GPM: Leveraging Persistent Memory from a GPU

Shweta Pandey*
shwetapandey@iisc.ac.in
Indian Institute of Science
Bangalore, India

Aditya K Kamath†*
adityakamath@iisc.ac.in
Indian Institute of Science
Bangalore, India

Arkaprava Basu
arkapravab@iisc.ac.in
Indian Institute of Science
Bangalore, India

ABSTRACT

The GPU is a key computing platform for many application domains. While the new non-volatile memory technology has brought the promise of byte-addressable persistence (a.k.a., persistent memory, or PM) to CPU applications, the same, unfortunately, is beyond the reach of GPU programs.

We take three key steps toward enabling GPU programs to access PM directly. First, enable direct access to PM from within a GPU kernel without needing to modify the hardware. Next, we demonstrate three classes of GPU-accelerated applications that benefit from PM. In the process, we create a workload suite with nine such applications. We then create a GPU library, written in CUDA, to support logging, checkpointing, and primitives for native persistence for programmers to easily leverage PM.

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**.

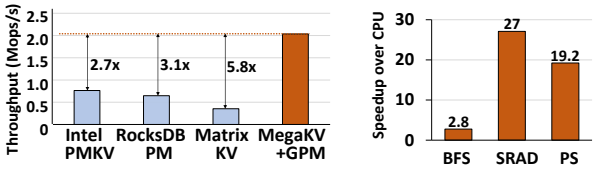
KEYWORDS

Graphics Processing Unit; Persistent Memory; Crash consistency

ACM Reference Format:
Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2022. GPM: Leveraging Persistent Memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507758

1 INTRODUCTION

Non-volatile memory (NVM) technologies promise to blur the long-held distinction between memory and storage by enabling byte-grained durability at latencies comparable to DRAM [77]. We define persistent memory (PM) as NVM accessible via loads and stores at a byte granularity [64]. Thanks to many years of research on the CPU's software and hardware stack for PM (*e.g.*,[18, 19, 29, 31, 42, 54, 93]), and with the recent commercialization of Intel's Optane NVM [37], PM's promise of revolutionizing computing through fast byte-grained persistence and recoverability seems close to reality.



(a) Throughput of pKVS. **(b) Speedup over CPU PM apps.**

Figure 1: Benefits of GPM over CPU with PM.

The graphics processing unit (GPU) is a key computing platform today but is deprived of direct access to PM. We find important applications that can benefit from both the GPU's parallelism and the fine-grained persistence of PM. Consider persistent key-value stores (KVS) that leverage PM's fine-grain persistence. Today, they are limited to CPUs [38, 79, 100]. Independently, researchers have shown that GPUs can significantly improve KVS's throughput [102]. Fine-grained persistence to PM from the GPU could *enable both*. Figure 1a provides a glimpse of its potential performance benefits. The first three bars show throughputs of batched SET operations (8B keys and values) on PM-optimized state-of-art commercial KVS (Intel's pmemKV [38] and RocksDB-pmem [79]) and academic KVS ([100]) on a many-core CPU. The fourth bar shows throughput with a GPU-enabled KVS, MegaKV [102], ported onto our system named GPM, to leverage PM's persistence with 23 lines of code changes. GPM (GPU with Persistent Memory) enables fine-grain persistence to GPU kernels (programs). GPM improves throughput by 2.7-5.8× over today's multi-threaded CPU alternatives.

Many other applications, *e.g.*, breadth-first search (BFS), image processing (SRAD), and prefix sum (PS) benefit from fine-grained persistence while leveraging GPU's parallelism. They speed up by 2.8-27× over their multi-threaded CPU alternatives that use PM for persistence (Figure 1b).

Today, if an application wishes to leverage PM's persistence, it would typically perform both computation on the CPU and ensure persistence of results from the CPU. Alternatively, one can use a GPU for computation then transfer the results to the CPU's memory and rely on the CPU to guarantee persistence for recoverability in presence of failures. We call this alternative that uses GPU as CPU-Assisted Persistence (CAP).

Unfortunately, CAP has several shortcomings. Its inability to efficiently guarantee persistence of PM-resident data structures at a fine granularity from the GPU impedes programmers' abilities to create recoverable GPU kernels. Second, GPUs accelerate computation through massive parallelism. Many benefits of parallelism are lost by relying on the CPU to persist results of GPU computation. Third, sometimes only a fraction of data is updated during

*Both authors contributed equally to this work.
†The author is currently affiliated with University of Washington. The author contributed toward this work when he was a research assistant at the Indian Institute of Science.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507758

computation. However, which data would be updated is not known apriori [2, 12, 14]. Since the GPU cannot directly persist results while computing, extraneous data could be transferred to and persisted by the CPU. In short, GPU kernels today cannot harness the full benefits of the byte-grained low-latency persistence of PM.

We thus create GPM where GPU kernels can directly manipulate PM-resident data structures and guarantee persistence wherever desired in the kernel without the CPU's or OS's help.

Currently, there is no hardware with NVM onboard the GPU. An incarnation of GPM is still realizable without new hardware. The NVM (Intel Optane) is placed alongside the DRAM, as in a typical Intel Xeon server, and can be accessed by a GPU over the PCIe interconnect. GPM leverages NVIDIA's Uniform Virtual Address (UVA) to map desired portions of NVM onto the virtual address space of a GPU kernel. Kernels can access and manipulate PM-resident data structures at byte granularity using loads and stores.

To compose programs that are recoverable in the presence of crashes or power failures, a programmer must be able to guarantee persistence of data to PM (i.e., a *persist*) wherever programs' semantics demand. A persist operation typically requires flushing contents of volatile cache lines to PM and waiting for these flushes to finish. Unfortunately, unlike CPUs, today's GPUs are not designed for PM and thus, do not have instructions to flush cache lines [34, 69]. However, we find that a fence operation with *system* scope (__threadfence_system() in CUDA) ensures that all writes to the system (host) memory before the fence are made visible to the entire system, including the host (CPU). While the original purpose of the fence was synchronization between the CPU and GPU, it provides the semantics needed for persist operations. This is because in GPM, the NVM is a part of the system memory, alongside DRAM.

The system-scoped fence alone is not sufficient to create persist operations for GPUs. Another seemingly unrelated feature stops writes from reaching PM immediately. When an Intel Xeon processor's Data Direct IO (DDIO) feature is enabled (default), writes to the system memory by IO devices, e.g., NIC, GPU, are cached in the CPU's last level cache (LLC) [33, 44, 96] . Consequently, the fence completes as soon as the writes reach the CPU's volatile LLC. GPM, therefore, *selectively* disables DDIO when persistence is needed. This ensures that the system-scoped fence completes only when the persistence of writes to PM is guaranteed.

In short, we use UVA to map PM to GPU's address space, and system-scoped fence with selective disabling of DDIO to create GPM out of Xeon servers with Optane NVM and NVIDIA GPUs.

A new system is useful only if there are important use cases for it. We find three categories of applications that benefit from GPM. Transactions in GPU-accelerated persistent KVS and relational databases benefit from fine-grained logging to PM. Long-running applications that iteratively invoke GPU kernels, *e.g.,* DNN training, benefit from faster checkpointing to PM for fault tolerance and preemption [63]. Finally, GPM enables GPU kernels to embed the logic to perform in-place byte-grained updates to PM-resident data structures while ensuring they remain recoverable (consistent) after a crash. These kernels can then resume, rather than restart computation upon recovery from a crash. GPU-accelerated BFS on PM-resident graphs is an example. In the process, we created a workload suite, named GPMbench, with 9 GPU workloads that leverage PM's persistence.

Our third contribution is a GPU (CUDA) library, libGPM, that enables GPU-optimized parallel logging, checkpointing, and persist operations. The cornerstone of libGPM is the Hierarchical Coalesced Logging (HCL), a write-ahead (undo) logging facility to implement transactions. It incorporates two GPU-specific optimizations. ① To scale logging for GPUs, where hundreds of thousands of threads may attempt to insert entries into the log concurrently, HCL mimics the GPU's execution hierarchy of threads in the log's structure. This allows every GPU thread to insert an entry at specifically computed indices in the log without locking. ② HCL leverages the GPU's hardware coalescer that merges parallel writes to a cache line by a set of threads executing in lockstep. It stripes log entries across cache-line-sized units such that logging by a set of threads coalesces into a single write to a cache line.

Our key contributions are as follows:

- We created GPM to bring fine-grain persistence to GPUs.
- We created GPMbench workload suite with three classes of GPU-accelerated applications leveraging PM.
- We created libGPM CUDA library for GPU-optimized logging, checkpointing, and in-place updates and persists to PM.
- We developed GPU-optimized HCL that speeds up logging by up to 6.1× over conventional distributed logging.
- Across diverse PM-optimized applications, including persistent KVS to BFS, GPM provides 2-27× performance improvement over their multi-threaded CPU alternatives.
- Over CAP, GPM enables performance improvements of up to 85× by leveraging GPU's parallelism in persisting data and by avoiding extraneous data movement.

2 BACKGROUND

Persistent memory: Intel's Optane Persistent Memory [37] is the first commercially available NVM technology that enables PM by allowing load/store accesses and persistence. Optane's access times are only 3-10× of DRAM [41] and it is much denser – supporting up to 3TB on a single socket [77]. A challenge for applications using PM is ensuring recoverability – maintaining consistency of PM-resident data structures in the face of a crash [18]. Persistence alone does not guarantee recoverability. Updates to PM could be cached in the processors' volatile caches. Consequently, the order in which data is written to the cache could be different from the order it reaches the PM. For example, failure during insertion into a doubly-linked list could lead to dangling pointers if the pointer update reaches PM before the data.

A *persist* operation is needed to guarantee writes to PM are durable. It is typically implemented using a combination of a flush and a drain operation. The x86 CPU architectures provide instructions (*e.g.,* CLFLUSHOPT) that flush a cache line to memory [32]. Alternatively, one can eschew benefits of caching using non-temporal (nt) stores to bypass caches. Durability, however, is only guaranteed when the flush (or nt store) is followed by a drain, *e.g.,* SFENCE, that ensures pending flushes (or nt stores) are complete [82].

