the corresponding entry in Completion Queues (CQs) to be aware of the completion. BaM enables GPUs to achieve high throughput and fine-grained SSD-GPU accesses without CPU involvement. However, it still has a severe issue that prevents it from efficient I/O.

TABLE I: Architectural design comparison

	Initialized by	Control plane	Data plane
POSIX I/O	CPU	CPU OS kernel	SSD-CPU memory -GPU memory
BaM	GPU	GPU User I/O queue	SSD-GPU memory
CAM	GPU	CPU User I/O queue	SSD-GPU memory

Issue 3: Serial Execution of Computation and I/O due to Low GPU SM Utilization. BaM [45] offers an array-based synchronous API (`bam::array`) to access SSD, which provides fine-grained, on-demand SSD access. However, this user interface design comes at a performance cost in three aspects: 1) Due to its synchronous interface, threads wait for their competition after submitting I/O requests instead of submitting other requests. 2) To fully exploit SSD bandwidth, BaM needs to launch a large number of GPU thread blocks to submit enough in-flight I/O requests and poll the completion. 3) SSD I/O incurs high latency, e.g., a random read latency of 15 microseconds and a random write latency of 82 microseconds [49]. Thus, the thread idle period is long. As a result, a large number of thread blocks in BaM are waiting idly.

Worse still, many application utilizes multiple SSDs to enlarge the SSD capacity and improve the aggregated I/O bandwidth, and the problem becomes more severe with the increase in SSD numbers. To illustrate this, Figure 4 shows the A100 SM (Streaming Multiprocessor, computing unit of a GPU) utilization to fully exploit the bandwidth of different numbers of SSDs. When the number of SSDs exceeds five, the BaM system engages nearly all available GPU streaming multiprocessors to initiate NVMe commands. This high utilization for the I/O process results in substantial contention between GPU computation and GPU-managed I/O. Consequently, the I/O and computation phases are executed serially, leading to low GPU utilization for the computation process.

To validate that BaM failed to overlap I/O and computation, we evaluated the relative execution time of I/O and computation in a real-world application. We profile each stage in the GNN training execution of GIDS, as shown in Figure 1. GIDS extracts node features based on the BaM's high-level interfaces. GIDS spends 40%-65 % of the overall training time on I/O. The training phase accounts for a significant portion of the execution time, ranging from 16% to 44% for each step. In conclusion, BaM's low GPU SM utilization fails to overlap computation and I/O.

III. Design of CAM

To address the above issues, we propose CAM, a GPU-initialized, CPU-managed system that efficiently offloads the SSD management to CPU *user space* while providing a set of APIs to keep programmability. Table I shows the architectural design comparison of CAM, BaM, and POSIX I/O. We have three design goals that motivate the design of CAM.

Goal 1: Minimum GPU SM Overhead for I/O Processing. Minimum GPU SM utilization for I/O processing

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.

indicates that more GPU SMs can be used by computation tasks and thus can directly reduce the task completion time.

Goal 2: Fully Utilizing SSDs' Bandwidth and Scalability. Our design must provide high scalability when managing multiple SSDs, so as to meet the fast processing requirements of the growing data volume.

Goal 3: Programming-Friendly Interfaces. Generally, asynchronous APIs are thought to be able to provide a better performance than synchronous APIs, however, an asynchronous interface comes with low programmability. As such, CAM intends to provide APIs as easy as synchronous APIs to simplify programming without sacrificing performance.

To achieve the above goals, we propose CAM, an asynchronous GPU-initiated, CPU-managed SSD management for batching storage access. CAM consists of 1) a GPU-initiated, CPU-managed SSD I/O stack and 2) a set of user-friendly synchronous programming APIs.

A. GPU-Initiated, CPU-Managed SSD I/O Stack

CAM's I/O stack consists of new designs in both control and data planes. Regarding the control plane, CAM proposes 1) a GPU-initiated asynchronous I/O submission technology, 2) a CPU user-space SSD control offloading technology, and 3) a thread-level synchronization technology. Regarding the data plane, CAM presents the direct data path between GPU and SSD.

Overall I/O Procedure. Figure 5 shows how GPUs access SSDs in CAM. Here, we take GPU reading data from an SSD as an example, while the writing procedure is the same, except for the data transfer direction. GPU first writes the LBA of the data to be prefetched in the next step into CPU memory (❶). Meanwhile, the CPU thread keeps polling until it receives the GPU signal that informs the new I/O request (❷). After issuing the LBA, GPU initializes the asynchronous I/O submission by sending the synchronization signal (❸). Then, the CPU issues the I/O request to SSDs and waits for SSDs to complete the data transfer (❹). Meanwhile, GPU polls for the complete prefetch signal when there are no ongoing computation tasks (❺). Upon the CPU receiving the completion signal from SSDs, it informs the GPU through the completion signal (❻). After the data is prefetched completely into GPU memory, the GPU starts the computation (❼). For the data path, data is directly transferred between SSDs and GPU memory.

GPU-Initiated Asynchronous I/O Submission.

To achieve Goal 1, we aim to reduce GPU SM utilization during SSD I/O. GPU computes the LBA of blocks to be prefetched. It takes tens of microseconds between issuing NVMe commands and waiting for the request to complete, thus such a long time brings a significant challenge to fully harness SM utilization. Therefore, we present an asynchronous initial method to let GPU send LBA to CPU without concerning the SSD control (e.g., creating SSD commands and sending commands to SSDs).

As illustrated in Figure 5, GPU threads compute LBA and write the LBA to CPU memory (❶). This is an asynchronous GPU initial block request without blocking. After each thread

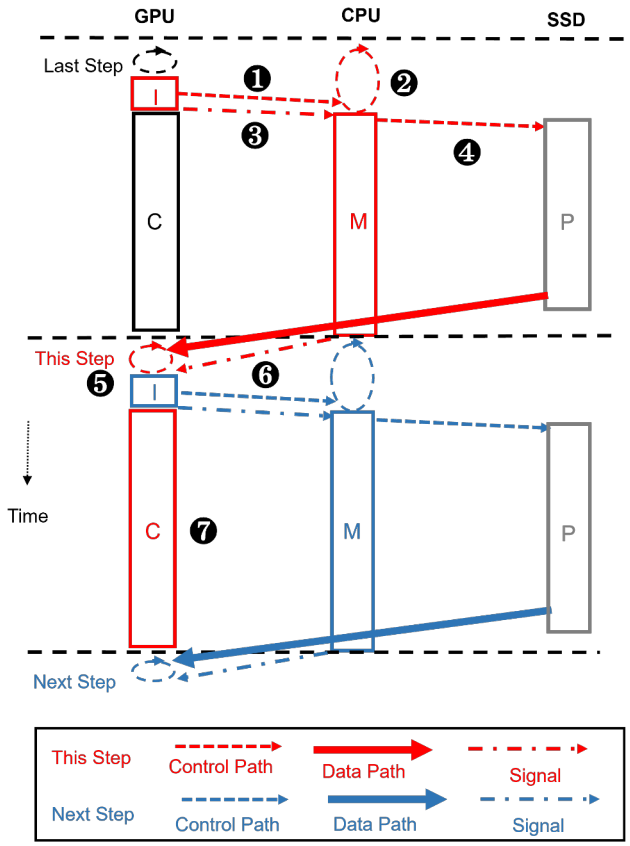


Fig. 5: CAM's GPU-initiated, CPU-managed disk I/O stack (I: GPU prefetch initialization; C: GPU computation; M: CPU SSD management; P: SSD request processing)

