Performance Analysis of Different IO Methods between GPU Memory and Storage

Tarun SREEPADA†, Tsuyoshi OZAWA††, Kiran UDAY RAGE†, and Kazuo GODA††

† The University of Aizu, 〒 965-0006 Fukushima, Aizuwakamatsu, Itsukimachi Oaza Tsuruga, Kamiiawase-90

†† The University of Tokyo, 〒 153-8505 Tokyo, Meguro-ku, Komaba 4-6-1

E-mail: †{m5281045,udayrage}@u-aizu.ac.jp, ††{ozawa,kgoda}@tkl.iis.u-tokyo.ac.jp

Abstract BaM, a new GPU system architecture, enables efficient, GPU-driven data transfers between Solid-State Drives (SSDs) and GPU memory by bypassing the CPU. Unlike traditional CPU-managed approaches that rely on sequential data parsing and memory transfers, BaM can leverage the GPU's parallelism and introduce optimized data pipelines for self-orchestrated storage access. It incorporates a fine-grained software cache to minimize I/O amplification and high-throughput queues to maximize system interconnect and storage utilization. This paper benchmarks BaM's performance using SSD microbenchmarks, demonstrating its ability to accelerate data-intensive workloads and improve overall system efficiency.

Key words GPU, GPUDirect, NVMe, SSDs, Memory Capacity, Benchmarking

1 Introduction

High-performance computing (HPC) workloads—including machine learning, scientific simulations, and real-time analytics—require substantial I/O resources to manage and process vast datasets efficiently. The introduction of compute accelerators, such as Graphics Processing Units (GPUs), has significantly improved processing speeds in computing clusters. However, data transfer between networked nodes and these accelerators remains a persistent challenge. Traditional CPU-managed transfer methods often create bottlenecks, as GPUs remain idle while waiting for the CPU to handle data movement, leading to increased latency and reduced system throughput.

Figure 1 illustrates the CPU-initiated data transfer process between an SSD and system memory using the NVMe protocol. The process begins when the CPU enqueues a read or write command into the Submission Queue (SQ) (Step 1) and notifies the NVMe controller by updating the doorbell register (Step 2). The controller then retrieves the command from the SQ (Step 3) and executes the requested read or write operation on the SSD (Step 4). Once completed, the controller updates the Completion Queue (CQ) and triggers an interrupt to notify the CPU (Step 5), ensuring efficient and low-latency communication between the CPU and SSD.

Once the data is copied to system memory, the CPU transfers it to the GPU. This typically involves staging the data in pinned memory—memory accessible to both the CPU and GPU—to facilitate efficient direct memory transfers. The CPU initiates a data copy to the GPU's memory using APIs such as CUDA's 'cudaMemcpy'. This transfer occurs over the PCIe bus, increasing the time required to move data from storage to the GPU.

As HPC, big data, and machine learning converge, the demand for scalable, low-latency I/O solutions continues to grow. In distributed computing environments, efficient data movement is critical to maximizing performance. Networking technologies like Remote Direct Memory Access (RDMA) offer a path toward high-speed data transfers but still introduce latency overhead due to protocol translations. Peripheral Component Interconnect Express (PCIe) remains the dominant standard for connecting storage and compute resources, and leveraging PCIe-based networking can help minimize performance bottlenecks by reducing protocol overhead. Additionally, emerging technologies such as GPUDirect Storage (GDS) allow GPUs to bypass the CPU entirely and access storage directly, reducing transfer latency and improving system efficiency. Addressing these challenges is essential for the next generation of high-performance and cloud computing architectures.

BaM (Big Accelerator Memory) [12, 15, 16] is a new GPU-driven system architecture designed to enable efficient data transfers between Solid-State Drives (SSDs) and GPU memory by bypassing the CPU. Unlike traditional CPU-centric approaches, BaM allows GPUs to initiate I/O operations directly, thereby reducing CPU involvement in essential tasks and leveraging the inherent parallelism of GPUs. This architecture integrates optimized data pipelines for self-

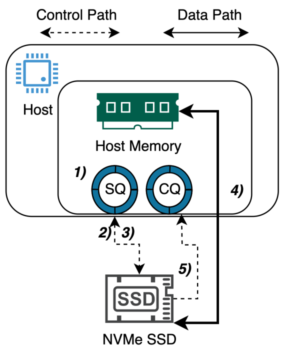


Figure 1: CPU initiated IO overview: (1) Enqueue read/write IO to SQ, (2) Doorbell update, (3) Controller retrieves command, (4) Read/write from/to SSD, (5) Update submission queue and trigger interrupt.

orchestrated storage access, a fine-grained software cache to minimize I/O amplification, and high-throughput queues to maximize system interconnect and storage utilization.