CPU's memory controllers' (MCs) ADR (Asynchronous DRAM Refresh) feature helps in hiding higher latency of NVM writes by buffering writes in a capacitor-backed on-chip Write-Pending Queue (WPQ). Durability is guaranteed as soon as writes are buffered

in WPQ. As long as free entries exist in the WPQ the extra latency of writing to NVM is hidden.

GPU's execution hierarchy: GPU's hardware resources are arranged in a hierarchy to scale to massive parallelism. Streaming multiprocessors (SMs) are its basic compute blocks. SMs contain many SIMD units that share an L1 cache and scratchpad. A GPUs contain tens of SMs. All SMs in a GPU share a L2 cache. GPU programming languages also expose an execution hierarchy. In CUDA, a thread runs on a single lane of a SIMD unit. A set of threads (*e.g.,* 32), typically executing in lockstep forms a warp. When threads in a warp execute SIMD loads/stores whose addresses fall on the same cache line (typically 128 bytes in GPU), the hardware coalesces them into a single cache access. Many warps make a threadblock that executes on a SM. Finally, work is dispatched to a GPU at the granularity of a grid (kernel) having several threadblocks.

Since GPU's L1 caches are not kept coherent, GPUs provide fence instructions (*e.g.,* __threadfence) to order memory operations for ensuring visibility of writes to other threads when desired [59]. Leveraging the execution hierarchy, GPUs added the ability to perform synchronization within only a subset of threads. For example, in CUDA, fence operations support three different *scopes* – block, device and system. An operation with a given scope is only guaranteed to be visible to threads within the scope of that operation. For example, a device-scope operation is only guaranteed to affect threads of a kernel on a GPU. The system scope affects all GPU and CPU threads, and those in other GPUs for multi-GPU kernels.

3 THE CASE FOR GPM AND ITS DESIGN

Before the advent of NVM technologies, only the filesystems atop block-storage devices could guarantee persistence. This would typically involve writing to a file and then issuing an *fsync* or an *msync* syscall for persistence. In contrast, the data on NVM can be mapped onto an application's address space. Applications can directly access that data using loads and stores and then guarantee persistence using user-space cache flush and drain when desired. The direct access to PM created a new class of CPU applications that provide fine-grain recoverability after a crash or a power failure [19, 64, 93].

Unfortunately, direct access to PM is still beyond the GPU's reach. Even on systems with PM, if a GPU kernel desires to persist results, it is *forced* to rely on the CPU to do so. Figure 2(a) depicts how a GPU-accelerated application can persist results of GPU computation today, in three broad steps. ① The GPU driver moves data from GPU's device memory to CPU's DRAM using DMA. ② The CPU then copies the data from host DRAM to the NVM. ③ The CPU finally guarantees the persistence of the data to PM by evicting contents from caches. We refer to this three step process as CPU-Assisted Persistence or CAP.

CAP can be realized in multiple ways. One could rely entirely on the filesystem, which we name CAP-fs. Here, after results are DMA-ed to DRAM, the CPU writes the results to a PM-resident file, and then guarantees persistence using fsync(). Alternatively, a PM-resident file can be memory-mapped onto the CPU's address space. The results can be transferred from GPU's memory to the memory-mapped file using *cudaMemcpy*. However, cudaMemcpy would not move the data directly to the file. Instead, it internally uses a pinned memory on DRAM as a bounce buffer to copy data without the possibility of page-faults [68]. It would copy data from GPU to the buffer and then write it to the file. Finally, CPU threads issue cache flushes and drains to guarantee persistence. This approach limits OS overheads but still relies on the CPU for persistence. We name it CAP-mm. Note that CAP-mm cannot use non-temporal stores because the CPU is not generating the data to be written. The data comes to the LLC from the GPU.

Limitations of CAP: ① The inability to directly access PM-resident data structures and persist while computing results (i.e., *in-kernel persistence*) precludes programmers from writing fine-grain recoverable GPU kernels. ② GPUs accelerate applications through massive parallelism. By relying on CPUs to persist results of GPU computation, many of the benefits of parallelism are lost. ③ The lack of in-kernel persistence could lead to *write-amplification*. Sometimes, only a part of the data that a kernel computes upon would be updated and, thus, persisted (*e.g.,* parts of a KVS). There is no way for a GPU programmer to efficiently transfer and selectively persist results to PM at byte granularities.

3.1 GPM's Design Philosophy

We therefore envision a system where GPUs can directly write and persist data to PM with no CPU involvement. We name it GPU with Persistent Memory (GPM). Figure 2(b) depicts how GPM works without CPU's assistance. It uses NVIDIA's Unified Virtual Address [67] to map desired parts of PM onto the GPU's virtual address space. This allows a GPU kernel to issue loads/stores to PM-resident data structures. To guarantee persistence from within a kernel, we use a system-scoped fence instruction that guarantees writes to the system memory are visible to the entire system (__threadfence_system() in CUDA). Although modern servers' (*e.g.,* Intel Xeons') memory controllers are ADR-enabled, guaranteeing visibility of writes is not enough to guarantee persistence due to DDIO [33, 44]. When DDIO is enabled (default), GPU's writes to system memory are cached in CPU's LLCs. They do not immediately proceed to the memory controllers [44]. Thus, GPM *selectively* turns off DDIO for GPUs when persistence is desired.

3.2 Understanding the Benefits of GPM

GPM's ability to persist data to PM at byte granularities from a GPU enables fine-grain recoverable kernels. Consider kernels to compute the prefix-sum of large arrays that form the backbone for many scientific and sorting applications. They can leverage fine-grain recoverability to *resume* instead of restarting computation after a crash (more in § 4.3). GPM also enhances programmability since kernels can persist (intermediate) results without coordinating with the CPU. One may wonder if fine-grain recoverability and ease of programmability comes at a performance cost over CAP's coarser-grain persistence. We analyze this performance question next.

Parallelism in persisting data: The key sale of a GPU is its massive parallelism that can hide longer latencies. An individual write, followed by system-scoped fence from a GPU can take longer than flushing a CPU cache line, followed by a drain [66]. However, a GPU can use massive parallelism to hide larger latency. To quantitatively demonstrate this, we create a microbenchmark that writes and persists 1 GB of data from the GPU to PM. On GPM, we vary the number of GPU threads, writing and persisting data at an 8-byte

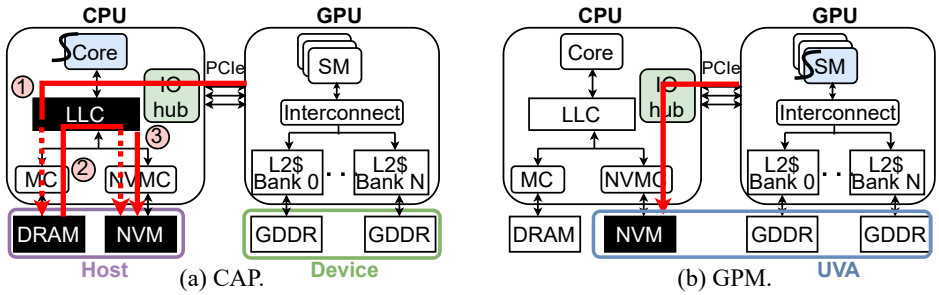


Figure 2: Overviews of CAP and GPM. Dotted lines denote natural cache evictions.

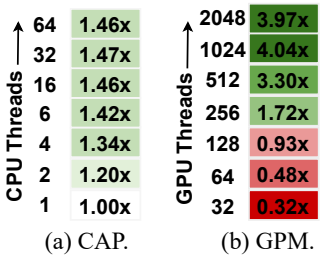


Figure 3: Scaling of persistence.

granularity. On CAP-mm, we vary the number of CPU threads to persist at a coarse granularity. Data is equally partitioned among the CPU threads, where each persists its portion at once.

Figure 3(a) shows how the microbenchmark speeds up with increasing number of CPU threads on CAP-mm. It plateaus at 1.47× over single-threaded persistence. Figure 3(b) shows the same on GPM with an increasing number of GPU threads. GPM's performance scales to 4× over the single-threaded CAP-mm. While a GPU can have many more threads, it typically supports a limited number of concurrent operations on the PCIe [1]. Thus, it does not scale beyond a point. In general, however, the experiment shows how GPM can use parallelism to hide the latency of writing and persisting better than CAP even at a finer grain.

Selective persistence: GPM enables kernels to *selectively* persist only as much data, and as and when needed by the program semantics for recoverability. Consider, batched SETs to a PM-resident KVS containing hundreds of millions of key-value pairs. The SETs running on the GPU would update only a fraction of those pairs. However, which pairs would be updated are known only upon computing the indices. Thanks to the fine-grained in-kernel persistence offered by GPM, a kernel can persist *only* the updated entries as the indices are computed.

On CAP however, guaranteeing persistence from within a kernel at a byte granularity is not possible. Consequently, after the execution of SETs on GPU, the entire KVS (or sections of it) needs to be transferred to CPU memory and be persisted by the CPU. While smaller granularities of transfer can moderate extraneous data movement in a few applications, the overhead of initiating fine-grain transfers from the CPU remains high enough to nullify any scope for improvement [61]. In general, applications with input-dependent irregular memory accesses can potentially incur extraneous data movement [2, 12, 14].

Direct access to PM: An obvious benefit of GPM is its ability to manipulate PM-resident data structures using direct loads and stores from the GPU. Hundreds of thousands of GPU threads can perform loads/stores concurrently to hide the latency of accessing PM. In contrast, flushing updates to PM via the CPU could have required thousands of CPU cache line flushes at a 64-byte granularity for even a few MBs of data. CPU's limited ability to hide latency adds to CAP's overheads.

3.3 Discussions

Alternatives to GPM: We are not the first to foresee the usefulness of non-volatility to GPU. However, no system today brings the byte-grained low-latency persistence of PM to the GPU. For example, NVIDIA's GPUDirect Storage [89] and its precursor SPIN [10] enable direct transfer of data between NVMe SSDs and GPU memory using DMA. They are applicable to block-storage devices with DMA. They also do not have a way to explicitly guarantee persistence of data at byte granularities. Further, GPUfs exposes file-related system calls (*e.g.,* gread, gwrite) to GPU kernels [87]. GAIA focuses on consistency protocols for sharing memory-mapped files across the CPU and the GPU [13]. Both GPUfs and GAIA rely on the CPU and the OS to persist data to storage, like in CAP. In § 6.1, we further quantitatively compare against GPUfs.

