RESEARCH-ARTICLE

GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture

ZAID QURESHI, University of Illinois Urbana-Champaign, Urbana, IL, United States

VIKRAM SHARMA MAILTHODY, University of Illinois Urbana-Champaign, Urbana, IL, United States

ISAAC GELADO, NVIDIA, Santa Clara, CA, United States

SEUNGWON MIN, University of Illinois Urbana-Champaign, Urbana, IL, United States

AMNA MASOOD, University of Illinois Urbana-Champaign, Urbana, IL, United States

JEONGMIN BRIAN PARK, University of Illinois Urbana-Champaign, Urbana, IL, United States

View all

Open Access Support provided by:

Stanford University

University of Illinois Urbana-Champaign

IBM Research

NVIDIA

University at Buffalo, The State University of New York

GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture

Zaid Qureshi[*][†]
zqureshi@nvidia.com
NVIDIA/UIUC
USA

Vikram Sharma Mailthody[*][†]
vmailthody@nvidia.com
NVIDIA/UIUC
USA

Isaac Gelado
igelado@nvidia.com
NVIDIA
USA

Seungwon Min[†]
davmin@nvidia.com
NVIDIA/UIUC
USA

Amna Masood[†]
ammasood@amd.com
AMD/UIUC
USA

Jeongmin Park
jpark346@illinois.edu
UIUC
USA

Jinjun Xiong
jinjun@buffalo.edu
University at Buffalo
USA

C. J. Newburn
cnewburn@nvidia.com
NVIDIA
USA

Dmitri Vainbrand
dvainbrand@nvidia.com
NVIDIA
USA

I-Hsin Chung
ihchung@us.ibm.com
IBM Research
USA

Michael Garland
mgarland@nvidia.com
NVIDIA
USA

William Dally
bdally@nvidia.com
NVIDIA/Stanford
USA

Wen-mei Hwu
whwu@nvidia.com
NVIDIA/UIUC
USA

ABSTRACT

Graphics Processing Units (GPUs) have traditionally relied on the host CPU to initiate access to the data storage. This approach is well-suited for GPU applications with known data access patterns that enable partitioning of their dataset to be processed in a pipelined fashion in the GPU. However, emerging applications such as graph and data analytics, recommender systems, or graph neural networks, require fine-grained, data-dependent access to storage. CPU initiation of storage access is unsuitable for these applications due to high CPU-GPU synchronization overheads, I/O traffic amplification, and long CPU processing latencies. GPU-initiated storage removes these overheads from the storage control path and, thus, can potentially support these applications at much higher speed. However, there is a lack of systems architecture and software stack that enable efficient GPU-initiated storage access. This work presents a novel system architecture, BaM, that fills this gap. BaM features a fine-grained software cache to coalesce data storage requests while minimizing I/O traffic amplification. This software cache communicates with the storage system via high-throughput queues that enable the massive number of concurrent threads in modern GPUs to make I/O requests at a high rate to fully utilize the storage devices and the system interconnect. Experimental results show that BaM delivers 1.0× and 1.49× end-to-end speed up for BFS and CC graph analytics benchmarks while reducing hardware costs by up to 21.7× over accessing the graph data from the host memory. Furthermore, BaM speeds up data-analytics workloads by 5.3× over CPU-initiated storage access on the same hardware.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**; • **Software and its engineering → Software organization and properties**; • **Computer systems organization → Architectures**; • **Information systems → Storage architectures**.

KEYWORDS

GPUs, GPUDirect, SSDs, Memory capacity, Memory hierarchy, Storage systems

---

[*]Both authors contributed equally to this research.
[†]Work was done while at UIUC.

ACM Reference Format:
Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, C. J. Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the

1 INTRODUCTION

After over a decade of phenomenal growth in compute throughput and memory bandwidth [33, 45], GPUs have become popular compute devices for HPC and machine learning applications. Emerging high-value data-center workloads such as graph and data analytics [39, 46, 59, 68], graph neural networks (GNNs) [22, 30], and recommender systems [1, 20, 40, 41, 74] can potentially benefit from the compute throughput and memory bandwidth of GPUs. These applications access massive datasets organized into array data structures whose sizes range from tens of GBs to tens of TBs today, and are expected to grow rapidly in the foreseeable future.

Storing these datasets as in-memory objects enables applications to naturally and efficiently process the data. However, the memory capacity of GPUs, in spite of a 53× increase from that of G80 to A100 [33, 44], is only at 80GB, far smaller than the required capacity to accommodate entire datasets of these workloads. Thus, some state-of-the-art approaches rely on CPU user/OS code to partition datasets into chunks and orchestrate the appropriate storage access and data transfers into the GPU memory for application processing.

Another approach is to rely on memory-mapped files and GPU page faults to active the CPU page fault handler to transfer data whenever the application accesses data not present in GPU memory. In this paper, we refer to both alternatives as a **CPU-centric approach**. Our experiments show that CPU-centric approaches are bottlenecked by CPU software and CPU/GPU synchronization overheads, resulting in poor performance (See § 5 and [50]).

To avoid the inefficiencies of the CPU-centric approaches, some state-of-the-art solutions use the host memory [39], whose capacity typically ranges from 128GB to 2TB today, or pool together multiple GPUs' memories [1] to hold the entire datasets. We refer to the use of host memory or pooling multiple-GPUs' memory to extend GPU memory capacity as a *DRAM-only* solution.

Extending the host memory to the level of tens of TBs is an extremely expensive proposition in the foreseeable future. Similarly, pooling multiple GPUs' memory to reach tens of TBs can be very expensive as well. The memory capacity of each A100 GPU is only 80GB, so a 10TB pool would require 125 A100 GPUs. Furthermore, using the host memory and pooling GPU memories both require pre-loading the dataset into the extended memory. Some of the preloaded data may not be ultimately used due to data-dependent accesses. We will show with results from graph analytics that the performance of a host-memory solution is comparable to or lower than our proposed approach despite its much higher cost.

Proposal: We propose a novel system architecture called BaM (Big accelerator Memory). BaM capitalizes on the recent improvements in latency, throughput, cost, density, and endurance of storage devices to realize another level of the accelerator memory hierarchy. The goal of BaM's design is to provide efficient abstractions for the GPU threads to easily make on-demand, fine-grained accesses to massive datasets in the storage and achieve much higher application performance than state-of-the-art solutions.

Prior attempts[57, 58] to enable GPU threads to generate on-demand storage requests achieved low throughput (∼823K IOPs on the NVIDIA A100 GPU), as shown in § 5.1. In this paper, we present and evaluate the effectiveness of the key components and the overall design of BaM in addressing two key technical challenges in efficient on-demand storage accesses for accelerator applications.

First, there is currently a lack of **fast** mechanisms for the GPU application code to generate on-demand storage access requests without incurring the CPU software bottlenecks such as the OS page fault handler. To fill this gap, BaM features a scalable, highly concurrent, high-throughput configurable software cache that takes advantage of the massive memory bandwidth and atomic operation throughput of modern GPUs. The software cache coalesces redundant on-demand accesses and facilitates reuse of storage data while providing high application-perceived throughput.

Second, while the CPU-centric approaches suffer from the low-degree of CPU thread-level parallelism available to page fault handlers and device drivers, there is currently a lack of GPU mechanisms for orchestrating storage accesses without relying on the CPU. To address this issue, BaM provides a user-level GPU library of highly concurrent submission/completion queues in GPU memory that enables GPU threads whose on-demand accesses do not hit in the software cache to make storage accesses in a high-throughput manner. This user-level approach removes the page fault handling bottleneck, and uses fine-grained synchronization and minimal critical sections to reduce software overhead for each storage access and support a high-degree of thread-level parallelism.

There is a trend towards increasing autonomy and asynchrony of GPUs. The GPUDirect Async family [42] of technologies accelerate the control path when moving data directly into GPU memory from memory or storage. Each transaction involves initiation, where structures like work queues are created and entries within those structures are readied, and triggering, where transmissions are signaled to begin. To our knowledge, BaM is the first ***accelerator-centric*** approach where GPUs can create on-demand accesses to data where it is stored, be it memory or storage, **without relying on the CPU to initiate or trigger the accesses.** Thus, BaM marks the beginning of a new variant of this family that is GPU kernel initiated (KI): *GPUDirect Async KI Storage*.

While the user-level storage device queues raise security concerns for traditional monolithic server architectures, the recent shift in data centers toward zero-trust security models that provide security guarantees through trusted hardware or software services have provided the new system framework for securing accelerator-centric user-level storage access models like BaM [26–28].

We have built a prototype BaM system through novel organization of off-the-shelf hardware components and development of a novel custom software stack that takes advantage of the advanced architectural features in recent GPUs. Evaluation using a variety of workloads with multiple datasets shows that BaM is on-par with a 21.7× more expensive host-memory DRAM-only solution and up to 5.3× faster than a state-of-the-art CPU-centric software solution. Overall, we make the following contributions. We

(1) propose BaM, an accelerator-centric system architecture in which GPU threads perform on-demand accesses to array data where it is stored, be it memory or storage, without relying on the CPU to initiate these accesses;

(2) enable on-demand, high-throughput fine-grained access to storage through a novel library of highly concurrent submission/completion protocol queues;

(3) provide high-throughput, scalable software-defined caching and software API for programmers to exploit locality and control data placement for their applications; and

(4) construct and evaluate a prototype design for GPUs to access massive storage data in a cost-effective manner.

We have published a full version of the paper that includes appendix covering in-depth analysis on limitations of current system with additional evaluations [50]. BaM is implemented completely in open-source, and both hardware and software requirements are publicly accessible [5, 36, 49].

