AGILE: Lightweight and Efficient Asynchronous GPU-SSD Integration

Zhuoping Yang
Brown University
Providence, USA
zhuoping_yang@brown.edu

Jinming Zhuang
Brown University
Providence, USA
jinming_zhuang@brown.edu

Xingzhen Chen
Brown University
Providence, USA
xingzhen_chen@brown.edu

Alex Jones
Syracuse University
Syracuse, USA
akj@syr.edu

Peipei Zhou
Brown University
Providence, USA
peipei_zhou@brown.edu

Abstract

GPUs are critical for compute-intensive applications, yet emerging workloads such as recommender systems, graph analytics, and data analytics often exceed GPU memory capacity. Existing solutions allow GPUs to use CPU DRAM or SSDs as external memory, and the GPU-centric approach enables GPU threads to directly issue NVMe requests, further avoiding CPU intervention. However, current GPU-centric approaches adopt synchronous I/O, forcing threads to stall during long communication delays.

We propose AGILE, a lightweight asynchronous GPU-centric I/O library that eliminates deadlock risks and integrates a flexible HBM-based software cache. AGILE overlaps computation and I/O, improving performance by up to 1.88× across workloads with diverse computation-to-communication ratios. Compared to BaM on DLRM, AGILE achieves up to 1.75× speedup through efficient design and overlapping; on graph applications, AGILE reduces software cache overhead by up to 3.12× and NVMe I/O overhead by up to 2.85×; AGILE also lowers per-thread register usage by up to 1.32×.

CCS Concepts

• **Information systems** → **Storage architectures**; • **Computing methodologies** → **Parallel computing methodologies**; • **Hardware** → **External storage**.

Keywords

GPUs, SSDs, Asynchronous I/O, Software-managed cache, Memory hierarchy, Storage systems

ACM Reference Format:
Zhuoping Yang, Jinming Zhuang, Xingzhen Chen, Alex Jones, and Peipei Zhou. 2025. AGILE: Lightweight and Efficient Asynchronous GPU-SSD Integration. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3712285.3759778

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759778

1 Introduction

Graphics Processing Units (GPUs) have become the de facto accelerator widely used for computationally intensive applications such as graphics rendering [26, 53], deep learning [26, 33], and high-performance computing [58, 59, 66]. However, modern applications are increasingly data-intensive, often processing data that far exceeds GPU memory capacity [19, 31, 50]. For example, training large-scale models like GPTs [1, 5] involves billions of parameters and terabytes of training data [49]. Similarly, analyzing large graphs for social networking or ranking websites touches on billions of vertices and trillions of edges [9]. Recommender systems also handle data ranging from gigabytes to petabytes [51]. Moreover, while GPUs' computational power has increased rapidly, their memory capacity has not kept the same pace [19]. These new trends necessitate innovative memory extension techniques and optimizations.

To expand GPUs' memory, existing solutions resort to CPU memory [2, 44, 52, 65]. For example, Nvidia Unified Memory enables GPUs and CPUs to share a single memory address space so that GPUs can access CPU memory without explicit memory copies [44]. However, scaling the CPU memory to tens of terabytes is still considered a challenge [48]. Another approach is extending GPU memory using SSDs [3, 4, 62], which provide much larger space but entail sophisticated designs for better performance. GPUDirect Storage [40] enables direct data transfers between GPUs and SSDs without involving the memory of the CPU as an intermediary, thereby eliminating the overhead of using CPU memory as a staging buffer. Microsoft proposes DeepNVMe [32], which offers additional optimizations, such as asynchronous I/O operations and integration with ZeRO-Infinity [50] for large neural networks. However, GPUDirect Storage and DeepNVMe still require the CPU to initiate the data transfer. As the computational workloads are offloaded onto the GPUs, the CPU lacks visibility to requests made by GPU threads in flight. Consequently, frequent synchronization between the GPU and the host CPU is necessary, leading to significant performance degradation [48].

The emerging interconnect technology CXL is built on top of PCIe and offers new protocols, such as `CXL.memory` and `CXL.cache`, to efficiently extend host memory [14]. `CXL.memory` allows devices to use `load` or `store` instructions to access other devices' memory or storage. `CXL.cache` further enables devices to coherently cache memory that physically resides on other devices. CXL-enabled

SSDs are promising candidates for helping maintain coherence for memory expansion with SSDs [63], but are not currently a complete solution for expanding GPUs' memory. This is because the flash memory access time is at the microsecond level [63], which is orders of magnitude higher than High-Bandwidth Memory (HBM), where CXL is primarily deployed. A solution to hide the latency of SSDs is still necessary.

Overlapping memory access with computation is a common technique used to tolerate slow data movement [8, 20, 62]. For example, ALCOP [22] utilizes the CUDA-provided asynchronous data movement API to explore multi-stage pipelining. This avoids GPU idle time caused by synchronous data movement. However, inside a GPU kernel, only asynchronous data movement from global memory (or pinned host memory) to shared memory can be initiated using existing CUDA APIs [30], and the GPU's shared memory is limited, e.g., 164 KB per Streaming Multiprocessor on an A100 GPU [41]. Using a larger buffer for asynchronous loads per thread has been demonstrated to have more performance benefits when using an overlapping technique [28].

GPU-centric storage access is another method to avoid the synchronization overhead between GPUs and CPUs. BaM [48] is the first GPU-centric method that enables GPU threads to directly initiate NVMe I/O requests while bypassing the host CPU. It tolerates long SSD access latency via massive concurrent I/Os enabled by the GPU's high parallelism. This approach eliminates CPU intervention overhead. However, it adopts a synchronous access model, and threads must wait for the I/O requests to be completed before concurrently starting computation or issuing other commands. As a result, communication time cannot be hidden in each GPU thread, and applications must rely on runtime warp scheduling to preempt stalled warps and schedule other ready warps to avoid wasting GPU cycles [25], which is not always effective and leaves space for further optimization opportunities.

In contrast, an asynchronous I/O model can better tolerate long latency in accessing SSDs by overlapping communication with computation [27]. However, designing a GPU-centric asynchronous I/O model is challenging, as the massive GPU threads may compete on shared resources, e.g., NVMe queues, software-defined cache, etc., leading to performance degradation. Using locks before accessing these shared resources is a common method to avoid resource conflicts, but in an asynchronous model, allowing threads to hold locks can lead to deadlock issues. For instance, if multiple threads asynchronously request SSD data, a request queue can fill prior to commands that check for completion and subsequently clear the completed request from the request queue, creating a deadlock. In addition, efficient lock handling is necessary to avoid performance degradation from the software API side.

Moreover, BaM [48] only supports a fixed cache policy for its software cache on GPUs' HBM. This limits the cache policy customization for various applications. As new caching policies [17, 35, 47] are continuously designed, it is important for storage systems to choose the best software-defined caching policy under various workloads and requirements [60].

To address these needs and challenges, we propose AGILE, a GPU-centric GPU-SSD integration that enables GPU threads to issue NVMe requests asynchronously and efficiently while eliminating deadlock risks.

Our contributions are highlighted as follows:

- We propose AGILE, enabling the GPU to issue NVMe commands asynchronously. To the best of our knowledge, AGILE is the first GPU-centric asynchronous I/O model.
- We implement a robust lock-based asynchronous transaction mechanism, which allows GPU threads to issue NVMe commands asynchronously without holding any locks. Our approach efficiently eliminates possible deadlocks and data hazards.
- We integrate a flexible software cache hierarchy in AGILE to utilize GPU HBM, which allows users to customize their cache policy and provides a simple interface for increased usability.
- We evaluate AGILE on micro-benchmarking and applications. The results show that AGILE enables overlapping at the thread level and achieves up to 1.88× speedup over a synchronous I/O model. Compared with state-of-the-art work, BaM, AGILE achieves up to 1.75× reduction in end-to-end execution time on DLRMs; in graph applications, AGILE demonstrates lower API overhead in managing software cache and NVMe I/O requests up to 3.12× and 2.85×, respectively; furthermore, AGILE consumes fewer registers and exhibits up to 1.32× reduction in the usage of registers.
- We open-source AGILE with detailed guides for users to leverage AGILE and customize AGILE components in various applications: https://github.com/arc-research-lab/AGILE

2 Background & Design Challenges

In this section, we first introduce the background of the NVMe protocol and how GPU threads are scheduled and hide memory access latency in CUDA. Then, we present the challenges in supporting an asynchronous I/O model on GPUs.

2.1 Background of NVMe Protocol

Non-Volatile Memory Express (NVMe) is a standard protocol that allows software to communicate with non-volatile memory via PCIe [46]. Software can access an NVMe SSD via an I/O queue pair, consisting of a submission queue (SQ) and a completion queue (CQ). With an I/O queue pair, the software is responsible for maintaining the SQ tail pointer, which indicates the next available SQ entry (SQE) for a new command, and the CQ head pointer, which is used to receive the next completion from the SSD. To issue an NVMe command, the software writes a new command to the next available SQE and notifies the changes in SQ to the SSD by moving the SQ tail pointer and updating the new SQ tail by writing to the corresponding SQ doorbell register in the SSD's PCIe Base Address Registers (BAR). Then, the SSD fetches the newly added command, and after execution, the SSD returns a completion to the next available CQ entry (CQE). After receiving a completion, the software needs to respond to SSD by increasing the CQ head pointer and updating the associated CQ doorbell register. This is necessary for SSDs to release the CQE and reuse it for another command; otherwise, the SSDs will stall while waiting for available CQEs. This queue-based approach also allows software to issue multiple commands in a batch and increase the SQ tail pointer by the number of newly inserted commands. The software can detect

and process the completion message by either polling the CQ or responding to an interrupt triggered by the SSD. To achieve high parallelism, NVMe SSDs allow multiple SQs/CQs to be registered and used concurrently.

