This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of current version 22 May, 2026.

Digital Object Identifier 10.1109/ACCESS.2026.0000000

NVMeNext: Rearchitecting a Virtual NVMe Device for GPU-centric Storage Systems

INJAE KANG, JAEHOON SHIM, KYU-JIN CHO, AND JIN-SOO KIM, (Member, IEEE)
Department of Computer Science and Engineering, Seoul National University, Seoul 08826, Republic of Korea

Corresponding author: Jin-Soo Kim (e-mail: jinsoo.kim@snu.ac.kr).

This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grants (No. RS-2021-II211363 and No. RS-2023-00222663) funded by the Korean government (MSIT) and in part by SK hynix. (Injae Kang, Jaehoon Shim and Kyu-Jin Cho are co-first authors.)

ABSTRACT As GPU memory pressure continues to intensify, the role of NVMe SSDs has evolved from that of mere persistent storage media to that of active GPU memory expanders that provide abundant memory capacity at low cost. However, the conventional software storage I/O stack managed by the host CPU has become a critical bottleneck under the tremendous I/O demand generated by massive thread-level parallelism of modern GPU devices. To address this limitation, NVIDIA has introduced the StorageNext vision centered on GPU-initiated I/O that enables GPUs to directly manage data movement without host CPU involvement. In such GPU-centric storage systems, NVMe SSDs are expected to sustain MIOPS-scale random read throughput for fine-grained accesses issued by GPU applications. Yet, no existing NVMe emulator is designed to sustain such highly concurrent, fine-grained GPU-intiated I/O workloads, highlighting the need for a scalable emulation platform for studying future GPU-centric storage systems. In this paper, we present NVMeNext, a scalable NVMe SSD emulator architecture motivated by GPU-centric workloads. NVMeNext adopts a streamlined architecture for scalable request processing and performance modeling by removing architectural bottlenecks from the I/O path. In addition, NVMeNext introduces phase-based batched request processing, which improves GPU memory access efficiency by amortizing GPU-side NVMe queues and buffer accesses through the Intel Data Streaming Accelerator (DSA) engine. The source code of NVMeNext is publicly available at https://github.com/snu-csl/nvmenext.

INDEX TERMS Emulator, NVMe, SSD, GPU-initiated I/O, Intel DSA

I. INTRODUCTION

Modern GPU workloads, ranging from large-scale graph analytics [1], [2] to large language model (LLM) serving [3], [4], [5], [6], are characterized not only by immense computational demand but also by their extremely large memory footprints. To overcome the limited capacity of High-Bandwidth Memory (HBM), recent systems increasingly exploit lower tiers in the memory hierarchy to expand the effective memory capacity available to GPUs. For example, recent approximate nearest neighbor search (ANNS) systems [7], [8], [9] offload billion-scale graph datasets from GPU HBM to CPU DRAM, or even to the storage tier. This rapid change of GPU computing landscape has shifted system design priorities from pure computation toward the joint optimization of computation and data movement across a heterogeneous memory hierarchy. In particular, owing to its low $/GB and ease of scaling out, the storage tier is evolving from a passive data persistence layer into a scalable, capacity-oriented extension of GPU memory.

This shift in the role of the storage tier has also transformed the data access characteristics and performance requirements imposed on modern storage systems. Traditionally, data transfers between GPUs and storage devices have been coordinated by the host CPU, which is responsible for submitting I/O requests and processing their completions (referred to as *CPU-initiated I/O*). However, as GPU architectures continue to exploit massive thread-level parallelism, CPU involvement in the I/O path has become a fundamental bottleneck and significantly constrains overall system throughput. To address this limitation, prior works [10], [11], [12], [13] have proposed *GPU-initiated I/O*, in which GPUs directly issue I/O requests on demand without CPU intervention. Such GPU-centric architectures demonstrate that GPUs can serve as highly parallel data access engines and saturate the bandwidth of the storage tier at high queue depths even for small, fine-grained data accesses.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

In this context, NVIDIA has publicly introduced the ''StorageNext'' vision with the SCaled Accelerated Data Access (SCADA) programming model [14]. SCADA exposes a storage access interface that allows GPU threads to directly issue I/O requests to NVMe-based devices. By moving both the control and data paths from the CPU to GPU, SCADA enables tens of thousands of GPU threads to generate highly concurrent I/O requests, thereby better utilizing PCIe bandwidth. Beyond system-level software support, storage vendors are also beginning to design new StorageNext-oriented device architectures. For example, SK hynix and Sandisk have announced High Bandwidth Flash (HBF) [15], a next-generation NAND-based memory layer formed by vertically stacking multiple NAND dies. Kioxia has likewise introduced a PCIe Gen7 SSD based on their XL-FLASH memory solution [16]. In these models, NVIDIA is driving storage devices toward 100 MIOPS-class performance for small, 512 B-granularity random reads [14], [17], [18].

Although the architectural transition of SSD toward StorageNext is becoming apparent, designing a new storage architecture and maturing it into a deployable product remain complex and time-consuming processes. As a result, it is challenging for GPU application developers to evaluate how emerging storage architectures and their performance characteristics can be exploited in practice, as well as what benefits they may provide at the application level. This gap motivates the need for a scalable NVMe SSD emulator that enables early evaluation of future storage designs under GPU-centric workloads, while capturing the key characteristics and performance requirements of GPU-centric storage systems.

Such an emulator must satisfy three key requirements. First, it must provide peer-to-peer (P2P) communication functionality with real GPU devices. This is because, in GPU-centric storage systems, NVMe queue pairs and target data buffers reside in GPU memory, and these memory regions are exposed to the NVMe device through the GPU's base address register (BAR) space. Second, it should remain scalable under the extreme concurrency and MIOPS-class throughput demands of GPU workloads. Third, it should provide a seamless performance model that accurately captures device-level timing behavior, including NAND read latency (i.e., *tR*) and channel occupancy. At the same time, the timing model itself must not become a centralized serialization point that limits overall emulator scalability. However, to the best of our knowledge, existing SSD emulators [19], [20], [21], [22], [23], [24] fall short of satisfying all of these requirements simultaneously.

In this paper, we propose NVMeNext, a scalable NVMe SSD emulator for GPU-centric storage systems. NVMeNext rearchitects NVMeVirt [22] to scale under highly concurrent, fine-grained data accesses from massive GPU threads. First, we introduce a *streamlined request-handling architecture* that improves scalability by removing architectural bottlenecks from the I/O path and eliminating synchronization overhead. Second, we propose a *global NAND performance model with fine-grained locking* that preserves a single shared performance model while supporting highly concurrent access. Finally, we adopt *phase-based batched request processing* to amortize accesses to GPU-resident NVMe queues and data buffers through bulk transfers using the batching capability of Intel Data Streaming Accelerator (DSA), thereby meeting MIOPS-class throughput demands.

Our system-level evaluation demonstrates the utility of NVMeNext as a scalable emulation platform for GPU-centric storage systems. Specifically, we analyze system-level implications, including how NVMe driver behavior and warp-level optimizations jointly affect end-to-end performance. In addition, using graph analytics workloads, we demonstrate that NVMeNext can deliver performance that reflects realistic characteristics in real-world GPU workloads.

The main contributions of this work are as follows:

- **Scalable NVMe emulator for GPU-centric systems.** We present NVMeNext, a scalable NVMe SSD emulator designed for highly concurrent, fine-grained GPU-initiated I/O. Through its multi-manager architecture and performance model with per-plane locking, NVMeNext effectively eliminates serialization and synchronization bottlenecks within I/O path.
- **DSA-accelerated batched request processing.** We introduce phase-based batching that utilizes Intel DSA to amortize GPU-memory access overhead across command fetching, data transfer, and completion handling for MIOPS-class throughput demands.
- **System-level evaluation and design insights.** We assess the usefulness of NVMeNext as a scalable emulator platform through GPU-centric workloads. We derive system-level implications and expose application-level optimization effects otherwise masked by emulator-side bottlenecks. The source code of NVMeNext and scripts are publicly available at https://github.com/snu-csl/nvmenext.

The remainder of this paper is organized as follows. Section II and Section III provide background and motivate our design. Section IV and Section V develop our design incrementally, pairing each component with its validation to establish a foundation for subsequent components and isolate the contribution of each change. Section VI presents system-level studies and derives implications for GPU-centric storage. Section VII discusses limitations and portability. Finally, Section VIII concludes the paper.

II. BACKGROUND

A. CPU-INITIATED I/O VS. GPU-INITIATED I/O

In traditional storage systems, the host CPU is responsible for both initiating storage accesses and orchestrating data movement. Fig. 1(a) illustrates how data stored on an SSD are delivered to the GPU through CPU-initiated I/O. The CPU first opens the target file on the SSD and submits I/O requests through the conventional kernel storage I/O stack including the file system and block layer. Upon receiving the requests, the SSD locates the requested data and performs a direct memory access (DMA) operation to transfer the data from NAND to CPU memory. After the DMA transfer

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

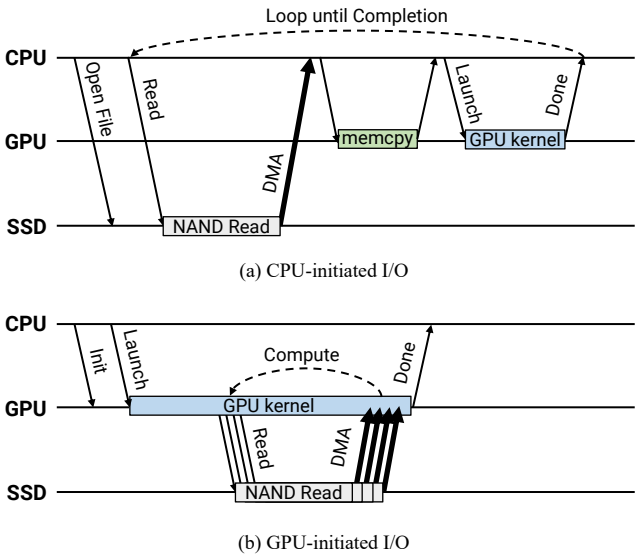


FIGURE 1. **Read path comparison between CPU-initiated I/O and GPU-initiated I/O.**

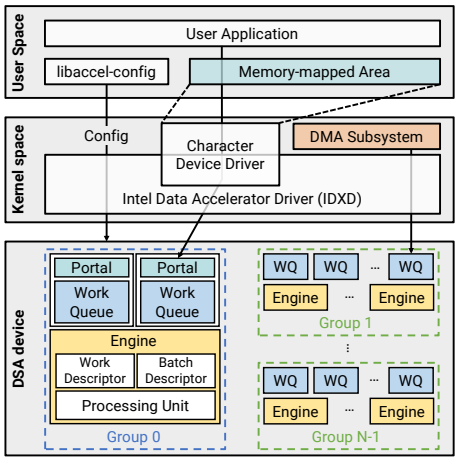


FIGURE 2. **Intel Data Streaming Accelerator (DSA).**

completes, the SSD signals completion to the CPU, which then copies the data into a GPU buffer and launches the corresponding GPU kernel. When a GPU kernel requires a large dataset, this process may occur multiple times, thereby incurring substantial overhead from repeated memory copies and CPU–GPU synchronization. Although GPUDirect Storage (GDS) [25] can eliminate intermediate memory copies by enabling direct DMA transfers between GPU memory and SSDs, it still relies on the host CPU for I/O orchestration, leaving CPU intervention on the critical path and limiting achievable throughput [10].