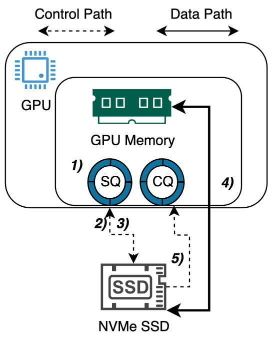


Figure 2: GPU(BaM) initiated IO overview: (1) Enqueue read/write IO to SQ, (2) Doorbell update, (3) Controller retrieves command, (4) Read/write from/to SSD, (5) Update submission queue and trigger interrupt.

In this work, we implement the BaM architecture within our system environment and develop benchmarks to evaluate its performance. Our benchmarks confirm that BaM can fully utilize SSD performance and achieve the expected throughput of the SSD. This demonstrates BaM's effectiveness in enhancing data transfer efficiency for GPU-intensive applications, mitigating CPU-induced bottlenecks, and improving overall system performance in HPC environments. By validating BaM's capabilities through our implementation and benchmarking efforts, we contribute to the ongoing efforts to optimize I/O performance in distributed, GPU-accelerated computing systems.

The remainder of this paper is organized as follows: Section 2 reviews background work on GPU-driven data transfer architectures and benchmarking methodologies. Section 3 describes our implementation of BaM and the benchmarking setup. Section 4 presents the benchmarking results and analysis. Finally, Section 5 concludes the paper and outlines future work.

2 GPUs in High-Performance Computing

Graphics Processing Units (GPUs) have become integral to high-performance computing (HPC) due to their ability to perform parallel computations efficiently [1, 10]. Initially designed for rendering graphics, GPUs have evolved to support various scientific and engineering applications, including machine learning, simulations, and data analytics. The massive parallelism offered by GPUs enables significant speedups for workloads that can exploit data-level parallelism, making them indispensable in modern HPC clusters.

2.1 GPU Memory and Its Limitations

Despite their computational prowess, GPUs are constrained by memory limitations, both in bus width and on-chip memory capacity [14]. The memory bandwidth of GPUs, determined by the width of the memory bus and the memory type, plays a crucial role in sustaining high computational throughput. For instance, NVIDIA's H100 GPU features a 5120-bit memory bus and 80GB of HBM2 memory, providing substantial bandwidth but still insufficient for vast datasets that are orders of magnitude larger [3]. These limitations necessitate efficient memory management and data transfer strategies to prevent memory bottlenecks and ensure that the GPU remains fully utilized [2].

2.2 GPU Memory Pooling

Various memory pooling techniques have been proposed to mitigate GPU memory constraints, including pinned and unified memory architectures. Pinned memory involves allocating a portion of the host memory directly accessible by the GPU, allowing faster data transfers through mechanisms like Direct Memory Access (DMA) [4]. On the other hand, Unified memory provides a single address space accessible by both the CPU and GPU, simplifying memory management and enabling more flexible data sharing [9]. These approaches aim to maximize memory utilization and reduce the overhead associated with data movement between the host and the GPU.

Additionally, leveraging multiple GPUs and pooling their memory using technologies like NVIDIA NVLink and NVSwitch allows for significant memory capacity and bandwidth scaling. NVLink enables high-speed interconnects between GPUs, facilitating seamless data sharing and reducing the latency of cross-GPU memory access. However, implementing this approach requires an additional GPU and compatible hardware to support NVLink or NVSwitch. This includes specialized interconnects and potentially a motherboard that supports multi-GPU configurations. This approach alleviates memory bottlenecks and enhances parallel processing capabilities, enabling efficient execution of memory-intensive tasks across multiple GPUs [8].

2.3 GPU RDMA and GPUDirect Storage

Remote Direct Memory Access (RDMA) and NVIDIA's GPUDirect Storage are advanced techniques designed to enhance data transfer efficiency between GPUs and storage devices [7, 11]. RDMA allows direct memory access from one computer to another without involving the CPU, reducing latency and CPU overhead. GPUDirect Storage extends this concept by enabling GPUs to communicate directly with storage devices such as NVMe SSDs, bypassing the CPU and minimizing data transfer latency [6]. These technologies facilitate high-throughput and low-latency data transfers, essential for data-intensive applications.

2.4 libnvm

The libnvm library [13] is a userspace API implemented in C for developing custom NVM Express (NVMe) drivers and high-performance storage applications. Similar in concept to SPDK, it transfers driver logic to userspace, utilizes hardware polling, and leverages direct memory mapping to eliminate the overhead associated with kernel-space context switching. This architecture enables zero-copy access, substantially reducing I/O operation latency compared to traditional Linux kernel file system abstractions.

libnvm provides a low-level block interface with extremely low latency and supports PCIe peer-to-peer communication by allowing arbitrary memory mappings to device memory. Furthermore, it offers a lockless interface that can be shared across multiple computing instances, aligning with NVMe's design for modern parallel computing architectures. Notably, libnvm integrates seamlessly with CUDA programs, facilitating direct high-performance storage access from CUDA kernels. By placing I/O queues and data buffers directly in GPU memory, the library bypasses the CPU in the I/O path, thereby enhancing throughput and minimizing latency.