completes the block initially, the GPU synchronizes with the CPU (❸). Following this initial process, the GPU can perform computations using data prefetched in the last step. This arrangement ensures that the GPU remains productive rather than lying idle awaiting the completion of I/O tasks.

CPU User-Space SSD Control Offloading. To achieve Goal 1, we should minimize the GPU SM utilization from I/O processing. So, we offload the SSD management to the CPU, achieving zero GPU streaming multiprocessor utilization during SSD I/O. To achieve Goal 2, we have two tasks: 1) to make an SSD achieve maximum throughput, and 2) to achieve scalability when the SSD's number increases. We have adopted the lightweight Storage Performance Development Kit (SPDK) to achieve higher SSD bandwidth. SPDK offers a set of tools and libraries for writing high-performance, scalable, user-mode storage applications. Users can simply write a request into a predefined ring buffer and signal to the SSD that there are pending requests. SPDK completely bypasses the operating system kernel, including the block device layer, file systems, and the page cache. With SPDK's capabilities, we have managed to avoid kernel I/O stacks and achieve high bandwidth for SSD data reading. To achieve scalability, we use the thread pool that allows each thread to control one or many

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.

TABLE II: CAM software API

API	Run On	Input	Description
`CAM_init`	Host	—	Initialize SSD
`CAM_alloc`	Host	size	Allocate GPU memory
`CAM_free`	Host	pointer	Free GPU memory
`prefetch`	Device	LBA array req_num dest addr	Prefetch data from SSDs to pinned GPU memory
`prefetch_synchronize`	Device	—	Synchronize the last prefetch function
`write_back`	Device	LBA array req_num dest addr	Write back data from pinned GPU memory to SSDs
`write_back_synchronize`	Device	—	Synchronize the last write_back function

SSDs and dedicate a single NVMe queue pair to each NVMe device. The NVMe driver takes no locks in the I/O path. So, it scales linearly in terms of performance per thread, as long as a queue pair and a CPU core are dedicated to each new thread. To optimize CPU usage, CAM can dynamically adjust the number of CPU cores for controlling SSD. If computation takes a longer time, the total execution time is bounded by computation because I/O time completely overlaps with computation time. Less I/O throughput may also be no longer than the computation time, allowing CAM to dynamically reduce the CPU cores without affecting performance. CAM records computation and I/O time. CAM records both computation and I/O times. CAM adjusts the number of cores for CPU-based SSD control according to the relative time of computation and I/O in the last batch.

Direct Data Path between GPU and SSD. To achieve Goal 2, we need to optimize the data path from storage to GPU, which is essential for performance efficiency in massive storage access. Our system tackles memory copy issues by establishing a direct data path from the SSD to GPU, bypassing the CPU and thus avoiding unnecessary memory staging at CPU memory. To implement this, we need to get the physical address of pinned GPU memory and then use the physical address to make SSD commands.

Our system incorporates the GDRCopy technology to pin GPU memory and get the physical address of the pinned GPU memory. Specifically, these pinned memory buffers can be mapped to the GPU memory through the function `nvidia_p2p_get_pages`. After this procedure, we can know the start physical address of this big chunk of memory, and the address is continuous. So, we can calculate the physical address from any virtual address in this chunk.

During the data transfer, our system issues NVMe Submission Queue Entries (SQEs) with the target addresses set to specific physical locations within pinned GPU memory. By directly targeting these pinned GPU memory's physical addresses when sending NVMe SQEs, the data path is directly between GPU and SSDs.

B. Asyn. API with Synchronous Programming Experience

In order to achieve Goal 3, CAM needs to offer a series of programming-friendly interfaces, while an asynchronous interface can lack programmability. However, CAM must have an asynchronous interface to overlap computation and communication while keeping high GPU utilization for compute kernels. Therefore, CAM exposes synchronous-like high-level application programming interfaces to guarantee high programmability. We aim to pipeline the I/O and computation phases as shown in Figure 5. Figure 6 shows how the steps are logically pipelined for the GNN workload. The pipeline programming procedure consists of a series of coordinated steps to manage data flow with SSDs and processing tasks. Firstly, node sampling enables us to identify which nodes should be prefetched. Secondly, the data is retrieved from SSDs and stored in a read buffer. Thirdly, users utilize this data to perform computations for model training purposes. These three stages overlap with each other, ensuring a seamless and efficient process. If the read is dependent on the prior compute, pipeline bubbles will appear. This is a limitation of the algorithm. Our system can't eliminate the pipeline bubbles caused by data dependencies. In this situation where data have dependencies, the compute and I/O are serial. Our system achieves high I/O throughput and thus achieves short I/O time.

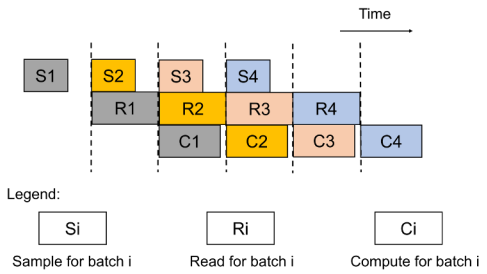


Fig. 6: CAM pipeline in GNN application

Table II shows our APIs. This section will describe the design for CAM API in the order of initialization, GPU memory management, and read-write-related functions, which are similar to the order of use. Finally, we will show a simplified example using CAM.

Initialization. CAM utilizes an `CAM_init` function to set up the data structure and manage threads for CPU-GPU synchronization and SSD management. CPU-GPU synchronization involves four main memory regions and a polling CPU thread. Regarding SSD management, the initialization function focuses on setting up the SSD controllers and mapping GPU memory regions, which we used for the `CAM_alloc` function.