2.2 GPU Threads Scheduling & Asynchronous Data Movement in CUDA

To meet the increasing high throughput demands, modern GPUs can execute tens of thousands of threads in parallel via Single Instruction, Multiple Threads (SIMT) [36]. The GPU threads are grouped into thread blocks, and the threads in each thread block will be scheduled onto the same Streaming Multiprocessor (SM) [38]. If the hardware resource, such as the number of registers and the shared memory, is enough for an SM to serve more than one thread block, each SM can accommodate multiple thread blocks simultaneously. Current GPUs adopt a static resource allocation model, which can cause SM underutilization. Once the thread blocks are scheduled onto SMs, they will occupy the SMs until their tasks are finished. This prevents new thread blocks from being scheduled, even if the scheduled thread blocks are stalled due to some high-latency operations. This problem of SM underutilization is mitigated by warp-level scheduling. The SM will schedule threads at the granularity of warps (typically 32 threads in a warp). If some warps stall due to high-latency operations such as fetching data from memory or SSDs, other ready warps from the same thread block or different thread blocks can be scheduled to keep the SM busy. However, this mechanism is not sufficient, especially when many warps encounter memory or I/O stalls.

To avoid stalls caused by memory access, users can use asynchronous data movement APIs such as `cuda::memcpy_async` [39] or `cp.async` [36] in CUDA to hide latency with computation tasks. However, these asynchronous data movement APIs only allow data transfers from GPU global memory or pinned host memory to shared memory in SMs [30]. Using larger buffers for asynchronous loads will lead to a higher performance increase [28], but the shared memory is limited in each SM, e.g., 164 KB per SM on an A100 GPU [41].

2.3 Design Challenges in Asynchronous GPU-SSD integration

2.3.1 Deadlock in NVMe Queues. Designing an efficient asynchronous model for GPU-SSD integration is challenging as a massive number of threads share limited resources such as NVMe queues and the software cache. Acquiring locks before accessing these resources is necessary to avoid conflicts, but can introduce deadlock.

For NVMe queues, when a thread puts a new NVMe command into an SQ, the corresponding SQ entry will remain locked to prevent other threads from using the same entry until the SSD has received the command.

Figure 1 illustrates an example of deadlock when Thread-1 and Thread-2 need to execute NVMe commands asynchronously in parallel. First, Thread-1 successfully acquires the SQ and places its read request into an available entry. However, before this thread can move to line 3, Thread-2 gains access to the SQ and adds its request to the last available entry, which fills the SQ ①. Now, because the SQ is full, both threads become stuck at Line 3, they

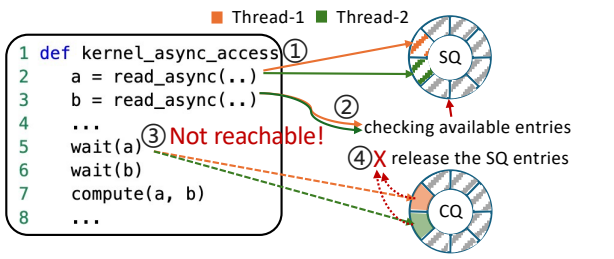

Figure 1: A deadlock example caused by sharing NVMe queues in asynchronous execution.

continue to check for the next available SQ entry ②. Therefore, both threads cannot reach ③, where they check the completions in CQ to confirm their issued commands have been processed by the SSD and then release locks in SQ. Even though the corresponding completions become available in the CQ, if Threads-1 and 2 own all the occupied SQ entries, none can be released ④, resulting in a deadlock.

For larger numbers of threads, this deadlock remains a concern as many threads will request multiple operands in line 3, hence, filling the queue prior to anyone reaching ③.

2.3.2 Deadlock in the Software Cache. AGILE promises to offer flexibility in the software cache policy, and therefore, eliminating the potential for deadlock caused by the software cache is necessary. A common scenario resulting in a deadlock is simultaneous threads accessing multiple cache lines. For example, one compute kernel needs multiple operands that are stored in different cache lines. To prevent redundant SSD accesses, once a thread checks the software cache and the requested data is found—i.e., a cache hit occurs—access to the corresponding cache lines must be atomic to avoid eviction before accesses in process are completed. When multiple threads block cache line eviction while requesting new cache lines, a deadlock could occur.

2.3.3 Potential Performance Degradation. Flash memory cannot be accessed randomly, and data is managed at a coarse-grained page level, typically 4KB per page [18]. Therefore, the software cache line should align with the SSDs' granularity. This alignment can avoid redundant I/Os when multiple threads access different parts of the same SSD page concurrently. To ensure correctness during accessing the same cache line simultaneously, atomic operations are required to avoid conflicts and data hazards. It is crucial to implement an efficient lock mechanism to prevent performance degradation and deadlock.

Furthermore, in NVMe queues, although multiple threads can insert their commands into the same SQ concurrently, updating the SQ doorbell register must be serialized. This is because concurrent writes to the same doorbell registers may cause inconsistent SQ tail values in SSDs. Besides, the serialization ensures memory consistency so that the newly submitted commands are visible in global memory before the SQ doorbell registers are updated. Improper handling of this serialization may also cause performance degradation.

Lastly, real-world SSD devices only support a small number of I/O queue pairs compared to the massive living GPU threads. For example, a maximum of 128 queue pairs in Samsung 980 PRO NVMe SSD [57]. Therefore, the completions from SSDs tend to concentrate in a small number of completion queues, which requires an efficient

and low-overhead mechanism to consume the completions to avoid stalls from SSDs.

3 AGILE Design & Implementation

In this section, we will first give an overview of AGILE in Section 3.1. Then, we present the main components of AGILE. In Section 3.2, we will discuss how AGILE avoids the deadlock problem resulting from NVMe queues and processes completions from SSDs in parallel. We will discuss how AGILE deals with the serialization process required by the NVMe SQs and coalescing redundant requests in Section 3.3. In Section 3.4, we will present the software-managed cache in AGILE and discuss how AGILE extends cache coherency to user-specified buffers. Finally, we will present an example program, illustrating how AGILE can be used, and introduce a debug option provided in AGILE.

3.1 Overview of AGILE System

Figure 2 presents an overview of the AGILE system, which enables efficient asynchronous GPU-SSD communication. The system involves three types of hardware, including SSDs, a GPU, and a host CPU. The host CPU manages admin queues, located in DRAM, to establish GPU-SSD PCIe peer-to-peer (P2P) communication. The NVMe SSDs are connected to the system via PCIe, their PCIe BARs are exposed to the host CPU for management, and their doorbell registers are registered to GPU for GPU-centric data transfers. Within the GPU, AGILE consists of a lightweight service to handle I/O queues for users (Section 3.2), a software controller to manage cached data in HBM (Section 3.4), and a Share Table to extend cache coherency to user-specified buffers (Section 3.4.1). Users can interact with AGILE through the AGILE controller (AGILE CTRL), which provides simple APIs for requesting or accessing data in SSDs or the software cache.

To establish the PCIe P2P communication, the SSDs and the GPU must be able to access the other device's memory. To let an NVMe SSD access I/O queues (SQs/CQs) and the software cache, we need to allocate a contiguous memory space on GPU HBM, pin the memory space to avoid being swapped out, and get the physical address of the memory space to enable Direct Memory Access (DMA) for the SSD to access the GPU HBM. GDRCopy [42] is designed for direct GPU memory access from third-party devices. It runs in kernel space and serves userspace calls for allocating and pinning contiguous memory on the GPU. We modify the GDRCopy kernel module and invoke `nvidia_p2p_put_pages` in the kernel space, enabling userspace applications to access the mapping table that translates virtual addresses into physical addresses of GPU memory. Then, the physical addresses of SQs/CQs are registered to SSDs via the admin queues on the host CPU. To let the GPU notify NVMe SSDs after generating new commands, we use memory-mapping (`mmap`) to expose the SSDs' PCIe BAR to userspace and then register the doorbell registers to the GPU using `cudaHostRegister` with the `cudaHostRegisterIoMemory` flag. After this initialization process, the GPU threads can insert NVMe commands to SQs in HBM and update the doorbell registers to notify the NVMe SSDs directly, and the NVMe SSDs are able to fetch commands in GPU HBM, process them, and update the completion messages to CQs in GPU HBM directly. In AGILE, the initialization process requires CPU intervention and must be performed at the beginning of the program using AGILE.

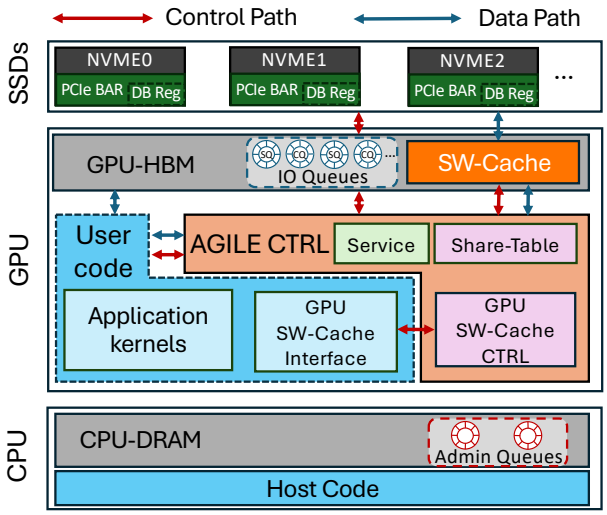

Figure 2: Overview of system architecture adopting AGILE.

AGILE provides two types of asynchronous APIs and an array-like synchronous API. The asynchronous API `prefetch(src)` is used to issue data requests from SSDs to the GPU software cache, and then the user threads can access the data directly in the GPU software cache. Another asynchronous API, `async_issue(src,dst)`, is similar to `cuda::memcpy_async` [39] or `cp.async` [36] in CUDA, but the `src` and `dst` in AGILE are more flexible and can be either SSDs' addresses or user-specified buffers in GPUs' global memory. By using user-specified buffers with `async_issue(src,dst)`, GPU threads can save multiple data chunks for later use safely without holding locks in the software cache, thereby avoiding the deadlocks described in Section 2.3.2. However, the increased flexibility of `src` and `dst` in `async_issue(src,dst)` may introduce data hazards, and we will present our solution in Section 3.4.1. The `async_issue(src,dst)` will return a barrier to let the user threads know if the data transfer is completed. Lastly, the array-like synchronous API allows users to simply view the SSDs as a two-dimensional array, and AGILE automatically checks the software cache and issues requests if the data is not available.

3.2 AGILE Service

As mentioned in Section 2.3, allowing threads to hold locks on NVMe queues is risky and can cause deadlock. However, locking SQs is necessary so that commands do not collide. To address this problem, we propose a lightweight AGILE service that runs in the background on the GPU and interacts with user threads.