Impact of upcoming technologies on GPM's design: Intel recently announced eADR (enhanced ADR) feature in their future server processors [36]. This hardware feature, along with 2nd generation Optane NVM will drain the entire contents of CPU caches to PM on power failures [3, 27, 36, 39]. This feature will obviate the need to flush cache blocks from CPU's caches in order to guarantee persistence in future processors. However, the fence is still needed to maintain ordering of writes to PM [39]. On a system with eADR, GPM would *not* require to disable DDIO to guarantee persistence of GPU writes. This is because persistence is guaranteed once the GPU writes reaches CPU's LLC. This will improve GPM performance significantly. On the other hand, for CAP, cache blocks would not need to be explicitly flushed to the PM but fences are still needed as mentioned earlier. In Section 6, we quantitatively project performance of GPM and CAP on a future system with eADR and show GPM remains useful.

Another key emerging technology is Compute Express Link (CXL) [80] – an open industry standard for interconnect enabling coherence and memory semantics between the CPU, GPUs and other memory devices for building disaggregated systems. CXL 2.0 provides support for PM and allows devices to directly (de-)allocate data on CXL-attached PM. CXL 2.0 provides a Global Persistent Flush (GPF) instruction that allows PM-aware applications to flush their data to the CXL-attached PM. However, GPF can only be issued from the host CPU and it flushes all persistent data from all device caches. In short, CXL-attached PM alone cannot enable fine-grain, in-kernel persistence from a GPU. We believe the design principles of GPM can be extended to CXL-attached PM.

Table 1: GPMbench: Workloads for GPM.

	Workload	Data structure on PM Input set; size	Parallel op on PM
Transactional	Key-value Store (gpKVS) [43, 79, 102]	Key-value store (25 batches of 2M SETs and, 100 batches of 2M 95:5 GET:SET; 4.1GB)	Batched parallel SET and GET operations
	GPU-accelerated DB (gpDB) [6]	Relational DB table (50M row insert, 2.5M row update; 3GB)	Insertion and updation of individual rows
Iterative long-running	DNN Training (DNN) [71, 88]	Checkpoint weights, biases (MNIST dataset [53]; 3.2MB)	Calculation of weights and biases
	Computational Fluid Dynamics (CFD) [15]	Checkpoint flux, momentum, etc. (Surface of missile; 8.9MB)	Compute flux, momentum at each point
	Black-Scholes (BLK) [70]	Checkpoint predicted prices (256M stock options; 4GB)	Calculating price of individual options
	Hotspot (HS) [15]	Checkpoint temperatures (16K * 16K power and temp matrix; 2GB)	Estimating temperature
Native	Breadth-First Search (BFS) [25]	Input graph, search sequence, cost (USA road network; 1GB)	Calculating cost from source for each node
	SRAD [25]	Diffuse noise per pixel (128K * 1K; 3GB)	Output image, diffusion matrix
	Prefix Sum (PS) [70]	Integer arrays (1K * 1M integers; 4GB)	Partial and final sum

4 GPMBENCH: USE CASES FOR GPM

NVM technologies enable both persistence and higher capacity. While it is easy to appreciate that many GPU applications could leverage larger memory capacity, it is less obvious which ones would benefit from both GPU's parallelism and persistence. Thus, a goal of this work is to identify such use cases.

We identify 9 such GPU-accelerated applications and kernels (referred to as workloads). These workloads, listed in Table 1, are chosen from well-known benchmark suites and open-source applications [6, 15, 25, 70]. We modified them to use PM and named this suite GPMbench. The workloads are categorized into three classes based on their use of persistence. The table also lists the key operations in each workload that leverage both parallelism and persistence. We report the minimal set of data structures that are persisted for meaningful recovery after a crash for each workload. The rest are mapped on the volatile memory for better performance. We will open-source GPMbench to aid future research in the area.

4.1 Transactional Updates to PM

Arguably, a transaction is the most common way for software to persist data in a recoverable manner, by ensuring the atomicity and durability of the updates. Logging is the primary mechanism to achieve these properties of a transaction.

Researchers have explored persistent KVS on PM [43, 55, 79, 103] and GPU-accelerated KVS [102] in isolation. GPM makes it possible to leverage them together. A PM-resident KVS can leverage higher capacity of PM and also utilize persistence for fault tolerance [43, 55, 79]. GPUs on the other hand, enable high throughput for batched SETs and GETs [102]. Transactions are typically used for recoverability when servicing SETs [38, 79, 100]. We extended a GPU-accelerated KVS, called MegaKV [102], to execute transactions of batched SETs and GETs on GPM (gpKVS).

Databases are one of the biggest proponents of transactions. Today's GPU-accelerated databases such as OmniSci [74], Virginian [6], and HippogriffDB [56] increase throughput of business analytics queries by executing primarily SELECT queries and perform database searches on GPU. However, they typically avoid executing transactions that modify the database as there is no efficient way to directly persist results from the GPU. GPM changes that. We extended Virginian GPU-accelerated database to implement transactions for batched update and insert queries covering millions of rows of a PM-resident relational database table (gpDB). In § 5, we detail our CUDA library that implements GPU-optimized write-ahead logging for creating transactions.

4.2 Iterative Long-running Kernels

Many long-running GPU-accelerated applications iteratively invoke kernels and checkpoint intermediate results for fault tolerance and for early termination. For example, DNN training often checkpoints partially trained weights and biases [63]. Checkpointing performance assumes heightened importance due to increasing training time of ever growing DNN models. Besides hardware and software failures, the use of cheaper preemptible VMs in public clouds for training makes job interruptions more frequent [65]. Consequently, DNN training needs fine-grain checkpointing to avoid loss of computing [63]. On GPM, such applications benefit from fast checkpointing to PM without CPU overheads. In GPMbench, we use NVIDIA's cuDNN kernels [71] to train a LeNet model [52] on the MNIST image data set [53]. After a user-defined number (*e.g.*, 10) of forward and backward passes, we checkpoint the weights and biases of the partially trained model on the PM.

Computational fluid dynamics (CFD) kernels are often iteratively invoked by long-running HPC applications. The CFD kernel, drawn from the Rodinia suite [15], implements a grid solver for Euler equation for inviscid and compression flow. The flux, momentum, and density are computed over many timesteps. We periodically checkpoint these to PM. Further, as listed in Table 1, financial applications like Black-Scholes and simulation applications like Hotspot can checkpoint results of intermediate computations for fault tolerance.

4.3 Native Persistence

GPM enables a new class of GPU kernels that performs fine-grain in-place updates to PM-resident data structures. The kernel that performs computation also persists (intermediate) results for recoverability. Thus, the persistence is native to it.

Consider a GPU-accelerated breadth-first search [25]. The kernel persists the node search sequence and cost of traversal for each node on PM. On a crash, applications using BFS can use the persisted partial traversals and the search sequence to resume the traversal. Prefix sum is another kernel that is often used in applications such as sorting networks and benefits from fine-grained persistence. It outputs an array on PM, where each element at index 'j' contains the sum of all elements of the input array preceding 'j'. The kernel divides the array into subarrays and assigns each subarray to a threadblock to compute its prefix sum. The partial sums are then used to compute the global sum. Another application SRAD (Table 1) benefits from GPM too but we omit details for brevity.

Table 2: libGPM APIs.

	Caller	Function
Primitive	CPU	gpm_map(path, size, create)
		gpm_unmap(path, addr, size)
		gpm_persist_begin() / gpm_persist_end()
	GPU	gpm_persist()
Logging	CPU	gpmlog_create_conv(path, size, n_partition)
		gpmlog_create_hcl(path, size, blocks, threads)
		gpmlog_open(path) / gpmlog_close(log)
	GPU	gpmlog_insert(log, addr, size, partition = -1)
		gpmlog_read(log, addr, size, partition = -1)
		gpmlog_remove(log, size, partition = -1)
		gpmlog_clear(log, partition = -1)
Checkpointing	CPU	gpmcp_create(path, size, elements, groups)
		gpmcp_open(path) / gpmcp_close(cp)
		gpmcp_register(cp, addr, size, group)
		gpmcp_checkpoint(cp, group)
		gpmcp_restore(cp, group)

We also note that it is important to avoid unnecessary accesses to PM for better performance since accesses to PM are slow. Hence, read-only data structures, *e.g.,* input graph in BFS, are read onto GPU's device memory (HBM) once, without affecting recoverability.

No single platform suits all types of applications perfectly. During our exploration, we found applications that are less suitable for GPM. Consider computation of binomial options, a pricing model for options valuation in financial markets [84]. In its GPU-accelerated computation, threads in a threadblock coordinate to compute a *single* value which is written by a single thread of a threadblock [48]. That leaves little parallelism to exploit in writing and persisting data to PM. GPM's fine-grained persistence brings fine-grained recoverability. However, GPM needs parallelism for good performance.

5 LIBGPM: THE LIBRARY ENABLING GPM

Our third key contribution is a CUDA library to aid programmers in leveraging GPM. We name it libGPM. libGPM abstracts away complex persistence logic and low-level GPU details from the programmer while providing GPU- and PM-specific optimizations for manipulating persistent data structures. Table 2 lists libGPM's API, categorized by usage type.

5.1 Persistency Primitives

First, libGPM enables allocation/deallocation of persistent memory using *gpm_map* and *gpm_unmap*. Typically, the memory needed for GPU kernels is statically allocated or deallocated on the CPU, before and after a kernel launch. Therefore, we do the same for allocating PM. To allocate memory on PM, a PM-resident file is memory-mapped using Intel PMDK's libpmem library [35]. Using CUDA's UVA [83], it maps the newly allocated memory to the GPU's address space, enabling direct access to PM via loads/stores. Memory is deallocated by unmapping the address range and the file.

Intel Xeon processors use DDIO [33] to cache writes from GPU in the LLC. This would break the persistence guarantee on GPM. Our library provides *gpm_persist_begin()* and *gpm_persist_end()*, that switches DDIO off and on for the GPU by writing to the I/O register perfctrlsts_0 [22]. The persistence guarantees by the library are valid only inside the regions marked by these routines, typically placed before and after a kernel launch.