In contrast, GPU-initiated I/O removes the host CPU from the I/O control plane. As shown in Fig. 1(b), the CPU is responsible only for launching the GPU kernel. When the kernel needs to access data stored on an SSD, GPU threads issue I/O requests on demand, and the requested data is transferred directly from the SSD to GPU memory via peer-to-peer DMA (P2P DMA). Moreover, only the GPU threads (or warps) waiting for the requested data are stalled, while the remaining threads can continue execution. This design allows storage access to effectively overlap with GPU kernel execution and more naturally matches the massive thread-level parallelism of modern GPUs.

The concept of GPU-initiated I/O was first demonstrated by prior work through BaM [10]. BaM enables GPUs to directly perform high-throughput, fine-grained storage accesses without incurring host-side CPU software overhead. Based on this concept, NVIDIA has publicly introduced SCADA under its StorageNext vision, extending the idea of GPU-initiated I/O into a broader programming model for future GPU-centric storage systems [14]. More recently, AGIO [13] extends this model with an enhanced API and runtime environment for asynchronous I/O operations. These works suggest that GPU-initiated I/O is not merely an optional design choice, but is emerging as a central direction for future GPU systems.

B. INTEL DATA STREAMING ACCELERATOR (DSA)

Intel Data Streaming Accelerator (DSA) [26] is an on-chip high-performance accelerator for data copy and transformation that was introduced in Intel's 4th Gen Xeon processors. It is designed to offload common data operations such as memory copy, fill, compare, and checksum-related primitives from CPU cores. Fig. 2 shows the overall architecture of DSA device and its software stack. Internally, a DSA device comprises *work queues (WQs)* that receive submitted descriptors and *engines* that execute the requested operations. A key abstraction in DSA is a *group*, which serves as the fundamental scheduling domain. Within hardware limits, each group may contain one or more WQs and one or more engines.

In Linux, DSA devices are managed by the Intel Data Accelerator Driver (IDXD). In addition to initializing and managing DSA devices, IDXD provides abstracted DSA instances through the Linux DMA subsystem APIs for in-kernel use [27]. For user-level applications, device initialization and runtime configuration are typically performed via the `libaccel-config` API library [28]. Since each WQ is exposed to applications as a character device by IDXD, applications can `mmap()` the corresponding WQ *portal* and submit work descriptors directly through the mapped area. Each 64-byte work descriptor contains the operation type, the addresses of target memory buffers, the transfer size, and the address of the completion record. DSA also supports batch descriptors, which can encapsulate multiple work descriptors to reduce the overhead of submitting multiple descriptors individually. By amortizing descriptor submission overhead, batch descriptors significantly improve the throughput of small DMA operations [29].

C. SSD SIMULATORS AND EMULATORS

To accurately reproduce the behavior of real NVMe SSDs, many prior works have been proposed in the

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

TABLE 1. Comparison of various simulation and emulation frameworks.

	Simulators		Emulators				
	Trace-replay [30],[31]	Full-system [32],[33]	Driver-based [20],[21]	HW-based [23],[24]	VM-based [19],[34]	NVMeVirt [22],[35]	NVMeNext
Execution speed	Fast	Slow	Fast	Fast	Unstable	Fast	**Fast**
Real environment deployment	✗	✔	✔	✔	✔	✔	✔
Accurate performance model	✔	✔	✗	✗	✗	✔	✔
Low-latency request handling	✔	✔	✔	✗	✗	✔	✔
Kernel bypassing	✗	✗	✗	✔	✔	✔	✔
GPU visibility	✗	✗	✗	✔	✔	✔	✔
P2P DMA functionality	✗	✗	✗	✔	✗	✔	✔
MIOPS scalability	✗	✗	✗	✗	✗	✗	✔

form of simulators [30], [31], [32], [33] and emulators [19], [20], [21], [22], [23]. In Table 1, we classify the features required for GPU-centric storage systems and compare existing work. Since simulators either rely on replaying traces collected from real systems or suffer from prohibitively slow execution, they are often regarded as unsuitable for accurately reproducing full-system behavior. Moreover, simulators cannot interact with real GPUs. Consequently, they cannot support GPU-initiated I/O at MIOPS-scale throughput, since this workload inherently depends on real GPU threads issuing massively concurrent I/O requests.

In contrast, emulators expose device instances that interact with applications and peripheral devices, enabling system-level studies of SSD behavior. However, supporting GPU-initiated I/O requires an emulator to expose a GPU-visible NVMe interface, allow GPU threads to submit commands without traversing the host kernel I/O stack, and access NVMe queues and data buffers residing in GPU memory. Beyond these functional requirements, emulating future GPU-centric storage devices requires scalable request processing that sustains MIOPS-class throughput.

Existing emulation approaches differ in their ability to meet these requirements. Driver-based approaches [20], [21] modify the kernel device driver to handle request processing, which precludes the kernel-bypass I/O paths required for GPU-initiated I/O. Although hardware-based emulators [23], [24] can natively support PCIe P2P communication and provide reliable performance by using real NAND hardware, their reliance on fixed hardware components, such as controllers, NAND dies, and interfaces, makes it difficult to keep pace with the rapidly evolving performance requirements of modern SSDs.

Another line of work adopts VM-based emulation [19], [34], which exposes a virtual NVMe device to a guest operating system (OS). For instance, FEMU, a representative VM-based NVMe SSD emulator, implements a NAND performance model on the QEMU/KVM [36] and allows applications in the guest OS to directly interact with the emulated SSD. However, its performance model remains relatively simplified as it primarily accounts for NAND latency while omitting contention for shared resources such as PCIe links and NAND channels. In addition, VM-based architecture incurs frequent context-switching overhead during request processing, limiting its scalability and performance stability. Although a physical GPU can be exposed to the guest OS via VFIO [37], the surveyed VM-based emulators lack the P2P DMA functionality needed to access GPU-resident NVMe queues and data buffers, preventing them from directly supporting GPU-initiated I/O.

NVMeVirt [22] is a state-of-the-art software-defined NVMe SSD emulated at the PCIe layer, which is exposed to the host system as a native NVMe device. Upon initialization, NVMeVirt registers itself as a PCIe device, exposing NVMe BARs and controller registers so that the host identifies the control block and interacts with NVMeVirt as if it were an ordinary NVMe SSD. Every NVMe-level interaction, such as queue creation, doorbell updates, command fetching, and completion posting, is served through these standard interfaces in accordance with the NVMe specification. Consequently, unmodified I/O stacks, ranging from the kernel NVMe driver and SPDK [38] to GPU-initiated I/O frameworks such as BaM [10], run on NVMeVirt without any code changes. This enables NVMeVirt to seamlessly support GPU-centric storage systems through GPU-visible PCIe presence, kernel bypass, and P2P DMA functionality. Among the software-based emulators in Table 1, NVMeVirt uniquely combines these functional capabilities with a configurable NAND performance model. We therefore choose NVMeVirt as the foundation for NVMeNext.

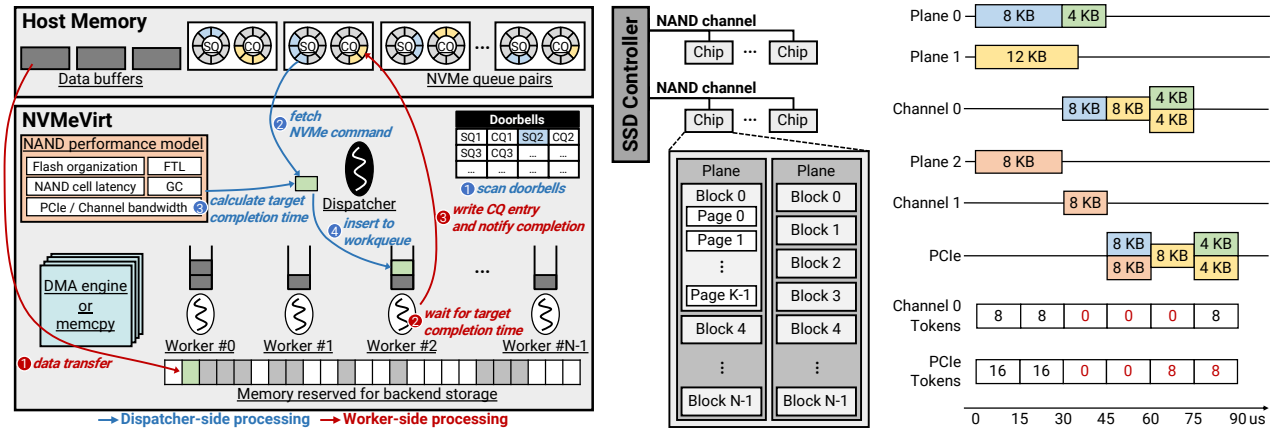
D. NVMEVIRT

In Fig. 3, we demonstrate the architecture and performance model of NVMeVirt to explain how it achieves high-fidelity emulation.

Dispatcher-worker architecture. NVMeVirt mainly consists of a *dispatcher* thread and multiple *worker* threads. The dispatcher manages the control path by polling the submission queues (SQs) and detecting host requests, while worker threads process those requests by performing data transfers. Fig. 3(a) illustrates the I/O request processing flow of NVMeVirt in detail. The dispatcher monitors the doorbell registers, and upon detecting a doorbell ring (❶), fetches an NVMe command from the corresponding SQ (❷). It then determines the command's target completion time using the performance model (❸) and forwards the request to a worker thread by inserting it into the worker's workqueue (❹).