3.2.1 Avoid deadlock from NVMe queues. To eliminate the deadlock risk, AGILE creates a lightweight kernel daemon on the GPU to keep checking completion queue entries (CQE) for all registered NVMe CQs in a non-blocking fashion. This service frees the user threads from the burden of processing completion messages and automatically releases shared resources for user threads after completion. Once the AGILE service receives a completion from the CQs, the corresponding locks in SQs will be released. This allows

additional SQ requests to proceed and avoids deadlock even when user threads issue multiple request commands.

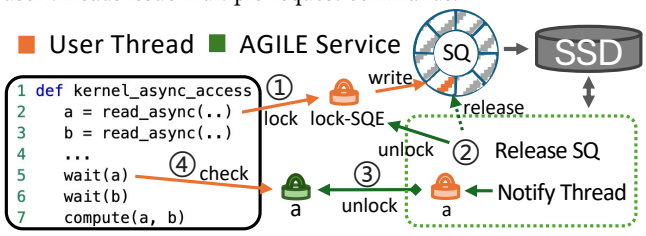

Figure 3: Avoiding NVMe Queue Deadlocks in AGILE.

Figure 3 illustrates the process of how the AGILE service assists user threads in issuing commands. In Figure 3 line 2, when a user thread successfully locks the SQ, it can safely enqueue the command into the SQ entry ①. Then, it will handoff `lock-SQE` to the AGILE service and receive back a barrier (lock `a`) representing the status of the transaction. Thus, when a thread reaches lines 2–3 and cannot add its requests to the SQ because it is full, once the AGILE service receives completions from SSDs, it can release the appropriate SQ entry directly and then clear the appropriate transaction lock ② so that the user thread will not be blocked forever. Meanwhile, the AGILE service will notify the corresponding barrier by clearing the lock `a` ③ to indicate that the transaction is finished. Finally, in line 5, if the thread arrives at line 5 prior to the SSD access completion, it will wait for the AGILE service to clear the lock `a` ④.

Since the completions may be returned out of order relative to the issued commands, the AGILE service tracks the mapping between each completion and its corresponding SQE via the Command Identifier (CID), which is a 16-bit field in the NVMe command and should be unique to identify commands within a batch using the same SQ.

3.2.2 Polling Completion Queues (CQs). Processing CQs efficiently is critical to sustain high throughput in a GPU-centric asynchronous I/O model. In practice, the NVMe SSDs only support a limited number of CQs. For example, the Samsung 980 PRO NVMe SSD supports up to 128 CQs [57]. In contrast, GPU applications typically involve a great number of threads, many of which need to share the same CQ. As a result, completions may tend to concentrate in a small number of CQs, which could lead to contention and performance bottlenecks. To ensure timely completion processing, AGILE increases intra-CQ polling parallelism by adopting a warp-centric CQ polling strategy, where each warp concurrently processes 32 CQEs within a CQ at every iteration. Meanwhile, AGILE only uses a small number of warps for CQ polling and rotates across all registered CQs in a round-robin fashion.

Algorithm 1 describes the warp-centric CQ polling routine used in the AGILE service to process CQs efficiently. When invoked, the warp is assigned with a specific CQ, and each thread is responsible for checking a single CQE within a 32-entry window. In the warp-centric CQ polling service, the threads first load the current polling offset, the CQ phase bit for monitoring the changes in CQEs, and a 32-bit mask that represents the completion status of the CQEs (line 2). If the corresponding bit in the mask is unset, which indicates the completion is not received, the threads will compare the CQEs' phase bit with the expected value. If new completions are found, the associate bits in the mask will be set to 1 (lines 5-6). When

Algorithm 1 Warp-centric CQ polling

```
1:  function CQ_Polling(cq_idx)
2:      offset, mask, phase_bit ← load_CQ(cq_idx)
3:      if mask[warp_idx] == 0 then
4:          pos ← offset + warp_idx
5:          valid ← process_CQE(cq_idx, pos, phase_bit)
6:          mask[warp_idx] ← valid
7:      end if
8:      if mask == 0xFFFFFFFF then
9:          mask ← 0
10:         update_CQ(cq_idx, offset)
11:     end if
12:     update_mask(cq_idx, mask)
13: end function
```

all threads in the warp detect valid completions, indicated by the mask being fully set, the polling service considers the window fully processed. If the window is fully processed, the warp will update the CQ doorbell register to notify the SSD and reset the mask for the next round (lines 9-10). The mask will be updated each time to save the current status of the target CQ (line 12). This warp-coordinated approach increases the parallelism in processing each CQ while minimizing the divergence across threads in a warp because all threads operate on physically contiguous CQEs and follow the same polling logic.

3.3 AGILE Request Issuing Mechanism

As discussed in Section 2.3.3, the SQs require an efficient serialization mechanism before updating the SQ doorbell registers to avoid performance degradation. In this subsection, we first present how user threads issue NVMe commands. Then, we illustrate how AGILE coalesces redundant requests at the warp level.

3.3.1 Serialization process in NVMe SQs. In AGILE, each SQE is associated with a lock that can have three possible states: `EMPTY`, `UPDATED`, and `ISSUED`. Algorithm 2 illustrates the serialization process for issuing NVMe commands. When a user thread needs to issue an NVMe command, it first selects an SQ associated with the target SSD based on its thread index and attempts to submit the command to this SQ if it has an available SQE for a new command (line 2). If the SQ is full, the thread will try to submit commands to another SQ by simply increasing the index of the target SQ. After enqueuing the commands to an SQ (line 6), AGILE sets the state of the corresponding SQE's lock to `UPDATED`, which indicates the command is now visible in the global memory and can be safely notified to the SSD. To ensure the SSD is properly notified, all threads will attempt to update the associated SQ doorbell register and verify whether their commands have been issued (line 9). A thread that successfully acquires the lock for the SQ doorbell register increases the SQ tail (line 15), during which it scans the SQEs in order and updates the SQEs' states from `UPDATED` to `ISSUED`. This process continues until it encounters an SQE in the `EMPTY` state, which either marks the end of the current batch of commands or indicates that the corresponding SQE is not visible in the global memory yet. Then, this thread will update the SQ doorbell register and release the lock (line 15). Finally, all threads verify the states of their respective SQEs (line 17) to confirm if the commands have been

successfully issued to the SSD. Once the completions are received by the AGILE service, the corresponding SQEs' states are reset to EMPTY, allowing them to be reused for future commands.

Algorithm 2 Serialization process in SQs

```
1:  function ATTEMPT_ENQUEUE(sq_idx, cmd)
2:      sqe = check_full(sq_idx)
3:      if sqe == −1 then
4:          return false
5:      end if
6:      enqueue_cmd(sq_idx, sqe, cmd)
7:      update_SQE(sq_idx, sqe, ENQUEUE)
8:      repeat
9:          status ← Attempt_SQDB(sq_idx, sqe)
10:     until status == SUCCESS
11:     return true
12: end function
13: function ATTEMPT_SQDB(sq_idx, sqe)
14:     if acquire_lock(sq_idx) then
15:         move_SQ_tail(sq_idx, sqe)
16:     end if
17:     return check_SQE(sq_idx, sqe)
18: end function
```

3.3.2 Coalescing identical requests at the warp level. To avoid redundant requests, AGILE coalesces identical data requests issued by different threads, which is essential because user threads may independently request the same data chunk from SSDs.

For `prefetch()` and the array-like interface, AGILE employs a two-level coalescing strategy. The first level occurs at the warp level, where CUDA warp-level primitives [45] are used to examine duplicate requests. Then, AGILE selects one thread to forward the request to the second-level coalescing stage. The second level is handled by the AGILE software cache (Section 3.4), which filters remaining redundant requests that are not eliminated in the first warp-level coalescing stage. AGILE prioritizes the warp-level coalescing since accessing the shared software cache requires atomic operations to maintain consistency, which creates critical sections and serializes execution. This serialization can cause stalls and different execution paths for threads in a warp, which introduces warp divergence and degrades overall GPU performance.

For `async_issue(src,dst)`, which mimics `cp.async` [36] or `cuda::memcpy_async` [39] in CUDA and no warp-level coalescing is performed. Even if threads in a warp request the same data, each thread will still obtain its own copy of the requested data. Therefore, in AGILE, the redundant requests are only coalesced at the software cache level, and AGILE delegates the warp level optimization to users. Moreover, `async_issue(src,dst)` provides more flexibility compared to the CUDA APIs, which can introduce potential data hazards. These data hazards are addressed through the Share Table mechanism, which will be described in Section 3.4.1.

3.4 AGILE Software Cache

A software-managed cache can significantly reduce SSD I/O traffic by storing frequently accessed SSD data on the device [21, 29, 48]. AGILE also enables this feature and provides built-in cache policies as well as interfaces for users to customize cache policies. In AGILE, all SSD data accesses are routed through the software cache to ensure coherency and to coalesce the redundant SSD requests.

In AGILE, each cache line has four possible states: `INVALID`, `BUSY`, `READY`, and `MODIFIED`. When user threads request any data, AGILE first checks the user-specified cache policy and obtains the target cache line index. There will be 4 possible cases: **(a) cache hit and data is valid.** If the state of the cache line is `READY`, or `MODIFIED`, it means the data is already in GPU HBM, and the threads can directly obtain the requested data. **(b) cache miss and no eviction required.** In this case, the state is `INVALID`, and the thread will issue an NVMe command to load data from SSD to HBM and change the cache line state to `BUSY`. **(c) cache hit, but the data is invalid.** This happens when the cache line state is `BUSY`. This means the data has already been requested by another thread, and this thread will either wait (synchronous APIs) or append its buffer to the corresponding linked list. **(d) cache miss and eviction required.** This occurs when the cache line is reserved, and the state is not `INVALID`. Then, AGILE will trigger a cache line eviction if the cache line state is `READY`, `MODIFIED`, or `BUSY`. AGILE will simply reset the cache line if the state is `READY`, and write `MODIFIED` cache line to the SSDs and change the state to `BUSY`. If the state is `BUSY`, the corresponding cache line cannot be evicted until the processing is finished, and AGILE will let the user-specified GPU software cache policy decide whether to wait or find another cache line.