The *gpm_persist()* routine ensures prior writes by a GPU thread are made persistent. It uses the system-scoped fence operation to wait until data reaches the system memory (i.e., CPU's memory controllers) over the PCIe. This is sufficient to guarantee persistence in GPM implemented on a server with Intel Xeon processors having ADR-enabled memory controllers.

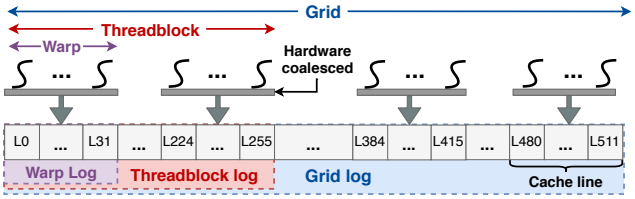


Figure 4: The log structure in hierarchical coalesced logging.

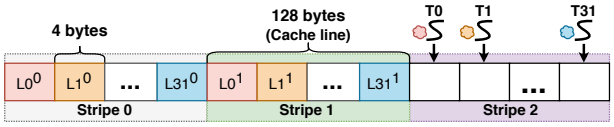


Figure 5: Striping for larger log entry. Ln^k denotes k^{th} 4-byte chunk of a log entry for n^{th} thread in a warp.

5.2 Enabling Logging to PM from GPUs

Logging is a key functionality that enables atomicity and durablity for transactional updates [62]. We implement write-ahead undo logging, where the undo log is persisted before an update. Efficiently scaling logging to GPU's hundreds of thousands of threads is a significant challenge.

Conventional logging on CPU typically uses sequential, lock-based accesses to a log file, but is impractical for parallel systems due to serialization overheads. Prior works [9, 11, 94] have proposed keeping multiple distributed log files (partitions). Insertion of log entries to different partitions can proceed concurrently, but that to the same partition is serialized. We improve upon this to avoid all serialization with two key *GPU-specific* optimizations as follows.

The execution in a GPU follows a hierarchy of warp, threadblock, and grid to scale to massive parallelism. We posit that logging on GPU should mimic the same hierarchy to harness its parallelism. Next, we observe that the GPU is optimized for SIMD execution. For example, 32 threads in a warp execute in lockstep, and the GPU hardware coalesces loads/stores that fall on the same 128-byte, aligned block into fewer load/store operations. This is critical to performance; accessing host memory over the PCIe is slow, and thus, the lesser the number of accesses the better is the performance. Prior work also demonstrated that PCIe is better utilized when a warp accesses data at a 128-byte, aligned granularity [1].

Driven by these observations, we design GPU-optimized logging as Hierarchical Coalesced Logging (HCL). In HCL, each GPU thread logs the data it is to modify in a data-parallel manner. HCL divides a PM-resident log file such that each thread has a *unique offset* to insert its entry. Consequently, threads can concurrently insert log entries without locks. We first explain how HCL achieves this by assuming each thread inserts only a 4-byte entry into the log and relax this constraint later.

Figure 4 depicts how the GPU's execution hierarchy is reflected in the structure of the HCL's log. It also shows how HCL leverages hardware coalescing while writing log entries from each warp. A log

```
1  __global__ void insert_into_kvs(...) {
2    int hash = getHash(data.key);
3    int loc = getLocation(hash, threadId % THRD_GRP_SZ);
4    if (loc != -1) {
5      log_entry entry;
6      entry.hash  = hash;
7      entry.loc   = loc;
8      entry.key   = kvs[hash].key[loc];
9      entry.value = kvs[hash].value[loc];
10     gpmlog_insert(&entry, sizeof(log_entry));
11     kvs[hash].key[loc]   = data.key;
12     kvs[hash].value[loc] = data.value;
13     gpm_persist();
14   }
15 }
```

(a) Simplified application kernel in CUDA.

```
1  __global__ void recover(...) {
2    log_entry entry;
3    gpmlog_read(log, &entry, sizeof(log_entry));
4    int hash             = entry.hash;
5    int loc              = entry.loc;
6    kvs[hash].key[loc]   = entry.key;
7    kvs[hash].value[loc] = entry.value;
8    gpm_persist();
9    gpmlog_remove(log, sizeof(log_entry));
10 }
```

(b) Recovery kernel in CUDA.

Figure 6: Use of logging in gpKVS application.

file is divided into 4-byte chunks with each thread logging 4-bytes of data into the log at a time. Since the GPU coalesces accesses to a 128-byte block [30], writes to the log by threads in a warp becomes a single store. This also improves NVM's endurance. As shown in the figure, each warp has a specific cache-line-aligned offset in the log to insert its entries. Similarly, each threadblock has a specific region in a log for inserting entries generated by its warps. In short, the offset of 4-byte chunk within a log where a thread will insert its entry is uniquely calculated based on its thread, warp, and threadblock ID. Thus, HCL needs no serialization.

If each thread needs to log more than 4 bytes, HCL stripes log entries across multiple blocks (Figure 5). Striping keeps the benefits of HCL and employs a sequence of SIMD stores to insert log entries. In a SIMD store, each thread in a warp inserts a 4-byte chunk of log into a stripe for that warp. For failure-atomicity, the entire entry or none of it should be visible in the log after a crash. Thus, a thread persists its entry, then increments and persists a tail index of its log. This tail index acts as the sentinel for the logs used during recovery.

The size of the individual log entry is known statically based on the data being logged. Therefore, the number of stripes needed is known apriori. Further, in typical GPU-accelerated software, (likely) parallel tasks (*e.g.,* thousands of key-value pair insertions in gpKVS) are launched on to the GPU in batches. The number of parallel tasks and thus, the number of GPU threads are known at the kernel launch. Consequently, the number of logging threads and their offset into HCL's log is known before the kernel starts execution.

The *gpmlog_insert* routine can be invoked by a GPU thread to insert a log entry. The routine calculates the offset in the log for inserting the entry based on the calling thread's IDs. It also correctly persists the log entry by invoking *gpm_persist*. In short, the intricacies of HCL and that of ensuring persistence are abstracted away from programmers.

HCL is useful for parallelly logging a massive amount of data, but not for logging small metadata (*e.g.,* the table size). Metadata is typically logged by one thread in a threadblock or a warp. Thus, libGPM supports both conventional logging and HCL. The routine *gpmlog_create_conv* creates log partitions with conventional logging and *gpmlog_create_hcl* creates partitions for HCL. While inserting an entry into a conventional log, instead of calculating the log offset, the *gpmlog_insert* routine automatically appends the entry to the specified log partition after acquiring a lock. Table 2 lists the full API for logging.

Example use of logging: We demonstrate how libGPM's logging APIs can be used for implementing transactions with the help of code snippets. Figure 6(a) shows a (simplified) CUDA kernel for recoverable batched insertions in gpKVS. This code is executed by every thread of the kernel. Before the kernel begins execution, a flag is set and persisted to indicate that a transaction on the GPU is active (not shown). gpKVS, which is derived from MegaKV [102], uses an 8-way set-associative structure as the KVS to limit collisions. Each thread calculates an insertion position (a set) by hashing the key (line 2). A group of eight threads (THRD_GRP_SZ=8) then coordinates to select one of the ways in the set and thus, the thread in the group that will insert the entry (line 3). The selected thread (line 4) creates a log entry with the current pair in the selected location (line 5-9) and inserts it into the log (line 10). The insertion routine ensures persistence of the log entry. The new pair is inserted into the KVS (line 11-12), then persisted (line 13).

Recovery using logs: On a crash, gpKVS undoes the last batch of partially completed batched insertions. During the recovery, if the transaction flag is not set, the crash did not occur during an active insertion, and logs can be truncated. Otherwise, the logs are used to undo the partial inserts. Figure 6(b) shows a simplified CUDA code of this recovery logic. Each GPU thread is responsible for undoing a single insertion in the failed batched insertion. Each thread reads one of the key-value pairs from the log (lines 2-3). The corresponding location in the KVS for a pair is obtained in line 4-5. That location is updated with the logged entry, i.e., previous value (line 6-7) and is persisted (line 8). To ensure recoverability during recovery itself, the log entry is only removed after successfully updating and persisting the gpKVS (line 9).

5.3 Enabling Checkpointing to PM for GPUs

The library enables an application to associate semantically related data structures with a checkpoint group. It exposes an API (Table 2) for a programmer to checkpoint and restore data structures in a given group together. Different groups of data structures are checkpointed/restored independently.

The *gpmcp_create* routine creates a checkpoint file and initializes the checkpoint's internal structures. The checkpoint structures are 128-byte aligned to maximize bandwidth to the NVM and across the PCIe. Users must specify the size of the data being checkpointed, the number of groups, and elements per group for the checkpoint. Checkpoint files are opened and closed from the CPU using *gpmcp_open*, and *gpmcp_close*. Once a checkpoint is created, the programmer should register data structures to checkpoint using *gpmcp_register* routine. Internally, the library notes the address and size of the data structure and its group.

```
1  void dnn_training(...) {
2    // d_weights is checkpointed in cp_weights
3    gpmcp *cp_weights;
4    if (!RECOVERY_MODE) {
5      cp_weights = gpmcp_create("./file_weights",
6        size_weights, numElements, numGroups);
7      gpm_register(cp_weights, d_weights,
8        size_weights, groupId);
9    }
10   else {
11     cp_weights = gpmcp_open("./file_weights");
12     gpm_register(cp_weights, d_weights,
13       size_weights, groupId);
14     gpmcp_restore(cp_weights, groupId);
15   }
16   while(!training_done) {
17     ... // Calculation
18     gpmcp_checkpoint(cp_weights, groupId);
19   }
20 }
```

Figure 7: Use of checkpointing for iterative GPU programs.

To initiate checkpointing, *gpmcp_checkpoint* should be called with a checkpoint group ID. This routine launches a GPU kernel that copies registered data structures of the group to PM. The routine *gpmcp_restore* copies the data from the PM-resident checkpoint to the corresponding volatile data structures on the device. The library internally employs double buffering, where two checkpointed copies of each data structure are kept on PM. One is the 'consistent' copy, while the other is the 'working' copy. When a checkpoint is initiated, data is copied into the working copy. On completion, libGPM persists the working copy, and atomically marks it as the consistent copy and vice-versa. The routine internally uses *gpm_persist* to persist checkpoints.