2 BACKGROUND

This section covers the limitations with the common solution that keeps large datasets in abundant CPU memory or pooled multi-GPU memory (§2.1) and then describe why GPUs can tolerate long storage access latency (§2.2).

2.1 Leveraging CPU or Pooled Multi-GPU Memory

Applications can leverage CPU memory or even pool together the memory of multiple GPUs to host large data structures. Previous works show that the GPU provides sufficient memory-level parallelism to tolerate the access latency of these memories and significantly out-perform the UVM [48] solution for graph traversal applications [39].

However, regardless of whether CPU memory or pooled GPU memory is used to host these data structures, this approach suffers from two major pitfalls. First, *data must still be loaded from the storage to the memory before any GPU computation can start.* Often this initial data loading can be the main performance bottleneck. (see the Target (T) system of Figure 7). Second, *hosting the dataset in CPU memory or pooled GPU memory requires scaling the available memory*, by either increasing the CPU DRAM size or the number of GPUs in the system, with the dataset size, *and thus can be prohibitively expensive for massive datasets.*

2.2 Tolerating Storage Access Latency

As the storage device latency is reduced due to technological advancements like Optane [23] or Z-NAND [55] media, software overhead is becoming a significant fraction of overall I/O access latency. Our experiments show that even with io_uring, a highly optimized CPU software stack[4], *as device latency decreases, OS kernel software overhead severely limits the storage access throughput and becomes a significant fraction, up to 36.4%, of the total storage access latency.*

To address this shift, emerging storage systems allow applications to make direct user-level I/O accesses to storage [17, 25, 26, 31, 34, 35, 53, 67, 69]. The storage system allocates user-level queue pairs, akin to NVMe I/O submission (SQ) and completion (CQ) queues, which the application threads can use to enqueue requests and poll for their completion. Using queues to communicate with storage systems forgoes the userspace to kernel crossing of traditional file system access system calls. Instead, isolation and other file

system properties are provided through trusted services running as trusted user-level processes, kernel threads, or even storage system firmware running on the storage server/controller [25, 26, 53, 67].

In such systems, the parallelism required to tolerate access latency and achieve full throughput of the device is governed by Little's Law: $T \times L = Q_d$, where T is the target throughput, L is the average latency, and Q_d is the minimal queue depth required at any point in time to sustain the target throughput. To achieve the full potential of the critical resource, PCIe ×16 Gen4 connection providing ~26GBps of bandwidth, then T is $26GBps/512B = 51M/sec$ and $26GBps/4KB = 6.35M/sec$ for 512B and 4KB access granularities, respectively. The average latency, L, depends on the SSD devices used and is measured when the SSD provides its maximal throughput. For the experiments reported in this paper, L is $11us$ and $324us$ for the Intel Optane and Samsung 980pro SSDs, respectively. From Little's Law, to sustain a desired 51M accesses of 512B each, the system needs to accommodate a queue depth of $51M/s \times 11us = 561$ requests (70 requests for 4KB) for Optane SSDs. For the Samsung 980pro SSDs, the required Q_d for sustaining the same target throughput is $51M \times 324us = 16,524$ (2057 for 4KB). Note that Q_d can be spread across multiple physical device queues. To sustain T over a computation phase, there need to be a substantially higher number of concurrently serviceable access requests than Q_d over time.

The emerging queue-based storage systems present major challenges to sustaining T in massively parallel execution paradigms. Take as an example using NVMe queues to make I/O requests. After requests are enqueued into a NVMe SQ, the queue's doorbell must be rung with the updated queue tail to notify the storage controller of the new request(s). As these doorbell registers are write-only, when a thread rings a doorbell, it must make sure that no other thread is writing to the same register and that the value it is writing is valid and is a newer value than any value written to that register before. Implementing queue insertion as a critical section, while simple, imposes significant serialization latency, which might be fine for the CPU's limited parallelism but can cause substantial overhead for thousands of GPU threads. Storage interfaces like [25, 26, 53] face similar serialization challenges. These challenges are addressed in the design of the BaM queues.

3 BAM SYSTEM AND ARCHITECTURE

The goal of BaM's design is to provide high-level abstractions for accelerators to make on-demand, fine-grained, high-throughput access to storage while enhancing the storage access performance. To this end, BaM provisions storage I/O queues and buffers in the GPU memory as shown in Figure 1, and builds on the recent memory-map capability of GPUs to map the storage doorbell registers to the GPU address space. Although doing so enables the GPU threads to access terabytes of data on storage, BaM must address three key challenges in providing an efficient and effective solution:

(1) As storage protocols and devices exhibit significant latency, BaM must leverage the GPU's massive parallelism (up to three orders of magnitude more than the CPU) to keep many requests in flight, efficiently tolerate such latency, and overlap it with computation. (See §3.3)
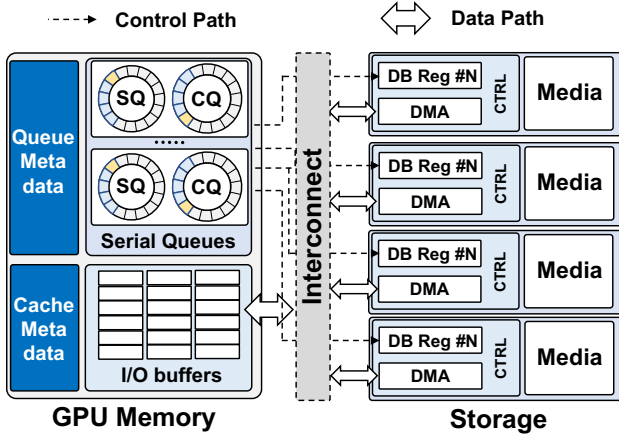
Figure 1: Logical view of BaM design.

Figure 2: Life of a GPU thread in BaM.

(2) As storage devices have relatively low bandwidth and GPUs have limited memory capacity, BaM must optimally utilize these resources. (See §3.4)

(3) As GPU kernels generally don't expect to make storage accesses, BaM must provide high-level abstractions that hide BaM's complexity and make it easy for the programmers to integrate BaM into their GPU kernels. (See §3.5)

Next, we provide an overview of BaM and then explain how BaM addresses these challenges.

3.1 BaM System Overview

BaM presents the bam::array high-level programming abstraction, enabling programmers to easily integrate BaM into their existing GPU applications. An application can call BaM APIs to map the bam::array to data on storage, akin to mmap'ing a file.

Figure 2 shows how a GPU thread uses BaM to access data. When a GPU thread accesses data with the bam::array abstraction ❶, it uses the abstraction to determine the offset, i.e. cache line, for the data being accessed ❷. Threads in a warp can coalesce their accesses ❸ if multiple threads access the same cache line. For each unique cache line being accessed, a single thread probes the cache line's metadata ❹ on behalf of the rest of the threads, improving cache access efficiency.

If an access hits in the cache, the thread can directly access the data in GPU memory. If the access misses, the thread needs to fetch data from the backing memory. The BaM software cache is designed to optimize the bandwidth utilization to the backing memory in two ways: (1) by eliminating redundant requests to the backing memory and (2) by allowing users to configure the cache for their application's needs.

If the storage system or device is backing the data, the GPU thread enters the BaM I/O stack to prepare a storage I/O request ❺, enqueues it to a submission queue ❻, and then waits for the storage controller to post the corresponding completion entry ❼. The BaM I/O stack aims to amortize the software overhead associated with the storage submission/completion protocol by leveraging the GPU's immense thread-level parallelism, and enabling low-latency batching of multiple submission/completion (SQ/CQ) queue entries
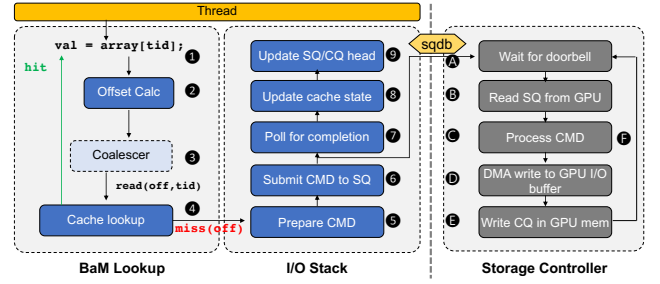
to minimize the cost of expensive doorbell register updates and reducing the size of critical sections in the storage protocol.

On receiving the doorbell update (A), the storage controller fetches the corresponding SQ entries (B), processes the command (C) and transfer the data between SSD and the GPU memory (D). At the end of the transfer, the storage controller posts a completion entry in the CQ (E). After the completion entry is posted, the thread updates the cache state ❽, update the SQ/CQ state ❾ and then can access the fetched data in GPU memory.

## 3.2 Comparison With the CPU-Centric Approaches

When compared to the proactive tiling CPU-centric approach shown in Figure 3a, BaM has three main advantages. First, with proactive tiling, the CPU ends up copying data between the storage and the GPU memory and launching compute kernels multiple times to cover a large dataset. Each kernel launch and termination incurs costly synchronization between the CPU and the GPU. BaM allows GPU threads to both compute and fetch data from storage, as shown in Figure 3b, which reduces the frequency of CPU-GPU synchronization and GPU kernel launches. Furthermore, the storage access latency of some threads can also be overlapped with the compute of other threads, which improves the overall performance.

Second, in proactive tiling, because the compute is offloaded to the GPU and the data movement is orchestrated by the CPU, the CPU cannot accurately determine which parts of the data are needed and when they are needed, thus it ends up fetching many unneeded bytes. In contrast, with BaM, a GPU thread fetches bytes only when they are used, reducing the I/O amplification overhead.