GPU Memory Management. The `CAM_alloc` and `CAM_free` functions are used to manage the GPU memory. Users should use our alloc interface instead of the `cudaMalloc` function to allocate the GPU memory. The alloc function returns an address registered by GDRCopy. The allocated buffer would be pinned and the SSDs can directly access these buffers.

Read and Write related Functions. CAM does not require persistent threads on the GPU. Instead, it requires a persistent thread on the CPU. We realize the synchronization between GPU and CPU by four pre-allocated memory regions. These regions are designated for prefetching and are allocated in the initialization function. (1) The first region contains an array of

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.

```
1: procedure __global__ kernel_function
2:     for i in iterations do
3:         prefetch_synchronize()
4:         //compute the blocks to be prefetched
5:         compute_buffer ← read_buffer
6:         read_buffer ← temp_buffer
7:         for blocks to be prefetched do
8:             array[i] ← (block_id)
9:         end for
10:        prefetch(array,num,read_buffer)
11:        /* computation phase
12:        // Sampling
13:        // computation for model training
14:        */
15:        temp_buffer ← compute_buffer
16:    end for
17: end procedure
```

(a) Kernel Funtion

```
1: procedure host_function
2:     CAM_init()
3:     void *read_buffer,*compute_buffer
4:     void *temp_buffer
5:     read_buffer ← CAM_alloc(BATCH_SIZE)
6:     compute_buffer ← CAM_alloc(BATCH_SIZE)
7:     void *p1 ← read_buffer
8:     void *p2 ← compute_buffer
9:     kernel function <<<>>>()
10:    CAM_free(p1)
11:    CAM_free(p2)
12: end procedure
```

(b) Host Funtion

Fig. 7: Programming example powered by CAM

logical blocks that need to be processed. (2) The second region stores arguments for the CPU to process a batch of requests. (3) The third region is used by the CPU to be informed when the GPU has finished writing all the block IDs; this region is written exclusively by the GPU and read only by the CPU. (4) The fourth region notifies the GPU when the CPU has processed all requests; the CPU writes to this region but reads from it only when requested by the GPU. The first three regions are implemented using unified memory, while the last region is located in GPU memory but has its copy stored in CPU memory. The first three regions are only written by the GPU and read by the CPU, whereas the last region is only written by the CPU and read by the GPU.

Before the `prefetch` function is called, the GPU threads fill the first region with logical block addresses. Within the `prefetch` function, only the leading thread writes the necessary arguments for the CPU to process this batch of requests into the second region. The leading thread also writes a signal to the third region, informing the CPU polling thread that the GPU has completed writing and is ready for the CPU to begin processing I/O requests. The `prefetch` function only needs the leading thread to perform these actions, while other threads need not do anything. In the `prefetch_synchronize` function, all threads are blocked and wait for the leading thread to check if the fourth region has been written. This region will be written to by the CPU polling thread once it has fully processed all requests. Regarding the write-related functions, `write_back` and `write_back_synchronize` operate similarly to the read functions.

Figure 7 demonstrates a simplified example using CAM to write an application that overlaps prefetching and computation through a structured, three-step process. The first step invokes the `prefetch_synchronize` function to ensure complete data fetching (Line 3). The second step entails preloading logical block addresses of SSDs' blocks to be prefetched and calling the `prefetch` function (Lines 7-10). The core computational work is carried out during the third step. In the host function, users only need to initialize the `initialize` function once (Line 2 in the host function). Users alloc and free buffers in GPU memory using `alloc` and `free`.

We observe that the user experience of sequentially reading and computing mirrors the familiar synchronous programming model. Essentially, CAM offers the programmer a natural workflow where reading is synchronized before data retrieval. This implicitly aligns with conventional programming practices, offering ease of use and reduced cognitive load.

C. Discussion

CAM has three limitations. Firstly, CAM requires SSDs to operate without a pre-existing filesystem. Any filesystem must be removed before CAM deployment, and concurrent access to the same data blocks by multiple processes risks data consistency issues. Secondly, the current prototype restricts data consumption capabilities to a single GPU configuration. Thirdly, CAM's architecture requires a linear scaling of CPU core allocation relative to the number of SSDs to fully utilize their aggregate bandwidth. This scalability model risks resource contention when CAM operates alongside concurrent applications that require almost all CPU cores.

IV. Evaluation

Our evaluations aim to answer the following questions:

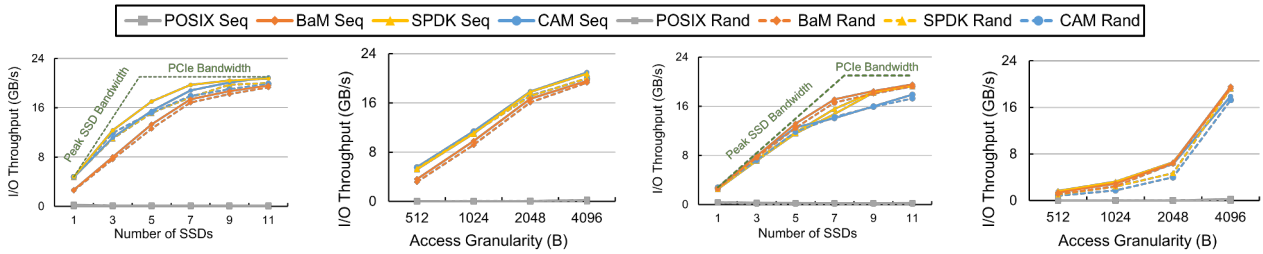
- What are the performance characteristics of CAM compared with existing CPU- and GPU-managed approaches (§IV-B)?
- What is the end-to-end performance in real world applications of CAM compared to baseline solutions (§IV-C, §IV-D, and §IV-E)?
- How user-friendly is the CAM API for programming purposes (§IV-F)?
- Does the user-friendly APIs sacrifice performance? (§IV-G)?
- What is the performance penalty of handling multiple NVMes with a single CPU thread (§IV-H)?
- How much is the CPU overhead for CAM? (§IV-I)?
- How does CAM compare with optimized SPDK (with overlapping) (§IV-J)?

A. Experimental Setting

Hardware Testbed. Table III summarizes the hardware and software configurations of our evaluation platform.

Workloads. In this section, we first conduct micro-benchmarks to evaluate the performance of CAM and different I/O stacks, then benchmark the end-to-end performance of CAM on three real-world workloads, namely GNN training,

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.