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776



(a) I/O request processing in NVMeVirt (b) Flash organization (c) NAND performance model

FIGURE 3. NVMeVirt architecture and performance model overview.

The worker first performs the data transfer using either `memcpy()` or the on-chip DMA engine (❶). After data transfer, the request is held in the workqueue until its target completion time is reached (❷). The worker periodically checks the target completion time and, once it is reached, posts a completion entry to the completion queue (CQ) and sends an inter-processor interrupt (IPI) to signal completion to the host (❸).

NAND performance model. NVMeVirt supports two performance models: a simple model and a NAND model. The simple model divides an I/O request into smaller chunks and assigns a fixed processing latency to each chunk. However, this abstraction is inadequate for reflecting internal resource contention and other complex SSD behaviors such as page-mapped FTLs, write buffering, channel- and PCIe-level contention, plane-level parallelism, and garbage collection [35]. We therefore focus on the NAND model, which preserves NAND flash organization (Fig. 3(b)) and models those key features of modern SSDs.

The NAND model is designed to capture both device-internal parallelism and resource contention in SSDs. Each I/O request is decomposed into smaller NAND operations that can be processed in parallel across multiple FTL instances, channels, and planes. However, these operations contend for two types of resources. First, each plane can independently perform NAND read operations, but only a single read can proceed at a time per plane due to the shared page buffer. This is modeled by tracking the next available scheduling time for each plane, representing the earliest time at which the buffer is cleared and ready to accept new data. Second, operations that target different planes contend for shared bandwidth on the NAND channels and the PCIe link. To accurately reflect this contention, NVMeVirt employs a *token-based contention model* that reflects bandwidth-limited data movement along shared paths.

Fig. 3(c) illustrates how the token-based contention model operates through an example in which a 20 KB read request is issued and split into an 8 KB read on plane 0 and a 12 KB read on plane 1. Each token represents the right to transfer a certain amount of data during a given time slot. In this example, we assume one token represents a time slot for transferring 1 KB data. Although the two NAND read operations can be served in parallel at the plane-level, they still contend for the same channel bandwidth. If the 8 KB transfer from plane 0 is scheduled first and fully consumes the tokens available in the current slot, the 12 KB transfer over the shared channel 0 must be deferred to the next slot.

Once the channel transfer latency is determined, the PCIe transfer latency is computed in the same manner. Because the PCIe link offers higher bandwidth than an individual channel, each PCIe time slot contains more tokens than a channel time slot, and its tokens are shared across all channels. Consequently, the latency of a read I/O is dictated by the longest NAND operation, which encompasses NAND read latency (*tR*), channel transfer latency (*tCh*), and PCIe transfer latency (*tPCIe*). These components collectively reflect the overhead incurred by resource contention.

III. MOTIVATION

Although NVMeVirt provides the functional capabilities required for GPU-initiated I/O, it was not originally designed for the high concurrency and throughput demands of GPU-centric storage systems. We identify two critical limitations that prevent it from meeting these demands:

L1: Serialized command fetching and performance modeling. As shown in Fig. 3, NVMeVirt uses a single dispatcher thread that first fetches NVMe commands from the SQ one at a time and then calculates the performance model using per-channel and per-PCIe tokens. When there are no remaining commands in the corresponding SQ, the dispatcher moves to the next SQ and repeats the same procedure. Although this serialized design is simple to implement and adequate for modeling earlier-generation PCIe SSDs, it is ill-suited to GPU-centric storage systems, where SSDs are required

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

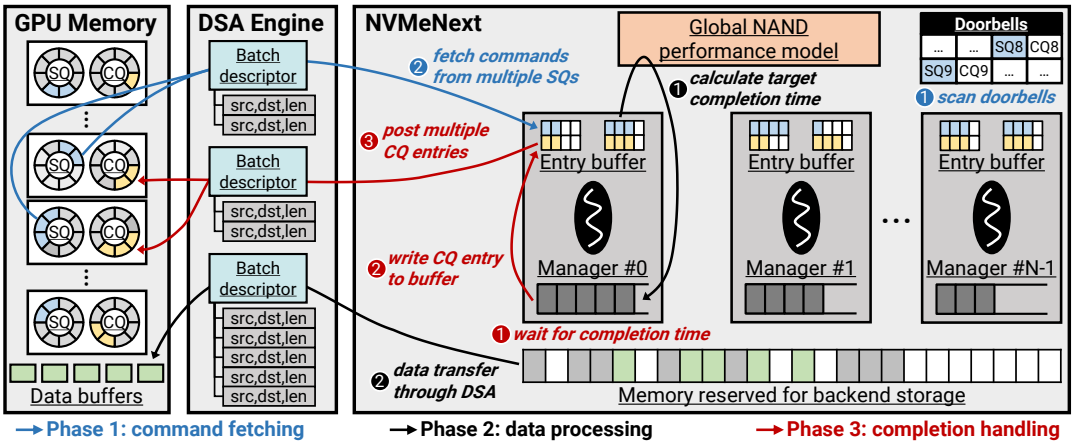


FIGURE 4. Overall architecture of NVMeNext.

to sustain extremely high IOPS for small-sized I/O requests issued by massively parallel GPU threads.

A naïve approach might be to increase the number of dispatchers and assign a subset of NVMe queue pairs to each of them. However, because NVMeVirt's NAND performance model is also implemented as a single instance and does not support concurrent token calculations by multiple dispatchers, accesses to the model must be serialized using a global lock, which critically degrades performance. Therefore, to support scalable command fetching with multiple dispatchers, the NAND performance model itself must also be redesigned to avoid becoming a bottleneck.

L2: Slow data copy engines. Since NVMeVirt is designed to use a reserved host memory area as its storage backend, it provides two data copy engines to support data movement between the reserved memory area and host-side data buffer. The first is the *memcpy engine*, which uses the standard `memcpy()` operation to copy data from the host buffer. However, because the host buffer location is specified as a physical address (PA) in the NVMe command, it must be remapped into the virtual address (VA) space before the data can be copied using `memcpy()` and unmapped after the copy completes. This approach incurs significant performance overhead because these mappings and unmappings should be performed for every NVMe operation. Therefore, the memcpy engine is not well suited to GPU-initiated I/O workloads.

To improve I/O processing performance, NVMeVirt also provides a *DMA engine* leveraging Intel I/O Acceleration Technology (I/OAT) [39]. I/OAT not only offloads data copy operations, but also eliminates the per-request remapping overhead required by the memcpy engine because it operates directly on PAs. Moreover, under GPU-initiated I/O, NVMe queue pairs and data buffers residing in GPU HBM are exposed in the host PA space through GPU BAR mappings. This architectural alignment suggests that leveraging I/OAT to offload data movement is a viable design direction for supporting GPU-initiated I/Os.

However, NVMeVirt's DMA engine has inherent constraints that limit throughput scalability when servicing tens of thousands of GPU threads. Specifically, NVMeVirt currently uses I/OAT in a synchronous manner, waiting for each DMA request to complete before issuing the next one. As a result, requests cannot be effectively pipelined even when a massive number of them are issued concurrently. Furthermore, for small I/O requests, the overhead of initializing each DMA request is non-trivial, making it difficult to achieve MIOPS-scale throughput. This problem becomes even more severe when I/OAT is used to fetch 64 B NVMe commands from the SQ, as the DMA setup overhead can be comparable to or even exceed the transfer cost itself. Unfortunately, unlike DSA, I/OAT lacks advanced features to effectively amortize this overhead, limiting its usefulness as a DMA engine for accelerating data movement between the GPU and SSD.

In the following sections, we explain how we rearchitect NVMeVirt to accommodate the new demands of GPU-initiated I/O. Section IV presents the overall system architecture of NVMeNext, including the redesigned I/O request processing flow and performance modeling architecture that address **L1**. Building on this scalable architecture, Section V addresses **L2** by introducing a DSA-based data copy acceleration mechanism for efficient data movement to and from GPU memory under GPU-initiated I/Os. We use its batch capability groups multiple DMA operations into a single submission to amortize per-operation overhead and enable effective pipelining, thereby improving throughput for highly concurrent GPU-initiated I/O.

IV. NVMENEXT DESIGN

A. OVERVIEW

NVMeNext is designed to support scalable request processing in GPU-centric storage environments, where sustaining high throughput for many concurrent small I/O requests across multiple queue pairs at high queue depths is of paramount importance. Fig. 4 depicts the overall architecture of NVMeNext. The key difference between NVMeVirt

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

and NVMeNext lies in the use of multiple *request manager* threads (hereafter referred to as managers). Rather than relying on a single dispatcher thread for command fetching and performance modeling, NVMeNext distributes queue pairs across multiple managers, each of which independently fetches commands and performs modeling. To further streamline request processing, each manager also handles data transfer and completion, eliminating the communication overhead between dispatchers and workers.

Based on this architecture, NVMeNext implements two optimizations. **(a)** *Phase-based batched request processing* improves DMA efficiency by grouping multiple I/O requests into a single transfer unit and reduces queue access overhead by performing bulk transfers of multiple queue-pair entries to and from the GPU memory. **(b)** *A NAND performance model with fine-grained locking* allows multiple managers to access a shared global performance model concurrently with minimal contention.

B. MULTI-MANAGER ARCHITECTURE

1) Design and Rationale

As described in Section II-D, NVMeVirt uses a single dispatcher thread for three tasks: (1) Fetching commands from SQs, (2) Computing target completion time through the performance model, and (3) Dispatching requests to worker threads. However, dispatching requests to workers incurs nontrivial synchronization overhead, as both the dispatcher and the target worker access the worker's workqueue. Although effective for modest throughput demands, this architecture does not scale to GPU-centric storage environments with high concurrency and low-latency requirements.

A straightforward way to improve scalability is to increase the number of dispatchers. However, this approach has two key drawbacks. First, communication and synchronization overhead between multiple dispatchers and workers deteriorates significantly under highly concurrent, latency-sensitive workloads. As the number of concurrent I/O requests increases, dispatchers need to access workers' workqueues more frequently to forward requests. In addition, because small I/O requests are typically completed with low latency, workers must dequeue requests from the workqueues more aggressively. Consequently, an increased number of dispatchers may exacerbate lock contention with workers over access to the workqueues.