Figure 7 shows checkpointing in the DNN training application. Here, '*cp_weights*' is a checkpoint for checkpointing DNN's weights, while '*d_weights*' is the volatile data structure containing weights. A programmer first creates a checkpoint file on PM for the weights (lines 5-6) and then registers '*d_weights*' with the checkpoint (lines 7-8), associating it with groupId. On reaching the checkpoint directive (line 18), '*d_weights*' is checkpointed as it belongs to groupId.
Recovery: To resume computation, we need to reconstruct the data structures from the last consistent copy of a persisted checkpoint. A programmer first opens an existing checkpoint file (line 11). Upon a crash, the mappings between the checkpointed copies of data structures and the addresses of those structures in process's address space are lost. The library, thus, relies on the order of registration of data structures to a checkpoint for identifying which data structure a checkpointed structure should be restored to. Hence, the programmer should maintain the same order of registering data structures during restoration (lines 12-13). Then, a programmer can use gpmcp_restore (line 14) to restore from a checkpoint. However, pointer-based data structures cannot be checkpointed.

5.4 Enabling Native Persistence for GPUs

Native persistence workloads do not require any additional support than what already described. To demonstrate how such applications leverage libGPM for recoverability, we use the example of computing the prefix sum of an array. The input array is divided among the threadblocks, and each thread in the threadblock finds the prefix sum for a single element within the subarray, called partial sums.

```
1  __global__ void partial_sums(...) {
2    // Partial sum of last thread in block exists, skip
3    if(pm_p_sums[(blkId + 1) * blkSize - 1] != EMPTY)
4      return;
5    // Compute partial sum for each thread
6    ...
7    // All but last thread in block persist partial sum
8    if(blkThreadId != blkSize - 1) {
9      pm_p_sums[globalThreadId] = thread_p_sum;
10     gpm_persist();
11   }
12   __syncthreads(); // threadblock barrier
13   // Last thread in block persists partial sum
14   if(blkThreadId == blkSize - 1) {
15     pm_p_sums[globalThreadId] = thread_p_sum;
16     gpm_persist();
17   }
18 }
```

Figure 8: Native persistence in prefix sum kernel.

Table 3: Configuration of the experimental platform.

CPU	4× Intel Xeon Gold 6242 (4 × 16 cores) @ 2.80GHz
GPU	NVIDIA Titan RTX (72 SMs, 24 GB GDDR6)
DRAM	768 GB DDR4 @ 2933 MHz
NVM	8 x 128 GB Intel Optane NVDIMM
Interconnect	PCIe 3.0 ×16
Software	Ubuntu 20.04, CUDA 11, PMDK 1.8, ext4-DAX

Figure 8 shows the CUDA snippet for prefix sum kernel. A persistent copy of each thread's partial sum is kept in the pm_p_sums array. At the start, threads check whether PM contains the partial sum for the last thread in the threadblock (line 3). We shall later see why. If not, each thread computes its partial sum. All threads except the last thread in a threadblock store and persist their partial sums into the pm_p_sums array (lines 9-10). The threads then wait at a threadblock barrier to ensure all completed their calculations (line 12). Then the last thread in the threadblock updates and persists its partial sum (lines 15-16).
Recovery: These workloads do not need separate recovery programs as the recovery logic is embedded within the workloads themselves. Consider prefix sum, the partial sum of the last thread in the threadblock is persisted only after all the other threads of the threadblock have finished persisting their sums. Therefore, after a crash, if a value is present in the array for the last thread (line 3), then all the threads would have had their values persisted. No recomputation is needed for that threadblock. Otherwise, recomputation is required for the subarray assigned to that threadblock.
Lines of code changed: As libGPM encapsulate intricacies of programming GPM, only limited modifications are needed for application kernels to leverage GPM. On average, 14 LOC changes were needed per application. Implementation of UPDATEs required most changes at 33 LOC, while prefix-sum required only 7 LOC.

6 EVALUATION

We implemented GPM on a system with Intel Xeon processors, Optane NVM and an NVIDIA GPU. Table 3 details the system.

6.1 Performance

We compare the performance of GPM with CAP-fs and CAP-mm – today's options for GPU applications to use PM's persistence. Recall both CAP systems rely on the CPU to persist results of GPU computation. While the former relies on the filesystem for persistence, the

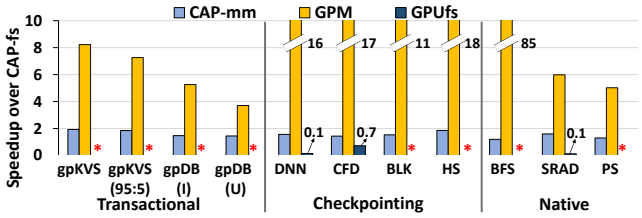


Figure 9: Speedup of CAP-mm, GPM, GPUfs normalized to CAP-fs. (*) indicates unsupported by GPUfs.

Table 4: Write-amplification (WA) in CAP over GPM.

	Transactional			Checkpointing	Native
	gpKVS	gpDB (I)	gpDB (U)	All workloads	All workloads
WA	39.38×	1.27×	19.88×	1.00×	1.00×

latter uses user-space cache flush and drain. We use PM-optimized ext4-DAX filesystem for CAP [75]. CAP-mm uses 2-32 CPU threads for persisting data. We choose the number of threads that provide the best performance for a given application. Later in the section, we also compare with a related academic work, GPUfs [87].

Figure 9 reports the speedups normalized to CAP-fs. There are up to three bars for each application: CAP-mm, GPM and GPUfs (* if failed to execute). Applications are divided into clusters by their type – transactional, long-running iterative kernels (checkpointing), and native persistence. We evaluate gpKVS with 100% SETs, a.k.a., gpKVS and with 95% GETs and 5% SETs, a.k.a., gpKVS (95:5). For gpDB, we show the numbers for INSERTs (gpDB (I)) and UPDATEs (gpDB (U)) separately as they exhibit distinct behaviors. We observe that CAP-mm improves performance over CAP-fs by 2× for gpKVS. This shows the benefits of avoiding OS overheads and that of parallelism in persisting data from the CPU.

Next, we focus on GPM. We observe that gpKVS speeds up by 7-8× on GPM over CAP-fs and by 4× over CAP-mm (Figure 9). GPM's ability to persist only the updated/new entries in the PM-resident KVS, due to its in-kernel selective persistence, is key to its performance. The indices in KVS that would be updated become known only upon computation in the kernel. These indices are often sparsely spread over tens of millions of KVS entries. On CAP, the entire KVS or its pre-defined large chunks have to be transferred to and be persisted from the CPU due to CAP's inability to directly persist updates from a GPU kernel at a byte granularity. Table 4 lists the amount of extraneous data persisted, *i.e.,* write-amplification (WA), by CAP over GPM. We observe that CAP persists 39× more data, explaining GPM's performance advantage for gpKVS. In gpKVS (95:5), GPM's advantages over CAP moderates as expected. GETs run in the same way on both CAP and GPM as they do not update data. GPM's features are useful when data is modified and needs to be persisted in a recoverable manner. However, GETs are mostly served out of the GPU's fast HBM while SETs involve persisting data to PM which dominates the runtime. Consequently, even with 5% SETs, GPM improves throughput.

Next, we observe that INSERTs and UPDATEs get faster by 3.6× and 2.6× over CAP-mm, respectively, on GPM. While speedup numbers for both types of queries are similar, the reasons are different. For INSERTs, the newly inserted rows are added at the end of the table. GPM utilizes GPU's parallelism to write millions of new rows and persist them in parallel from the kernel. In contrast, CAP designs are unable to leverage GPU's parallelism as it persists from the CPU. Further, initializing the DMA engine and transferring rows from GPU to CPU memory adds overheads.

For UPDATEs, similar to gpKVS, GPM's ability to selectively persist only the updated rows from the GPU helps it speed up over alternatives. As in gpKVS, the updated rows of the relational table can be distributed over millions of rows and their locations are known only upon computation. Thus, the ability to selectively persist data at byte granularities from the GPU helps avoid write-amplification. The same is evident from write-amplification reported in Table 4 – CAP-mm incurs ~ 20× more data transfer and persisting than GPM.

Beyond performance, GPM provides stronger recoverability due to fine-grained logging from GPU. On CAP, there was *no* logging and thus, it cannot guarantee recoverability if a crash happens during the process of persisting results to the PM.

Figure 9 next shows speedups for checkpointing long-running iterative workloads. Checkpointing speeds up on GPM by 11-18×. Here, the benefits stem from the GPU's massive parallelism in directly writing the checkpoint to PM. In contrast, software overheads of initiating DMA and writing the checkpoint to PM using few CPU threads throttles the performance on CAP. Note that both CAP and GPM persist an equal amount of data as the size of checkpoint does not change (write-amplification=1). Note that speedups in the total execution time that includes time to both compute and checkpoint intermediate results depends upon the chosen checkpointing frequency. For example, the DNN training speeds up by 61% and 40%, when we checkpointed weights and biases after every 10^{th} and 20^{th} pass, respectively. In the specific case of DNN training, it takes about 8.26 milliseconds to run 10 iterations. Whereas it takes 0.221 milliseconds to checkpoint and 0.342 milliseconds to restore from the checkpoint. This demonstrate how GPM can make checkpointing feasible even at a fine granualrity. In general, various workloads' total execution times improved by 19%-122% over different checkpointing frequencies.

Finally, native persistence workloads speed up by 5-85×. Most prominent is BFS. It is an iterative kernel (here, 6000 iterations) that persists costs of visited nodes and queues after every iteration. The overheads of initiating DMA at every iteration and persisting via CPU are significant. On GPM, however, the kernel performing the search directly writes and persists the results to PM in each iteration avoiding these overheads.