Third, with proactive tiling, programmers expend effort to partition the application's data and overlap compute with data transfers to hide storage access latency. BaM allows the programmer to naturally access the data through the array abstraction and harness GPU thread parallelism across large datasets to hide the storage access latency.

3.3 High-Throughput I/O Queues

BaM leverages GPU's massive thread-level parallelism and fast hardware scheduling to maintain the queue depths needed to hide storage access latency and to achieve peak storage throughput. However, ringing doorbells after enqueueing commands or cleaning up SQ entries, in the existing storage I/O protocols requires serialization. A critical section that encompasses the process of enqueuing a command and ringing the doorbell, albeit simple, would

(a) CPU-Centric Model.

(b) BaM Model.
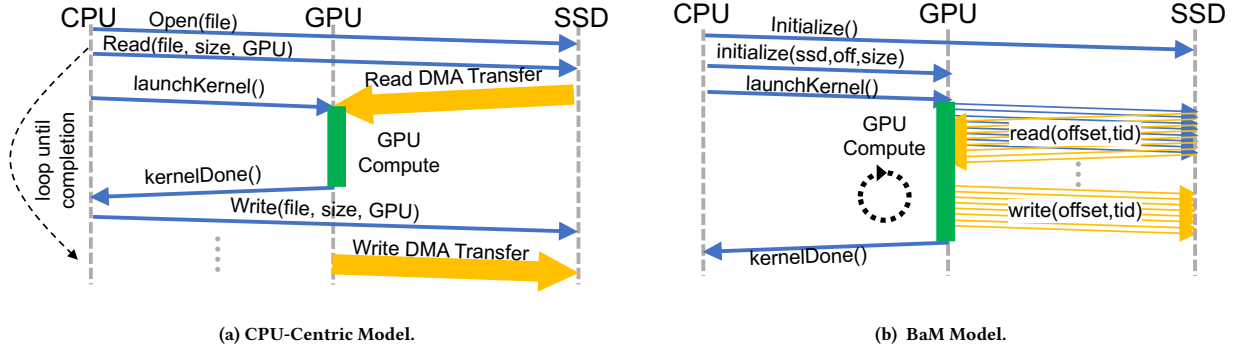
Figure 3: Comparison between the traditional CPU-centric and BaM computation model is shown in (a) and (b). BaM enables GPU threads to directly access storage enabling compute and I/O overlap at fine-grain granularity.

result in poor throughput and significant serialization latency when thousands of threads enqueue I/O requests concurrently. Instead, BaM uses fine-grained memory synchronization enabling many threads to enqueue into the SQ, poll the CQ, or mark queue entries for cleanup in parallel without large critical sections.

We will use the SQ implementation in BaM as an example to explain how fine-grained memory synchronization helps to achieve the above design goals and principles. BaM keeps the following metadata per SQ in the GPU memory: 1) local copies of the queue's head and tail, 2) atomic `ticket` counter, 3) `turn_counter` array, an integer array of the same length as the queue, 4) `mark` bit-vector of the same length as the queue, and 5) a `lock`.

To enqueue requests, threads first atomically increment the ticket counter by two. The returned `ticket` value is an index into a virtual queue with 2^{32} entries and can be divided by the physical queue size to assign each thread an `entry` in the physical queue, the remainder, and a `turn` value along with a valid bit for that entry, the quotient. Using its `entry` and `turn`, each thread can wait for both the enqueue and dequeue operations of all previous commands in its assigned `entry` in the queue. Each thread uses its `entry` to index into the `turn_counter` array, and polls on the location until the counter equals the thread's `turn`. That is, the `turn_counter` array tracks the mapping of each physical queue entry to an virtual queue entry. It effectively creates sub-queues whose waiting threads are ordered by their `turn` value for each physical queue entry.

When it is the thread's turn, i.e., the virtual queue entry assigned to the thread becomes active, the thread can copy its I/O access command into its assigned (`entry`) position in the physical queue. Afterwards, each thread sets the position's corresponding bit in the `mark` bit-vector. The `turn_counter` array enables as many threads as the size of the physical queue to copy their commands in parallel.

The threads call the `move_tail` routine to complete the insertion of their requests. One thread will successfully acquire the lock and will move the tail past the consecutive new entries inserted by the calling threads and reset the `mark` bits associated with these entries with the `reset_marks` routine. The thread then rings the queue's doorbell at the storage controller with the new tail and releases the lock. This coalesces the doorbell writes, which are expensive operations over the PCIe interconnect, for multiple calling threads and

improves the effective throughput. The rest of the calling threads will return if their `mark` bits are reset, signifying the SQ's tail will be moved past their enqueued entries. If the `mark` bit for a thread has not been reset, the thread keeps trying to take the queue's `lock` and complete the insertion of its request. Once the thread knows that its position's `mark` bit has been reset, it atomically increments the position's `turn_counter` value by one (to an odd value).

After the thread's command is submitted, the thread can poll, without any lock, on the CQ to find the completion entry for its submitted request. When it finds the completion entry, it marks the entry for dequeuing by the next movement of the CQ head in the CQ's `mark` bit-vector. The CQ's head and doorbell are managed similarly as the SQ's tail, except a thread stops trying to reset its CQ entry's `mark` bit if it notices the CQ's head has already been moved past the entry it marked for dequeueing. A winning thread among the calling threads rings the doorbell with a new `head` position to communicate forward progress to the storage system.

The storage controller communicates forward progress by specifying a new SQ head in each CQ entry. The thread with the CQ `lock` reads this field from the last CQ entry whose mark bit was reset by it, and then iterates from the current SQ head until the specified new head, incrementing each position's `turn_counter` value by one (to an even value), allowing threads waiting for those positions to enqueue their commands. The thread then updates the SQ head and releases the CQ `lock`.

3.4 BaM Software Cache

The BaM software cache is designed to enable optimal use of the limited GPU memory and storage-access bandwidth. Traditional OS-kernel-mode memory management (allocation and translation) implementations must support diverse, legacy application/hardware needs. As a result, they contain large critical sections that limit the effectiveness of multi-threaded implementations. BaM addresses this bottleneck by allocating all the virtual and physical memory required for the software cache when starting each application. This approach reduces critical sections to only require a lock when inserting or evicting a cache line, which allows the BaM cache to support many more concurrent accesses.

The BaM cache aims to minimize the number of redundant I/O requests to the storage, avoiding unnecessary I/O traffic. To this end, when a thread probes the cache with an offset, it checks the corresponding cache line's state. If the accessed cache line is not in the cache, the thread locks the cache line, finds a victim to evict, and requests the cache line from the backing memory. When the request completes, the requesting thread unlocks the cache line by making its state valid and incrementing its reference count. This locking forces other threads that access the same cache line to wait until the cache line has been inserted and thus prevents multiple requests to the backing memory for the same cache line, exploiting locality and minimizing the number of requests to the backing memory. If, when the thread probes the cache line state, it is valid, the thread atomically increments the cache line's reference count. When the thread is done using the cache line, it decrements it's reference count.

To avoid contention among concurrent evictions, the BaM cache uses a clock replacement algorithm [14]. The cache has a global counter that gets incremented when a thread needs to find a cache slot. The returned value of the counter assigns the thread a cache slot to use, allowing concurrent threads to evict unique cache slots in parallel. If the assigned cache slot is currently mapped to a cache line that has a non-zero reference count, that is it is pinned, the thread will increment the counter again to attempt to replace another cache slot until it finds a cache slot that is not mapped to a pinned cache line. The thread will then mark the cache line invalid and change the mapping of the cache slot to point to the newly inserted cache slot.

Warp Coalescing: Threads in a warp may contend among themselves in accessing the BaM software cache, especially when consecutive threads try to access contiguous bytes in memory. This contention incurs significant overhead when the needed cache line is already in the GPU memory. To overcome this, BaM's cache implements coalescing in software using the `__match_any_sync` warp primitive to synchronize among threads in the warp and a mask is computed letting each thread know which other threads in the warp are accessing the same offset. From that group, the threads decide on a leader and only the leader queries the cache and manipulates the requested cache line's state. The threads in the group synchronize using the `__shfl_sync` primitive, and the leader broadcasts the address of the requested offset in the GPU memory to the group.

Compared to warp coalescing implemented in prior works [57] where the unique cache probes in a warp are serialized and require many instructions, BaM's coalescer enables all threads in a warp to divide themselves into groups according to the cache lines they are accessing with a single instance of the new `__match_any_sync` warp synchronization primitive. A leader is determined for each group in parallel and each leader can independently probe the cache for the group's needed cache line, with no inter-group dependencies or synchronization.

3.5 BaM Abstraction and Software APIs

BaM's software stack provides the programmer an array-based high-level API (`bam::array<T>`), consistent with array interfaces defined in modern programming languages (e.g. C++, Python, or

```
__global__
void kernel(bam::array<float> data, size_t n,
bam::array<float> out, bam::array<int> randidx) {
size_t tid = ...;
...
for(; tid < n; tid += (blockIdx.x * blockDim.x))
        out[tid] = data[randidx[tid]];
};
```

Listing 1: Example GPU kernel with `bam::array<T>`.

Rust). As GPU kernels operate on such arrays, BaM's abstraction minimizes the programmer's effort to adapt their kernels, as shown in Listing 1. `bam::array`'s overloaded subscript operator enables the accessing threads to coalesce their accesses, query the cache, make I/O requests on misses, and returns the appropriate element of type T to the calling function. The `bam::array<T>` can also be used to develop higher-level abstractions that can transparently provide optimizations like cache line reference reuse so threads do not need to probe the cache more than once when accessing the same cache line multiple times. In contrast, the proactive tiling CPU-centric model requires full, non-trivial application rewrites to decompose the compute and data transfers into tiles that fit within the limited GPU memory.