Second, although dispatchers require more computing resources than workers in such workloads, at most half of the threads can be allocated to dispatchers, as each dispatcher must be paired with at least one dedicated worker to avoid excessive communication overhead. In addition, since the dispatcher and worker rely on polling-based execution, each thread requires a dedicated core to sustain rapid processing. This inflates the total core requirement, reducing the emulator's deployment efficiency and accessibility.

To address these limitations, NVMeNext unifies the roles of the dispatcher and worker into a single manager that independently handles command fetching, performance modeling, data transfer, and completion handling. Queue pairs are statically sharded across managers, which parallelizes command fetching while completely eliminating inter-thread synchronization. We adopt static sharding rather than dynamic queue pair assignment to avoid unnecessary scheduling complexity, since GPU-centric systems typically sustain high queue depths and a large number of outstanding requests, making load imbalance across queue pairs unlikely.

Fig. 4 illustrates how manager threads handle I/O request processing in a GPU-centric storage system environment. To efficiently handle a large number of concurrent requests under the unified design, each manager adopts phase-based execution and handles requests in batches at each phase rather than processing each request individually from end to end. NVMeNext divides the I/O request processing flow into three phases: (1) Phase 1: command fetching, (2) Phase 2: data processing, and (3) Phase 3: completion handling.

Specifically, during the command fetching phase (❶-❷), the manager scans the doorbells of its assigned SQs and accesses pending requests in bulk. Since accessing each entry requires a PCIe transfer from GPU memory, we copy valid SQ entries in batches into the *entry buffer* to improve transfer efficiency (detailed in Section V-B). The manager then moves on to the data processing phase, where it calculates the target completion time through the performance model and inserts the request into its local workqueue (❶). After computing the target completion times for all requests in the batch, it performs the data transfers between the reserved backend and GPU-side data buffers (❷).

Finally, during the completion handling phase, each manager checks the target completion times of requests in its local workqueue (❶). For requests whose completion time has passed, NVMeNext first writes the corresponding CQ entries to the entry buffer (❷) and then posts them in batches (❸), similar to how commands are fetched from the SQ. After dequeuing completed requests from the workqueue, the manager returns to Phase 1 and resumes doorbell scanning.

2) Evaluation

Impact of multi-manager architecture. To evaluate the effectiveness of the proposed multi-manager architecture, we run the FIO [40] benchmark to measure 512 B random read performance under various conditions. Throughout the paper, all experiments are conducted on a 64-core machine equipped with two Intel Xeon Gold 6526Y processors and 512 GB of DRAM. Each experiment is repeated 10 times, and we report the average value. We dedicate NUMA node 0 to FIO threads and the other node to NVMeNext's managers to minimize performance interference.

To clearly isolate the impact of the architectural changes, we conduct the evaluation in an idealized CPU-initiated I/O configuration in which data movement is handled by the memcpy engine and performance modeling is disabled. This setup is intended to approximate the theoretical upper bound of NVMeNext's request processing performance. Furthermore, we use the FIO's `SPDK` engine [38] that eliminates the

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

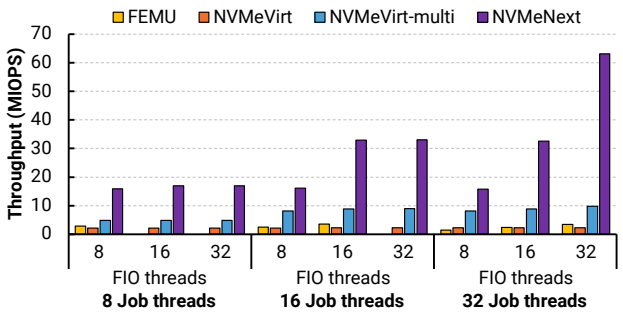


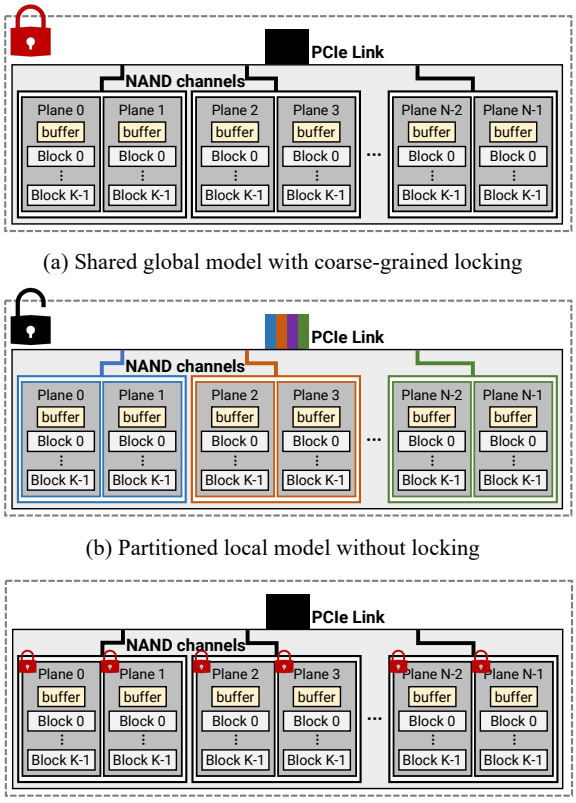
FIGURE 5. 512 B random read performance.

overhead of kernel stack and uses polling for I/O completion, which exhibits similar request submission/completion characteristics to GPU-initiated I/O. Under the `SPDK` engine, each FIO thread creates a dedicated NVMe queue pair with a per-queue depth of 256.

In this evaluation, we compare NVMeNext against FEMU and two NVMeVirt configurations: NVMeVirt and NVMeVirt-multi. NVMeVirt-multi adopts a multiple-dispatcher, multiple-worker design, in which one worker is assigned to each dispatcher. We use the term *job threads* to denote the total number of emulator threads involved in request processing. In NVMeVirt, 8 job threads correspond to one dispatcher and seven workers, whereas in NVMeVirt-multi, they correspond to four dispatchers and four workers. In NVMeNext, each job thread corresponds to a single manager. In FEMU, the number of job threads cannot exceed the NVMe queue count. Therefore, we report FEMU results only for configurations in which the number of FIO threads does not exceed the number of job threads. In addition, FEMU requires a separate performance-modeling thread beyond the job threads, which we deploy but exclude from the job thread count in the results.

Fig. 5 presents the read throughput results while varying the numbers of FIO threads and job threads. Both FEMU and NVMeVirt show saturated performance below 4 MIOPS. This is because both rely on a centralized modeling thread, which remains the primary bottleneck under highly concurrent workloads. FEMU further exhibits high run-to-run variance, with standard deviations reaching up to 49% of the average throughput due to context-switching overhead from its virtualization-based design, as reported in prior work [22]. NVMeVirt, in contrast, maintains standard deviations below 4% across all runs providing the stable execution needed for faithful emulation, but still falls short of the target throughput.

NVMeVirt-multi achieves a 1.8× throughput improvement as the number of job threads increases from 8 to 16. However, it shows no further performance gains beyond 16 job threads, nor does it exhibit any performance scaling with respect to the number of FIO threads. This result strongly demonstrates that, although multiple dispatchers can improve command-fetching throughput, synchronization overhead between dispatchers and workers becomes the dominant bottleneck to



(a) Shared global model with coarse-grained locking

(b) Partitioned local model without locking

(c) Shared global model with per-plane fine-grained locking

FIGURE 6. Comparison of NAND performance model designs.

throughput scaling. In contrast, we observe that NVMeNext achieves near-linear throughput scaling as the number of managers increases, reaching 64 MIOPS with 32 managers. The result highlights the scalability benefits of a unified architecture that combines the dispatcher and worker roles within a single manager. In summary, these results confirm that multi-manager architecture effectively addresses the scalability bottlenecks of dispatcher-worker designs, providing a scalable foundation for NVMe device emulation in GPU-centric storage environments.

C. PERFORMANCE MODEL WITH PER-PLANE LOCKING

1) Design and Rationale

The multi-manager architecture enables NVMeNext to sustain high throughput under highly concurrent small-I/O workloads. However, when the NAND performance model is enabled, throughput drops sharply from 64 MIOPS to 2 MIOPS. This degradation stems from two bottlenecks in the NAND performance model. First, the NAND model maintains globally shared state (e.g., token counts, L2P mapping table), requiring managers to acquire a global lock for every request being processed (Fig. 6(a)). As a result, contention among managers increases, leading to significantly reduced performance. Second, as described in Section II-D, the NAND performance model performs token counting to model NAND

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

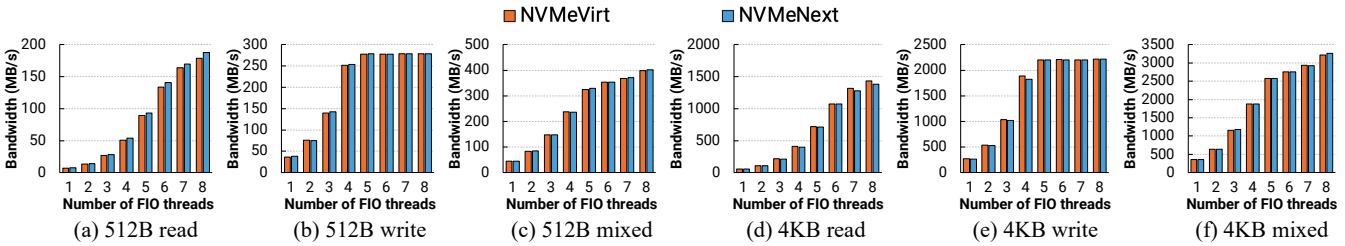


FIGURE 7. Performance comparison between NVMeVirt and NVMeNext.

channel and PCIe bandwidth contention. To ensure that bandwidth limits are not exceeded, the model sequentially traverses the slots to find an available slot with sufficient tokens. This costly sequential traversal exacerbates the global-lock bottleneck by increasing lock hold time.

To eliminate the global lock, we can partition the model across managers. Under this design, each manager maintains an independent local model with its own L2P mapping table, chips, and planes, while shared resources—NAND channel and PCIe bandwidth—are evenly divided among N managers. Fig. 6(b) illustrates the partitioned local model with four managers. Each manager is assigned a sharded subset of planes, and PCIe bandwidth is evenly partitioned across managers. Although this design is lock-free and load imbalance is unlikely under highly concurrent GPU-initiated I/O, such partitioning-based approaches are impractical for real workloads with mixed reads and writes. For example, in graph workloads, there is no guarantee that the manager serving a read request is the same one that previously handled the corresponding write request. In such cases, the partitioned model fails to locate the target data for a read request and completes the request without issuing any NAND read operations, producing inaccurate and unreliable results.