3.4.1 Extending Coherency to User-specified Buffers. It is worth noting that `async_issue(src,dst)` in AGILE is conceptually similar to `cuda::memcpy_async` [39] or `cp.async` [36] in CUDA, which enables asynchronous data movement to hide memory latency. However, the CUDA's asynchronous APIs are limited to specific memory paths, i.e., transferring data from the global memory or pinned host memory into the shared memory on the SM [30]. In contrast, `async_issue(src,dst)` in AGILE provides greater flexibility in the source and destination addresses, both of which can be SSD data or user-specified GPU buffers. This enhanced flexibility, however, introduces potential data hazards. For example, a thread may issue an `async_issue(src,dst)` to fetch data from SSD directly to the user-specified buffer, while other threads can concurrently access the same data from the AGILE software cache. If the user-specified buffer or software cache is modified without coordination, data hazards such as read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) can occur, where threads may observe stale or partially updated data.

To address these data hazards, AGILE provides a compile-time option to enable the user-specified buffers to be integrated into the AGILE software cache and safely shared among multiple user threads. If enabled, by default, AGILE will maintain a hashtable-based Share Table to track user-specified buffers' ownership and apply a software-managed coherency protocol inspired by the MOESI model [56] to ensure consistency across different access paths.

Unlike the original MOESI model, where each thread maintains its own copy of data, AGILE maintains the coherency by sharing the pointers to the user-specified buffers, which allows all threads to access the same physical memory region. This eliminates redundant data duplication and avoids extra copies between threads. In AGILE, the MOESI is reinterpreted to reflect the relationship and

responsibility between user threads and their shared buffers. Specifically, when a thread requests data for its buffer, the thread receives exclusive ownership of that buffer. Meanwhile, the Share Table records the source of the data in the buffer and stores the pointer to this buffer. When other threads request the same source of data, the Share Table will return the existing pointer to that buffer and increment a corresponding reference counter of the shared buffer to indicate the usage. If any threads attempt to modify the buffer, the buffer will switch to the Modified State, and the original owner of the buffer will be responsible for propagating the updates to L2 cache – software cache in GPU HBM – after other threads finish using the buffer.

When this Share Table is enabled, it will have the highest priority in the AGILE software cache hierarchy. When new requests arrive, AGILE will first consult the Share Table to determine if any user thread owns a valid buffer of the requested data. If no record is found, AGILE will fall back to the software cache or issue a new request to the SSD and register this buffer in the Share Table. Similar to the flexible customization in software cache, AGILE allows users to design their own sharing policy and integrate it into AGILE seamlessly to meet various application needs.

3.5 AGILE Abstraction and Software APIs

Listing 1 shows an example GPU program that uses AGILE. Users can define their software cache policy (line 1) or directly choose the built-in software cache policies and specify the software cache policy in line 2. To provide flexibility in software cache and share table policies, AGILE employs the curiously recurring template pattern (CRTP) to implement the software cache and share table control logic. CRTP enables compile-time polymorphism and avoids using virtual functions. The software cache and the Share Table policies are specified in line 2.

Because AGILE allows users to provide customized policies, where processing on locks is necessary and may introduce new deadlock risks, AGILE provides a debug option at compile time to track acquired locks within each thread using a lock chain implemented as a linked list (line 6). If this debug option is enabled, when a thread tries to acquire a target lock but fails, it will scan all previously acquired locks and mark these acquired locks are dependent on the target lock to release. Then, it will check if any acquired lock exists in the dependency chain of the target lock – if a circular dependence results in a deadlock. If a circular dependency happens, AGILE will report it to users.

Lines 8 - 19 present the three methods to access SSDs in AGILE. Line 9 is an example of the `prefetch()`, which asynchronously loads the data from a target SSD to the software cache. Line 12 shows how users can register a user-specified buffer to AGILE and use `async_issue(src,dst)` to load or store data asynchronously (lines 13 - 15). For `asyncRead()`, users need to verify if the transfer is completed before using (line 14), while the `asyncWrite()` will ensure the data is updated to the software cache and the write command is issued, and the buffer is available right away for other purposes. AGILE also provides an array-like synchronous API that views the SSDs as a two-dimensional array, where the first dimension specifies the SSD indices and the second dimension is the data position to access (lines 18 - 19).

```
1  class GPUCache:public GPUCacheBase<GPUCache>{...};
2  #define AGILE_CTRL AgileCtrl<GPUCache, ShareTable>
3  __global__
4  void kernel(AGILE_CTRL * ctrl, void * data){
5    ...
6    AgileLockChain chain;
7
8    // Method-1: AGILE prefetch
9    ctrl->prefetch(dev_idx, blk_idx, chain);
10
11   // Method-2: AGILE async_issue
12   AgileBufPtr buf(data + tid * ctrl->line_size);
13   ctrl->asyncRead(dev_idx, blk_idx, buf, chain);
14   buf.wait();
15   ctrl->asyncWrite(dev_idx, blk_idx, buf, chain);
16
17   // Method-3: AGILE array-like synchronous API
18   auto agileArr = ctrl->getArrayWrap<int>(chain);
19   int val = agileArr[dev_idx][idx];
20 }
21
22 int main(int argc, char** argv){
23     // GPU Configurations
24     AGILE_HOST host(...);
25     // Policy Configurations
26     SHARE_TABLE_IMPL s_table(...);
27     GPU_CACHE_IMPL g_cache(...);
28     host.setGPUCache(g_cache);
29     host.setShareTable(s_table);
30     // Add and open target SSDs in the program
31     host.addNvmeDev("/dev/AGILE-xxx", ...);
32     host.addNvmeDev("/dev/AGILE-xxx", ...);
33     host.initNvme();
34     // Initialize AGILE controller
35     host.initializeAgile(...);
36     // CUDA kernel parallelism configurations
37     host.configKernelParallelism(...);
38     host.queryOccupancy(kernel);
39     // Start the lightweight AGILE service
40     host.startAgile();
41     // Execute the CUDA kernel
42     host.runKernel(kernel, args...);
43     // Stop AGILE service
44     host.stopAgile();
45     // Close the opened SSDs
46     host.closeNvme();
47 }
```

Listing 1: Example GPU program using AGILE.

Lines 22 - 47 demonstrate the AGILE host-side code executed on the CPU. At Line 24, users specify the GPU configurations, e.g., selecting which GPU to use for the program. Lines 26 - 29 handle the initialization of AGILE's GPU software cache and share table policies. AGILE allows multiple NVMe SSDs to be configured and used simultaneously in the program, as shown in Lines 31 - 33. To utilize SDDs with AGILE, the devices must be bound to the AGILE-provided NVMe SSD driver, which creates a device file, `/dev/AGILE-NVMe-${PCIe-BDF}`, for each SSD. AGILE supports customized NVMe queue configurations for users to enable prioritization control across SSDs. At Line 35, AGILE allocates physically contiguous memory on HBM for NVMe I/O queues and registers

these queues to the SSDs. Lines 37 - 38 configure the application kernel's launch configurations (i.e, `gridDim`, `blockDim`), compile the application kernel, and report the maximum number of active blocks per SM. The AGILE lightweight runtime service, described in Section 3.2, must be started (Line 40) and properly terminated (Line 44) before and after kernel execution (Line 42). Finally, the opened SSDs need to be closed at Line 46.

4 Evaluation

In the experiments, we first use a micro-benchmark to demonstrate the advantages of the asynchronous model over a synchronous model under different workload characteristics. Then, we evaluate the scalability of AGILE using 4KB random read and write on various numbers of SSDs. To demonstrate the usability of AGILE, we compare AGILE with the state-of-the-art work BaM on Deep Learning Recommendation Models (DLRMs) and use various configurations. We further evaluate the API overhead of AGILE against BaM on graph applications to demonstrate AGILE's efficiency. Lastly, we report the pre-thread register usage of AGILE and BaM, which shows that AGILE is more lightweight in terms of GPU resource consumption.

4.1 Experimental Setup

We evaluate AGILE on a Dell R750 server running Ubuntu 20.04, equipped with an Nvidia RTX 5000 Ada GPU [43], a Dell Ent NVMe AGN MU AIC 1.6TB SSD [15], and two Samsung 990 PRO 1TB SSDs [54]. The GPU and SSDs are attached to the server via PCIe Gen4x16 and Gen4x4, respectively. The Nvidia Driver 550.54 and the CUDA 12.8 are installed on the server for experiments. The modified Linux kernel drivers used in AGILE are tested on Linux 5.4.0-200-generic.

4.2 Comparison between asynchronous I/O and synchronous I/O

We first demonstrate how AGILE's asynchronous I/O model enables overlapping between computation and communication to reduce end-to-end execution time. In this experiment, 1024 threads within a block are launched to issue 64 NVMe commands and perform computation on the returned data. In the synchronous mode, computation begins only after all data has been fetched. In contrast, the AGILE asynchronous mode enables computation and communication overlapping at the thread level. Ideally, when computation and communication perfectly overlap with each other, the speedup can be defined by Equation 1:

$$\text{Ideal Speedup} = \begin{cases} 1 + \text{CTC}, & 0 \le \text{CTC} \le 1 \\ 1 + \dfrac{1}{\text{CTC}}, & \text{CTC} > 1 \end{cases} \quad (1)$$

As shown in Figure 4, we illustrate the effectiveness of AGILE's thread-level asynchronous model by varying the computation-to-communication (CTC) ratio from 0 to 2 by increasing the number of computation iterations. AGILE asynchronous version can achieve up to 1.88x improvement over the synchronous baseline. The observed speedup increases with CTC until it reaches a peak where CTC is close to 0.9 and then gradually decreases when CTC further increases, which aligns with the theoretical trend. The peak speedup occurs below CTC equals 1 because certain portions of the asynchronous pipeline stages, such as prefetching and the issuing logic, cannot be fully hidden by either computation or communication, which limits the ideal overlap. The experimental results demonstrate that AGILE's asynchronous I/O model is effective in hiding communication time, especially when the computation and communication are balanced.

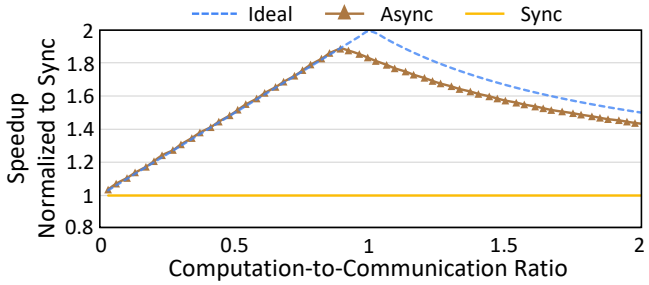

Figure 4: Speedup comparison of asynchronous I/O over synchronous I/O on workloads with different Computation-to-Communication Ratio (CTC).