2.5 BaM: Big Accelerator Memory

Building upon the libnvm driver, BaM introduces enhancements in robustness, error checking, memory alignment, and performance optimization for handling large requests. Similar to the approach employed by SmartIO [13], BaM enables peer-to-peer data transfers between NVMe SSDs and GPU memory, bypassing the CPU entirely in the data path.

BaM [12,15,16] proposes a new GPU-driven architecture to address inefficiencies associated with traditional CPU-centric data transfer methods. To enable GPU threads to access data on NVMe SSDs directly, BaM employs the following key strategies:

1. **Migrating NVMe Queues and I/O Buffers to GPU Memory:** BaM relocates NVMe queues and I/O buffers from host CPU memory to GPU memory by leveraging GPUDirect RDMA APIs. These APIs pin and map the queues and buffers directly into the GPU's address space.

2. **Enabling GPU Threads to Interact with SSD Doorbells:** BaM permits GPU threads to write directly to the queue doorbell registers in the NVMe SSD's Base Address Register (BAR) space. This is accomplished using GPUDirect Async [5], which maps the NVMe SSD doorbells into the CUDA address space. GPU threads can then "ring" the doorbells on demand. The SSD's BAR space is first memory-mapped into the application's address space and subsequently mapped into CUDA's address space using the `cudaHostRegister` API.

Implementing BaM within our system environment involves developing comprehensive benchmarks to evaluate its performance. These benchmarks confirm that BaM can fully utilize SSD performance, achieving up to 100% of the expected throughput. This demonstrates BaM's effectiveness in mitigating CPU-induced bottlenecks and improving overall system performance in HPC environments.

3 GPU I/O Benchmark

3.1 Objective

This study evaluates whether the BaM architecture can meet the maximum performance of an NVMe SSD and examines how different GPU configurations (threads per block and number of blocks) affect SSD performance. The hardware configuration is listed in Table 1.

3.2 Experimental Setup

a) SQ/CQ and Page Cache

Using BaM APIs, Submission Queues (SQ) and Completion Queues (CQ) are initialized for direct GPU–SSD interaction. Page cache buffers are allocated in GPU memory to streamline read/write operations.

b) Access Methodology

Threads map to SSD pages via a GPU-side array. The CPU generates random page numbers for random access, while sequential access uses thread IDs directly.

c) GPU Execution

Threads are grouped into blocks; each warp handles one SQ. The kernel execution steps include (1) retrieving or computing a logical page number, (2) mapping it to physical SSD pages, (3) preparing the NVMe command, and (4) issuing the I/O operation. Latency is measured by recording each operation's start and end times in GPU clock cycles.

3.3 Preconditioning & Performance Testing

a) Preconditioning

The SSD is brought to a steady state with 4 KB random and 128 KB sequential read/write operations.

b) Testing Conditions

All tests maintain a constant total thread count. Each thread executes one I/O per iteration with a queue depth of 1024. Random operations use 4 KB page size, while sequential operations use 128 KB.

c) Variables and Metrics

We vary (1) threads per block, (2) number of blocks, and (3) total data accessed (1 GB, 16 GB, 64 GB). We measure:

- **Throughput:** IOPS and GB/s,
- **Latency:** GPU clock cycles.

Component	Description
GPU	NVIDIA H100 (80 GB HBM2e)
SSD	Micron 7450 Pro, 3.8 TB, PCIe Gen 4
CPU	Dual AMD EPYC 9224
Memory	377GB DDR5

Table 1: Hardware configuration

The GPU connects via PCIe Gen 5 (64 GB/s) and the SSD via PCIe Gen 4 (8 GB/s), enabling direct GPU–SSD interaction.

4 Experiments

4.1 Random Workloads

Random accesses use 4 KB pages. Threads per block vary (1, 4, 8, 16), with larger blocks omitted if run time exceeds 50 min. Table 3 summarizes the configurations.

$$\text{Number of Threads} = \frac{\text{Data Size}}{\text{Page Size}}$$

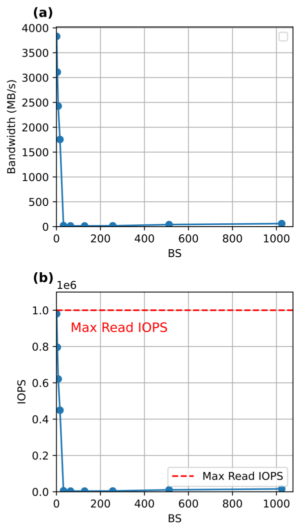
$$\text{Number of Blocks} = \frac{\text{Number of Threads}}{\text{Block Size}}$$

Total Data	Threads/Block	Blocks
1 GB	1–1024	Computed accordingly
16 GB	1, 4, 8, 16	Computed accordingly
64 GB	1, 4, 8, 16	Computed accordingly

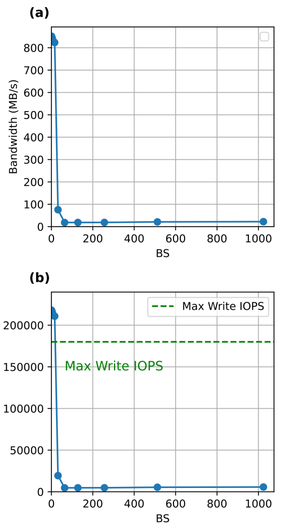
Table 2: Thread configurations for random workloads.