The coefficient matrix and the partial sums are persisted in SRAD and PS, respectively. The amount of data persisted remains the same under CAP and GPM. On GPM, GPU threads persist individual data items in parallel after individual computations. On CAP, however, data is transferred to the CPU in bulk after the entire computation (kernel boundary), for the CPU to persist it. Importantly, due to fine-grain persists, applications can *resume* computation where they left off after a crash on GPM. For example, if a crash occurs during computation of prefix sum, the entire computation needs to be restarted on CAP. On GPM, the kernel can resume computation only for sub-arrays that were yet to be persisted before the crash.

Comparison with GPUfs: We attempt to compare against GPUfs, which enables GPU kernels to invoke filesystem system calls. It relies on the CPU and OS to guarantee persistence. Most workloads

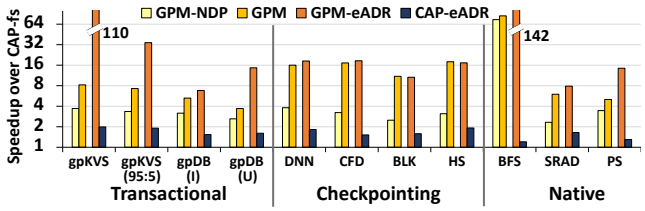
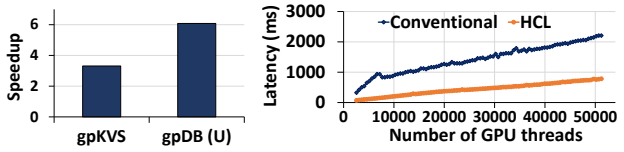


Figure 10: Understanding GPM's performance and eADR.



(a) Speedup due to HCL. **(b) Latency for microbenchmark.**

Figure 11: HCL's performance against conventional logging.

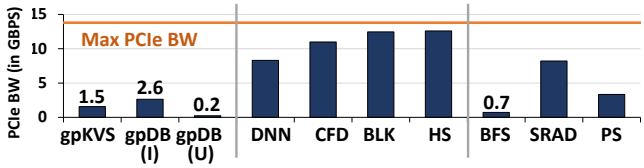


Figure 12: PCIe write bandwidth with GPM.

fail to run on GPUfs as it was never designed for fine-grain persistence. GPUfs requires *all* threads in a threadblock to invoke GPUfs's API – these calls are ordered by barriers. Applications deadlock if individual threads try to read/write data. Workloads that persist entire data structures at a coarse-grain, *e.g.,* checkpointing ones and SRAD, run on GPUfs. However, overheads of repeatedly invoking system calls from the GPU and filesystem slows down applications. As GPUfs only supports file sizes upto 2GB, BLK and HS fail.

Benefits over CPU-only persistence: One can perform both computation and leverage PM using only the CPU, as it is typically done today. However, as seen in Figure 1, GPM, aided by libGPM, can speed up applications by 3-27× over their CPU-only counterparts while ensuring the same recoverabilty guarantees. However, there is no meaningful 'CPU' counterparts to checkpointing workloads beyond CAP since data being checkpointed is generated on the GPU. For databases, their performance depends largely on the specific query engine. For a fair comparison, we converted the CUDA implementation of gpDB to OpenMP implementation that can leverage many core CPUs. We observed that GPM sped up gpDB (I) and gpDB (U) by 3.1× and 6.9×, respectively, while maintaining the same recoveribilty properties through write-ahead logging.

Analyzing GPM's performance and eADR: A key aspect of GPM is its ability to guarantee persistence from within a GPU kernel, beyond performing load/store to PM from the GPU. To tease out the importance of this aspect in GPM's performance, we created GPM-NDP. We force kernels to rely on the CPU to guarantee persistence as in CAP-mm (i.e., No Direct Persistence) but they can still directly load/store to PM as in GPM. We also keep DDIO enabled since it is only needed when guaranteeing persistence from the GPU.

Figure 10 shows speedup over CAP-fs, as before (note log scale in y-axis). We observe that GPM speeds up over GPM-NDP by up to 6×. This demonstrates that guaranteeing persistence from the GPU helps performance beside enabling fine-grain recoverabilty for GPU kernels. We observe most workloads, particularly checkpointing, slowed down significantly with GPM-NDP. Guaranteeing persistence for the entire checkpoint from the CPU adds significant serialization as CPU threads have to flush individual cache lines (64 bytes). The difference is slightly lesser for gpKVS (still 40 - 60%) since it uses more synchronizations in the kernel. That limits the amount of parallelism it can exploit in persisting updates.

We now investigate how eADR will impact both GPM and CAP. In GPM-eADR, we project GPM's performance with eADR by not disabling DDIO. GPM-eADR improves performance over GPM by up to 13× (Figure 10). However, improvements are not even across all applications. The applications with many ordering points due to frequent persists, *e.g.,* those due to logging, benefit the most since fences complete as soon as data reaches LLC. In contrast, checkpointing required a single persist after writing the entire checkpoint and, thus, is mostly agnostic to eADR.

We also created CAP-eADR, a version of CAP-mm but without cache flushes on the CPU as they are unnecessary with eADR. The difference between GPM-eADR and CAP-eADR shows the likely impact on GPM's relative advantage over CAP in future systems. Although eADR eliminates the need for flushing data from the CPU under CAP, it provides limited benefits to CAP as most of the time is spent in transferring data from GPU to CPU and not in persisting data. In Figure 10, we observe that GPM-eADR is 24× faster than CAP-eADR, on average. This is because GPM's advantages of leveraging GPU's parallelism in writing to PM and avoiding write-amplification are unaffected by eADR. At the same time, the fact that eADR allows GPM to guarantee persistence even in the presence of DDIO brings significant performance uplifts.

Importance of HCL in logging: Figure 11(a) shows speedup due to HCL over conventional distributed logging for transactional workloads. gpKVS speeds up by 3.3× with HCL over conventional logging. gpDB (U) that logs old values for every updated record witnesses a 6.1× speedup. We skip INSERTs since it only logs the table size. GPU's hardware coalescing of writes to logs and massively parallel log entry insertions propel performance under HCL. For gpKVS, one in eight threads inserts a key-value pair and, thus, logs. Thus, it does not exploit HCL's parallelism like gpDB (U) does. We further studied scalability of HCL with a microbenchmark. Figure 11(b) shows how HCL scales compared to conventional distributed logging. The x-axis and y-axis show the number of concurrent logging threads and the latency, respectively. HCL's logging latency remains stable with thread count but that for conventional logging jumps. On average, HCL lowers latency by ~ 3.6×.

Write bandwidth to PM: Only the data structures required for recovery reside on the PM (Table 1). The rest are placed on the GPU's volatile memory. Therefore, only accesses to the PM cross the PCIe. The writes to the PM captures the bandwidth utilized for data updates and recoverability (*e.g.,* logging).

Figure 12 shows the measured *write* bandwidth from GPU to PM on the PCIe. We find that the bandwidth utilization is well below the achievable total PCIe 3.0 bandwidth (~ 13 GBps) for transactional workloads. Upon analysis, we find that these workloads are

Table 5: Restoration latency (RL) in GPM.

	Transactional			Checkpointing			
	gpKVS	gpDB (I)	gpDB (U)	DNN	CFD	BLK	HS
RL	18.96%	0.01%	10.43%	0.12%	0.30%	0.80%	1.65%

bottlenecked at PM. Optane enables maximum bandwidth and lowest latency when accesses are sequential and aligned at 256-byte boundaries [27, 99, 103]. This is because it internally buffers writes at 256 bytes to hide latency [95, 99]. Using a microbenchmark, we observed that one can achieve 12.5 GBps bandwidth with sequential accesses aligned at 256 bytes. However, if the accesses are not 256-bytes-aligned then it drops to 3.13 GBps. Further, if accesses are to random addresses then bandwidth drops to 0.72 GBps. For gpDB (U), gpKVS, gpKVS (95:5), updates to the tables/KVS are sparse and unaligned that lead to low bandwidth at PM. That then show up as low PCIe bandwidth utilization. For gpDB (I), the accesses are unaligned but sequential as new rows are contiguous, and thus, enable better bandwidth.

The checkpointing workloads achieve better PCIe bandwidth usage because streaming writes to contiguous and aligned checkpoint memory fully leverages Optane's bandwidth. In BFS, writes to PM happen at random addresses based on the nodes being updated. In contrast, while persisting the co-efficient matrix in SRAD, writes to PM are streaming but not necessarily aligned. In short, we find that access patterns to PM and their interactions with idiosyncrasies of the Optane are key determinants to bandwidth utilization. Note that bandwidth presented is not the total bandwidth utilization but only for writes to the PM. For example, total bandwidth utilization to GPU's HBM for applications like BLK is ∼ 250GBps.

6.2 Recovery Analysis

We stress-tested recoverability for all workloads under GPM by injecting crashes at random points during kernel execution using NVIDIA's fault injection tool NVBitFI [28, 72]. It is a binary instrumentation tool that automatically injects fault at a given frequency. We successfully recovered the state of every program after crashes.

To measure recovery performance, we define restoration latency as the latency to execute the recovery kernel to recover after a crash. Table 5 shows the restoration latency as the percentage of a workload's operation time. The operation time includes kernel(s) execution time and time for recurring operations like loading data but excludes one-time setup cost. We skip native workloads since they do not have separate recovery kernels as their recovery logic is embedded in the application itself.

For transactional workloads, the restoration latency is the time to undo updates. We measure the worst-case cost latency by crashing just before transactions commit. The restoration latency is at most 19% of computation time and is often much lesser. gpDB (I) restores quickly since recovery only involves restoring table metadata. For checkpointing, the restoration latency is the time to copy data from the last checkpoint to the corresponding in-memory data structures. Table 5 shows that these workloads are quick to restore thanks to direct access to PM.

7 RELATED WORK

The advent of PM inspired significant research in exploring its use across many domains such as key-value stores [43, 47, 54, 55], databases [5, 16, 92], and filesystems [4, 42, 98]. Numerous interfaces have been proposed to enable programmers harness capabilities of PM. Mnemosyne [93] and NV-Heaps [19] explored techniques for transactional interactions with PM. Intel offers PMDK [35], a library to leverage Optane NVM. Prior works explored techniques to lower overheads of persistence, either through software [29, 31, 49, 50, 85, 97] or hardware [51, 58, 64]. While they provide useful insights, they are not directly applicable in the context of GPUs.