BaM initialization requires allocating a few internal data structures that are reused during the application's lifetime. Initialization can happen implicitly through a library construction if no customization is needed. Otherwise, the application specializes the memory through template parameters to BaM's initialization call, a standard practice in C++ libraries. However, in most cases, specialization and fine-tuning are unnecessary, as we show later in § 5 where only BaM's default parameters are used.

4 BAM PROTOTYPE

We use off-the-shelf hardware including NVIDIA GPUs and arrays of NVMe SSDs to construct a BaM prototype and show the benefits of allowing GPUs to directly access storage with enough random access bandwidth to take full advantage of a GPU's PCIe Gen4 x16 link. *Once this level of data access bandwidth is achieved, a storage-based solution is as good as a host memory accessed through PCIe in terms of performance but much cheaper.* For simplicity, we describe the prototype assuming bare-metal, direct access to the NVMe SSDs.

4.1 Enable Direct NVMe Access From GPU Threads

For simplicity, we will use the NVMe SSD controllers as a simple example of storage controllers to explain the key features of the BaM prototype. In order to enable GPU threads to directly access data on NVMe SSDs we need to: 1) move the NVMe queues and I/O buffers from the host CPU memory to the GPU memory and 2) enable GPU threads to write to the queue doorbell registers in the NVMe SSD's BAR space. To this end, we create a custom Linux driver that creates a character device per NVMe SSD in the system, like the one by SmartIO [38]. Applications use BaM APIs to open the character device for each SSD they wish to use.

In the custom Linux driver, BaM leverages GPUDirect RDMA APIs to pin and map NVMe queues and I/O buffers in the GPU

Table 1: BaM prototype system specification

BaM Config	Specification
System	Supermicro AS-4124GS-TNR
CPUs	2× AMD EPYC 7702 64-Core Processors
DRAM	1TB Micron DDR4-3200
GPU	NVIDIA A100-80GB PCIe
PCIe Expansion	H3 Platform Falcon-4016 [21]
SSDs	Refer to Table 2
Software	Ubuntu 20.04 LTS, NVIDIA Driver 470.82, CUDA 11.4

memory. This enables the SSD to perform peer-to-peer data reads and writes to the GPU memory.

We leverage GPUDirect Async [42] to map the NVMe SSD doorbells to the CUDA address space so GPU threads can ring the doorbells on demand. This requires the SSD's BAR space to be first memory-mapped into the application's address space. Then it is mapped to CUDA's address space with the cudaHostRegister API. Other storage systems can be enabled similarly.

4.2 Scalable Hardware

Scaling BaM using the PCIe slots available within a data-center grade 4U server is challenging as the number of PCIe slots available in these machines is limited. For instance, Supermicro AS-4124 system has five PCIe Gen4 ×16 slots per socket. If a GPU occupies a slot it can only access 4 ×16 PCIe devices without crossing the inter-socket fabric and suffering significant performance degradation. However, as no single NVMe SSD can match the throughput of a PCIe x16 Gen4 link, the BaM hardware must scale the number of NVMe devices to provide the necessary throughput and match the bandwidth of the GPU's ×16 PCIe Gen4 interconnect.

To address this, we built a custom prototype machine for the BaM architecture using the off-the-shelf components as shown in Table 1. The BaM prototype uses a PCIe expansion chassis with custom PCIe topology for scaling SSDs. The PCIe switches provide low-latency, high-throughput peer-to-peer access. The expansion chassis has two identical drawers that can be connected independently to the host. Each drawer supports 8 ×16 PCIe slots. We use one ×16 slot in each drawer for an NVIDIA A100 GPU and the rest of the slots are populated with different types of SSDs. Currently, each drawer can only support 10 U.2 SSDs as they take significant space. With PCIe bifurcation, a multi-SSD riser card can enable more than 16 M.2 SSDs per drawer.

SSD Technology trade-offs: Table 2 lists the metrics that significantly impact the design, cost and efficiency of BaM systems for three types of off-the-shelf SSDs. The $/GB is based on the current list price per device, the expansion chassis, and the risers needed to build the system. A comparison of these metrics across SSD types shows that the consumer grade NAND Flash SSDs are inexpensive with more challenging characteristics, while the low-latency drives such as Intel Optane SSD and Samsung Z-NAND are more expensive with desirable characteristics. For example, for write intensive applications using BaM, Intel Optane drives provide the best throuput and endurance.

Irrespective of the underlying SSD technology, as shown in Table 2, the BaM prototype provides 4.3-21.8× advantage in cost-per-GB, even with the expansion chassis and risers, over a DRAM-only solution. Furthermore, this advantage grows with additional capacity added

per device, which makes BaM highly scalable as SSD capacity and application data size increase.

4.3 BaM Raw Throughput

We establish that BaM can generate sufficient I/O requests to saturate the underlying storage system by measuring raw throughput of BaM using microbenchmarks with Intel Optane SSDs and an NVIDIA A100 GPU. We allocate all the available SSD SQ/CQ queues into the GPU memory with a queue depth of 1024. We then launch a CUDA kernel with each thread requesting a random 512-byte block from the SSD via a designated queue. The requests are uniformly distributed across all queues with round-robin scheduling. We vary the number of threads and SSDs mapped to the GPU. For multiple SSDs, the requests are uniformly distributed across SSDs using round-robin scheduling. We measure I/O operations per second (IOPs) as the ratio of the number of requests submitted and the kernel execution time.

Results: Figure 4 presents the measured IOPs for 512B random read and write benchmarks. BaM can reach peak IOPs per SSD and linearly scale with additional SSDs for both reads and writes. With a single Optane SSD, BaM only requires about 16K-64K GPU threads to reach near peak IOPs (see Table 2). With ten Optane SSDs, BaM achieves 45.8M random read IOPs and 10.6M random write IOPs, the peak possible for 512B accesses to the Intel Optane SSDs. This is 22.9GBps (90% of the measured peak bandwidth for Gen4 ×16 PCIe links) and 5.3GBps of random read and write bandwidth, respectively. Further improvements in write bandwidth, which isn't yet hitting PCIe limits, can be achieved by scaling to more SSDs. Similar performance and scaling is observed with Samsung SSDs and also at 4KB access sizes but are not reported here due to space constraints. *These results validate that BaM's infrastructure software can match the peak performance of the underlying storage system.*

4.4 Discussion

GPUDirect RDMA I/O Consistency: As prior works[43, 61] have noted, when a third-party device writes into the GPU's memory with GPUDirect RDMA over PCIe, without a PCIe read following the writes, the order of these writes might not be preserved from the viewpoint of the GPU threads running concurrently. In the BaM prototype, we allow a GPU thread to submit a second I/O request after successfully polling for the first one's completion. As the storage device must read a submitted request over PCIe before writing the completion entry for it, when the thread finds the completion entry for the second command, it knows a PCIe read has been completed after all the writes for the first command. However, doing so incurs a 100% performance overhead.

To reduce the total number of additional I/O requests submitted, we implement a shared global virtual queue with a lock. All threads who find the CQ entries for their commands race for the lock of the virtual queue, and the winner enters the BaM I/O stack to submit an additional I/O request on behalf of all competing threads whose request can be coalesced with its own. After the winner finds the completion entry for the second request, it notifies the threads whose second requests were coalesced and releases the lock. Other threads continuously poll to see if their respective second requests are covered through coalescing. If not they try to

Table 2: Comparison of different types of SSDs with DRAM DIMM. Prices are taken from a well known retailer [13] and near the time of writing this paper. The cost of required PCIe expansion hardware [21] is included for systems using SSDs.

Technology	Sources	Product	RD IOPs (512B, 4KB)	WR IOPs (512B, 4KB)	Latency (μs)	DWPD	$/GB	Gain
DRAM	multiple	DIMM (DDR4)	>10M	>10M	O(0.1)	>1000	11.13	1.0×
Optane [23]	single	Intel P5800X	5.1M, 1.5M	1M, 1.5M	O(10)	100	2.54	4.4×
Z-NAND [55]	single	Samsung PM1735	1.1M, 1.6M	351K, 351K	O(25)	3	2.56	4.3×
NAND Flash [54]	multiple	Samsung 980pro	700K-800K, 700K-800K	172K, 172K	O(100)	0.3	0.51	21.8×

obtain the `lock` again. We find that this solution incurs only less than 8% performance overhead.

Programming Model: BaM provides the GPU threads a memory-like abstraction, with the same memory consistency properties as the GPU memory. If the application threads read and write to the same words in BaM-backed memory, then it is up to the application to implement the necessary synchronization to avoid races, just as if the threads were reading and writing to the GPU memory.

BaM supports writes at all levels of its software stack: block write I/O requests, tracking dirty cache lines, and the write method in the high-level abstractions. When the high-level abstraction's write method is called, the access to the cache line's state will set the dirty bit. The BaM cache is a write-back cache and has APIs for the user to flush a specific or all dirty cache lines.

If the system crashes in the middle of a GPU kernel that writes, there are no guarantees unless the application itself takes the responsibility of check-pointing application state and data, which is common practice in GPU accelerated applications.

CPU-GPU Data Sharing: If the application instantiates a BaM cache in the GPU memory and another BaM cache in the host CPU memory, it is up to the application to implement the appropriate synchronization and communication to maintain a consistent view of shared data, just like when the application uses GPU memory without BaM.