(a) Read I/O throughput w.r.t SSD number (b) Read I/O throughput w.r.t access granularity (c) Write I/O throughput w.r.t SSD number (d) Write I/O throughput w.r.t access granularity

Fig. 8: I/O throughput of CAM compared to BaM, SPDK, and POSIX I/O

TABLE III: Experimental platform

Configuration	Specification
CPU	Intel(R) Xeon(R) Gold 5320 CPU (2 × 52 threads) @ 2.20GHz
CPU Memory	768 GB
GPU	80GB-PCIe-A100
SSD	12 × 3.84TB Intel P5510
PCIe	Gen 4 x16
S/W	Ubuntu 22.04 LTS, NVIDIA Driver 550.54 CUDA 11.4 DGL 0.10 Pytorch 1.13.0

TABLE IV: Real-world dataset used for evaluating CAM

Dataset	Paper100M	IGB-full
Nodes Num	111,059,956	269,364,174
Edges Num	1,615,685,872	3,995,777,033
Feature Dimension	128	1024
Feature Size	56 GB	1.1 TB

merge sort, and general matrix multiplication (GEMM). We will describe the concrete workload and baseline settings in their individual subsections.

B. I/O Stack Microbenmarks

We first present the performance of CAM in comparison with (1) BaM, (2) POSIX I/O, and (3) SPDK. CAM manages each SSD using one CPU thread. We evaluate the performance of BaM using 262144 CUDA threads, a CUDA block size of 64, a queue depth of 1024, and the number of queues per controller of 128. We evaluate the performance of POSIX I/O with O_DIRECT. To measure the scalability of this SSD management, we create a widely adopted method of RAID 0 array to support multiple SSDs because POSIX I/O doesn't support varying SSD numbers. The goal is to examine the achieved disk I/O throughput. We examine the achieved I/O throughput on our platform with Intel P5510 SSDs [49] in different SSD numbers and access granularity. From Figure 8, we make 3 major observations.

Firstly, Figure 8a shows CAM achieves similar read throughput to that of SPDK and BaM. All three systems outperform POSIX I/O because they can bypass the overhead of the OS kernel. The measured peak PCIe bandwidth (21 GB/s) is lower than the theoretical value (32 GB/s) due to 1) PCIe header and control signal overhead and 2) PCIe traffic contention between multiple SSDs.

Secondly, the write I/O throughput is slower than the read throughput for all the measured SSD managements. This is because SSD itself has higher read throughput than write.

Thirdly, our findings also show that the I/O throughput increases with increases in access size in all workloads using 12 SSDs, as shown in Figures 8b and 8d. The throughput increase is facilitated by the NVMe protocol's efficiency, where more data are retrieved from the SSDs using a single Submission Queue Entry (SQE). This has a lower overhead in the flash translation layer [15].

In conclusion, CAM performs higher I/O throughput POSIX I/O and has similar performance to SPDK and BaM. When configured with 12 SSDs and an access granularity of 4096, CAM is capable of achieving 20GB/s throughput. Additionally, as CAM employs CPU resources exclusively to orchestrate the SSDs, it does not engage any GPU SMs. Consequently, during the computation phase, all available GPU SMs can be dedicated to computational tasks without any reservation or hindrance.

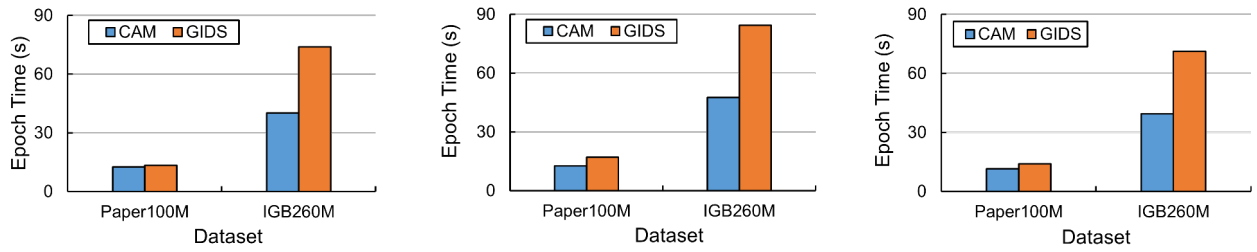
Parameter	Setting
GNN Task	Node classification
Sampling Method	2-hop random neighbor sampling
Sampling Fan-outs	25, 10
Hidden Layer Dimension	128
Batch Size	8000

TABLE V: Configuration Details in the GNN Experiment

C. Comparison of GNN Training Epoch Time

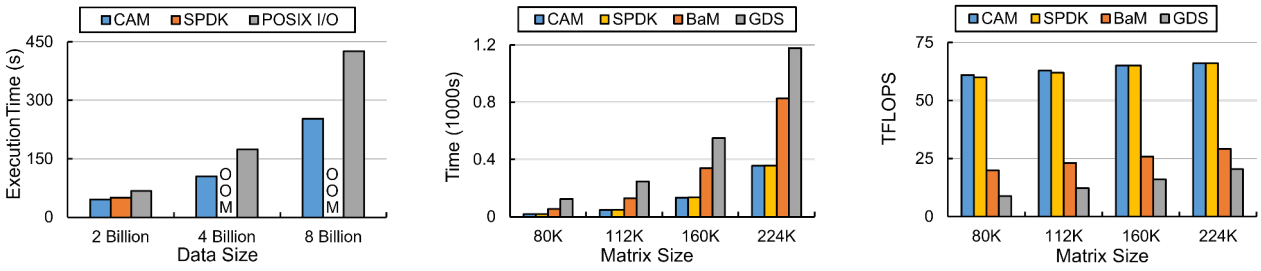
We compare CAM with the state-of-the-art out-of-core GNN training systems, GIDS [43], regarding the GNN training epoch time. We run three GNN models (GCN [26], GAT [53], and GRAPHSAGE [16]). Each model is tested in two datasets (Paper100M and IGB-Full). The node number, edge number, and feature dimension of Paper100M and IGB-Full are shown in Table IV. We use 12 SSDs to store the datasets. Neither GIDS nor our code uses the CPU memory cache. The configuration details in the GNN experiment are shown in Table V.

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.



(a) GCN (b) GAT (c) GRAPHSAGE

Fig. 9: End-to-end performance comparison between CAM and GIDS in GNN training



(a) Sort Time (b) GEMM Time (c) GEMM Flops

Fig. 10: End-to-end performance comparison in Sort and GEMM workloads

Figure 9 shows the results. We make 3 observations. Firstly, our approach has consistently outperformed state-of-the-art implementations across various models and datasets. The reason is that the I/O overlaps with computation, and thus shortens I/O time, because CAM can achieve higher throughput than BaM. Secondly, with the Paper100M dataset, our solution can achieve greater speed in the GAT model than GCN and GRAPHSAGE. This is because I/O time is slightly longer than the computation time. We find that the GAT involves the most intensive computations when evaluating different models. Due to this characteristic, CAM can overlap more time with the GAT model over others, such as GCN and GraphSAGE, which have lower computational demands. Thirdly, we have observed that the CAM achieves a greater speed-up on the IGB dataset than the Paper100M dataset. This is primarily because the I/O operations consume more time on the IGB dataset than the Paper100M dataset. For the IGB dataset, the I/O time is slightly longer than the computation time. In the ideal situation, if the I/O and computation parts fully overlap, the total time is I/O bound. CAM can take advantage of the available SSD throughput better than BaM.