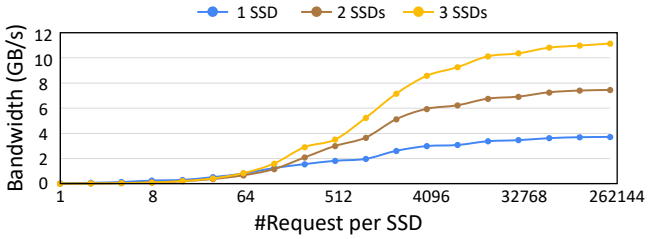

Figure 5: AGILE 4KB random read on multiple SSDs

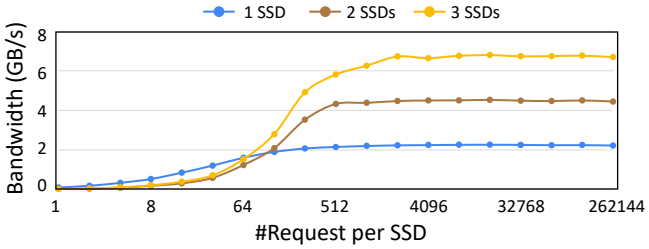

Figure 6: AGILE 4KB random write on multiple SSDs

4.3 AGILE 4KB random read and write on multiple SSDs

We evaluate the scalability of AGILE using 4 KB random read and write using 1, 2, and 3 SSDs, as shown in Figure 5 and Figure 6, respectively. For experiments with more than one SSD, different SSDs are accessed in an interleaved manner. For example, requests 0, 2, 4, etc. are issued to SSD1, while requests 1, 3, 5, etc. are directed to SSD2. In both 4 KB random read and write, AGILE exhibits scalable performance as the number of requests increases and can leverage multiple SSDs effectively. For 4KB random reads in Figure 5, the aggregate bandwidth saturates at 3.7 GB/s, 7.4 GB/s, and 11.1 GB/s with 1 SSD, 2 SSDs, and 3 SSDs, respectively, after approximately 32K concurrent requests per device. Figure 6 depicts the aggregate write bandwidth achieved by AGILE in the 4 KB random write workload, and AGILE saturates at 2.2 GB/s, 4.4 GB/s, and 6.7 GB/s with 1 SSD, 2 SSDs, and 3 SSDs, respectively.

4.4 Evaluation on DLRM inference

We further evaluate AGILE against BaM [48] on Deep Learning Recommendation Model (DLRM) inference. We use the Criteo 1TB Click Logs dataset [12] and construct the categorical feature vocabulary using the first three days of data. To ensure consistent and efficient computation across all experiments, we use cuBLAS [37] for matrix multiplications. BaM and AGILE are used to fetch embedding vectors to HBM, and their kernels are integrated into the CUDA stream pipeline with cuBLAS kernels. We keep the same clock replacement cache policy [10] and set the software cache size to 2GB for all experiments unless otherwise specified. For NVMe I/O queue configurations, we use 128 queue pairs, and the queue depth of each queue is set to 256 by default across all experiments unless otherwise specified. We use AGILE in both the synchronous mode (AGILE sync) and the asynchronous mode (AGILE async). For AGILE sync and BaM implementation, we request data and perform computation on the requested data within the same epoch. For AGILE async, we prefetch data for the next epoch to enable overlapping of communication and computation.

We adopt DLRM architecture from [34] and evaluate several variants. In addition to projection layers (for dimensional alignment in matrix multiplication) and activation layers, the bottom MLP in Config-1 has three matrix multiplication kernels with dimensions 512-512-512, and the top MLP consists of three layers with sizes of 1024-1024-1024. Config-2 reduces the number of matrix multiplications to one in both the bottom MLP and the top MLP to represent a less computationally intensive model. In Config-3, we repeat the matrix multiplications six times to emulate a more computationally intensive workload. In all configurations, we measure the end-to-end execution time using a batch size of 2,048 and an epoch size of 10,000.

Figure 7 illustrates the speedup comparison of AGILE in both synchronous and asynchronous modes relative to BaM across three DLRM configurations. AGILE sync shows consistent improvement over BaM, achieving speedups of 1.3×, 1.39×, and 1.27× in Config-1, Config-2, and Config-3, respectively. The AGILE async further improves the performance by overlapping data movement with computation and reaches 1.48×, 1.63×, and 1.32× speedups in the same configurations.

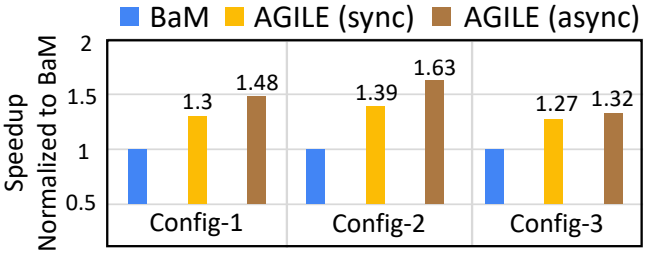

Figure 7: Speedup comparison of AGILE (async and sync modes) over BaM on different recommendation models.

To understand how AGILE performs under different workload granularities, we evaluate AGILE's speedup across a wide range of batch sizes using DLRM Config-1, which assesses the scalability of AGILE and BaM. Figure 8 depicts the speedup of AGILE in sync and async modes normalized to the BaM baseline across batch sizes ranging from 1 to 2048. AGILE sync mode shows stable gains over BaM with speedup from 1.18× to 1.30×. AGILE async also consistently outperforms AGILE sync across all batch sizes and reaches the peak speedup to 1.75× at a batch size of 16. These results demonstrate AGILE's ability to overlap computation and computation at scale. The results also indicate that the AGILE async benefits more when the batch size is smaller and near 16 in this DLRM inference, where the opportunity to hide communication is more significant.

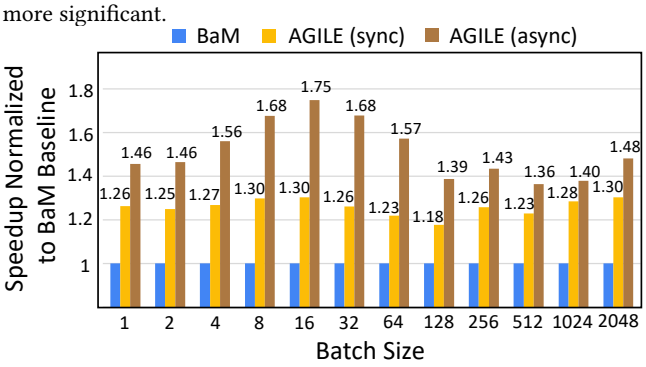

Figure 8: Speedup comparison of AGILE (async and sync modes) and BaM across varying batch sizes in DLRM inference.

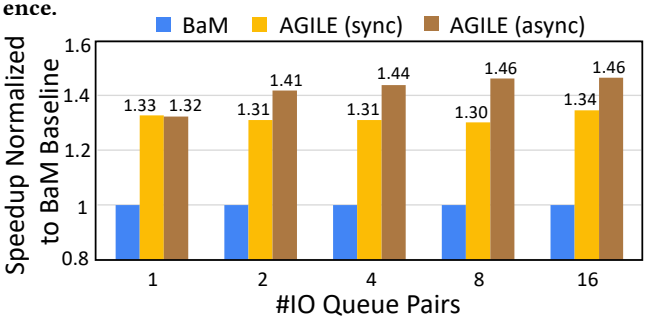

Figure 9: Speedup comparison of AGILE (async and sync modes) and BaM under varying numbers of I/O queue pairs in DLRM inference.

We further study the sensitivity of NVMe queue settings for both AGILE and BaM using DLRM Config-1 and a batch size of 2048. Specifically, we reduce the queue depth to 64 and sweep the number of queue pairs from 1 to 16, which introduces greater contention in the NVMe queues. Figure 9 demonstrates that both AGILE sync and async modes consistently outperform the BaM baseline across all configurations. When only one queue pair is used, the AGILE async mode provides only marginal speedup over the AGILE sync mode. This phenomenon arises because the number of available SQEs is too small to support all the requests issued in an epoch. As a result, in the prefetch stage, the threads must wait until the AGILE service receives completions from the SSD and recycles SQEs. Consequently, this waiting degrades the asynchronous mode, causing it to exhibit a similar behavior to the synchronous mode in AGILE. As the number of queue pairs increases, more SQEs are available for each epoch. This reduces contention during the prefetch stage and allows the prefetch stage to proceed without stalls. Therefore, the speedup of AGILE async over the synchronous mode becomes more significant.

Lastly, we evaluate the impact of software cache size on the DLRM inference using DLRM Config-1 and a batch size of 2048.

We sweep the software cache size from 1 MB to 2 GB and compare the speedup of AGILE against the BaM baseline. Figure 10 illustrates the changes in the speedup under different software cache sizes. The AGILE sync mode consistently outperforms BaM across all cache sizes, achieving a peak speedup of 1.48× at 256 MB software cache size. In contrast, AGILE async mode initially lags behind both the BaM baseline and the AGILE sync mode when the software cache size is small. However, the AGILE async mode surpasses the synchronous mode after the software cache reaches a certain threshold, around 64 MB. This behavior stems from the software cache contention. When the software cache is too small, each epoch may access more data that cannot fit in the software cache size. In this case, the prefetch stage in AGILE async will not only wait for available cache lines to make new requests but also evict the previously requested data intended for the next epoch. Therefore, when that data is needed in the next epoch, it has already been evicted, and additional requests become necessary during the computation phase. The delays in the prefetch stage degrade the asynchronous mode to behave more like the synchronous version, and the extra requests during the computation phase make the performance worse. As the software cache size keeps increasing, more cache lines are available to support concurrent prefetching without evictions. This allows the prefetch stage to complete soon after the commands are issued. Therefore, the data movement time can be hidden by the computation again, exhibiting consistent speedup over the synchronous mode again. These results indicate that the asynchronous mode does not always outperform the synchronous one because an improper software cache size will cause stalls and introduce extra NVMe commands. Therefore, when applying asynchronous mode in real-world applications, it is essential to estimate both the capacity of the software cache size and the data access demands per epoch to fully leverage the benefits of asynchronous mode.