5 EVALUATION

BaM excels on applications that have data-dependent data access patterns. Data dependent access patterns are widely used in popular applications, e.g. graph analytics, and frameworks, e.g. RAPIDS for accelerating analytical queries. In this section, we present an evaluation of the prototype BaM system using these two workloads with a variety of datasets and show that a) BaM's performance is either on-par with or outperforms the state-of-the-art solutions (see § 5.2 and § 5.3). b) BaM's design is agnostic to the SSD storage medium used, enabling application-specific cost-effective solutions. c) BaM reduces I/O amplification and CPU orchestration overhead significantly for data-analytics workloads (see § 5.3).

5.1 Comparison With GDS and ActivePointers

We evaluate BaM's performance benefits over NVIDIA GDS [47] for different I/O block sizes, ranging from 4KB to 1MB. For GDS, we use `fio` to benchmark sequential access performance to transfer 128GB of data from four SSDs to GPU memory with 16 CPU threads. In the case of BaM, each warp is assigned a cache-line, the same size as the I/O blocks used for GDS, and consecutive warps access consecutive cache-lines in the 128GB dataset. Recall that in BaM the cache-line size defines the I/O access granularity. As shown in
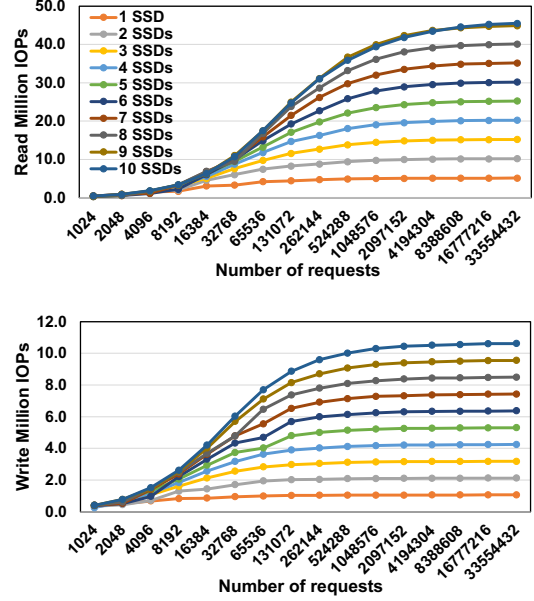
Figure 4: 512B random read (top) and write (bottom) benchmark scaling with BaM on Intel Optane P5800X SSDs. BaM's I/O stack can reach peak IOPs per SSD and linearly scale for random read and write accesses.

Figure 5, GDS can only saturate the GPU's PCIe link at the large I/O granularity of 32KB and only reaches 23.6% of the PCIe bandwidth at 4KB. Regardless of the number of CPU threads used, GDS is limited by the high overhead incurred by the Linux software stack. In contrast, BaM easily achieves 25GBps using four SSDs, which is the measured peak bandwidth of the GPU's PCIe link.

Next, we compare BaM and ActivePointers [57]. For this evaluation, a warp is assigned to read 1024 contiguous 8-byte elements in a file where threads access the elements in a coalesced manner. With ActivePointers, the file is pinned in the Linux page cache in CPU memory, favoring ActivePointers as *any misses in the Active-Pointers cache only require data transfers from CPU memory to GPU memory, avoiding storage I/O requests and latencies*. For BaM, the data is kept on four SSDs from where it is requested in case of a miss in the BaM cache, incurring storage access latency. Due to space constraints, we only show the results for 64K and 1 Million threads in Figure 6; measurements with up to 32 Million threads show similar trends.

With a cold cache, ActivePointers' I/O performance is greatly hindered by the GPUfs [58] mechanism that GPU threads use to request the data transfers from the CPU, achieving only an effective
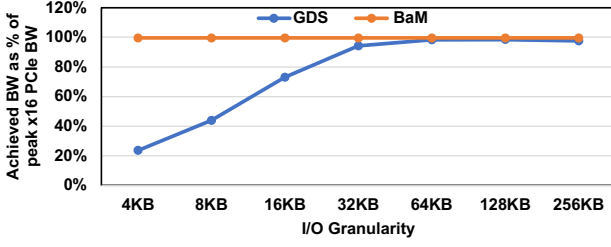
Figure 5: Performance of BaM compared with NVIDIA GDS [47]. With granularities less than 32KB, GDS is not able to saturate the PCIe interface due to the overheads of the traditional CPU software stack. In contrast, BaM is able to saturate the interface (∼25GBps) at even 4KB I/O granularity.
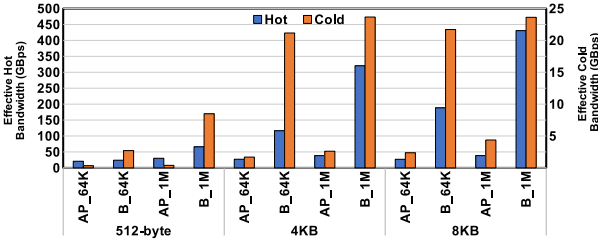

Figure 6: Performance of BaM (B) and ActivePointers+GPUfs [57, 58] (AP) with 64K and 1Million GPU threads with a 8GB cache in the GPU memory, in both hot and cold state, using 512-byte, 4KB, and 8KB cache-line sizes. BaM provides a peak miss-handling throughput of 17MIOPs with 4 Optane SSDs, which is 20.7× higher than ActivePointers' peak miss-handling throughput with the fast CPU memory. BaM's provides a peak hot cache bandwidth of 430GBps, 11.2× higher than ActivePointers' cache.

Table 3: Graph Analytics Datasets.

Graph	Num. Nodes	Num. Edges	Size (GB)
GAP-kron (**K**) [7]	134.2M	4.22B	31.5
GAP-urand (**U**) [7]	134.2M	4.29B	32.0
Friendster (**F**) [68]	65.6M	3.61B	26.9
MOLIERE_2016 (**M**) [59]	30.2M	6.67B	49.7
uk-2007-05 (**Uk**) [10, 11]	105.9M	3.74B	27.8

bandwidth of 4.4 GBps for 8 KB data transfers out of CPU memory. ActivePointers provides a peak miss-handling throughput of 823 KIOPs, with 512-byte cache-lines. In contrast, BaM nearly saturates the GPU's PCIe link at around 24 GBps with 4 KB and 8 KB cache-lines. Even with 512-byte cache-lines, BaM achieves 85% of the peak throughput supported by the 4 SSDs, at 17 MIOPs. Furthermore, when both caches are hot, BaM can provide an effective bandwidth of up to 430 GBps, 11.2× more than ActivePointers' peak bandwidth. BaM outperforms ActivePointers by more than an order of magnitude in both cache miss handling and hit data delivery.

5.2 Performance Benefit for Graph Analytics

We evaluate the performance benefit of BaM for graph analytics applications. We use the graphs listed in Table 3 for the evaluation. K, U, F, M are the four largest graphs from the SuiteSparse Matrix collection [16] while the Uk is taken from LAW [9].

A goal of BaM is to provide competitive performance against the host-memory-based DRAM-only graph analytics solution. To this end, *optimistic* target baseline **T** allows the GPU threads to directly perform coalesced fine-grain access to the graph data stored in the host-memory [39]. As there is sufficient CPU memory for the input graphs used in this experiment, we can make a direct performance comparison between BaM and **T**.

We run two graph analytics algorithms, Breadth-first-search (BFS) and Connected Components (CC), on the target system and BaM with different SSDs listed in the Table 2. Porting the state-of-the-art GPU implementations for both applications [39] to BaM requires minimal code changes as described in §3.5. For BFS, we report the average run time after running at least 32 source nodes with more than two neighbors. We do not execute CC on the Uk dataset since CC operates only on undirected graphs. Finally, we fix the BaM software cache size to 8GB, the cache-line size to 4KB and unless explicitly stated, we use four Intel Optane SSD with 128 queue-pairs at 1024 depth.

Overall performance with Intel SSD: Figure 7 shows the performance of both the target system (**T**) and BaM with single (B_1I) and four (B_4I) Intel Optane SSDs. For the single SSD (B_1I) configuration, BaM is on average slower by 1.43× and 1.27× for BFS and CC workload, respectively. This is because of the limited storage throughput available for BaM with single SSD (×4 PCIe Gen4 interface).

Scaling the number of SSDs to four (B_4I) and replicating data increases the BaM's aggregate bandwidth and provides similar bandwidth as the ×16 Gen4 PCIe interface as in the target system. Comparing with the target system **T** with file-loading time, *BaM with four Optane drives provides on average 1.00× and 1.49× speedup on BFS and CC applications, respectively.*

In both workloads, BaM overlaps the SSD data transfers for some threads with the compute of other threads, fully utilizing PCIe Gen4 ×16 bandwidth while incurring much less I/O amplification. In contrast, the target system **T** needs to wait until the file is loaded into memory before it can offload the compute to GPU. The superior host-memory bandwidth of the **T** system cannot overcome the initial file loading latency. As a result, BaM achieves either on-par or higher end-to-end performance.

Compared to the single-SSD BaM configuration, the four-SSD configuration scales to 3.48× for BFS and 4× for CC. The main detractor of scaling for BFS is the Uk dataset. Many nodes in the Uk graph have tiny neighborlists, resulting in deep BFS traversal (>100+ iterations) with small frontiers. Consequently, the total number of overlapping I/O requests in each iteration of BFS is insufficient to tolerate the latency.