Logging is a key enabler of transactions. Prior works [11, 21, 24, 40, 94] have explored ways to reduce logging overheads. Checkpointing is often used for fault tolerance. Previous works for checkpointing on NVM [20, 23, 46] focused on minimizing the checkpointing latency and bandwidth. However, we explored GPUs and studies on CPUs do not directly apply.

Prior works proposed different ways and interfaces to directly transfer data between SSDs and GPU memory [8, 86, 90, 101]. However, none of these works concern themselves with byte-grained direct access and persistence to PM. A recent work [57] adapts existing CPU persistency models [76] for GPUs. They propose to completely replace GPU memory with NVM which seems impractical. DRAGON [60] exploits NVM's capacity (and not persistence) for GPU through Unified Memory [81]. Chen et al. [17] addresses the challenge of NVM's limited write bandwidth with new hardware. HeteroCheckpoint [45] leverages PM for checkpointing GPU programs and addresses issues of low NVM bandwidth and high persist latencies. However, it relies CPU for checkpointing. CheckFreq [63] proposes faster checkpointing of DNN state from the GPU. GPM goes much beyond checkpointing by enabling generic fine-grain in-kernel persistence. Researchers have also proposed adding new instruction in GPU hardware for supporting PM [7, 26]. In contrast, GPM needs no hardware modifications.

8 CONCLUSION

GPU is an important computing platform but is unable to harness the fine-grain persistence of PM today. We address this shortcoming by creating GPM that enables in-kernel fine-grain persistence for GPU-accelerated applications. Toward this, we explore three key challenges in bringing benefits of PM to GPU programs. We glue together a GPU and NVM to enable GPU kernels directly access PM. We then contribute in two ways to create an ecosystem around the systems. We create a workload suite of GPU programs that benefits from PM. We create a GPU library for easily leveraging PM by providing support for logging, checkpointing, and basic primitives.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Samira Khan for their constructive feedback. We thank Ashish Panwar, Vinod Ganapathy, R. Govindarajan, and Mainak Chaudhuri for their feedback on an earlier draft of this work. This work is supported by research grants from VMware Inc and Intel Labs. Arkaprava is supported by a Young Investigator Fellowship by Pratiksha Trust, Bangalore.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact provides source for the proposed system - GPM, which allows a GPU to leverage PM. The artifact consists of GPMbench and libGPM. GPMbench comprises of 9 GPU-accelerated workloads divided into three categories. libGPM provides a CUDA library that allows programmers to leverage PM from GPU. The artifact allows users to reproduce key results from the paper, including Figure 1, Figure 9 and Table 5. The hardware must contain both an NVIDIA GPU and an Intel Optane DCPMM attached to an Intel Xeon server to run GPM and reproduce the results. The measurement infrastructure is provided using Makefiles and bash scripts.

A.2 Artifact Check-list (Meta-information)

- **Compilation:** CUDA 11, GCC 9.3.0, CuDNN 8.2.
- **Binary:** Included for x86-64.
- **Data set:** Scripts are provided to download/generate datasets.
- **Run-time environment:** Workloads need CUDA 11, driver version >= 450 and CuDNN 8.2. The Optane PMEM should be configured in app-direct mode with all DIMMs interleaved. The scripts are written for Ubuntu 20.04 and need sudo privilige.
- **Hardware:** ① NVIDIA Turing GPU (preferably Titan RTX 72 SMs, 24 GB GDDR6) ② Intel Optane DCPMM (preferably 8 x 128 GB Intel Optane NVDIMM, with 2 DIMMs per socket) attached to ③ Intel Xeon server (preferably Xeon Gold 6242 (4×16 cores) @ 2.80GHz) ④ PCIe 3.0 x 16 interconnect between the CPU and the GPU
- **Execution:** CUDA kernels are executed on the GPU which directly access the PM. The execution framework is provided using Makefiles and bash scripts.
- **Output:** A tab separated file generated for each experiment.
- **How much disk space required (approximately)?:** GPMbench requires approx 50GBs of persistent memory.
- **How much time is needed to prepare workflow (approximately)?:** 30 mins
- **How much time is needed to complete experiments (approximately)?:** 6 hours
- **Publicly available?:** Yes. https://github.com/csl-iisc/GPM-ASPLOS22.git and https://doi.org/10.5281/zenodo.5847956
- **Archived (provide DOI)?:** Yes. 10.5281/zenodo.5847956

A.3 Description

The artifact contains the source for GPMbench and libGPM. It allows to reproduce results the following results:

- Figure 1: Benefits of GPM over CPU with PM. Both Figure 1a and Figure 1b.
- Figure 9: Speedup of CAP-mm, GPM and GPUfs over CAP-fs.
- Figure 10: Understanding GPM's performance and eADR.
- Figure 11a: Speedup due to HCL.
- Table 5: Restoration latency of GPM.

A.3.1 How to Access. The artifact is made available in the GitHub repository https://github.com/csl-iisc/GPM-ASPLOS22 and also at https://doi.org/10.5281/zenodo.5847956.

A.3.2 Hardware Dependencies. To support GPM, a system must have both GPU and NVM. We recommend the following hardware configuration: ① NVIDIA Turing GPU (preferably Titan RTX 72 SMs, 24 GB GDDR6) ② Intel Optane DCPMM (preferably 8×128 GB Intel Optane NVDIMM, with 2 DIMMs per socket) in app-direct mode and interleaved memory across all DIMMs attached to a ③ Intel Xeon server (preferably Xeon Gold 6242 (4×16 cores) @ 2.80GHz) ④ PCIe 3.0×16 interconnect between the CPU and the GPU

A.3.3 Software Dependencies. The artifact needs CUDA 11, CuDNN 8.2 and NVIDIA driver version >= 450. Follow the instructions in the README or from NVIDIA [73] to download and install CuDNN 8.2. The NVM setup requires the following libraries - NDCTL, DAXCTL [91], IPMCTL [78]. The libraries with all dependencies are maintained in dependencies.sh and can be installed as follows:

```
$ sudo ./dependencies.sh
```

The script also installs the dependencies needed for turning DDIO on and off, it is needed for persisting data from GPU.

To configure Optance PMM in app-direct mode, follow the steps mentioned below with sudo privilege.

```
$ cd pmem-setup/
$ #This step will tear down any older PMEM
config (e.g., memory-mapped).
$ sudo ./teardown.bashrc
$ #This step is going to interleave the DIMMs,
then reboot the machine.
$ sudo ./preboot.bashrc
$ #Once the system has rebooted run the
following commands as root to set Optane PMM
in app-direct mode.
$ sudo su
# sudo ./config.bashrc
```

All the experiments are performed assuming Ubuntu 20.04 and a GCC version of 9.3.

A.3.4 Data Sets. Scripts are provided to generate or download data sets.

A.4 Installation

The artifact can be downloaded and accessed as -

```
$ git clone https://github.com/csl-iisc/ \
GPM-ASPLOS22.git
$ cd GPM-ASPLOS22
```

A.5 Experiment Workflow

The outermost directory consists of two folders: Figure1 and GPM-Bench_LibGPM. Figure1 contains the run-time infrastructure for both Figure 1a and Figure 1b. GPMBench_LibGPM contains the source for both GPMbench and libGPM along with the run-time infrastructure for Figure 9, 10, 11a and Table 5.

The experiments must be run exclusively i.e., no compute or memory-intensive application should be running concurrently. We provide a Makefile that compiles, executes and generates a tab-separated report for the figures 1, 9, 10, 11a and table 5. Individual experiments can be run as:

```
$ make figure_1 #To run figure 1 (1a and 1b)
$ make figure_9 #To compile and run figure 9
```

```
$ make figure_10  #To compile and run figure 10
$ make figure_11a  #To compile and run figure 11a
$ make table_5  #To compile and run table 5
$ make all #To run all figures
```

The raw numbers can be obtained from the results folder present in the outermost directory.

A.6 Evaluation and Expected Results

For each key result, a tab separated file is generated. The reports/ folder contains all the generated reports. The reports can be matched against the figures reported in the paper. The generated reports are named: out_figure1a.txt, out_figure1b.txt for Figure 1, out_figure9.txt for Figure 9, out_figure10.txt for Figure 10, out_figure11a.txt for Figure 11 and out_table5.txt for Table 5. To obtain the reports, use the following command:

```
$ make out_figure_1
$ make out_figure_9
$ make out_figure_10
$ make out_figure_11a
$ make out_table_5
```

REFERENCES

[1] 2020. EMOGI: Efficient Memory-Access for out-of-Memory Graph-Traversal in GPUs. *Proc. VLDB Endow.* 2 (Oct. 2020), 114–127. https://doi.org/10.14778/3425879.3425883

[2] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast Key-Value Stores: An Idea Whose Time Has Come and Gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 113–119. https://doi.org/10.1145/3317550.3321434

[3] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 111–124. https://doi.org/10.1109/HPCA51647.2021.00019

[4] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. 2019. Assise: Performance and Availability via NVM Colocation in a Distributed File System. arXiv:1910.05106 [cs.DC]

[5] Joy Arulraj and Andrew Pavlo. 2017. How to Build a Non-Volatile Memory Database Management System. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1753–1758. https://doi.org/10.1145/3035918.3054780

[6] Peter Bakkum and Srimat Chakradhar. 2012. Efficient Data Management for GPU Databases. http://pbbakkum.com/virginian/paper.pdf.

[7] Arkaprava Basu, Dibakar Gope, Sooraj Puthoor, and Mitesh Meswani. 2019. Scoped persistence barriers for non-volatile memories. https://patents.google.com/patent/US10324650B2/en.

[8] Stephen Bates. November 2016. Project Donard. https://github.com/sbates130272/donard.