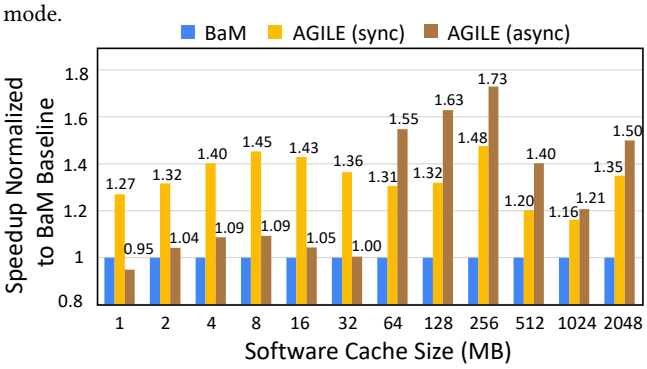

Figure 10: Speedup comparison of AGILE (async and sync modes) and BaM under varying software cache sizes in DLRM inference.

4.5 Evaluate AGILE API overhead on graph applications

The overhead resulting from the implementation is also an important factor that influences overall performance. We evaluate the AGILE's API overhead covering both the software cache access and request issuing against BaM on two graph applications: Breadth-First Search (BFS) and sparse matrix vector multiplication (SpMV). The execution time for both BFS and SpMV is dominated by data movement due to their irregular access patterns and low arithmetic intensity [6, 13], making them appropriate benchmarks for assessing API-level overhead. In our experiments, we implement the baseline versions of BFS and SpMV using BaM and AGILE without any application-level optimization, which ensures that the observed performance differences are attributed solely to the underlying software infrastructure, including the API overhead, cache access behavior, and request issuing & completion polling mechanism. We use GAP Benchmark Suite [55] to generate the uniform random graphs and Kronecker graphs to emulate realistic graphs. All graph structures and weights (if applicable) are stored in the compressed sparse row (CSR) format.

To measure the API overhead, we conduct the following three-step experiment:

(1) We first measure the execution times of the application kernels without using BaM or AGILE, and the graph data is directly stored inside HBM and accessed using the native CUDA API. (Kernel time)
(2) Then, we integrate BaM and AGILE into the application kernels and measure the total runtime, which includes the data transfer time and the overhead from both software cache access and NVMe command issuing. (I/O API time)
(3) Finally, to obtain the overhead in software cache access, we preload all graph data into the software cache before kernel execution, eliminating the NVMe requests during runtime. (Cache API time)

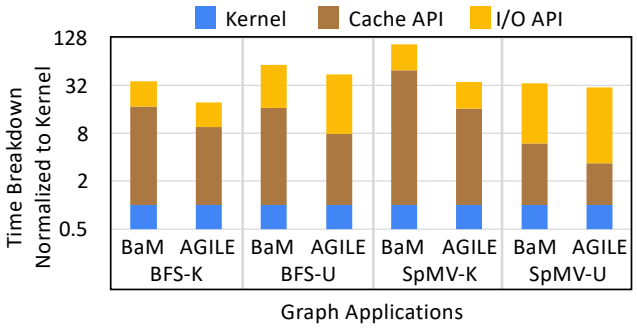

Figure 11: Execution time breakdown of BaM and AGILE across various graph applications.

Figure 11 illustrates the execution time breakdown of BFS and SpMV using different graph types, where '-K' denotes the Kronecker graphs (K-graph) with skewed degree distribution, and '-U' denotes uniform random graphs (U-graph) with regular structures. The bars are segmented into kernel execution, cache API, and I/O API time. All measured execution times are normalized to the kernel runtime. Across all graph types, AGILE consistently achieves lower execution time compared with BaM by effectively reducing both the cache API and I/O API overheads. For BFS, AGILE reduces the software cache overhead by 2.27× on U-graph and 1.93× on K-graph. and cuts the I/O API overhead by 1.16× and 1.86×, respectively. For SpMV, AGILE achieves even greater reductions – 2.11× and 3.17× in software cache overhead, and 1.06× and 2.85× in I/O overhead on U-graph and K-graph, respectively. These results underscore AGILE's efficiency in handling memory-intensive workloads by minimizing the overhead from the API implementation regardless of graph structure.

4.6 Evaluate AGILE per thread register usage across CUDA kernels

To further evaluate AGILE's efficiency on GPU resources, we examine its per-thread register usage across different CUDA kernels. Since register usage directly affects warp occupancy and scheduling flexibility, optimizing it is crucial on GPUs. Figure 12 depicts the number of registers used per thread in different CUDA kernels implemented using BaM or AGILE. We do not impose any constraints to limit the register usage, and both BaM and AGILE use identical kernel implementations for fair comparison.

Compared to BaM, AGILE achieves a reduction in per-thread register by 1.04×, 1.22×, and 1.32× in Vector Mean, BFS, and SpMV kernels, respectively. This improvement stems from the efficient implementation of AGILE and the offloading of the CQ polling logic to the dedicated AGILE service kernel, which alleviates pressure on application kernels and enables more efficient register utilization. Moreover, the AGILE service kernel is lightweight, which consumes 37 registers per thread and can assist multiple CUDA kernels simultaneously.

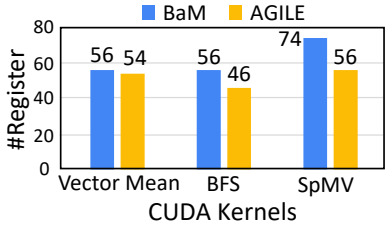

Figure 12: Per-thread register usage comparison between BaM and AGILE across various CUDA kernels.

5 Discussion

While AGILE demonstrates significant performance improvement over existing work and exhibits strong scalability with multiple SSDs, several opportunities remain for extending AGILE to broader and more complex system architectures.

First, extending the software cache hierarchy to incorporate CPU DRAM as an additional tier is a natural and well-motivated enhancement, as demonstrated in prior work [7, 21, 29]. AGILE is designed with the flexibility to support such an extension. In its current implementation, AGILE includes reserved APIs that enable integration of CPU DRAM as an additional level of the software-managed cache, complementing the existing GPU HBM cache. We will optimize and incorporate this functionality in our open-sourced GitHub repository soon.

Second, AGILE currently targets a single-GPU with multiple SSDs scenario, but AGILE has all the capabilities to be extended to support multiple GPUs with multiple SSDs. To simply share one SSD among GPUs, different I/O queue pairs of the target SSD can work independently and be assigned to different GPUs. It only requires some modifications to the Host APIs, while the AGILE service and interfaces on the CUDA kernel do not need any change. Allowing one GPU to issue peer-to-peer data transfers between another GPU and SSDs or populating data from one GPU directly to another GPU is also doable if the GPU knows the PCIe BARs of the other GPUs. However, it may require further investigation and optimization to handle data transfer and synchronization efficiently without performance degradation. In a multi-GPU system, enabling one GPU to view other GPUs' HBM as a remote cache and leverage NVLink to transfer cached data may also be worth investigating. This additional cache level in HBMs (shared among GPUs) needs further study on the cache coherency among GPUs, which involves dealing with the cache line metadata and analyzing its performance benefits.

Third, extending AGILE to support more heterogeneous systems with accelerators such as FPGAs could provide more performance gains on diverse workloads with various computation and IO characteristics. For example, by leveraging the FPGA's flexibility and advantages in network processing, FpgaNIC [61] develops a GPU-oriented SmartNIC on FPGA to accelerate a broad range of distributed applications on distributed GPUs. Besides, FPGA exhibits good energy efficiency as hardware accelerators [64, 67, 69, 70], and is a good fit for real-time systems, where determinism is critical [16, 23, 24]. Collaboration between FPGAs and GPUs may offer both high throughput and lower energy consumption while meeting stringent deadline requirements. Such an extension, however, introduces new challenges in coordinating multiple devices and requires more sophisticated system designs. We leave this extension to future versions of AGILE.

Fourth, AGILE may also enable new research in compiler-level optimizations. For applications involving multiple data communications and computations within a single kernel, the programmers need to explore the overlap opportunity manually. While currently AGILE functions as an asynchronous I/O library, it can be extended with compiler support to automatically analyze dependencies and perform code transformations. Existing research works have explored similar optimizations. For example, the compiler identifies the data dependency and reorders instructions for better overlapping [11, 68]. AGILE serves as a foundational first step toward that goal of developing a compiler that enables static dependency analysis to automatically explore overlapping opportunities.

Fifth, supporting AGILE in virtualized environments, such as virtual machines or Docker containers, is important for improving portability, scalability, and ease of deployment in shared computing infrastructures. However, this requires further development and investigation into the associated performance implications, particularly with respect to I/O virtualization, device passthrough, and potential overhead introduced by the virtualization layer.

6 Conclusion

In this paper, we propose AGILE, a lightweight and efficient asynchronous library for GPU-SSD integration. AGILE is the first work that enables GPU threads to issue NVMe commands asynchronously and allows users to customize software cache policy. AGILE enables overlapping at the thread level and achieves up to 1.88× reduction in execution time by hiding data transfer with computation. AGILE exhibits up to 1.75× improvement on DLRMs and shows 3.12× and 2.85× API overhead reduction in software cache and NVMe IO requests compared with the state-of-the-art GPU-centric work, BaM [48]. AGILE is also lightweight and consumes up to 1.32× fewer registers in various CUDA kernels.

ACKNOWLEDGEMENTS – This work is supported in part by Brown University New Faculty Start-up Grant, and NSF awards #2213701, #2217003, #2328972, #2511445, #2536952.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Tyler Allen and Rong Ge. 2021. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[3] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W Lee. 2021. {FlashNeuron}:{SSD-Enabled}{Large-Batch} training of very deep neural networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 387–401.

[4] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. 2019. SPIN: Seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. *ACM Transactions on Computer Systems (TOCS)* 36, 2 (2019), 1–26.

[5] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954* (2024).

[6] Aydin Buluç and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[7] Chia-Hao Chang, Jihoon Han, Anand Sivasubramaniam, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-Mei Hwu. 2024. Gmt: Gpu orchestrated memory tiering for the big data era. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 464–478.

[8] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. 2024. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 178–191.