BaM performance breakdown: We now slice up the overall execution time of BaM to understand how each BaM component contributes to the overall execution time, as shown in Figure 7. We first load the entire dataset in the GPU-HBM memory and measure the execution time (Compute in green color). Next, we measured the
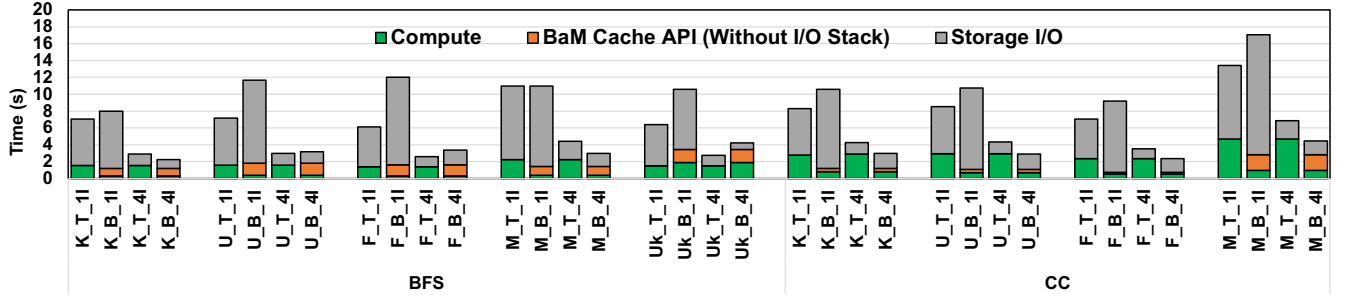
Figure 7: Graph analytics performance of BaM (CL size 4KB) and the Target (T) system with a single Intel Optane SSD. On average, BaM's end-to-end time is 1.0× (BFS) and 1.49× (CC) faster than the Target.
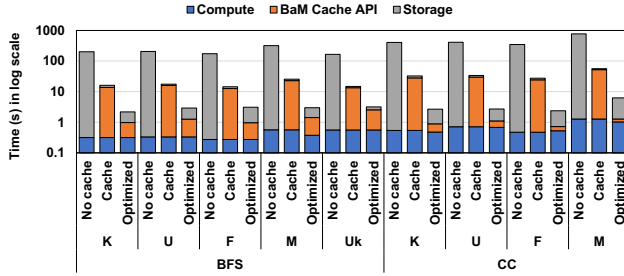

Figure 8: Sources of performance improvement in BaM. Adding a naive cache in BaM provides on average 11.9× (BFS) and 12.65×(CC) speedup over not having a cache. When the application leverages and exploits BaM's warp coalescing and reference reuse efficiently (`Optimized`), the application is sped up further by 6.07× (BFS) and 11.24× (CC) on average.

Figure 9: The slowdown observed by BaM with four Samsung DC PM1735 (S1735) and Samsung 980pro SSDs when compared with four Intel Optane SSDs.

total execution time when all the data is in the GPU-HBM memory, but each application access must go through the BaM cache API. This captures the best case performance one can achieve with BaM cache with no I/O requests. Subtracting `Compute` time from this measured execution time provides the cache API overhead (shown in orange color). Next, we constrain the BaM cache to 8GB and measure the total execution time. Storage I/O time (shown in grey color) can be computed by subtracting `Compute` and cache API time from this total time.

From Figure 7, we make the following key observations. First, the cache overhead is about 2-15% for a single SSD and goes to 4-45% for four SSDs. With a single SSD, the application performance is bounded by the storage throughput, and additional SSDs help to alleviate this problem by improving bandwidth, thus significantly minimizing the storage access overhead. The rest of the cache overhead is from cache metadata contention, long latency atomic operation, and warp scheduling in the face of polling threads. Even with this overhead, BaM outperforms the *most optimistic baseline system available*. Second, with four SSDs, BaM is still bounded by the storage I/O throughput (5-6.2M IOPs which is >80% of peak storage throughput), but there is an upper limit beyond which scaling SSDs will not help. This limit is bounded by the I/O request generation rate and how efficiently the applications utilize the cache. Further performance improvement requires application modification in
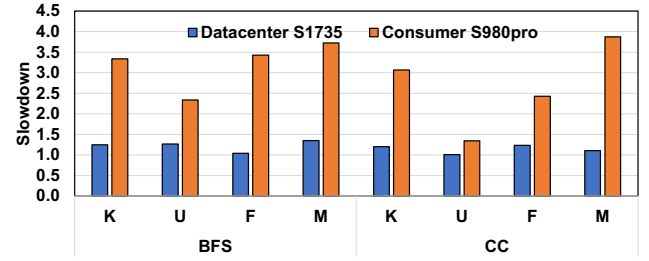
work assignment/scheduling to trigger the BaM cache misses earlier during execution.

Sources of performance improvement: Figure 8 shows the sources of performance improvement with BaM. Using a naive cache (without warp coalescing or cache-line reference reuse) in BaM provides on average 11.9× (BFS) and 12.65× (CC) speedup over a no-cache implementation. This is because the BaM cache minimizes I/O amplification. When the application exploits BaM's warp coalescing and reference reuse efficiently, we observe an additional 6.07× (BFS) and 11.24× (CC) speedup on average.

Impact of SSD type: We now evaluate BaM with different types of SSDs: datacenter grade Samsung DC 1735 and consumer grade Samsung 980pro. Figure 9 shows the slowdown observed by BaM with each type using four SSDs when compared to Optane drives. Samsung DC 1735 and the Intel Optane SSD have similar performance for almost all workloads, since both achieve similar peak 4KB read IOPs. With the Samsung 980pro SSD, BaM prototype is on average 3.21× and 2.68× slower for BFS and CC workload. These results are very encouraging as the consumer-grade SSDs provide by far the (lowest cost) among all the SSD technologies.

Sensitivity Analysis: Next, we evaluate the effect of varying the cache size and the number of I/O queue pairs on the application performance with BaM. We limit our discussions to the K dataset as similar trends are observed for other datasets.

Cache Size: As shown in Figure 10, even with a 1GB cache, BaM does not experience any performance degradation as the cache can still capture the same level of spatial and temporal locality as 8GB.
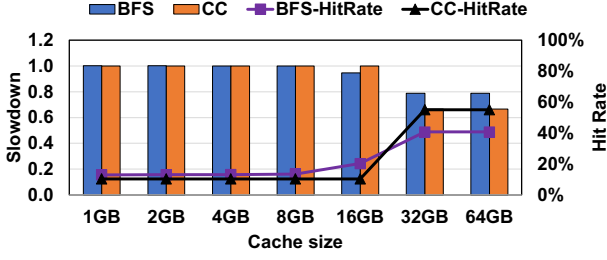
Figure 10: Impact of BaM cache capacity for K dataset relative to an 8GB cache. BaM maintains the same performance with a 1GB cache as with a 8GB cache.
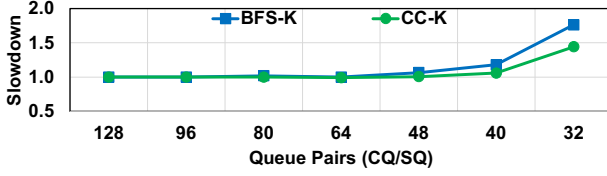

Figure 11: Impact of the number of NVMe SQ/CQ queue pairs on performance for K dataset relative to 128 queue pairs. BaM maintains the same performance as the number of queue pairs decreases and only starts degrading with 40 queue pairs.

Increasing the cache size to 32GB and 64GB enables the BaM cache to keep the entire working-set and only incur the cold cache misses.

Number of Queue Pairs: As shown in Figure 11, the application performance holds up well as the number of queue pairs decreases and degrades only at 40 (or lower) queue pairs, when the queue contention and the NVMe protocol serialization required per queue start to impact performance.

5.3 I/O Amplification Benefit for Data Analytics

Next, we evaluate the performance benefit of the BaM prototype for enterprise data analytics workloads to illustrate BaM's benefits in reduced I/O amplification and reduced software overhead while working on large structured datasets. These emerging data analytics are widely used to interpret, discover or recommend meaningful patterns in data that is collected over time. Although data-analytics applications are currently a small subset of all GPU applications, it is worth noting that the market size for this workload was $205 billion in 2020 and $230 billion in 2021 [18].

Setup: For this evaluation we use the NYC taxi ride dataset [60] and six queries, to compare the performance of BaM against the state-of-the-art GPU accelerated data analytics framework, RAPIDSv21.12 [46]. The queries used for evaluation altogether answer the final query: "Q5: What is the average dollar/mile the driver makes for trips that are at least 30 miles?" We start with a scan of the distance column (Q0), and add the total cost (Q1), surcharges (Q2), hail fee (Q3), tolls (Q4), and taxes (Q5) for only the trips that are at least 30 miles to get the penultimate query. For RAPIDS, we pin the entire dataset file in the Linux CPU page cache, enabling RAPIDS to read the file contents directly from the CPU DRAM without issuing an I/O request to the storage. This captures
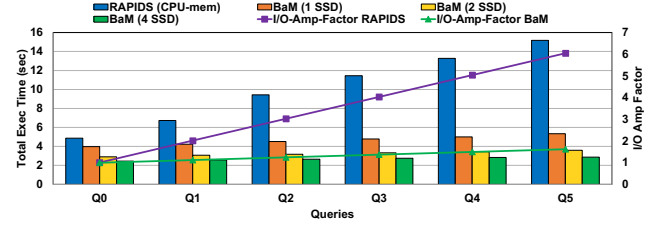

Figure 12: Performance of BaM and RAPIDS for data analytics queries on NYC Taxi dataset. BaM is up to 5.3× faster than the RAPIDS framework due to the reduced I/O amplification and reduced overhead in managing GPU memory.

the best performance modern systems can achieve. For evaluating BaM, we replicate data across SSDs, using up to four Intel Optane P5800X SSDs with 4KB cache-lines and 8GB cache capacity.

Results: Even with the single SSD, BaM outperforms RAPIDS performance for all queries as shown in Figure 12. For Q0, BaM observes a speed up of 1.22× over baseline even without any I/O amplification benefit. This is because, despite having the entire dataset preloaded into the CPU page cache, baseline RAPIDS experiences software overheads on the CPU to find and move data and manage the GPU memory.