In summary, CAM enables the overlap of SSD I/O and computation to achieve better performance in out-of-core GNN training applications compared to state-of-the-art BaM-powered out-of-core GNN training system GIDS.

D. Comparison of Sort Performance

We examine the sort time of the system based on CAM compared with the sort baseline, which is an essential database operation that is suitable to be executed in GPUs [7], [24].

We implement a sort algorithm based on the NVIDIA ModernGPU library [4]. The modern GPU library is a high-performance library. We compare CAM against SPDK and POSIX I/O. Our solution is structured into two distinct phases to optimize efficiency. In the initial phase, we leverage the advanced sorting capabilities of the ModernGPU library to methodically combine data blocks, each containing a substantial volume of 1 billion int32 entries. Following this preliminary step, we embark on the second phase, which involves the pairwise merging of these pre-sorted blocks in a systematic fashion until all data entries are fully organized in a sequential manner.

Figure 10a shows the mergesort time comparison of CAM and baselines on the mergesort workload. CAM outperforms POSIX I/O and achieves a similar execution time to SPDK. CAM performs better than POSIX I/O because CAM can achieve higher I/O throughput. CAM and SPDK achieve similar execution times because 1) they can achieve similar I/O throughput in this application, and 2) they overlap computation and I/O.

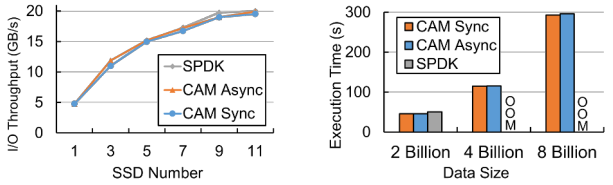
E. Comparison of GEMM Performance

We examine the General Matrix-matrix Multiplication (GEMM) performance of systems based on CAM, BaM, NVIDIA GPUDirect Storage (GDS) [40], and SPDK. GEMM is the core computational task of most deep learning models in training and inference. Accelerating GEMM has become a major goal of hardware accelerator design. Since three huge

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.

TABLE VI: Lines of code in real-world applications

Lines of Code / Workloads / SSD Management	GNN Training	Sort	GEMM
POSIX I/O	/	644	/
GDS	/	/	158
BaM	65	/	165
CAM	66	510	130



(a) Random Read (b) Sort Application

Fig. 11: Throughput and execution time comparison of CAM and other asynchronous APIs

matrices cannot fit into GPU memory entirely, we need to divide these matrices into smaller blocks.

Figures 10b and 10c show that our system can be used to accelerate GEMM compared to BAM and GDS solutions. GDS demonstrates slower throughput than BaM and CAM because GDS relies on a complex file system to deal with the EXT4 File System, NVFS Management, and CUDA library-related tasks. These I/O unrelated operations account for 70% of the total processing time. The substantial time spent on the file system and I/O mapping layers limits the number of concurrent NVMe commands that can be sent to SSDs, resulting in lower performance. For example, GDS achieves a throughput of only 0.8 GB/s with 12 SSDs, whereas CAM can attain nearly 20 GB/s. CAM outperforms BaM because CAM can overlap I/O with computation.

From the above three applications, we conclude that CAM has wide applicability and good performance.

F. Programming-Friendly APIs

We validate the programmability of our APIs by comparing the lines of the code we use with their baselines, as shown in Table VI. In the GNN training workload, we compare the code lines of the SSD-related I/O stack and the training one-step function, which are related to SSD management. CAM needs slightly longer lines of code than that of BaM, which relies on a synchronous API. In the context of the mergesort workload, the central processing loop of the CAM implementation comprises just 510 lines of code, compared to the 644 lines of the traditional version. Within the GEMM workload, the core loop of the CAM implementation is executed in 130 lines of code, showing a reduction of around 30 lines of code compared to the BaM and GDS solutions.

Irrespective of the algorithmic complexity, each implementation is streamlined, requiring fewer lines of code than traditional approaches or similar code lines than the synchronous

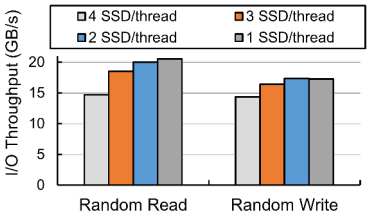


Fig. 12: I/O throughput using one thread to control multiple SSDs

code. CAM empowers developers to remain data-centric, minimizing the need to manage intricate asynchronous operations. CAM effectively meets Goal 3 of enhancing programmability without sacrificing efficiency, offering a more accessible and less labor-intensive coding experience.

G. Comparison with Asynchronous APIs

The main goal of the synchronous APIs is to allow users (regardless of their skill level) to write application code easily without sacrificing performance.

We run a sort application that sorts billions of integer elements in SSDs. We implement a version that adopts SPDK asynchronous APIs, a version with raw asynchronous CAM APIs (CAM-Async), and a version with our carefully designed synchronous APIs (CAM-Sync). For all three implementations, we use the same configurations. We let them use the same number of CPU threads. Figure 11a shows the achieved read throughput with different numbers of SSDs, and Figure 11b shows the execution time on datasets with different sizes. We observe that CAM-Sync can achieve nearly the same performance as CAM-Async/SPDK, indicating that our synchronous APIs would not harm performance while preserving programmability.

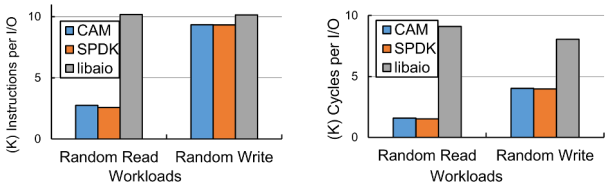
H. Effect of Handling Multiple NVMes with one CPU thread

To show the performance penalty of handling multiple NVMes with one CPU thread, we test the achieved random read and random write I/O throughput with different numbers of CPU threads using 12 SSDs.