4.1.1 Random Read and Write Metrics

For 1 GB read and write tests (Figure 3), performance drops sharply once the block size exceeds 32. Below this threshold, random reads nearly reach the SSD's rated peak of 1 M IOPS(Figure 3a), and random writes slightly surpass the expected 180 K IOPS(Figure 3b). The sudden decline indicates that larger thread-block configurations adversely affect performance, likely due to warp-level scheduling constraints in CUDA. Hence, for all remaining experiments, block sizes

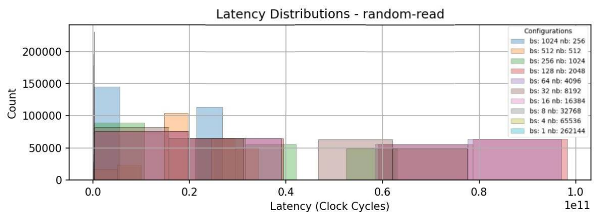


(a) Bandwidth and IOPS vs. Block Size (Read)

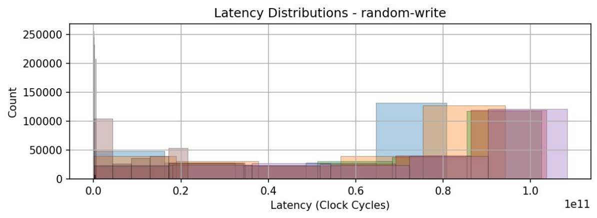


(b) Bandwidth and IOPS vs. Block Size (Write)

Figure 3: Random workload performance for 1GB data access: Read (left) and Write (right) Bandwidth and IOPS vs. Block Size.



(a) Latency histogram for Random Read.



(b) Latency histogram for Random Write.

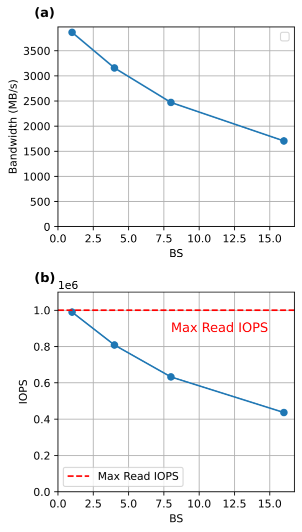
Figure 4: Latency distributions for random workloads for 1 GB access (Read and Write).

are limited to the range of 1–16.

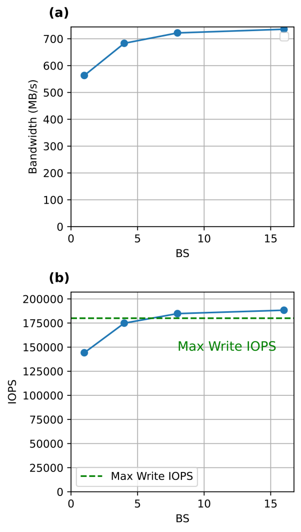
Analyzing the latency histogram(Figure 4) reveals substantial overlap for block sizes $\geqq 32$, whereas smaller block sizes appear as minor peaks concentrated on the left side of the graph.

Observing the 16 GB plot (Figure 5), a similar trend to the 1 GB case emerges. The initial read IOPS reaches the SSD's maximum before experiencing a sharp decline, whereas the write IOPS gradually increases until it reaches the SSD's peak performance.

The latency histogram for the 16 GB workload (Figure 6) reveals that the latency distribution becomes more spread

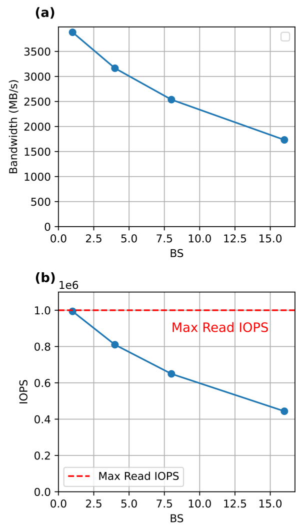


(a) Bandwidth and IOPS vs. Block Size (Read)

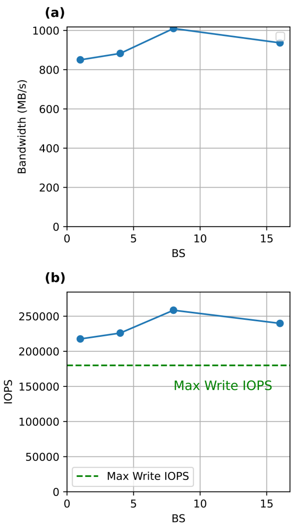


(b) Bandwidth and IOPS vs. Block Size (Write)

Figure 5: Random workload performance for 16GB data access: Read (left) and Write (right) Bandwidth and IOPS vs. Block Size.