With each additional data-dependent metrics (Q1 through Q5), the BaM performance advantage over RAPIDS increases as shown in Figure 12. The additional performance gain is attributed to BaM's reduced I/O amplification due to on-demand data fetching, while RAPIDS must transfer entire columns to the GPU memory. With additional data-dependent metrics, as shown in Figure 12, the baseline suffers from increased I/O amplification. In contrast, BaM's ability to make on-demand access to data as well as overlap compute, cache management, and many I/O requests helps it to handle multiple data-dependent columns nearly as efficiently as a single data-dependent column.

BaM's end-to-end application time scales by up to 1.46× with two SSDs and 1.62× with four SSDs when compared to a single SSD BaM configuration. The sub-linear scaling is due to BaM's setup overhead for pinning and mapping I/O queues and I/O buffers for DMA in GPU memory becomes more significant as more SSD's are used. Nevertheless, with four SSDs, BaM achieves up to 5.3× speed-up over the baseline.

5.4 VectorAdd Workload

In this section, we evaluate BaM on vectorAdd, a write-intensive workload. The vectorAdd workload takes two input arrays with four billion elements each, where each element is of eight-byte size to generate one output array of four billion elements. We assume the input vectors are in storage, and output requires to be written to the storage. For the baseline, we use a proactive tiling approach and split the four billion elements into five tiles. This allows the baseline to fully overlap the output write operation of the current tile with the loading of input vectors for the next tile. For BaM, the GPU kernel works on an entire dataset (no tiling), where each warp is assigned to a cache-line of the output vector.
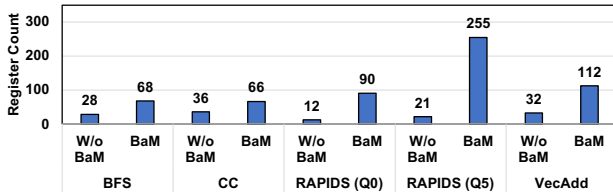
Figure 13: Per-thread register usage in applications. Even though BaM requires more registers per thread, all studied applications are storage I/O bound and register spilling or the reduced occupancy does not bottleneck performance.

BaM is 1.51× slower than the baseline implementation. BaM currently does not support overlapping between read miss handling and write-back activities, thus exposing the entire write latency to the application. We can address this by enabling asynchronous write-back in the BaM system, which we leave as future work.

5.5 SM Resource Utilization

Figure 13 shows the register usage per thread with and without BaM in all studied applications. Register spilling is observed in the RAPIDS workload. Neither BaM nor the applications use any shared memory. We do not enforce any occupancy constraints to limit register usage.

6 RELATED WORK

6.1 Optimized CPU-Centric Model

Most existing GPU programming models and applications were designed with the assumption that the working dataset fits in the GPU memory. If it doesn't, application specific techniques are employed to process large data on GPUs [8, 12, 19, 24, 29, 32, 47, 52, 56, 62, 63].

SPIN [8] and NVMMU [70] propose to enable peer-to-peer (P2P) direct memory access using GPUDirect RDMA from SSD to GPU and exclude the CPU from the data path. SPIN integrates the P2P into the standard OS file stack and enables page cache and read-ahead schemes for sequential reads. GAIA [12] further extends SPIN's page cache from CPU to GPU memory. Gullfoss [62] provides a high-level interface that helps in initializing and using GPUDirect APIs efficiently. Hippogriffdb [32] provides P2P data transfer capabilities to the OLAP database system. GPUDirect Storage [47] is a product that removes the CPU memory from the data path between SSDs and GPUs using GPUDirect RDMA technology. AMD has had similar efforts in RADEON-SSG products [3]. These works still require the CPU to orchestrate data movement. BaM allows any GPU thread to initiate data accesses to the SSDs.

6.2 Prior Accelerator-Centric Systems

ActivePointers [57], GPUfs [58], GPUNet [29], and Syscalls for GPU [64] have previously attempted to enable accelerator-centric model for data orchestration. GPUfs [58] and Syscalls for GPU [64] first allowed GPUs to request file data from the host CPU. Active-Pointers [57] added a memory-map like abstraction on top of GPUfs to allow GPU threads memory-like access to file data. Dragon [37] incorporated storage access to the UVM [48] page faulting mechanism. However, as shown in § 5 and [50], these approaches use

less-parallel CPUs to handle data demands from the massively parallel GPU and result in poor overall performance.

We also acknowledge the prior work in moving network control to the GPU [15], however storage presents a new set of challenges. With BaM, we enable GPUs to efficiently and directly access storage and show the performance and cost benefits for real applications.

6.3 Hardware Extensions

Prior work has proposed to replace or integrate GPU's global memory with non-volatile memories [65, 66, 71–73]. DCS [2] proposed enabling direct access between storage, network, and accelerators with an FPGA providing the required translation services for coarse-grain data transfers. Enabling persistence within the GPU has recently been proposed [6]. We acknowledge these efforts and further validate the need to enable large memory capacity for emerging workloads. More importantly, the BaM prototype aims to use emerging disaggregated storage hardware components to provide significant performance advantages to end-to-end applications with very large real-world datasets.

7 CONCLUSION

In this work, we make a case for enabling GPUs to orchestrate high-throughput, fine-grained accesses to storage, without the CPU software overhead, in a new system architecture called BaM. BaM mitigates the I/O amplification problem by allowing the GPU application compute code to read or write at finer granularities on-demand. As BaM supports the storage access control plane functionalities including caching, translation, and protocol queues on the GPU, it avoids the costly CPU-GPU synchronization, OS kernel crossing, and software bottlenecks that have limited the achievable storage access throughput. Using off-the-shelf hardware components, we built a prototype of BaM and show on multiple applications and datasets that BaM is a viable/superior alternative to DRAM-only and other state-of-the-art solutions.

A ARTIFACT APPENDIX

A.1 Abstract

BaM is a novel system architecture that defines mechanisms for GPU threads to efficiently initiate and orchestrate storage accesses. As such, with the artifact, i.e., a prototype implementation of BaM, the authors demonstrated the following: (1) BaM is functional with a single SSD and provides the contributions described in the paper, (2) BaM is functional with multiple SSDs - up to 2 SSDs can be tested with the provided system, (3) BaM is functional with different types of SSDs - consumer-grade Samsung 980 Pro and datacenter-grade Intel Optane SSDs available in the provided system, (4) BaM works with different applications - microbenchmarks and graph applications (with real datasets) that are provided for artifact evaluation. Based on these demonstrations, this paper has been awarded the Artifact Available and Artifact Functional badges.

A.2 Artifact Checklist (Meta-information)

- **Compilation:** For the artifact evaluation reviewers, the authors provided access to a machine where the required compilation tool chains was already set up. For general details, refer to software dependencies (§A.3.3).
- **Dataset:** As the dataset size is over 500GB, the datasets were pre-loaded on the SSDs in the provided system for the artifact evaluation reviewers. For general details refer to (§A.3.4).
- **Run-time environment:** Linux Kernel 5.8.x. CUDA, etc. Refer to software dependencies (§A.3.3) for details. Root access is required.
- **Hardware:** For the artifact evaluation reviewers, the authors provided access to a machine where the required hardware was already set up. For general details refer to hardware dependencies (§A.3.2).
- **Run-time state:** Yes
- **Execution:** Only one artifact evaluation reviewer can execute the evaluation applications at a time on the provided machine. 2-3 hours aggregate run time.
- **Metrics:** Execution time, bandwidth and IOPS.
- **Output:** Console. See provided log files for expected output.
- **Experiments:** Follow the instructions at https://github.com/ZaidQureshi/bam/tree/master/asplosae#readme
- **How much disk space required (approximately)?:** >500GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 3 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD-2-Clause
- **Archived at:** https://doi.org/10.5281/zenodo.7217356 [51]

A.3 Description

A.3.1 Access. BaM's codebase is publicly available here: https://github.com/ZaidQureshi/bam.git

A.3.2 Hardware dependencies. Refer to https://github.com/ZaidQureshi/bam#hardwaresystem-requirements

A.3.3 Software dependencies. Refer to https://github.com/ZaidQureshi/bam#system-configurations

A.3.4 Datasets. Refer to https://github.com/ZaidQureshi/bam#example-applications

```
# Building the BaM library and applications
$ git submodule update --init --recursive
$ mkdir -p build; cd build
$ cmake ..
$ make libnvm -j      # builds library
$ make benchmarks -j  # builds benchmark program

# Building the BaM kernel module
$ cd module
$ make -j
```

Listing 2: Building BaM library, applications and kernel module.

```
$ dmesg | grep nvme0
[126.497670] nvme nvme0: pci function 0000:65:00.0
[126.715023] nvme nvme0: 40/0/0 default/read/poll queues
[126.720783] nvme0n1: p1
[190.369341] EXT4-fs (nvme0n1p1): ...

# Unbind the SSD from kernel NVMe driver
$ echo -n "0000:65:00.0" >
    /sys/bus/pci/devices/0000\:65\:00.0/driver/unbind

# Load the BaM driver.
$ cd build/module
$ sudo make load
```

Listing 3: Identifying SSD to use and binding it to the BaM driver.

```
$ sudo ./bin/nvm-block-bench --threads=262144 --blk_size=64 --
    reqs=1 --pages=262144 --queue_depth=1024 --page_size=512 --
    num_blks=2097152 --gpu=0 --n_ctrls=1 --num_queues=128 --
    random=true
```

Listing 4: Basic test.

A.4 Installation

The complete installation instructions can be found at the following link https://github.com/ZaidQureshi/bam/blob/master/asplosaoe/README.md#compiling. We briefly describe them in this section.