Thus, NVMeNext adopts a global model with per-plane locking and static transfer modeling. This design is motivated by a key observation: under small, concurrent I/Os, NAND read latency (tR) dominates overall request latency, while channel and PCIe bandwidth remain far from saturation. Consequently, precise contention modeling for channel and PCIe transfers is less critical. By replacing the token-based delay modeling with statically computed transfer latencies, the model's shared state is reduced to per-plane availability tracking, requiring synchronization only at the granularity of individual planes (Fig. 6(c)).

2) Evaluation

Impact of global model with per-plane locking. We first validate the key observation underlying static transfer modeling by comparing NVMeNext against NVMeVirt with a token-based contention model. We use NVMeVirt's default NAND model parameters: 3.3 GB/s PCIe bandwidth, eight channels with 800 MB/s bandwidth each, two planes per channel, and 36 us NAND read latency (tR), which have been shown to accurately emulate real NVMe device behavior [22]. Following the FIO evaluation methodology in NVMeVirt [22], we use the `psync` engine while increasing the number of FIO threads. For NVMeNext, we use 32 manager threads. We evaluate read-only, write-only, and mixed read/write workloads with I/O sizes of 512 B and 4 KB. Fig. 7 presents the results. Both NVMeNext and NVMeVirt exhibit nearly identical performance, with an average difference below 4%. This confirms that static transfer modeling accurately captures device behavior for small I/Os targeted by GPU-centric storage systems [10], [13], and that the global model with per-plane locking preserves this accuracy under multiple manager threads.

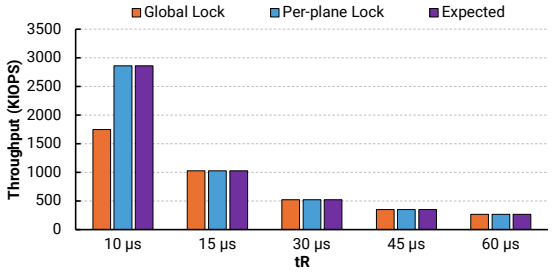


FIGURE 8. Comparison of expected and measured performance of NVMeNext.

Performance model scalability validation. We next evaluate whether NVMeNext achieves the maximum performance predicted by the NAND model. The expected throughput T_{exp} is given by

$$T_{\text{exp}} = \left(\frac{1}{tR + tCh + tPCIe} \right) \times N_{\text{planes}} \tag{1}$$

where tR, tCh and $tPCIe$ denote latencies of NAND read, channel transfer, and PCIe transfer respectively, and N_{planes} denotes the total number of planes. Note that the fractional term in (1) represents the number of NAND page reads that a single plane can serve per second. Multiplying this by the total number of planes gives the ideal throughput when all planes are fully utilized.

We use NVMeVirt's default NAND model parameters while varying tR from 10 us to 60 us. We compare two locking strategies for the shared NAND performance model: fine-grained per-plane locks and a single coarse-grained global lock. Fig. 8 presents the FIO results for 512 B I/O using the `SPDK` engine with 32 FIO threads. Per-plane locking success-

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

TABLE 2. 512 B random read throughput (MIOPS) under various NAND configurations.

Planes per channel	*tR* = 15 us			*tR* = 20 us			*tR* = 25 us			*tR* = 30 us		
	4 chan	8 chan	16 chan	4 chan	8 chan	16 chan	4 chan	8 chan	16 chan	4 chan	8 chan	16 chan
4	1.05	2.12	4.21	0.79	1.59	3.16	0.63	1.27	2.53	0.53	1.06	2.12
8	2.12	4.21	8.29	1.59	3.16	6.24	1.27	2.53	5.07	1.06	2.11	4.17
16	4.21	8.3	16.1	3.16	6.24	12.1	2.53	4.99	10.1	2.12	4.17	8.1
32	8.2	16.1	33.2	6.25	12.1	25.1	5	9.71	19.8	4.17	8.1	16.3

TABLE 3. Throughput of 512 B data copy operations with DSA under various configurations.

Config.	Interface	Memory Domain	Op. Type	Throughput
A	Subsystem API	CPU → CPU	Single	2.7 MIOPS
B	Low-level API	CPU → CPU	Single	39 MIOPS
C	Low-level API	CPU → GPU	Single	16 MIOPS
D	Low-level API	CPU → GPU	Batch	92 MIOPS

fully matches the expected throughput across all *tR* values. Global locking, however, falls short at *tR* = 10 us, achieving only 1700 KIOPS versus the expected 2900 KIOPS. These results confirm that NVMeNext can fully realize the performance derived by the NAND model despite using a globally shared model, enabled by per-plane locking.

Parameter study of the NAND model. With the accuracy of the performance model established, we explore the NAND parameter design space to study the impact of *tR* and NAND organization parameters, and to identify configurations that meet the requirements of GPU-centric storage systems. We provision ample interconnect bandwidth—3.6 GB/s per NAND channel and 28 GB/s for the PCIe link, following a recently proposed high-performance SSD [41]—so that *tR* and NAND organization parameters dominate the measured throughput. Table 2 presents the 512 B random read throughput using FIO `SPDK` engine with 64 FIO threads at a queue depth of 256. Across all 48 configurations, the measured throughput matches the analytical expectation from (1) within 5%, confirming that the static transfer model remains accurate across NAND organizations.

We observe that throughput varies significantly across different *tR* values and NAND organization parameters, indicating that these parameters strongly influence achievable performance. This tunability enables researchers to explore SSD configurations that meet the performance requirements of their GPU-centric storage systems and evaluate how future storage architectures may influence application performance. For the remainder of this paper, we use *tR* = 15 us, 16 channels, and 32 planes per channel as the default model parameters for evaluating GPU-initiated I/O workloads.

V. DSA-ACCELERATED DATA MOVEMENT

As discussed in **L2** of Section III, NVMeVirt's current DMA engine is inadequate for supporting GPU-initiated I/Os where data buffers and queue pairs are placed in GPU memory because its synchronous I/OAT-based DMA mechanism is not designed for highly concurrent, fine-grained I/O requests from GPUs. Furthermore, for small I/O requests, the overhead of DMA request initialization can account for a substantial fraction of the command processing time, thereby reducing achievable IOPS below the target level. To overcome this limitation, we fully redesign the DMA mechanism in NVMeNext by leveraging Intel's on-chip DSA engine.

A. ANALYSIS OF DSA PERFORMANCE

Prior to adopting DSA for data movement acceleration, we first characterize its raw performance by measuring data copy operations using a single DSA device. The experiment is conducted on the same testbed used in Section IV-B, where each CPU processor provides a single DSA device supporting up to 8 work queues (WQs). We use 32 job threads to issue data copy operations to the DSA device, distributing them across 8 WQs so that each WQ is shared by 4 threads. To ensure stable throughput measurements, all threads are pinned to the same NUMA node as the DSA device.

Table 3 presents the measured throughput of 512 B data copy operations under various configurations. The term *Memory Domain* refers to the locations of the source and destination buffers; for example, CPU → GPU indicates a 512 B copy from CPU memory to GPU memory. In Config A, we measure throughput using the Linux kernel DMA subsystem APIs (*subsystem APIs* for short), which provide an abstracted API for DSA, to copy data between two CPU-side buffers. Surprisingly, the measured throughput is only 2.7 MIOPS, which falls short of our expectations.

This is due to several limitations of the subsystem API. First, although DSA provides both interrupt-based and polling-based completion handling, the subsystem API forces the use of interrupt-based completion by setting the `DMA_COMPLETION_NO_ORDER` flag when using the DSA. This flag indicates that the abstracted DMA engine may complete work descriptors out of submission order. Therefore, it strictly requires users to receive completion by interrupt and identify each completion using a unique identifier [27]. However, DSA can guarantee in-order completion under certain conditions when descriptors are submitted to the same WQ without intermixing descriptor types, that is, when all submitted descriptors are either single work descriptors or batch descriptors [42]. Nevertheless, the subsystem APIs for DSA do not expose a polling-based completion option, even though polling may be more effective than interrupt-based completion for latency-sensitive, small DMA transfers [43].

Second, the subsystem API inevitably forces users to invoke `device_prep_dma_memcpy()` for descriptor allocation and initialization on every DMA operation, even

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

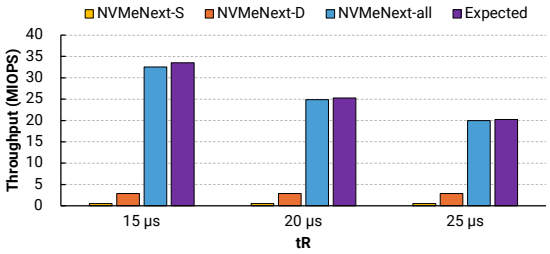


FIGURE 9. Ablation study of batched request processing in NVMeNext under *tR* values of 15, 20, and 25 us.

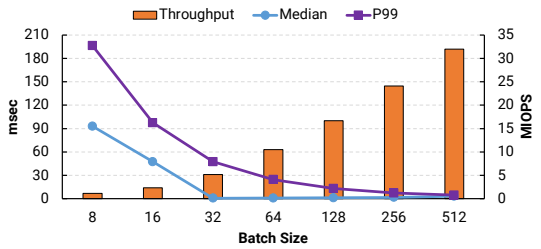


FIGURE 10. Effect of the batch size on the throughput and latency of batched request processing.

though allocated descriptors could be reused. It incurs redundant preparation overhead that non-trivially increases command processing time and degrades overall throughput.

To better utilize a DSA device, we decide to directly use low-level DSA APIs and apply several optimizations. By using low-level APIs, we adopt polling-based completion handling while complying with the conditions for in-order completion. Moreover, we allow each job thread to reuse its descriptor across requests, thereby eliminating redundant descriptor setup overhead. With the above optimizations, Config B achieves 39 MIOPS, which is 14.4× higher than Config A.

However, this throughput is not sustained when the memory domain changes to CPU → GPU, as shown in Config C. This is because accesses to GPU-side data buffers by DSA must traverse the PCIe hierarchy and incur additional latency compared with accesses to CPU memory, leading to lower throughput. To further derive the full performance potential of DSA in a GPU-centric environment, Config D exploits the batch-processing capability of the device by aggregating up to 512 data copy operations. The result reaches 92 MIOPS, demonstrating that the DSA engine can be effectively utilized to emulate GPU-initiated I/O.