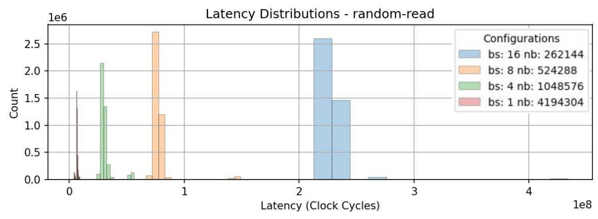


(a) Bandwidth and IOPS vs. Block Size (Read)

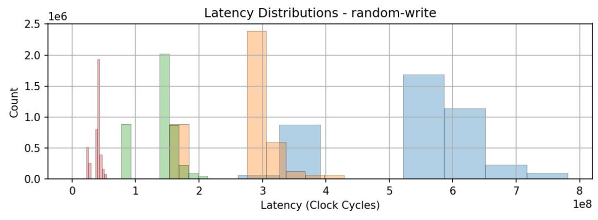


(b) Bandwidth and IOPS vs. Block Size (Write)

Figure 7: Random workload performance for 64GB data access: Read (left) and Write (right) Bandwidth and IOPS vs. Block Size.

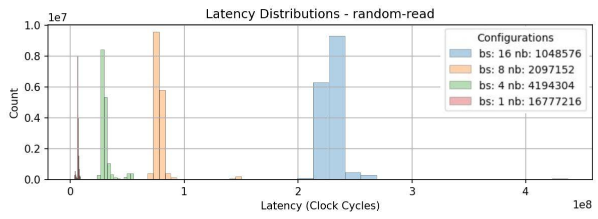


(a) Latency histogram for Random Read.

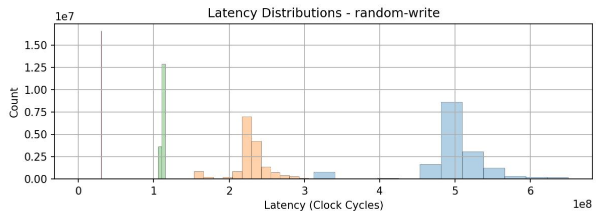


(b) Latency histogram for Random Write.

Figure 6: Latency distributions for random workloads for 16 GB access (Read and Write).



(a) Latency histogram for Random Read.



(b) Latency histogram for Random Write.

Figure 8: Latency distributions for random workloads for 64 GB access (Read and Write).

out as block sizes increase. This trend was not observable in the 1 GB case due to the higher number of block size variations, which made visualization more challenging.

The performance trends observed in the 1 GB and 16 GB experiments persist in the 64 GB workload(Figures 7 and 8). Although the measured IOPS exceeds the SSD's nominal maximum of 180 k, reaching approximately 250 k IOPS, this is an anticipated behavior. Factors such as SSD internal caching mechanisms can temporarily boost performance beyond the rated specifications. These reassuring results confirm that the experimental setup operates within the expected parameters.

Random workload experiments for 1 GB, 16 GB, and 64 GB show that SSD performance is highly sensitive to GPU thread block configuration. For 1 GB tests, random read IOPS nearly reach the SSD's peak with block sizes up to 32 before dropping sharply; latencies remain low for block sizes below 32. The 16 GB tests display a similar pattern, with an initial IOPS peak and more dispersed latencies as block sizes increase. In the 64 GB workload, trends persist, with IOPS reaching about 250 k—likely boosted by internal caching. These results validate our setup and justify limiting block sizes to 1–16 in future experiments.

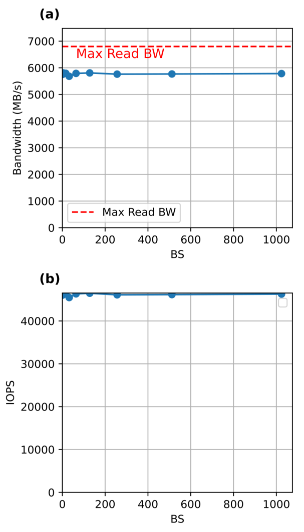
4.2 Sequential Workload

Sequential accesses use 128 KB pages. Threads per block vary (1, 4, 8,...). Table 3 summarizes the configurations.