A polling thread is used in each implementation and is not counted. We change the number of managed SSDs for each thread and measure the achieved throughput with the different number of threads. Figure 12 shows the result of CAM's I/O throughput when using different numbers of cores to control 12 SSDs at random read and random write workloads. We observe that when using a thread to control 2 SSDs, the I/O throughput is similar to that of one thread managing 1 SSD. When a single thread controls more than two SSDs, the performance begins to decline. The I/O throughput using one thread to control 4 SSDs is 75% of the throughput using one thread to control one SSD.

In conclusion, CAM allows a single thread to control two SSDs without performance degradation. However, 4 SSDs per thread could incur a 25% throughput degradation. In a cloud

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.



(a) Instructions per I/O (b) Cycles per I/O

Fig. 13: The process SSD cost of CAM, SPDK, and libaio

environment with N SSDs, users should guarantee that at least N/2 threads could be used to manage SSDs to avoid performance degradation.

I. Cost of CPU Processing

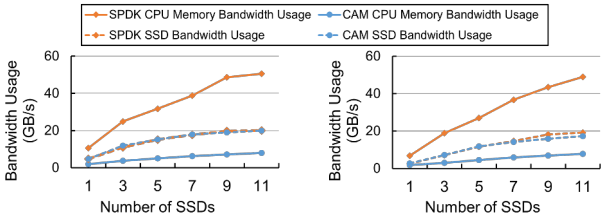
We test the cost of CPU processing SSD, in terms of cycles and instructions of the CPU to process each request. We compare the performance of CAM with SPDK and libaio. It's important to note that our experiment does not include BaM, as it utilizes GPU processing for SSDs, making comparisons with CPU instructions and cycles irrelevant. The results are shown in Figure 13.

We have four key observations. Firstly, CAM and SPDK use fewer instructions than `libaio` because CAM and SPDK bypass the OS kernel and avoid complex OS-related tasks, which results in a reduction in the number of executed instructions. Secondly, CAM and SPDK consume fewer CPU cycles compared to libaio. CAM's efficiency stems from having fewer processing instructions and achieving higher I/O throughput. Thirdly, CAM and SPDK require fewer instructions and cycles for random read workloads than for random write workloads because random read operations can utilize more bandwidth than random write operations. Fourthly, when comparing random write workloads, CAM and SPDK incur slightly fewer instructions but significantly fewer cycles than libaio. The substantial reduction in cycles arises from the much higher SSD throughput achieved by CAM and SPDK. The reason they only save a few instructions is that they are polling-based; they continuously check for SSD completion information. This polling method has a high instructions per cycle (IPC) ratio, resulting in significantly reduced cycles. In contrast, libaio is interrupt-based and does not require polling for SSD completion information. In conclusion, CAM costs less CPU resources than libaio and achieves a similar cost to SPDK.

J. Discussion on SPDK

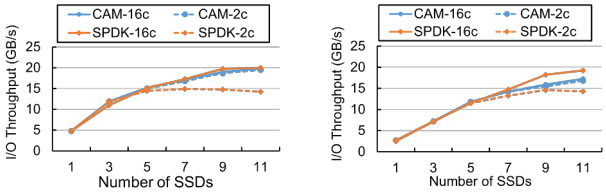
Relying on SPDK to overlap kernel invocations is effective only when 1) CPU memory bandwidth is sufficient and 2) the access granularity is large enough. If any of these conditions are not fulfilled, Performance degradation will occur. In the following, we show the memory bandwidth limitation and access granularity limitation when using SPDK.

Memory Bandwidth Limitation. When the GPU reads from an SSD, the SSD data is first written to CPU memory and



(a) Random Read (b) Random Write

Fig. 14: CPU memory bandwidth usage and SSD bandwidth usage comparison of CAM and SPDK



(a) Random Read (b) Random Write

Fig. 15: I/O throughput of CAM and SPDK at different CPU memory bandwidth. nc indicates running with n memory channels.

then copied from CPU memory to GPU memory. Similarly, when a GPU writes to an SSD, the process is analogous. Reading from SSDs consumes two times the CPU memory bandwidth, while writing to SSDs also consumes two times the CPU memory bandwidth. Saturating a single PCIe 4.0x16 GPU's read/write SSD throughput (21 GB/s) would consume nearly 42 GB/s CPU memory bandwidth. As such, the CPU memory bandwidth would easily become a system bottleneck if other co-located applications also heavily consume CPU memory bandwidth or multiple GPUs are reading/writing SSDs concurrently. To demonstrate this, we calculate the CPU memory bandwidth during data transfer and test the throughput change when the memory bandwidth is insufficient.

We first measure the real-time CPU memory bandwidth consumption when the GPU is reading/writing (Random read-/Random write) SSDs using CAM and SPDK. Figure 14 shows that the CPU memory bandwidth of SPDK is nearly twice the bandwidth of SSDs. The CAM's CPU memory bandwidth increases at a much slower pace. CAM requires much less CPU memory bandwidth than SPDK to fill the bandwidth. To further demonstrate the potential effect of the insufficient CPU memory bandwidth, we measure the achieved GPU-SSD throughput with 2 and 16 CPU memory channels (labeled "2c" and "16c", respectively). We use random read and random write workloads. Figure 15 shows that SPDK's throughput decreases when the CPU memory bandwidth is limited at both workloads, while CAM is not affected by the limited CPU memory bandwidth.

Access Granularity Limitation. SPDK's additional memory copy would call a `cudaMemcpyAsync` function and thus increase the I/O latency. When the destination buffer is not continuous, the `cudaMemcpyAsync` function needs to be

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.

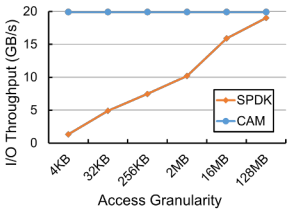


Fig. 16: I/O throughput of CAM and SPDK at different access granularity

called multiple times. The extra overhead and increased I/O latency would require a larger access granularity to hide them by overlapping.

To demonstrate this, we measure the I/O throughput with the different access granularities when the destination buffer is not continuous. Figure 16 shows that when the destination buffer is not continuous, the application with an access granularity of less than 128MB will decrease performance significantly. When accessing is in 4KB granularity, it can only achieve 1.3GB/s bandwidth, which is 93.5% lower than the CAM's achieved bandwidth.

In conclusion, SPDK with overlapping can only achieve ideal performance when the CPU memory bandwidth is sufficient, and when the application has a relatively large access SSD granularity.

V. Related Work

To our knowledge, CAM is the first asynchronous SSD management for batching storage access that offloads the SSD controller from GPU to CPU. In the following, we contrast CAM and existing works in the following aspects: GPU-managed direct SSD access, CPU-managed SSD access, and massive storage access applications.