B. REQUEST PROCESSING WITH BATCHING CAPABILITY

1) Design and Rationale

As shown in Fig. 4, NVMeNext exploits the batching capability of the DSA engine for data transfers between host buffers and reserved backend memory. To this end, each manager maintains its own DSA context which includes a DSA channel, a reusable batch descriptor, and the current batch-processing state. Compared with issuing multiple individual descriptors for data copy operations, this design reduces the number of DMA submissions and completion checks, while also lowering DMA initialization overhead.

To further improve overall performance, NVMeNext also uses DSA-based batching to transfer multiple SQ and CQ entries in bulk. Because NVMe SQ and CQ entries are small (64 B and 16 B, respectively), issuing a separate DMA operation for each entry is inefficient, as the DMA setup overhead can be comparable to, or even exceed, the actual transfer cost itself. Thus, NVMeNext also utilizes the batching capability of DSA to optimize command fetching and completion handling and maintains internal buffers for SQ and CQ entries.

During Phase 1, multiple contiguous SQ entries across multiple SQs are fetched from GPU memory into the SQ entry buffer in a single batched transfer, after which each command is fed into the performance model to determine its target completion time. Once the target completion time of an I/O request has elapsed, its corresponding completion entry is temporarily stored in the CQ entry buffer. Then, in Phase 3, the accumulated CQ entries are flushed to GPU memory using a single batched transfer, in the same manner as command fetching. Because massively parallel GPU threads issue I/O requests concurrently and maintain many in-flight requests, batched transfers are particularly effective at amortizing per-request transfer overhead.

2) Evaluation

Impact of batched request processing. To evaluate the benefit of batched request processing, we run a 512 B random read workload of BaM, in which GPU threads directly read data on demand. In our experiments, NVMeNext is configured to use DSA engines located on the same NUMA node as the GPU. Fig. 9 shows the performance differences among NVMeNext configurations with different levels of batching applied across the request-processing phases. Unless otherwise noted, all configurations aggregate up to 512 operations per batch descriptor.

In the figure, NVMeNext-S denotes the configuration that only adopts NVMeNext's scalable multi-manager architecture described in Section IV-B without DSA acceleration, whereas NVMeNext-D applies DSA batch processing only to data transfers in Phase 2. Finally, NVMeNext-all additionally applies batching to command fetching and completion handling in Phase 1 and Phase 3, which corresponds to the final design proposed in this paper. For all configurations, we enable the performance model to examine how variations in *tR* affect the achievable throughput. Along with the results for NVMeNext-all, we also compare its throughput with the ideal throughput calculated from (1) for three values of *tR*, while fixing the remaining parameters to 16 channels and 32 planes per channel.

Compared to NVMeNext-S, NVMeNext-D improves throughput by 5.5× through batched data movement. However, it still falls well short of the target performance of next-generation SSDs, and the effect of *tR* remains largely hidden by other dominant overheads. In contrast, NVMeNext-

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

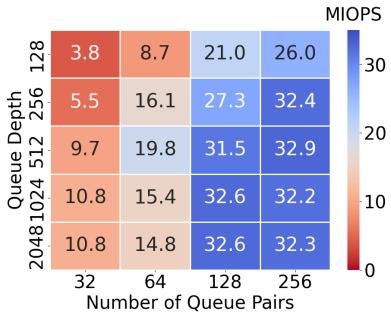


FIGURE 11. Impact of the numbers of NVMe queue pairs and queue depths on throughput.

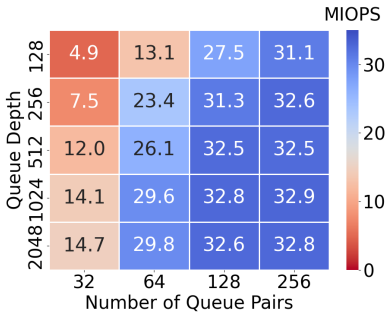


FIGURE 12. Throughput variation with warp-level coalescing.

all achieves 11.2× higher throughput at tR=15 us. Only NVMeNext-all exhibits throughput trends that reflect variations in tR while approaching the expected throughput, which indicates that efficient processing of SQ and CQ entries is a key factor in meeting the target performance.

Fig. 10 shows the effect of the batch size on the throughput and latency of batched request processing in the NVMeNext-all. We observe that larger batches amortize the fixed per-iteration cost of descriptor enqueue and completion polling across more operations. Thus, throughput scales linearly before saturating at 33 MIOPS from a batch size of 512 onward. The effect on median and P99 tail latency is more pronounced. At small batch sizes, latency reaches several hundred milliseconds because the manager threads fail to drain queued requests promptly, stalling GPU threads until free entries become available in the queue. As the batch size increases, the improved request-processing rate effectively mitigates this queuing delay. Consequently, at sufficiently large batch sizes, latency is dominated by device service time rather than queuing delay, with the median and P99 tail latencies converging to 3 ms and 4 ms, respectively.

VI. SYSTEM-LEVEL EVALUATION

To assess the usefulness of NVMeNext as a scalable emulation platform for GPU-initiated I/O and to derive system-level implications from it, we conduct various system-level evaluations leveraging workloads in the open-sourced BaM benchmarks [44]. Rather than merely modeling device-side latency and bandwidth, NVMeNext allows us to further study how cross-layer analysis guides SSD design principles and jointly determine end-to-end I/O throughput.

The benchmarks are conducted on the same testbed described in Section IV-B, which is equipped with an NVIDIA A100 PCIe 40 GB GPU. To minimize performance interference from NUMA effects, we use DSA engines located on the same NUMA node as the GPU.

A. IMPACT OF THE NVME QUEUE PAIR TOPOLOGY

Since GPU-initiated I/O exploits massive thread-level parallelism to issue a large number of I/O requests concurrently, the SSD must provide sufficient queue depth at all times to sustain the target throughput. While the minimum total queue depth can be derived from Little's Law [45] as also discussed in BaM, the same requirement must be revisited in the context of an NVMe SSD emulator. This is because the total queue depth is determined by the number of queue pairs multiplied by the queue depth of each queue pair. These two factors are configured by the NVMe device driver (e.g., *libnvm* [46] in BaM) when a user launches a GPU application and such a queue pair topology directly affects end-to-end I/O throughput even for the same total queue depth, depending on how it is composed.

To quantitatively examine the impact of queue pair configuration, we run a random read workload of BaM while varying the number of queue pairs and the queue depth on NVMeNext. Fig. 11 presents a throughput heatmap over different combinations of queue pair count and per queue depth, (x: queue pair count, y: queue depth). We can see that, even with the same number of total queue depth, each combination shows substantial performance gap. For example, although (64, 2048) and (128, 1024) reserve the same number of 128 K total queue depth, the latter shows 2.2× higher throughput than the former. This is because the more GPU threads share the same queue pair, the greater the synchronization overhead incurred in updating the SQ tail for I/O submission and per-queue pair doorbell.

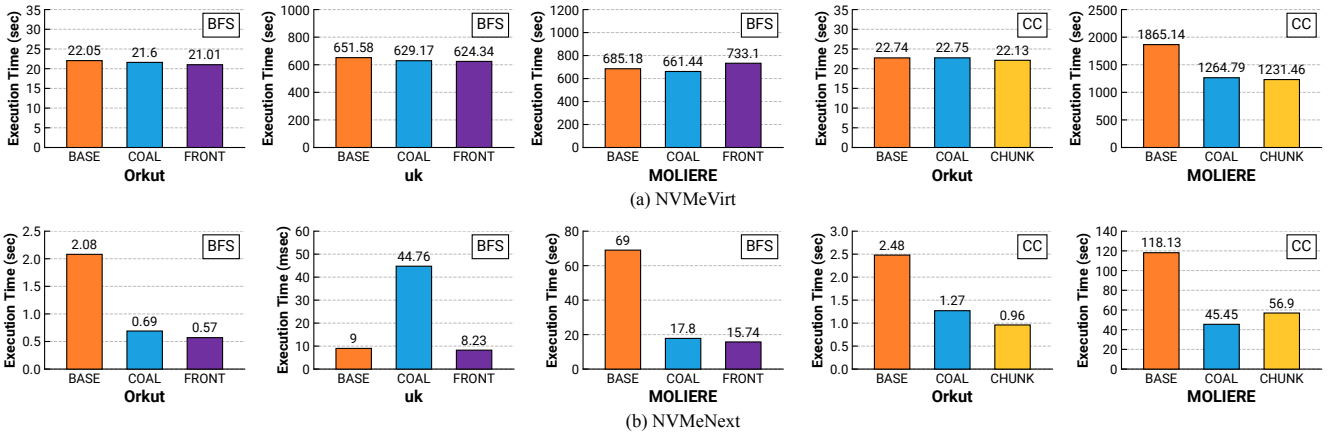
These results lead to a clear system-level implication: under GPU-initiated I/O, throughput is governed not only by the total queue depth, but also by how that depth is distributed across queue pairs. Because a real SSD can support only a limited number of queue pairs due to device-internal resource constraints, this tradeoff directly affects SSD resource provisioning and end-to-end I/O performance. NVMeNext is particularly useful to examine this correlation because its software-defined architecture allows us to explore queue-pair topologies beyond typical hardware limits and derive practical provisioning guidelines for future SSDs designed for GPU-centric storage systems.

B. WARP-LEVEL COALESCING

Through the experiment in Section VI-A, we observed that the achievable throughput depends not only on the total queue depth, but also on the specific combination of queue pair count and per queue depth due to the synchronization

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776



(a) NVMeVirt

(b) NVMeNext

FIGURE 13. Execution time of two graph analytics workloads across three graph datasets on NVMeVirt and NVMeNext.

TABLE 4. Graph analytics datasets.

Graph	# Nodes	# Edges	File Size
`Orkut` [47]	3.1 M	234.3 M	1.7 GiB
`uk-2007-05` [48]	105.9 M	3.37 B	66 GiB
`MOLIERE_2016` [49]	30.2 M	6.67 B	83 GiB

overhead among GPU threads. To mitigate this overhead, we adopt *warp-level coalescing*, an application-level optimization that designates the lane-0 GPU thread in each warp as the leader, which coordinates I/O submissions and completions, and handles critical-section operations (e.g., SQ slot allocation, command ID assignment, and SQ tail/CQ head doorbell updates) on behalf of the other threads in the same warp. This reduces the number of GPU threads entering the critical section, thereby alleviating contention on shared NVMe queues and doorbell registers.