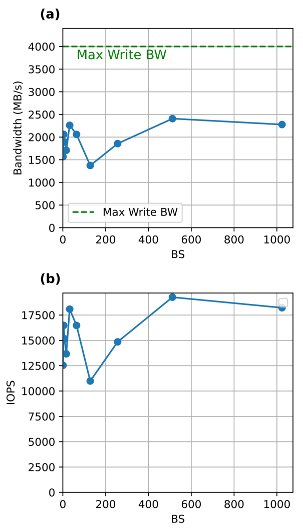
Total Data	Threads/Block	Blocks
1 GB	1–1024	Computed accordingly
16 GB	1–1024	Computed accordingly
64 GB	1–1024	Computed accordingly

Table 3: Thread configurations for random workloads.

4.2.1 Sequential Read and Write Metrics



(a) Bandwidth and IOPS vs. Block Size (Read)

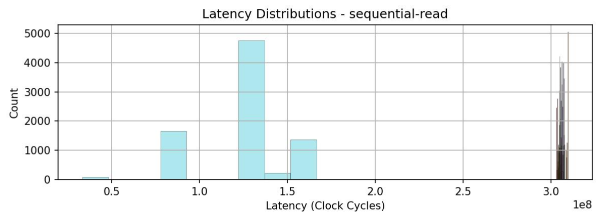


(b) Bandwidth and IOPS vs. Block Size (Write)

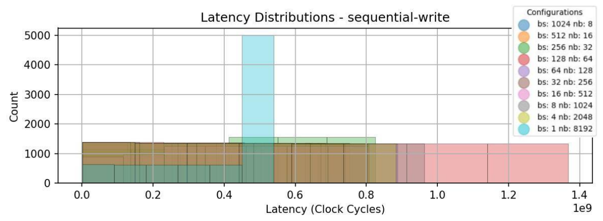
Figure 9: Sequential workload performance for 1GB data access: Read (left) and Write (right) Bandwidth and IOPS vs. Block Size.

For 1 GB sequential reads (Figure 9a, the observed peak performance reaches 6000 MB/s, slightly below the rated 6800 MB/s but close to full saturation. Sequential writes achieve 2500 MB/s out of the rated 4000 MB/s, which, while lower, remains within an expected range. Read performance remains consistently stable at 6000 MB/s across all block sizes, indicating that the SSD is fully saturated, and the benchmark is operating at its maximum capacity. However, write performance varies across block sizes, suggesting potential influences from thread scheduling, warp behavior, or other GPU-side execution dynamics.

Examining the latency histogram, read access latencies are predominantly clustered around 3×10^8 cycles, except block size 1, where some threads exhibit access latencies around 1.0×10^8. For sequential writes, latencies are more uniformly distributed between 1×10^8 and 1.4×10^9, with block size 1 again showing a subset of threads with access latencies around 5×10^8. This behavior is reminiscent of how block



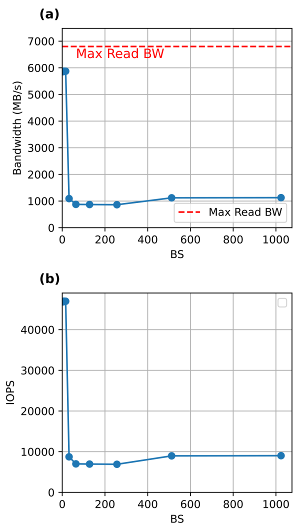
(a) Latency histogram for sequential Read.



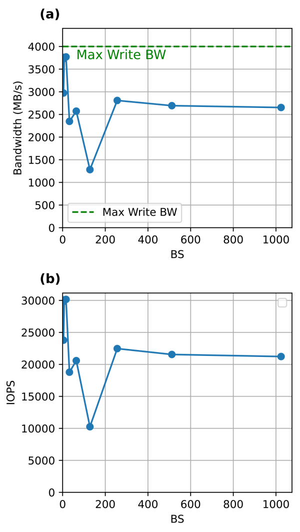
(b) Latency histogram for sequential Write.

Figure 10: Latency distributions for sequential workloads for 1GB access(Read and Write).

size 1 demonstrates relatively decent performance in random operations.



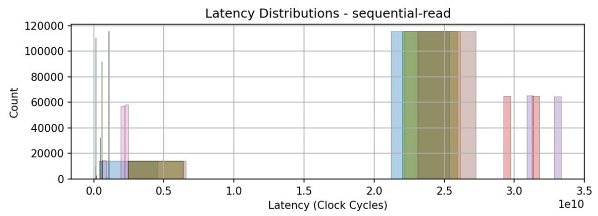
(a) Bandwidth and IOPS vs. Block Size (Read)



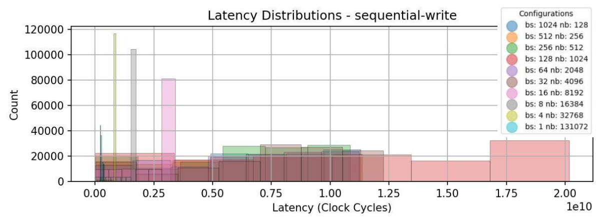
(b) Bandwidth and IOPS vs. Block Size (Write)

Figure 11: Sequential workload performance for 16GB data access: Read (left) and Write (right) Bandwidth and IOPS vs. Block Size.