GPU-Managed Direct SSD Access. In the realm of database query optimization, HippogriffDB [31] has effectively harnessed the power of direct GPU-SSD transfers. On a similar front, Morpheus [52] and GPUKV [23] have refined data serialization and the performance of key-value store applications through the use of in-storage computation. BaM [45] has introduced a synchronous model for GPU-initiated SSD access. Distinctively, CAM offloads the SSD management to CPU user space. This approach unlocks the GPU resource, significantly boosting the efficiency of GNN training processes.

CPU-Managed SSD Access. Recent advancements in technologies such as NVMMU [60] have enabled GPUs to directly transfer data to and from SSDs using the GPUDirect [39] technology. But it still requires a kernel/user mode switch. Our system does not need any mode switch. More importantly, this work is latency-oriented, and our work is throughput-oriented. Recent works [3], [15], [27], [30], [36], [54], [57] study the characteristics of SSDs and modern hardware and guide the design for data management tasks. Jun et al. [22] explain why the content of a file is not always mapped to continuous blocks. Didona et al. [11] presents the first systematic study and comparison of storage APIs on top of raw block devices. In contrast, our Asynchronous API, CAM, can help these systems tackle SSD I/O problems by overlapping the I/O process with computation while providing easy programming.

Massive Storage Access Applications. For DLRM training, RecTS [9] implements vector-based log-structured management to increase the cache hit ratio. RecSSD [58] is the first NDP-based SSD system specifically designed for recommendation inference. It assigns embedding vectors to specific SSD pages. This can increase SSD throughput due to SSD's character. Muhammad et al. [1] optimize the caching of frequently accessed embeddings. Existing practices have optimized several data management tasks, such as database buffer management [5], [8], [10], [12]–[14], [17], [18], [20], [28], [29], [31], [34], [41], [42], [47], [64] and information retrieval [56]. Marius Graph Embeddings [38] offloads node embedding parameters into SSDs and uses traditional CPU management. LuWu [51] optimizes parameter reading by assigning each layer its data buffers and retrieving parameters directly from the SSDs to the parameter buffer. Ratel [32] introduces SSD-CPU communication as an additional optimization dimension. Two notable examples of out-of-core GNN training systems are Ginex [44] and MariusGNN [55]. These platforms rely on the CPU to manage disk I/O operations. BaM-based GIDS [43] represents an attempt at creating a GPU-managed disk-based system that still struggles with the serial execution of node feature extraction and training due to BaM's synchronous interface. Their primary approach focuses on utilizing CPU memory to cache data to reduce the data amount to be accessed in the SSD without considering the SSD access process. In contrast, CAM can accelerate the procedure of SSD data access and help applications make I/O overlap with computation.

VI. Conclusion

In this work, we propose CAM, an asynchronous GPU-initialized, CPU-managed SSD management for batching storage access. CAM provides a series of APIs for data transfer between GPU and SSDs, minimizing GPU streaming multiprocessor utilization during I/O processes, enabling simultaneous maximization of GPU interface bandwidth, and overlapping computation and I/O operations. Our strategy involves a GPU-issued, CPU-managed SSD management approach, along with asynchronous APIs that provide a synchronous programming experience. Experimental results show that our system can fully utilize the bandwidth of the GPU interface. In the end-to-end experiments, CAM can perform GNN models training, mergesort, and GEMM up to 1.84×, 1.5×, and 1.84× faster, compared to the existing state-of-the-art GPU systems, while keeping high programmability. CAM is available at https://github.com/RC4ML/CAM.

Acknowledgement. The work is supported by the following grants: the National Key R&D Program of China (Grant No. 2022ZD0119301), the National Natural Science Foundation of China under the grant numbers (62236007, 62472384, 62441236, U24A20326). Zeke Wang is the corresponding author.

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. Heterogeneous acceleration pipeline for recommendation system training, 2024.

[2] Saurabh Agarwal, Chengpo Yan, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 348–363, New York, NY, USA, 2023. Association for Computing Machinery.

[3] Gustavo Alonso, Natassa Ailamaki, Sailesh Krishnamurthy, Sam Madden, Swami Sivasubramanian, and Raghu Ramakrishnan. Future of database system architectures. In *Companion of the 2023 International Conference on Management of Data*, pages 261–262, 2023.

[4] Sean Baxter. moderngpu 2.0. https://github.com/moderngpu/moderngpu/wiki, 2016.

[5] Nils Boeschen and Carsten Binnig. Gacco-a gpu-accelerated oltp dbms. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1003–1016, 2022.

[6] Sebastian Breß and Gunter Saake. Why it is time for a hype: a hybrid query processing engine for efficient gpu coprocessing in dbms. *Proc. VLDB Endow.*, 6(12):1398–1403, August 2013.

[7] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *ACM J. Exp. Algorithmics*, 14, January 2010.

[8] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, Ning Bao, and Yushi Liang. Ldc: a lower-level driven compaction method to optimize ssd-oriented key-value stores. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 722–733. IEEE, 2019.

[9] Cheng-Yu Chen, Jui-Nan Yen, You-Ru Lai, Yun-Ping Lin, and Chia-Lin Yang. Rects: A temporal-aware memory system optimization for training deep learning recommendation models. In *Proceedings of the 17th ACM International Systems and Storage Conference*, SYSTOR '24, page 104–117, New York, NY, USA, 2024. Association for Computing Machinery.

[10] Jiajia Chu, Yunshan Tu, Yao Zhang, and Chuliang Weng. Latte: A native table engine on nvme storage. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1225–1236. IEEE, 2020.

[11] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage apis: a systematic study of libaio, spdk, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, page 120–127, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Jaeyoung Do, Ivan Luiz Picoli, David Lomet, and Philippe Bonnet. Better database cost/performance via batched i/o on programmable ssd. *The VLDB Journal*, 30:403–424, 2021.

[13] Carl Duffy, Jaehoon Shim, Sang-Hoon Kim, and Jin-Soo Kim. Dotori: A key-value ssd based kv store. *Proceedings of the VLDB Endowment*, 16(6):1560–1572, 2023.

[14] Xiaopeng Fan, Song Yan, Yuchen Huang, and Chuliang Weng. Tengine: A native distributed table storage engine. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 3782–3795. IEEE, 2024.

[15] Gabriel Haas and Viktor Leis. What modern nvme storage can do, and how to exploit it: high-performance i/o for high-performance storage engines. *Proceedings of the VLDB Endowment*, 16(9):2090–2102, 2023.

[16] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS 2017*, 2017.

[17] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. Rethinking logging, checkpoints, and recovery for high-performance storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 877–892, 2020.