To quantify this effect, we run the same random read workload under the same queue topologies as in Section VI-A, while enabling warp-level coalescing. Compared with the no-coalescing case in Fig. 11, the throughput of most configurations in Fig. 12 improves by up to 101%, and even configurations with smaller total queue depths achieve saturated throughput. This result shows that reducing GPU-side synchronization overhead can effectively relax the SSD-side queue-depth requirement for sustaining high throughput. In other words, achieving the target throughput depends not only on device-side properties, but also on GPU-side I/O optimizations. Overall, NVMeNext enables cross-layer analysis spanning next-generation SSD and GPU applications, thereby providing system-level insights for the design of GPU-centric storage systems.

It is worth noting that, as shown in Fig. 11 and Fig. 12, NVMeNext successfully reaches a throughput ceiling of 33 MIOPS for GPU workloads under the configured model parameters (i.e., tR = 15 us, 16 channels, and 32 planes per channel). This demonstrates that NVMeNext can serve as a MIOPS-scale NVMe SSD emulator for GPU-centric storage systems without being bottlenecked by emulator-side overheads. However, we identified that scaling beyond this point requires higher aggregate DSA throughput than our testbed can provide. Although the DSA device in our testbed can sustain up to 92 MIOPS with batching capability, as shown in Table 3, each NVMe command requires multiple DSA operations for command fetching, data movement, and completion handling. Therefore, although we were unable to measure throughput at this scale due to a lack of hardware resources, we expect NVMeNext to scale toward 100 MIOPS-class throughput on servers provisioned with more DSA devices such as the Intel Xeon 6732P platform [50].

C. GRAPH ANALYTICS WORKLOADS

To demonstrate NVMeNext's usefulness when evaluating real-world workloads, we run graph analytics workloads of BaM using Breadth-First Search (BFS) and Connected Components (CC) while varying coalescing mechanisms. We use three graph datasets listed in Table 4: `Orkut`, `uk-2007-05` (`uk`), and `MOLIERE_2016` (`MOLIERE`). We exclude `uk` from the CC experiments because the CC implementation assumes an undirected graph.

We measure the average execution time by running each workload on three source vertices with more than two neighbors. By default, a single GPU thread is assigned to each vertex and sequentially scans all edges of that vertex (BASE). We also evaluate edge coalescing (COAL), in which a single warp processes all edges of a vertex to improve thread parallelism and memory access efficiency. On top of COAL, we further apply workload-specific coalescing optimizations, frontier coalescing (FRONT) for BFS and chunk coalescing (CHUNK) for CC. For BFS, frontier coalescing maintains a separate frontier queue that stores vertices to be explored in subsequent traversal steps, thereby guiding the progression of the traversal. For CC, chunk coalescing assigns eight adjacent vertices to a single warp, which reduces redundant GPU kernel launch overhead and improves warp utilization.

Fig. 13 compares the execution time of graph analytics

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

workloads on NVMeVirt and NVMeNext. Both emulators use BaM's default configuration of 128 NVMe queue pairs, each with a queue depth of 1024. In Fig. 13(a), NVMeVirt shows only marginal execution time variation across coalescing schemes because its emulator-side bottleneck masks the benefits of GPU-side optimizations. These results underscore the need for a scalable SSD emulator that delivers stable, high-throughput performance, allowing researchers to accurately characterize application behavior in GPU-centric storage systems.

In contrast, Fig. 13(b) shows that NVMeNext exhibits distinct results across datasets and coalescing configurations. For BFS with `Orkut` and `MOLIERE`, COAL achieves up to a 3.87× speedup over BASE, while FRONT further reduces execution time by 12% by avoiding redundant GPU-side processing of previously visited vertices. However, `uk` performs worse with COAL, as its sparsity with edge coalescing causes redundant GPU threads to be launched in a warp and reduces GPU execution efficiency.

For the CC workload, COAL achieves up to a 2.59× speedup over BASE. However, CHUNK does not consistently improve performance and lengthens execution time on `MOLIERE`. This is because `MOLIERE` contains a number of very high-degree vertices and CHUNK assigns eight adjacent vertices to a single warp for sequential processing. If a high-degree vertex is included in a chunk, it can dominate the chunk's execution time and introduce intra-warp serialization which forces the other vertices in the same warp to wait. These results further show that NVMeNext can execute realistic GPU workloads with sufficient emulation throughput and without prohibitive emulator-side overhead, while enabling quantitative analysis of the impact of GPU application-level optimizations.

VII. DISCUSSION

A. LIMITATIONS OF THE STATIC TRANSFER MODEL

The static transfer modeling assumes the condition where queueing delay on the NAND channels and PCIe link is negligible compared to NAND read latency (tR). This condition is naturally satisfied in GPU-centric storage workloads, which are dominated by small (512 B–4 KB) random reads where tR accounts for the vast majority of request latency. The same condition can be extended to mixed read/write workloads at small granularity: most writes are absorbed by the write buffer, and NAND program latency likewise dominates the write buffer flush time. Moreover, because PCIe bandwidth roughly doubles with each successive generation, its higher bandwidth broadens the regime in which the static transfer model remains valid.

In contrast, large I/O sizes that drive the aggregate bandwidth close to channel or PCIe capacity make transfer time no longer negligible, causing statically computed latencies to underestimate queuing delay. Accurately emulating such workloads would require incorporating dynamic contention modeling, such as NVMeVirt's token-based scheme, at the cost of the scalability benefits. Since such scenarios fall outside the design target of GPU-centric storage systems, we adopts static transfer modeling for NVMeNext.

B. FIDELITY RELATIVE TO A PHYSICAL NVME ENDPOINT

NVMeNext emulates an NVMe SSD entirely in software using host-side manager threads and the DSA engine. Therefore, it is not obviously identical to a physical NVMe endpoint performing SSD-to-GPU P2P DMA. In this section, we clarify which aspects of endpoint behavior NVMeNext reproduces exactly, and which it approximates.

Interface-level behavior. From the GPU's perspective, NVMeNext exercises the same NVMe-over-PCIe transaction sequence as a physical SSD: doorbells are updated through posted writes, SQ entries are fetched through DMA reads, and data and CQ entries are delivered through DMA writes. Furthermore, NVMeNext executes the entire data plane by a hardware DMA engine of DSA rather than CPU load/store instructions, mirroring behaviors of a physical SSD in which firmware orchestrates the control plane while dedicated DMA engines move data. Consequently, GPU-initiated I/O stacks run on NVMeNext without any modification.

PCIe requester behavior and routing. Although the transaction sequence matches that of a physical SSD, the requester's location in the PCIe fabric and the resulting route path differ. A physical SSD issues P2P transfers from an endpoint behind a root port or PCIe switch and the data does not traverse host DRAM. In contrast, NVMeNext issues transfers through a root-complex-integrated DSA engine using data stored in backend DRAM. The GPU-side segment of the route, namely the GPU's own link and the inbound path into GPU memory, is the only segment visible to GPU-side software and is exercised identically in direction, volume, and destination. Reading the backend DRAM corresponds to the internal media-to-buffer transfer of a real SSD.

The SSD-side segment, including switch- or root-complex-level peer forwarding whose efficiency varies across platforms, is not physically reproduced. Instead, its bandwidth constraint is instead represented by the PCIe-link parameter of the performance model (Section IV-C).

NUMA placement. NVMeNext can reflect NUMA effects by using a DSA engine attached to a specified NUMA node, allocating the reserved backend memory on that node, and pinning the manager threads to CPUs on the same node. These placement controls allow NVMeNext to approximate the NUMA-affinity relationship between a physical SSD and the GPU. Co-locating them with the GPU approximates a deployment in which the SSD and GPU are attached to the same NUMA node, whereas remote placement introduces additional cross-socket traffic. In our evaluation, we co-locate all three components on the same NUMA node as the GPU. When the DSA engine is placed on a remote NUMA node, throughput decreases by up to 13% because of the resulting cross-socket traffic.

C. PORTABILITY TO OTHER CPU PLATFORMS

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

Currently, the NVMeNext implementation is based on Intel Xeon platform with DSA available starting with the 4th Gen Xeon Scalable family. However, only the batched data-movement layer described in Section V is specific to DSA. The multi-manager architecture, phase-based request processing, and the NAND performance model with per-plane locking are platform-agnostic. On platforms without a comparable DMA engine, the memcpy engine provides an alternative, although its throughput depends on the cost of accessing GPU memory.

On AMD EPYC platforms equipped with PTDMA [51] or AE4DMA [52] engines, the Linux kernel DMA subsystem provides a potential interface for porting the data-movement layer of NVMeNext [27]. Achieving throughput comparable to DSA would require efficient GPU-memory access and submission mechanisms that amortize per-transfer overhead, such as descriptor batching in Section V-B.

On ARM platforms, emerging cache-coherent CPU–GPU interconnects offer another implementation path. For example, NVIDIA Grace Hopper [53] connects the CPU and GPU through NVLink-C2C [54], enabling direct CPU access to appropriately mapped GPU memory. This chip-to-chip connectivity could allow manager threads to access GPU-resident NVMe queues using load/store instructions and transfer data using memcpy at a rapid speed, reducing the need for a separate DMA engine. Porting NVMeNext to these platforms and quantitatively evaluating the resulting performance remain future work.

VIII. CONCLUSION

We present NVMeNext, a scalable emulation platform for GPU-centric storage systems, which addresses the challenges of existing emulators under highly concurrent, small-sized requests demanded by the GPU-centric storage systems. First, NVMeNext proposes a multi-manager architecture that unifies the dispatcher and worker roles into a single thread, eliminating dispatcher-worker synchronization overhead and achieving near-linear throughput scaling. Second, we introduce a global NAND performance model with per-plane fine-grained locking and static transfer modeling, which preserves the accuracy while enabling concurrent access from multiple managers. Third, we adopt DSA-based batched request processing across all phases of the end-to-end pipeline, significantly accelerating both data transfers and queue accesses. Through system-level evaluations, we demonstrate that NVMeNext enables full system-level analysis while providing actionable insights for future GPU-centric storage systems research.

REFERENCES

[1] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2018. DOI: 10.1109/TPAMI.2018.2889473.

[2] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang, "Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 4236–4247, IEEE, 2024.