A.4.1 Building the Project. From the project root directory, follow the steps shown in Listing 2. The CMake configuration is supposed to auto-detect the location of CUDA, Nvidia driver and project library. After this, we should also compile the custom libnvm kernel module for NVMe devices as shown in the last two lines in Listing 2.

A.4.2 Loading/Unloading the Kernel Module. In order to use the custom kernel module for the NVMe device, we need to unbind the NVMe device from the Linux NVMe driver. To do this, first we need to find the PCIe ID of the NVMe device. If the required NVMe device want is mapped to the /dev/nvme0 block device, Listing 3 shows how to find its PCIe ID.

Next, we need to unbind the SSD from the Linux NVMe driver, as shown in Listing 3. Repeat the process for all NVMe devices that are to be remapped to the BaM driver. We next load the BaM kernel module from the build directory. This should create a /dev/libnvm* device file for each NVMe SSD that is not bound to the NVMe driver.

A.4.3 Basic Test. The evaluator can run the command available at https://github.com/ZaidQureshi/bam/blob/master/asplosaoe/README.md#running-the-io-stack-component (also shown in Listing 4) as a basic test. The expected output (sans performance numbers) can be found at https://github.com/ZaidQureshi/bam/blob/master/asplosaoe/nvm_block_bench_1_sam.log.

A.5 Evaluation

A.5.1 Experimental workflow. Each benchmark is an executable compiled in the process described earlier. Each benchmark has various command line parameters (explained in its help) and outputs relevant performance metrics like time and throughput.

A.5.2 Evaluation and expected results. Evaluation should validate the goals that are defined in the artifact evaluation abstract. For each of the goals, we have prepared a set of commands, their descriptions and expected results at https://github.com/ZaidQureshi/bam/tree/master/asplosaoe#readme.

REFERENCES

[1] B. Acun, M. Murphy, X. Wang, J. Nie, C. Wu, and K. Hazelwood. 2021. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE Computer Society, Los Alamitos, CA, USA, 802–814.

[2] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. 2015. DCS: A fast and scalable device-centric server architecture. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, New York, NY, USA, 559–571.

[3] AMD. 2021. RADEON-SSG API Manual. https://www.amd.com/system/files/documents/ssg-api-user-manual.pdf.

[4] Jens Axboe. 2020. Efficient IO with io_uring.

[5] BaM 2022. BaM GitHub Repository. https://github.com/ZaidQureshi/bam.

[6] Ardhi Wiratama Baskara Yudha, Keiji Kimura, Huiyang Zhou, and Yan Solihin. 2020. Scalable and Fast Lazy Persistency on GPUs. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 252–263.

[7] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[8] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. 2017. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 167–179.

[9] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: a scalable fully distributed Web crawler. *Software: Practice and Experience* 34, 8 (2004), 711–726.

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[11] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[12] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. 2019. GAIA: An OS Page Cache for Heterogeneous Systems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 661–674.

[13] CDW 2022. CDW. https://www.cdw.com.

[14] F. J. Corbato. 1968. A Paging Experiment With The Multics System. *Technical Report, Massachusetts Institute of Technology, Cambridge, Project MAC* (1968).

[15] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrdma: GPU-Side Library for High Performance Networking from GPU Kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers* (Kyoto, Japan) *(ROSS '16)*. Association for Computing Machinery, New York, NY, USA.

[16] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (Dec. 2011), 25 pages.

[17] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, 478–493.

[18] Fortune Business Insights. 2021. Big Data Analytics Market | 2021 Size, Growth Insights, Share, COVID-19 Impact, Emerging Technologies, Key Players, Competitive Landscape, Regional and Global Forecast to 2028. https://tinyurl.com/2p8a8sbx.

[19] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. 2010. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 347–358.

[20] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 488–501.

[21] H3 Platform 2022. H3 Platform. https://www.h3platform.com.

[22] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. *arXiv preprint arXiv:2103.09430* (2021).

[23] Intel. 2021. Intel® Optane™ Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[24] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. 2012. Dynamically Managed Data for CPU-GPU Architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. Association for Computing Machinery, New York, NY, USA, 165–174.

[25] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 494–508.

[26] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. 2018. Designing a True Direct-Access File System with DevFS. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA.

[27] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA.

[28] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 756–771.

[29] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 201–216.

[30] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR'17)*.

[31] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Shanghai, China.

[32] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proceedings of the VLDB Endowment* 9, 14 (Oct. 2016), 1647–1658.

[33] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (March 2008), 39–55.

[34] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 819–835.

[35] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. 2016. DAOS and Friends: A Proposal for an Exascale Storage System. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 585–596.

[36] Vikram Sharma Mailthody. 2022. *Application Support And Adaptation For High-throughput Accelerator Orchestrated Fine-grain Storage Access.* Ph. D. Dissertation. University of Illinois Urbana-Champaign.

[37] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. 2018. DRAGON: Breaking GPU Memory Capacity Limits with Direct NVM Access. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, 13 pages.

[38] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. 2021. SmartIO: Zero-Overhead Device Sharing through PCIe Networking. *ACM Transactions on Computing System* 38, 1–2, Article 2 (jul 2021), 78 pages.

[39] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen mei W. Hwu. 2020. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs. *Proceedings of VLDB Endowment* 14 (2020), 114–127.

[40] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2021. Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models.

[41] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019).

[42] Nvidia 2016. State of GPUDirect Technologies. https://on-demand.gputechconf.com/gtc/2016/presentation/s6264-davide-rossetti-GPUDirect.pdf.

[43] Nvidia 2019. How to make your life easier in the age of exascale computing using NVIDIA GPUDirect technologies. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9653-how-to-make-your-life-easier-in-the-age-of-exascale-computing-using-nvidia-gpudirect-technologies.pdf.

[44] Nvidia 2020. NVIDIA DGX A100. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[45] Nvidia 2020. NVIDIA Tesla A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[46] Nvidia 2021. CUDA RAPIDS: GPU-Accelerated Data Analytics and Machine Learning. https://developer.nvidia.com/rapids.

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

[48] Nvidia 2022. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners.

[49] Zaid Qureshi. 2022. *Infrastructure to Enable and Exploit GPU Orchestrated High-Throughput Storage Access on GPUs*. Ph. D. Dissertation. University of Illinois Urbana-Champaign.

[50] 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. 2022. GPU-Orchestrated On-Demand High-Throughput Storage Access in the BaM System Architecture. arXiv. https://arxiv.org/abs/2203.04910

[51] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelago, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, Michael Garland, William Dally, and Wen-mei Hwu. 2022. *GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture*. https://doi.org/10.5281/zenodo.7217356 This zendo version has the ASPLOS AEC evaluation. As this project is in continuous development, the most updated version of the project can be accessed in the following link: https://github.com/ZaidQureshi/bam.git.

[52] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. 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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, 14 pages.

[53] Yujie Ren, Changwoo Min, and Sudarsun Kannan. 2020. CrossFS: A Cross-layered Direct-Access File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 137–154.

[54] Samsung. 2021. Samsung 980 PRO SSD. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/.

[55] Samsung ZNAND 2021. Samsung Z-NAND Technology Brief. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[56] Hyunseok Seo, Jinwook Kim, and Min-Soo Kim. 2015. GStream: A Graph Streaming Processing Method for Large-Scale Graphs on GPUs. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. Association for Computing Machinery,

New York, NY, USA, 253–254.

[57] Sagi Shahar, Shai Bergman, and Mark Silberstein. 2016. ActivePointers: A Case for Software Address Translation on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 596–608.

[58] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 485–498.

[59] Justin Sybrandt, Michael Shtutman, and Ilya Safro. 2017. MOLIERE: Automatic Biomedical Hypothesis Generation System. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17)*. Association for Computing Machinery, New York, NY, USA, 1633–1642.

[60] The City of New York. 2021. TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[61] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. *Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers*. Association for Computing Machinery, New York, NY, USA, 117–131.

[62] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. 2015. Gullfoss : Accelerating and Simplifying Data Movement among Heterogeneous Computing and Storage Resources.

[63] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 53–65.

[64] Ján Veselý, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H. Loh, Mark Oskin, and Steven K. Reinhardt. 2018. Generic System Calls for GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 843–856.

[65] Jeffrey S. Vetter and Sparsh Mittal. 2015. Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing. *Computing in Science Engineering* 17, 2 (2015), 73–82.

[66] Bin Wang, Bo Wu, Dong Li, Xipeng Shen, Weikuan Yu, Yizheng Jiao, and Jeffrey S. Vetter. 2013. Exploring hybrid memory for GPU energy efficiency through software-hardware co-design. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 93–102.

[67] Weka.io. 2021. WekaFS Architecture Whitepaper. https://www.weka.io/wp-content/uploads/files/2017/12/Architectural_WhitePaper-W02R6WP201812-1.pdf.

[68] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. *CoRR* (2012).

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

[70] Jie Zhang, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. 2015. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 13–24.

[71] Jie Zhang and Myoungsoo Jung. 2020. ZnG: Architecting GPU Multi-Processors with New Flash for Scalable Data Analysis. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 1064–1075.

[72] Jie Zhang and Myoungsoo Jung. 2021. Ohm-GPU: Integrating New Optical Network and Heterogeneous Memory into GPU Multi-Processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 695–708.

[73] Jie Zhang, Miryeong Kwon, Hyojong Kim, Hyesoon Kim, and Myoungsoo Jung. 2019. FlashGPU: Placing New Flash Next to GPU Cores. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, 6 pages.

[74] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. 2020. Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems. In *Third Conference on Machine Learning and Systems*.