[18] Haochen He, Erci Xu, Shanshan Li, Zhouyang Jia, Si Zheng, Yue Yu, Jun Ma, and Xiangke Liao. When database meets new storage devices: Understanding and exposing performance mismatches via configurations. *Proceedings of the VLDB Endowment*, 16(7):1712–1725, 2023.

[19] Weihua Hu, Matthias Fey, Marinka Zitnik, et al. Open graph benchmark: Datasets for machine learning on graphs. In *NIPS 2020*, 2020.

[20] Yuchen Huang, Xiaopeng Fan, Song Yan, and Chuliang Weng. Neos: A nvme-gpus direct vector service buffer in user space. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 3767–3781. IEEE, 2024.

[21] 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, 2024.

[22] Yuhun Jun, Shinhyun Park, Jeong-Uk Kang, Sang-Hoon Kim, and Euiseong Seo. We ain't afraid of no file fragmentation: causes and prevention of its performance impact on modern flash ssds. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies*, FAST '24, USA, 2024. USENIX Association.

[23] Min-Gyo Jung, Chang-Gyu Lee, Donggyu Park, et al. Gpukv: an integrated framework with kvssd and gpu through p2p communication support. In *SAC 2021*, 2021.

[24] Ben Karsin, Volker Weichert, Henri Casanova, John Iacono, and Nodari Sitchinava. Analysis-driven engineering of comparison-based sorting algorithms on gpus. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 86–95, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Arpandeep Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, et al. Igb: Addressing the gaps in labeling, features, heterogeneity, and size of public graph datasets for deep learning research. *arXiv preprint arXiv:2302.13522*, 2023.

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

[27] Artem Kroviakov, Petr Kurapov, Christoph Anneser, and Jana Giceva. Heterogeneous intra-pipeline device-parallel aggregations. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*, pages 1–10, 2024.

[28] Bohyun Lee, Mijin An, and Sang-Won Lee. Lru-c: Parallelizing database i/os for flash ssds. *Proceedings of the VLDB Endowment*, 16(9):2364–2376, 2023.

[29] Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. X-ssd: A storage system with native support for database logging and replication. In *Proceedings of the 2022 International Conference on Management of Data*, pages 988–1002, 2022.

[30] Alberto Lerner and Gustavo Alonso. Data flow architectures for data processing on modern hardware. In *2024 IEEE 40th International Conference on Data Engineering*, pages 5511–5522. IEEE, 2024.

[31] Jing Li, Hung-Wei Tseng, Chunbin Lin, et al. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. 2016.

[32] 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, 2024.

[33] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. Understanding capacity-driven scale-out neural recommendation inference. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 162–171, 2021.

[34] Yanfei Lv, Bin Cui, Bingsheng He, and Xuexuan Chen. Operation-aware buffer management in flash-based systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 13–24, 2011.

[35] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 527–543, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Fabio Maschi and Gustavo Alonso. The difficult balance between modern hardware and conventional cpus. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*, pages 53–62, 2023.

[37] Meta. Torchrec. https://github.com/pytorch/torchrec/, 2022. Accessed: October 5, 2024.

[38] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 533–549. USENIX Association, July 2021.

[39] NVIDIA. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect, 2011.

[40] Nvidia. GPUDirect Storage: A Direct Path Between Storage and GPU Memory. https://developer.nvidia.com/blog/gpudirect-storage/, 2022.

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.

[41] Tarikul Islam Papon. Enhancing data systems performance by exploiting ssd concurrency & asymmetry. In *Proceedings of the IEEE International Conference on Data Engineering PhD Symposium*, 2024.

[42] Tarikul Islam Papon and Manos Athanassoulis. Aceing the bufferpool management paradigm for modern storage devices. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 1326–1339. IEEE, 2023.

[43] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. Accelerating sampling and aggregation operations in gnn frameworks with gpu initiated direct storage accesses. *arXiv preprint arXiv:2306.16384*, 2023.

[44] Yeonhong Park, Sunhong Min, and Jae W Lee. Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching. In *VLDB 2022*, 2022.

[45] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, et al. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *ASPLOS 2023*, 2023.

[46] 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*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Sagar Shedge, Nishant Sharma, Anant Agarwal, Mohammed Abouzour, and Güneş Aluç. An extended ssd-based cache for efficient object store access in sap iq. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1861–1873. IEEE, 2022.

[48] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

[49] Solidigm. D7-P5510 high-performing, standard-endurance PCIe 4.0 NVMe SSD drive. https://www.solidigm.com/products/data-center/d7/p5510.html, 2021.

[50] Young-Kyoon Suh, Junyoung An, Byungchul Tak, and Gap-Joo Na. A comprehensive empirical study of query performance across gpu dbmses. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(1), February 2022.

[51] Mo Sun, Zihan Yang, Changyue Liao, Yingtao Li, Fei Wu, and Zeke Wang. Luwu: An end-to-end in-network out-of-core optimizer for 100b-scale model-in-network data-parallel training on distributed gpus, 2024.

[52] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, et al. Morpheus: Creating application objects efficiently for heterogeneous computing. *ACM SIGARCH Computer Architecture News*, 2016.

[53] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[54] Leonard Von Merzljak, Philipp Fent, Thomas Neumann, and Jana Giceva. What are you waiting for? use coroutines for asynchronous i/o to hide i/o latencies and maximize the read bandwidth! In *ADMS@ VLDB*, pages 36–46, 2022.

[55] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. Mariusgnn: Resource-efficient out-of-core training of graph neural networks. In *EuroSys 2023*, 2023.

[56] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Evaluating list intersection on ssds for parallel i/o skipping. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1823–1828. IEEE, 2021.

[57] Jia Wei and Xingjun Zhang. How much storage do we need for high performance server. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 3221–3225. IEEE, 2022.

[58] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. Recssd: near data processing for solid state drive based recommendation inference. ASPLOS '21, page 717–729, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. Agl: a scalable system for industrial-purpose graph machine learning. *Proc. VLDB Endow.*, 13(12):3125–3137, aug 2020.

[60] Jie Zhang, David Donofrio, John Shalf, et al. Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. In *PACT 2015*, 2015.

[61] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems, 2020.

[62] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. KDD '18, page 1059–1068, New York, NY, USA, 2018. Association for Computing Machinery.

[63] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *Proc. VLDB Endow.*, 12(12):2094–2105, aug 2019.

[64] Tobias Ziegler, Carsten Binnig, and Viktor Leis. Scalestore: A fast and cost-efficient storage engine using dram, nvme, and rdma. In *Proceedings of the 2022 International Conference on Management of Data*, pages 685–699, 2022.

Authorized licensed use limited to: Zhejiang University. Downloaded on October 27,2025 at 00:06:45 UTC from IEEE Xplore. Restrictions apply.