For the 16 GB workload (Figure 11), read performance drops sharply beyond 32 block sizes, similar to the trend observed in random reads. However, there is a slight recovery at 512 and 1024 block sizes, though the increase is minimal. In contrast, write performance initially declines from 4000 MB/s to 1200 MB/s before recovering to approximately 2700 MB/s—still below its original peak but showing noticeable improvement. Unlike the random workload, where read and write performance degraded significantly, leading to the exclusion of larger data sizes, we continue measuring at higher block sizes (1024 and beyond) to observe performance trends without limiting the maximum block size to 16.



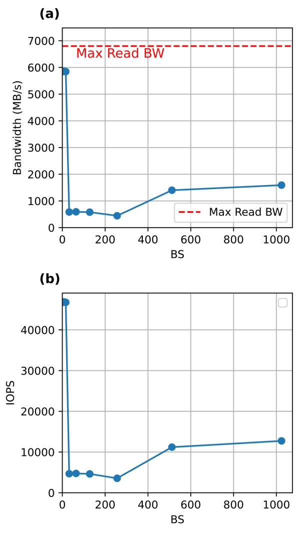
(a) Latency histogram for sequential Read.



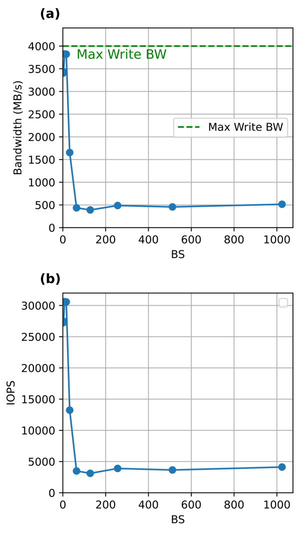
(b) Latency histogram for sequential Write.

Figure 12: Latency distributions for sequential workloads for 16GB access (Read and Write).

Read latencies (Figure 12) are primarily clustered around 1×10^9 and 2.5×10^{10}. The first cluster corresponds to block sizes smaller than 32, which exhibit faster read speeds, while the second cluster represents block sizes greater than 32, where latencies are significantly higher, leading to reduced read performance. In contrast, write latencies are more uniformly distributed, reflecting the relatively stable write bandwidth across most block-size configurations.



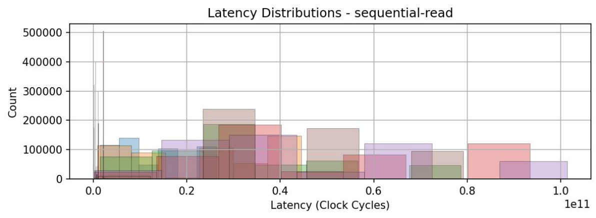
(a) Bandwidth and IOPS vs. Block Size (Read)



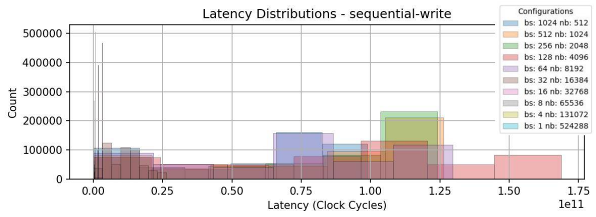
(b) Bandwidth and IOPS vs. Block Size (Write)

Figure 13: Sequential workload performance for 64GB data access: Read (left) and Write (right) Bandwidth and IOPS vs. Block Size.

For the 64 GB workload (Figure 13), both read and write performance decline without recovery, similar to the random workloads at 1 GB. Initially, performance approaches the SSD's rated capabilities before gradually deteriorating. Sequential reads exhibit a slight recovery, mirroring the behavior observed in the 16 GB workload, but overall, the trend remains one of sustained performance loss across larger block sizes.



(a) Latency histogram for sequential Read.



(b) Latency histogram for sequential Write.

Figure 14: Latency distributions for sequential workloads for 64 GB access (Read and Write).

Regarding latencies (Figure 14, the distribution is relatively uniform due to the consistent bandwidth and performance across most block sizes. However, block sizes smaller than 32 exhibits significantly lower latencies, as they achieve faster read and write speeds before the performance decline.

For the 1 GB sequential tests, read performance consistently saturates at around 6000 MB/s—just below the rated 6800 MB/s—while sequential writes average about 2500 MB/s. Read latencies are primarily concentrated around 3×10^8 cycles (with block size 1 even lower), whereas write latencies are more uniformly spread. In the 16 GB workload, read performance falls sharply beyond block sizes of 32, with only a minor recovery at very high block sizes, and write performance initially drops from 4000 MB/s to 1200 MB/s before partially recovering to roughly 2700 MB/s. Latency distributions in this case show two distinct clusters for reads (corresponding to block sizes below and above 32), while write latencies remain relatively uniform. For 64 GB, both read and write performances deteriorate further, with sequential reads showing only a slight recovery. Overall, the data indicate that while sequential read operations tend to saturate the SSD, write performance is more variable and strongly influenced by block size.