[9] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815.

[10] Fernando J Corbato. 1968. *A paging experiment with the multics system*. Massachusetts Institute of Technology.

[11] Neal C Crago, Sana Damani, Karthikeyan Sankaralingam, and Stephen W Keckler. 2024. Wasp: Exploiting gpu pipeline parallelism with hardware-accelerated automatic warp specialization. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1–16.

[12] Criteo AI Lab. 2025. Download Criteo 1TB Click Logs dataset - Criteo AI Lab. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/

[13] John D Davis and Eric S Chung. 2012. SpMV: A memory-bound application on the GPU stuck between a rock and a hard place. *Microsoft Research Silicon Valley, Technical Report14 September* 2012 (2012).

[14] Debendra Das Sharma and Ishwar Agarwal. 2023. CXL_3.0_white-paper_FINAL. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL_3.0_white-paper_FINAL.pdf

[15] Dell. 2021. Dell Enterprise Agnostic NVMe Drive Technical Specifications. https://dl.dell.com/manuals/all-products/esuprt_data_center_infra_int/esuprt_data_center_infra_storage_adapters/dell-poweredge-exp-fsh-nvme-pcie-ssd_Users-Guide7_en-us.pdf

[16] Peiyan Dong, Jinming Zhuang, Zhuoping Yang, Shixin Ji, Yanyu Li, Dongkuan Xu, Heng Huang, Jingtong Hu, Alex K Jones, Yiyu Shi, et al. 2024. EQ-ViT: Algorithm-hardware co-design for end-to-end acceleration of real-time vision transformer inference on Versal ACAP architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (2024), 3949–3960.

[17] Gil Einziger, Roy Friedman, and Ben Manes. 2017. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)* 13, 4 (2017), 1–31.

[18] Eran Gal and Sivan Toledo. 2005. Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR)* 37, 2 (2005), 138–163.

[19] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. 2024. AI and memory wall. *IEEE Micro* (2024).

[20] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben Van Werkhoven, and Henri E Bal. 2023. Optimization techniques for GPU programming. *Comput. Surveys* 55, 11 (2023), 1–81.

[21] Jeongmin Hong, Sungjun Cho, Geonwoo Park, Wonhyuk Yang, Young-Ho Gong, and Gwangsun Kim. 2024. Bandwidth-effective dram cache for gpu s with storage-class memory. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 139–155.

[22] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. 2023. Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus. *Proceedings of Machine Learning and Systems* 5 (2023), 680–694.

[23] Shixin Ji, Xingzhen Chen, Jinming Zhuang, Wei Zhang, Zhuoping Yang, Sarah Schultz, Yukai Song, Jingtong Hu, Alex Jones, Zheng Dong, and Peipei Zhou. 2025. ART: Customizing Accelerators for DNN-Enabled Real-Time Safety-Critical Systems. In *Proceedings of the 2025 ACM Great Lakes Symposium on VLSI (GLSVLSI '25)*.

[24] Shixin Ji, Zhuoping Yang, Xingzhen Chen, Wei Zhang, Jinming Zhuang, Alex K Jones, Zheng Dong, and Peipei Zhou. 2025. CLARE: Deterministic Cycle-Level Accelerator on REconfigurable platforms in DNN-Enabled Real-Time Safety-Critical Systems. In *The 46th IEEE Real-Time Systems Symposium, 2025 (RTSS 2025) (RTSS '25)*.

[25] Diya Joseph, Juan Luis Aragón, Joan-Manuel Parcerisa, and Antonio Gonzalez. 2024. Wasp: Warp scheduling to mimic prefetching in graphics workloads. *arXiv preprint arXiv:2404.06156* (2024).

[26] Mark J Kilgard and Jeff Bolz. 2012. GPU-accelerated path rendering. *ACM Transactions on Graphics (TOG)* 31, 6 (2012), 1–10.

[27] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. 2019. Asynchronous {I/O} stack: A low-latency kernel {I/O} stack for {Ultra-Low} latency {SSDs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 603–616.

[28] Ruihao Li, Sanjana Yadav, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J Yadwadkar, and Lizy K John. 2023. Performance Implications of Async Memcpy and UVM: A Tale of Two Data Transfer Modes. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 115–127.

[29] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. 2017. Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures. In *Proceedings of the International Conference on Supercomputing*. 1–10.

[30] Matthieu Tardy and Carter Edwards. 2020. Controlling Data Movement to Boost Performance on the NVIDIA Ampere Architecture. https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/

[31] Avinash Maurya, Jie Ye, M Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Breaking the memory wall: A study of i/o patterns and gpu memory utilization for hybrid cpu-gpu offloaded optimizers. In *Proceedings of the 14th Workshop on AI and Scientific Computing at Scale using Flexible Computing Infrastructures*. 9–16.

[32] Microsoft. 2025. DeepNVMe. https://www.deepspeed.ai/tutorials/deepnvme/

[33] Sparsh Mittal and Shraiysh Vaishay. 2019. A survey of techniques for optimizing deep learning on GPUs. *Journal of Systems Architecture* 99 (2019), 101635.

[34] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[35] Giovanni Neglia, Damiano Carra, and Pietro Michiardi. 2018. Cache policies for linear utility maximization. *IEEE/ACM Transactions on Networking* 26, 1 (2018), 302–313.

[36] Nvidia. 2025. 1. Introduction — PTX ISA 8.8 documentation. https://docs.nvidia.com/cuda/parallel-thread-execution/

[37] Nvidia. 2025. cuBLAS | NVIDIA Developer. https://developer.nvidia.com/cublas

[38] Nvidia. 2025. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[39] Nvidia. 2025. cuda::memcpy_async — libcudacxx 3.1 documentation. https://nvidia.github.io/cccl/libcudacxx/extended_api/asynchronous_operations/memcpy_async.html?utm_source=ainews&utm_medium=email&utm_campaign=ainews-a-quiet-weekend

[40] Nvidia. 2019. GPUDirect Storage: A Direct Path Between Storage and GPU Memory | NVIDIA Technical Blog. https://developer.nvidia.com/blog/gpudirect-storage/

[41] Nvidia. 2025. NVIDIA Ampere GPU Architecture Tuning Guide. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html

[42] Nvidia. 2025. NVIDIA/gdrcopy: A fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology. https://github.com/NVIDIA/gdrcopy

[43] Nvidia. 2025. RTX 5000 Ada Generation Graphics Card | NVIDIA. https://www.nvidia.com/en-us/design-visualization/rtx-5000/

[44] Nvidia. 2013. Unified Memory in CUDA 6 | NVIDIA Technical Blog. https://developer.nvidia.com/blog/unified-memory-in-cuda-6/

[45] Nvidia. 2018. Using CUDA Warp-Level Primitives | NVIDIA Technical Blog. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

[46] NVM Express. 2025. NVM Express. https://nvmexpress.org/

[47] Stéfani Pires, Adriana Ribeiro, and Leobino N Sampaio. 2024. On learning suitable caching policies for in-network caching. *IEEE Transactions on Machine Learning in Communications and Networking* (2024).

[48] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, Chris J Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. 2023. GPU-initiated on-demand high-throughput storage access in the BaM system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 325–339.

[49] Mohaimenul Azam Khan Raiaan, Md Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. 2024. A review on large Language Models: Architectures, applications, taxonomies, open issues and challenges. *IEEE Access* (2024).

[50] 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*. 1–14.

[51] Shaina Raza and Chen Ding. 2019. Progress in context-aware recommender systems—An overview. *Computer Science Review* 31 (2019), 84–97.

[52] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[53] Xiaowei Ren and Mieszko Lis. 2021. Chopin: Scalable graphics rendering in multi-gpu systems via parallel image composition. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 709–722.

[54] Samsung. 2025. Samsung 990 PRO PCIe 4.0 SSD | Samsung Semiconductor Global. https://semiconductor.samsung.com/consumer-storage/internal-ssd/990-pro/

[55] Scott Beamer. 2024. sbeamer/gapbs: GAP Benchmark Suite. https://github.com/sbeamer/gapbs

[56] Paul Sweazey and Alan Jay Smith. 1986. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *ACM SIGARCH Computer Architecture News* 14, 2 (1986), 414–423.

[57] Tom's Hardware. 2021. Samsung 980 Pro M.2 NVMe SSD Review: Redefining Gen4 Performance | Tom's Hardware. https://www.tomshardware.com/reviews/samsung-980-pro-m-2-nvme-ssd-review

[58] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. 2021. Grus: Toward unified-memory-efficient high-performance graph processing on gpu. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 2 (2021), 1–25.

[59] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[60] Yang Wang, Jiwu Shu, Guangyan Zhang, Wei Xue, and Weimin Zheng. 2010. Sopa: Selecting the optimal caching policy adaptively. *ACM Transactions on Storage (TOS)* 6, 2 (2010), 1–18.

[61] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. 2022. {FpgaNIC}: An {FPGA-based} versatile 100gb {SmartNIC} for {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 967–986.

[62] Kun Wu, Jeongmin Brian Park, Xiaofan Zhang, Mert Hidayetoğlu, Vikram Sharma Mailthody, Sitao Huang, Steven Sam Lumetta, and Wen-mei Hwu. 2024. SSDTrain: An Activation Offloading Framework to SSDs for Faster Large Language Model Training. *arXiv preprint arXiv:2408.10013* (2024).

[63] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin-Yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S Kim. 2023. Overcoming the memory wall with {CXL-Enabled}{SSDs}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 601–617.

[64] Zhuoping Yang, Jinming Zhuang, Jiaqi Yin, Cunxi Yu, Alex K. Jones, and Peipei Zhou. 2023. AIM: Accelerating Arbitrary-precision Integer Multiplication on Heterogeneous Reconfigurable Computing Platform Versal ACAP. In *ICCAD*.

[65] Haoyang Zhang, Yirui Zhou, Yuqi Xue, Yiqi Liu, and Jian Huang. 2023. G10: Enabling an efficient unified gpu memory and storage architecture with smart tensor migrations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 395–410.

[66] Yu Zhang, Yuxuan Liang, Jin Zhao, Fubing Mao, Lin Gu, Xiaofei Liao, Hai Jin, Haikun Liu, Song Guo, Yangqing Zeng, et al. 2022. Egraph: efficient concurrent GPU-based dynamic graph processing. *IEEE Transactions on Knowledge and Data Engineering* 35, 6 (2022), 5823–5836.