[3] vllm-project, "vllm: Easy, fast and cheap llm serving for everyone." https://github.com/vllm-project/vllm, 2023. Accessed: Nov. 1, 2026.

[4] sgl-project, "Sglang." https://github.com/sgl-project/sglang, 2024. Accessed: Nov. 1, 2026.

[5] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. 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*, pp. 1–14, 2021.

[6] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu," in *International Conference on Machine Learning*, pp. 31094–31116, PMLR, 2023.

[7] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[8] A. Singh, S. J. Subramanya, R. Krishnaswamy, and H. V. Simhadri, "Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search," *arXiv preprint arXiv:2105.09613*, 2021.

[9] B. Tian, H. Liu, Y. Tang, S. Xiao, Z. Duan, X. Liao, H. Jin, X. Zhang, J. Zhu, and Y. Zhang, "Towards high-throughput and low-latency billion-scale vector search via cpu/gpu collaborative filtering and re-ranking," in *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pp. 171–185, 2025.

[10] Z. Qureshi, V. S. Mailthody, I. Gelado, S. Min, A. Masood, J. Park, J. Xiong, C. J. Newburn, D. Vainbrand, I.-H. Chung, *et al.*, "Gpu-initiated on-demand high-throughput storage access in the bam system architecture," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 325–339, 2023.

[11] Z. Song, J. Zhang, J. Sun, M. Sun, Z. Yang, Z. Zhang, X. Chen, F. Wu, H. Tang, and Z. Wang, "Cam: Asynchronous gpu-initiated, cpu-managed ssd management for batching storage access," in *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pp. 2309–2322, IEEE, 2025.

[12] C.-H. Chang, J. Han, A. Sivasubramaniam, V. Sharma Mailthody, Z. Qureshi, and W.-M. Hwu, "Gmt: Gpu orchestrated memory tiering for the big data era," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 464–478, 2024.

[13] J. Han, A. Sivasubramaniam, C.-H. Chang, V. S. Mailthody, Z. Qureshi, and W.-M. Hwu, "Asynchrony and gpus: Bridging this dichotomy for i/o with agio," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 208–222, 2026.

[14] nVIDIA, "Speed-of-light data movement between storage and the gpu." https://www.nvidia.com/en-us/on-demand/session/gtc25-s73012/, 2025.

[15] SK Hynix, "Sk hynix and sandisk begin global standardization of next-generation memory 'hbf'." https://news.skhynix.com/sk-hynix-and-sandisk-begin-global-standardization-ofnext-generation-memory-hbf/, 2025.

[16] Kioxia, "Xl-flash: Storage class memory (scm)." https://americas.kioxia.com/en-us/business/memory/xlflash.html, 2026.

[17] C. J. Newburn, "Opening pathways to storage-driven insight." https://www.spectrumscaleug.org/wp-content/uploads/2025/06/ISC-05-NVIDIA-Opening-pathways-to-storage-driven-insight.pdf, 2025.

[18] Micron, "Sc25 performance breakthrough: 230m iops in a single server." https://www.micron.com/about/blog/storage/ssd/sc25-performance-breakthrough-230m-iops-in-a-single-server, 2025.

[19] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The case of femu: Cheap, accurate, scalable and extensible flash emulator," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pp. 83–90, 2018.

[20] K. T. Malladi, M. Awasthi, and H. Zheng, "Flexdrive: A framework to explore nvme storage solutions," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 1115–1122, IEEE, 2016.

[21] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for flash-based ssds with nameless writes," in *FAST*, p. 1, 2012.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776

[22] S.-H. Kim, J. Shim, E. Lee, S. Jeong, I. Kang, and J.-S. Kim, "Nvmevirt: A versatile software-defined virtual nvme device," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pp. 379–394, 2023.

[23] Y. H. Song, S. Jung, S.-W. Lee, and J.-S. Kim, "Cosmos openssd: A pcie-based open source ssd platform," *Proc. Flash Memory Summit*, pp. 1–30, 2014.

[24] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, "Cosmos+ openssd: Rapid prototype for flash storage systems," *ACM Transactions on Storage (TOS)*, vol. 16, no. 3, pp. 1–35, 2020. DOI: 10.1145/3385073.

[25] nVIDIA, "Gpudirect storage." https://docs.nvidia.com/gpudirect-storage/, 2026.

[26] Intel, "Intel® data streaming accelerator (intel® dsa)." https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/data-streaming-accelerator.html, 2026.

[27] The Linux Kernel, "Dmaengine controller documentation." https://docs.kernel.org/driver-api/dmaengine/provider.html, 2026.

[28] Intel, "idxd-config: Utility library for dsa and iaa." https://github.com/intel/idxd-config, 2021. Accessed: Nov. 1, 2026.

[29] R. Kuper, I. Jeong, Y. Yuan, R. Wang, N. Ranganathan, N. Rao, J. Hu, S. Kumar, P. Lantz, and N. S. Kim, "A quantitative analysis and guidelines of data streaming accelerator in modern intel xeon scalable processors," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 37–54, 2024.

[30] Parallel Data Lab, "The disksim simulation environment (v4.0)." https://www.pdl.cmu.edu/DiskSim/index.shtml, 2008.

[31] K. T. Malladi, M.-T. Chang, D. Niu, and H. Zheng, "Flashstoragesim: Performance modeling for ssd architectures," in *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pp. 1–2, IEEE, 2017.

[32] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "Mqsim: A framework for enabling realistic studies of modern multi-queue ssd devices," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pp. 49–66, 2018.

[33] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung, "Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 469–481, IEEE, 2018.

[34] K. Han, H. Gwak, D. Shin, and J. Hwang, "Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 147–162, 2021.

[35] S.-H. Kim, J. Shim, E. Lee, S. Jeong, I. Kang, and J.-S. Kim, "Empowering storage systems research with nvmevirt: A comprehensive nvme device emulator," *ACM Transactions on Storage*, vol. 19, no. 4, pp. 1–26, 2023. DOI: 10.1145/3625006.

[36] QEMU, "A generic and open source machine emulator and virtualizer." https://www.qemu.org/, 2026.

[37] Q. Zhang, J. Wang, Y. Zhou, P. Xu, K. Lu, J. Wan, F. Wu, and T. Lu, "Cemu: Enabling full-system emulation of computational storage beyond hardware limits," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 323–341, 2026.

[38] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 154–161, 2017.

[39] Intel, "Accelerating high-speed networking with intel® i/o acceleration technology." https://www.intel.com/content/dam/doc/white-paper/i-o-acceleration-technology-paper.pdf, 2006.

[40] J. Axboe, "fio: Flexible i/o tester." https://github.com/axboe/fio, 2005. Accessed: Nov. 1, 2026.

[41] Tech Power Up, "Micron 9650 max 6 tb." https://www.techpowerup.com/ssd-specs/micron-9650-max-6-tb.d2694, 2026.

[42] Intel, "Intel® data streaming accelerator architecture specification." https://cdrdv2.intel.com/v1/dl/getContent/671116, 2026.

[43] J. Yang, D. B. Minturn, and F. T. Hady, "When poll is better than interrupt," in *FAST*, vol. 12, pp. 3–3, 2012.

[44] ZaidQureshi, "Bam." https://github.com/ZaidQureshi/bam, 2023. Accessed: Nov. 1, 2026.

[45] J. D. Little and S. C. Graves, "Little's law," in *Building Intuition: Insights from Basic Operations Management Models and Principles*, pp. 81–100, Springer, 2008.

[46] J. Markussen, L. B. Kristiansen, P. Halvorsen, H. Kielland-Gyrud, H. K. Stensland, and C. Griwodz, "Smartio: Zero-overhead device sharing through pcie networking," *ACM Transactions on Computer Systems (TOCS)*, vol. 38, no. 1-2, pp. 1–78, 2021. DOI: 10.1145/3462545.

[47] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, pp. 1–8, 2012.

[48] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th International Conference on World Wide Web*, pp. 587–596, 2011.

[49] J. Sybrandt, M. Shtutman, and I. Safro, "Moliere: Automatic biomedical hypothesis generation system," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1633–1642, 2017.

[50] Intel, "Intel® xeon® 6732p processor." https://www.intel.com/content/www/us/en/products/sku/242641/intel-xeon-6732p-processor-144m-cache-3-80-ghz/specifications.html, 2026.

[51] S. R. Mehta, "dmaengine: ptdma: Initial driver for the amd ptdma." https://lkml.iu.edu/2108.0/00595.html, 2026.

[52] B. Natikar, "dmaengine: ae4dma: Add amd ae4dma controller driver." https://lore.kernel.org/r/20241025095931.726018-3-Basavaraj.Natikar@amd.com, 2026.

[53] nVIDIA, "Nvidia grace hopper superchip architecture in-depth." https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/, 2026.

[54] NVIDIA, "Nvlink-c2c." https://www.nvidia.com/en-us/data-center/nvlink-c2c/, 2026. Accessed: Nov. 3, 2026.



INJAE KANG received the B.S. degree in mechanical and aerospace engineering and the M.S. degree in computer science and engineering, both from Seoul National University, Republic of Korea, in 2020 and 2022, respectively. He is currently pursuing the Ph.D. degree in computer science and engineering with Seoul National University. His research interests include operating systems and storage systems.



JAEHOON SHIM received the B.S. degree in computer engineering from Sejong University, Republic of Korea, in 2019, and the M.S. degree in computer science and engineering from Seoul National University, Republic of Korea, in 2021. He is currently pursuing the Ph.D. degree in computer science and engineering with Seoul National University. His research interests include operating systems, storage systems, and database systems.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2026.3755776



KYU-JIN CHO received the B.S. degree in computer science and engineering from Sogang University, Seoul, Republic of Korea, in 2019, and the Ph.D. degree in computer science and engineering from Seoul National University (SNU), Seoul, Republic of Korea, in 2025. He is currently a postdoctoral researcher with the Department of Computer Science and Engineering, SNU. His research interests include storage systems, operating systems, and distributed systems.



JIN-SOO KIM (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer engineering from Seoul National University, Republic of Korea, in 1991, 1993, and 1999, respectively. He is currently a Professor with the Department of Computer Science and Engineering, Seoul National University (SNU), South Korea. Before joining SNU, he was a Professor with Sungkyunkwan University from 2008 to 2018, and an Associate Professor with the Korea Advanced Institute of Science and Technology from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a Senior Member of the research staff, and with the IBM T. J. Watson Research Center as an Academic Visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/