5 Conclusion

Our experiments reveal that random and sequential SSD I/O performance is susceptible to GPU thread block configurations and workload size. For the 1 GB tests, we observed that random read IOPS nearly reached the SSD's rated maximum when using block sizes up to 32, with performance dropping sharply beyond that threshold—likely due to warp-level scheduling constraints in CUDA. This observation and the clustered latency patterns for smaller block sizes led us to constrain subsequent experiments to block sizes in the range of 1–16.

In the 16 GB workload, similar trends persist: initial read IOPS peak before a steep decline, while write IOPS gradually increase toward the SSD's maximum. The latency distributions for reads and writes further illustrate the impact of block size, with larger block sizes yielding higher and more dispersed latencies. In contrast, sequential operations exhibit more consistent read performance across block sizes, although write performance shows significant variation—suggesting influences from threads scheduling and warp execution.

At the 64 GB level, both read and write performances initially approach the SSD's capabilities but then degrade continuously, with only a modest recovery in sequential read performance. The latency profiles for these larger workloads are generally uniform, except for the significantly lower latencies observed at block sizes below 32, which continue to deliver faster speeds.

Furthermore, our results demonstrate that BaM could saturate the SSD's IOPS and throughput in multiple cases. This highlights its efficiency in leveraging direct GPU-to-SSD communication for high-performance I/O. Moving forward, these findings suggest that carefully considering BaM's capabilities should be integrated into the design of future applications to maximize performance when utilizing GPU-driven storage access.

Overall, these results underscore the critical role of GPU thread configuration in optimizing SSD performance. The observed deviations—such as measured IOPS exceeding the nominal SSD ratings, likely due to internal caching effects—confirm that our experimental setup operates within the expected parameters. Fine-tuning block sizes, therefore, is essential for balancing throughput and latency, ensuring that the full potential of GPU-accelerated SSD I/O is realized.

6 Future Work

Future work will focus on enhancing our GPU-SSD system's scalability and performance. One key direction is developing a dynamic memory manager that supports in-kernel memory allocation via CUDA. Such a manager would enable the extension of GPU memory space onto the SSD through an efficient swapping mechanism, thereby facilitating the execution of larger workloads that exceed the native GPU memory capacity.

In addition, we plan to implement specialized database and file parsers designed to accelerate extensive file processing. This improvement is expected to boost overall throughput and reduce latency in data-intensive applications, enhancing the system's responsiveness and utility in real-world scenarios.

Together, these enhancements will broaden the applicability of our approach and address emerging challenges in high-performance computing.

Acknowledgments

This work was supported in part by Cross-ministerial Strategic Innovation Promotion Program (SIP) on Integrated Health Care System.

References

[1] NVIDIA HPC Application Performance.

[2] Anastasiia Arefyeva, David Broneske, and Gunter Saake. Memory management strategies in cpu/gpu database systems: A survey. In *14th International Conference BDAS 2018 Held at the 24th IFIP World Computer Congress WCC 2018*, pages 129–141. Springer, 2018.

[3] NVIDIA Corporation. Nvidia h100 tensor core gpu architecture, 2022. Accessed: 2025-01-06.

[4] NVIDIA Corporation. *CUDA C++ Best Practices Guide*, 2023.

[5] NVIDIA Corporation. *GPUDirect Async*, 2023. Accessed: 2025-01-06.

[6] NVIDIA Corporation. *GPUDirect RDMA*, 2023. Accessed: 2025-01-06.

[7] NVIDIA Corporation. *NVIDIA GPUDirect Storage Overview Guide*, 2023.

[8] NVIDIA Corporation. *NVIDIA NVLink and NVSwitch: Fastest HPC Data Center Platform*, 2023. Accessed: 2025-01-06.

[9] Mark Harris. Unified memory in cuda 6, 2013.

[10] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[11] Patrick MacArthur. An introduction to remote direct memory access, 2013.

[12] Vikram Sharma Mailthody. *Application Support And Adaptation For High-throughput Accelerator Orchestrated Fine-grain Storage Access.* PhD thesis, University of Illinois Urbana-Champaign, 2022.

[13] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. Smartio: Zero-overhead device sharing through pcie networking. *ACM Transactions on Computer Systems*, 38(1–2), jul 2021.

[14] Onur Mutlu, Saugata Ghose, and Rachata Ausavarungnirun. Recent advances in overcoming bottlenecks in memory systems and managing memory resources in GPU systems. *CoRR*, abs/1805.06407, 2018.

[15] Zaid Qureshi. *Infrastructure to Enable and Exploit GPU Orchestrated High-Throughput Storage Access on GPUs.*

PhD thesis, University of Illinois Urbana-Champaign, 2022.

[16] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seung Won Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *Proceedings of the Twenty-Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '23, 2023.