[67] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. 2023. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. In *The 2023 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '23)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3543622.3573210

[68] Jinming Zhuang, Shaojie Xiang, Hongzheng Chen, Niansong Zhang, Zhuoping Yang, Tony Mao, Zhiru Zhang, and Peipei Zhou. 2025. ARIES: An Agile MLIR-Based Compilation Flow for Reconfigurable Devices with AI Engines. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '25)*. Association for Computing Machinery, New York, NY, USA, 92–102. doi:10.1145/3706628.3708870

[69] Jinming Zhuang, Zhuoping Yang, Shixin Ji, Heng Huang, Alex K. Jones, Jingtong Hu, Yiyu Shi, and Peipei Zhou. 2024. SSR: Spatial Sequential Hybrid Architecture for Latency Throughput Tradeoff in Transformer Acceleration. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24)*. 55–66.

[70] Jinming Zhuang, Zhuoping Yang, and Peipei Zhou. 2023. High Performance, Low Power Matrix Multiply Design on ACAP: from Architecture, Design Challenges and DSE Perspectives. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. doi:10.1109/DAC56929.2023.10247981

Appendix: Artifact Description

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

- C_1 We propose AGILE, enabling GPU to issue NVMe commands asynchronously. To the best of our knowledge, AGILE is the first GPU-centric asynchronous IO model.
- C_2 We implement a robust lock-based asynchronous transaction mechanism, which allows GPU threads to issue NVMe commands asynchronously without holding any locks. Our approach efficiently eliminates possible deadlocks and data hazards.
- C_3 We integrate a flexible software cache hierarchy in AGILE to utilize GPU HBM, which allows users to customize their cache policy and provides a simple interface for increased usability.
- C_4 We evaluate AGILE on micro-benchmarking and applications. The results show that AGILE enables overlapping at the thread level and achieves 1.88× speedup over a synchronous IO model. Compared with SOTA work BaM on DRLMs, AGILE achieves up to 1.75× reduction in end-to-end execution time. In graph applications, AGILE demonstrates lower API overhead in managing software cache and NVMe IO requests up to 3.12× and 2.85×, respectively. Furthermore, AGILE consumes fewer registers and exhibits up to 1.32× reduction in the usage of registers.

A.2 Computational Artifacts

A_1 https://zenodo.org/records/17260393

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1 - C_4	Figure 4-12

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

Artifact A_1 contains the AGILE source code, benchmarking code for comparison with the baseline BaM, and scripts to reproduce the results in the paper. The source code supports the paper's main contributions C_1 - C_3, and the benchmarking code and scripts support the main contribution C_4. Contribution C_1 and C_2 are the prerequisites for C_3 and C_4. Contribution C_3 gives users flexibility when using AGILE in various application settings. Contribution C_4 highlights AGILE's superiority over the SOTA work BAM for the GPU-centric on-demand high-throughput storage access system.

Expected Results

In the experiments, we implement the same software cache policy with BaM using the AGILE software cache interface (C_3). We evaluate AGILE in various applications and different settings to support C_4 in the following aspects:

(1) AGILE's asynchronous IO model achieves better performance on synthetic workloads by overlapping computation and communication under workloads with various computation-to-communication (CTC) ratios. (Figure 4)
(2) AGILE exhibits scalable performance in 4KB random read-/write as the number of requests increases and can leverage multiple SSDs effectively. (Figure 5 & Figure 6)
(3) AGILE achieves better performance in Deep Learning Recommendation Models (DLRM) with various settings against the SOTA GPU-centric system architecture, BaM. (Figure 7 - Figure 10)
(4) AGILE's implementation, including the software cache system, NVMe queue handling, and lock mechanism, is more efficient than the SOTA work BaM. (Figure 11)
(5) AGILE requires fewer GPU hardware resources in terms of per-thread register usage on various CUDA kernels. (Figure 12)

As the hardware settings include GPU and SSD types, NUMA configurations, and PCIe configurations may influence the reproduced results, we will provide our server to the evaluators for reproducing our experimental results. The evaluators can contact us and request server access via email: zhuoping_yang@brown.edu.

Expected Reproduction Time (in Minutes)

The estimated time for setting up the environment and compiling code for both AGILE and BaM is within 30 minutes. The estimated time for preparing DLRM input data and graph data would take approximately 360 minutes. Reproducing the experimental results in Figure 4 - 12 can be finished in 120 minutes.

Artifact Setup (incl. Inputs)

Hardware. AGILE requires an Nvidia GPU and several NVMe SSDs. The Nvidia GPU and NVMe SSDs are installed in the same server via PCIe. In the experiments, we use an Nvidia RTX 5000 Ada GPU, a Dell Ent NVMe AGN MU AIC 1.6TB SSD, and two Samsung 990 PRO 1TB SSDs.

Software. AGILE is a CUDA library and requires a modified version of GDRCopy (https://github.com/NVIDIA/gdrcopy), which is included in the artifact.

Datasets / Inputs. The Criteo dataset is used in the DLRM evaluations, and the Criteo dataset can be downloaded from https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/. We use the GAP Benchmark Suite (https://github.com/sbeamer/gapbs) to generate uniform random graphs and Kronecker graphs.

Installation and Deployment. We use NVIDIA (R) Cuda compilation tools (release 12.8, V12.8.93) for all experiments. The Nvidia driver version is 550.54.14. The operating system is Ubuntu 20.04 with a Linux kernel version of 5.4.0-214-generic. For setting up BaM, please refer to https://github.com/ZaidQureshi/bam. AGILE's host code requires a continuous physical memory region, which is reserved in `/etc/default/grub` by adding the `GRUB_CMDLINE_LINUX` option. For example, `GRUB_CMDLINE_LINUX="memmap=1G\\\$128G"` will reserve 1 GB DRAM memory starting at 128 GB. After changing `/etc/default/grub`, executing `sudo update-grub` and `sudo reboot` to apply the modification.

Artifact Execution

Before conducting experiments using AGILE, it is necessary to find the target SSDs and their corresponding PCIe BARs and the BAR sizes. For example, Listing 2 shows an example of obtaining the BAR and BAR size for the NVMe SSD device at `/dev/nvme0n1`.

```
username@host$ readlink -f /sys/block/nvme0n1
/sys/devices/pci0000:4a/0000:4a:02.0/0000:4b:00.0/
    nvme/nvme0/nvme0n1
username@host$ lspci -vvs 4b:
4b:00.0 Non-Volatile memory controller: Samsung
    Electronics Co Ltd Device a80c (prog-if 02 [
    NVM Express])
        Subsystem: Samsung Electronics Co Ltd
            Device a801
        Control: I/O- Mem+ BusMaster+ SpecCycle-
            MemWINV- VGASnoop- ParErr- Stepping-
            SERR- FastB2B- DisINTx+
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr-
            DEVSEL=fast >TAbort- <TAbort- <MAbort-
             >SERR- <PERR- INTx-
        Latency: 0
        Interrupt: pin A routed to IRQ 18
        NUMA node: 0
        Region 0: Memory at 95400000 (64-bit, non-
            prefetchable) [size=16K]
        Capabilities: <access denied>
        Kernel driver in use: nvme
        Kernel modules: nvme
```

Listing 2: Obtain SSDs' BAR and the BAR size

From the output of "readlink -f /sys/block/nvme0n1", the PCIe BAR of the target SSD is `0x95400000` and the BAR size is `16384`, which will be specified in the AGILE host code or the command line arguments. Then, unbind the default `nvme` driver for this NVMe SSD by executing "echo "0000:4b:00.0" | sudo tee /sys/bus/pci/devices/0000:4b:00.0/driver/unbind". Before executing applications using AGILE, the modified `gdrcopy` needs to be installed by executing `insmod.sh` located at `AGILE-SC25-AD/driver/gdrcopy/`. To execute applications using BaM for experimental comparison purposes, `libnvm` should be installed according to https://github.com/ZaidQureshi/bam.

To reproduce the experiments for Figure 4 - 11, we provide several scripts (in `AGILE-SC25-AD/experiments`) that execute AGILE and BaM automatically and save the output files to `./results` for future analysis. Table 1 shows the matching between the experiments and the scripts.

To reproduce the per-thread register usage for various CUDA kernels in Figure 12, please compile the DLRM application, BFS, and SpMV in the "AGILE-SC25-AD/benchmarks" folder for AGILE and "AGILE-SC25-AD/baseline/bam/benchmarks" folder for BaM and read the output report from `nvcc`.

Artifact Analysis (incl. Outputs)

For Figure 4 - 11, we mainly focus on the reported end-to-end execution time. The example outputs of BFS using AGILE and BaM are shown in Listing 3 and Listing 4, respectively. The output will report the number of issued NVMe commands, execution time, and correctness check via MD5 digest. Then, we can plot the normalized speedup using the execution time.

Table 1: Experimental Bash scripts for reproducing results for Figure 4 - 11.

Figures	Corresponding Scripts
Figure 4	run_ctc.sh
Figure 5	rand_read.sh
Figure 6	rand_write.sh
Figure 7 - 10	run_dlrm.sh & auto_dlrm.sh
Figure 11	run_bfs*.sh & run_spmv*.sh

For Figure 12, we have added the compilation flag "-w -Xptxas -v" to the Makefile for both AGILE and BaM. During the compilation process, `nvcc` will report per-thread register usage for various CUDA kernels.

```
......
run: 0 prefetch_hit: 0 prefetch_relaxed_hit: 0
    prefetch_relaxed_miss: 0 prefetch_issue:
    472724 runtime_issue: 0 warp_master_wait: 0
issued_read: 472724 issued_write: 0 attempt_fail:
    367832
......
BFS time: 0.960216 seconds.
035fbc46fa4a11baf426d8408e632aa5  res-bfs.bin
```

Listing 3: An example output of BFS using AGILE

```
......
BFS time: 1.76495 seconds.
#READ IOs: 472724 #Accesses:484068261 #Misses
    :661416  Miss Rate:0.00136637  #Hits:
    483406845  Hit Rate:0.998634 CLSize:4096
********************************
Bandwidth: 1.09707e+09 GB/s
035fbc46fa4a11baf426d8408e632aa5  gpu-res-bfs.bin
```

Listing 4: An example output of BFS using BaM