[9] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon) *(SC '09)*. Association for Computing Machinery, New York, NY, USA, Article 21, 12 pages. https://doi.org/10.1145/1654059.1654081

[10] 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. https://www.usenix.org/conference/atc17/technical-sessions/presentation/bergman

[11] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Scaling a File System to Many Cores Using an Operation Log. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 69–86. https://doi.org/10.1145/3132747.3132779

[12] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2019. CoNDA: Efficient Cache Coherence Support for near-Data Accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 629–642. https://doi.org/10.1145/3307650.3322266

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

[14] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 79–92. https://doi.org/10.1145/3445814.3446713

[15] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[16] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *CIDR'11: 5th Biennial Conference on Innovative Data Systems Research* (cidr'11: 5th biennial conference on innovative data systems research ed.). https://www.microsoft.com/en-us/research/publication/rethinking-database-algorithms-for-phase-change-memory/

[17] Sui Chen, Faen Zhang, Lei Liu, and Lu Peng. 2019. Efficient GPU NVRAM Persistence with Helper Warps. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 155, 6 pages. https://doi.org/10.1145/3316781.3317810

[18] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 228–243. https://doi.org/10.1145/2517349.2522726

[19] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[20] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. 2011. Hybrid Checkpointing Using Emerging Nonvolatile Memories for Future Exascale Systems. *ACM Trans. Archit. Code Optim.* 8, 2, Article 6 (June 2011), 29 pages. https://doi.org/10.1145/1970386.1970387

[21] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. 2011. High Performance Database Logging Using Storage Class Memory. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, USA, 1221–1231. https://doi.org/10.1109/ICDE.2011.5767918

[22] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2020. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 673–689. https://www.usenix.org/conference/atc20/presentation/farshin

[23] Shen Gao, Bingsheng He, and Jianliang Xu. 2015. Real-Time In-Memory Checkpointing for Future Hybrid Memory Systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 263–272. https://doi.org/10.1145/2751205.2751212

[24] Shen Gao, Jianliang Xu, Theo Härder, Bingsheng He, Byron Choi, and Haibo Hu. 2015. PCMLogging: Optimizing Transaction Logging and Recovery Performance with PCM. *IEEE Transactions on Knowledge and Data Engineering* 27, 12 (2015), 3332–3346.

[25] Juan Gómez-Luna, Izzat El Hajj, Victor Chang, Li-Wen Garcia-Flores, Simon Garcia de Gonzalo, Thomas Jablin, Antonio J Pena, and Wen-mei Hwu. 2017. Chai: Collaborative Heterogeneous Applications for Integrated-architectures. In *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*. IEEE.

[26] Dibakar Gope, Arkaprava Basu, Sooraj Puthoor, and Mitesh Meswani. 2018. A Case for Scoped Persist Barriers in GPUs. (2018).

[27] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 626–639. https://doi.org/10.14778/3436905.3436921

[28] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. 2017. SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 249–258. https://doi.org/10.1109/ISPASS.2017.7975296

[29] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 775–788. https://doi.org/10.1145/3373376.3378472

[30] Mark Harris. 2013. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. *NVIDIA Developer Blog* (Jan 2013). https://tinyurl.com/global-mem-cuda.

[31] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-Threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 468–482. https://doi.org/10.1145/3064176.3064204

[32] Chunyang Hui. 2019. Enabling Persistent Memory in the Storage Performance Development Kit (SPDK). (2019). https://tinyurl.com/intel-spdk.

[33] Intel. 2012. Intel Data Direct I/O Technology. https://www.intel.in/content/www/in/en/io/data-direct-i-o-technology.html.

[34] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. (September 2016).

[35] Intel. 2017. Persistent Memory Development Kit. https://pmem.io/pmdk/.

[36] Intel. 2021. eADR: New Opportunities for Persistent Memory Applications. (2021). https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html.

[37] Intel. 2021. Intel Optane Persistent Memory. https://www.intel.in/content/www/in/en/architecture-and-technology/optane-dc-persistent-memory.html. Accessed: 2021-05-05.

[38] Intel. 2021. Intel PmemKV. https://github.com/pmem/pmemkv.

[39] Intel. September 2021. Persistent Memory Learn More Series Part 2. https://www.intel.com/content/www/us/en/developer/articles/training/pmem-learn-more-series-part-2.html.

[40] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 427–442. https://doi.org/10.1145/2980024.2872410

[41] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. http://arxiv.org/abs/1903.05714.

[42] 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* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 494–508. https://doi.org/10.1145/3341301.3359631

[43] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-Ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (Boston, MA, USA) *(FAST'19)*. USENIX Association, USA, 191–204.

[44] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and Solutions for Fast Remote Persistent Memory Access. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 105–119. https://doi.org/10.1145/3419111.3421294

[45] Sudarsun Kannan, Naila Farooqui, Ada Gavrilovska, and Karsten Schwan. 2014. HeteroCheckpoint: Efficient Checkpointing for Accelerator-Based Systems. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. IEEE Computer Society, USA, 738–743. https://doi.org/10.1109/DSN.2014.76

[46] Sudarsun Kannan, Ada Gavrilovska, Karsten Schwan, and Dejan Milojicic. 2013. Optimizing Checkpoints Using NVM as Virtual Memory. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. IEEE Computer Society, USA, 29–40. https://doi.org/10.1109/IPDPS.2013.69

[47] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 821–837. https://www.usenix.org/conference/atc21/presentation/kassa

[48] Craig Kolb and Matt Pharr. 2005. Binomial Option Pricing Model. https://developer.nvidia.com/gpugems/gpugems2/part-vi-simulation-and-numerical-algorithms/chapter-45-options-pricing-gpu. Accessed: 2021-05-06.

[49] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-Level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

[50] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[51] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated Persist Ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 58, 13 pages.

[52] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[53] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. (2010). http://yann.lecun.com/exdb/mnist/

[54] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[55] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 447–461. https://doi.org/10.1145/3341301.3359628

[56] J. Li, Hung-Wei Tseng, Chunbin Lin, Y. Papakonstantinou, and S. Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proc. VLDB Endow.* 9 (2016), 1647–1658.

[57] Zhen Lin, Mohammad Alshboul, Yan Solihin, and Huiyang Zhou. 2019. Exploring Memory Persistency Models for GPUs. In *In Proc of International Conference on Parallel Architectures and Compilation Techniques (PACT-28)*.

[58] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. 2019. Janus: Optimizing Memory and Storage Support for Non-Volatile Memory Systems. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/3307650.3322206

[59] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 257–270. https://doi.org/10.1145/3297858.3304043

[60] 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, Article 32, 13 pages.

[61] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen mei Hwu. 2021. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. arXiv:2103.03330 [cs.LG]

[62] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992), 94–162. https://doi.org/10.1145/128765.128770

[63] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 203–216. https://www.usenix.org/conference/fast21/presentation/mohan

[64] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. wift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xiaposan, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[65] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Analysis and Exploitation of Dynamic Pricing in the Public Cloud for ML Training. (2020).

[66] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1047–1064. https://www.usenix.org/conference/osdi20/presentation/neal

[67] NVIDIA. 2011. Peer-to-Peer & Unified Virtual Addressing. https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf.

[68] NVIDIA. 2012. How to Optimize Data Transfers in CUDA C/C++. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

[69] NVIDIA. 2019. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2019-11-15.

[70] NVIDIA. 2019. CUDA Software Development Kit Samples. https://docs.nvidia.com/cuda/cuda-samples/index.html.

[71] NVIDIA. 2021. NVIDIA cuDNN. (2021). https://developer.nvidia.com/cudnn.

[72] NVIDIA. April 2020. NVBitFI. https://github.com/NVlabs/nvbitfi.

[73] Nvidia. November 2021. cuDNN Archive. https://developer.nvidia.com/rdp/cudnn-archive.

[74] OmniSci. 2017. OmniSci DB. https://www.omnisci.com/.

[75] Linux Kernel Organization. 2021. DAX:Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[76] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 265–276.

[77] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System Evaluation of the Intel Optane Byte-Addressable NVM. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 304–315. https://doi.org/10.1145/3357526.3357568

[78] PMEM.io. August 2021. IPMCTL User Guide. https://docs.pmem.io/ipmctl-user-guide/.

[79] RocksDB. 2021. RocksDB. (2021). https://rocksdb.org/.

[80] Andy Rudoff. 2021. Persistent Memory on CXL. https://www.snia.org/educational-library/persistent-memory-cxl-2021.

[81] Nikolay Sakharnykh. 2016. Beyond GPU Memory Limits with Unified Memory on Pascal. *NVIDIA Developer Blog* (December 2016). https://tinyurl.com/pascal-um.

[82] Steve Scargall. 2020. *libpmem: Low-Level Persistent Memory Support.* Apress, Berkeley, CA, 73–79. https://doi.org/10.1007/978-1-4842-4932-1_6

[83] Tim C. Schroeder. 2011. Peer-to-Peer & Unified Virtual Addressing. *CUDA Webinar* (2011). https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf.

[84] Shobhit Seth. 2006. Understanding the Binomial Option Pricing Model. https://www.investopedia.com/articles/investing/021215/examples-understand-binomial-option-pricing-model.asp. Accessed: 2021-05-06.

[85] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 323–337. https://doi.org/10.1145/3127479.3128610

[86] Mustafa Shihab, Karl Taht, and Myoungsoo Jung. 2014. GPUdrive: Reconsidering Storage Accesses for GPU Acceleration. (2014).

[87] 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. https://doi.org/10.1145/2451116.2451169

[88] Tal Ben-Nun. 2017. CuDNN Training. https://github.com/tbennun/cudnn-training.

[89] Adam Thompson and CJ Newburn. 2019. GPUDirect Storage: A Direct Path Between Storage and GPU Memory. *NVIDIA Developer Blog* (August 2019). https://developer.nvidia.com/blog/gpudirect-storage/.

[90] 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. https://doi.org/10.1109/ISCA.2016.15

[91] Usenix. August 2021. NDCTL User Guide. https://docs.pmem.io/ndctl-user-guide/.

[92] Stratis D. Viglas. 2014. Write-Limited Sorts and Joins for Persistent Memory. *Proc. VLDB Endow.* 7, 5 (Jan. 2014), 413–424. https://doi.org/10.14778/2732269.2732277

[93] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[94] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *Proc. VLDB Endow.* 7, 10 (June 2014), 865–876. https://doi.org/10.14778/2732951.2732960

[95] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 496–508. https://doi.org/10.1109/MICRO50266.2020.00049

[96] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 523–536. https://www.usenix.org/conference/atc21/presentation/wei

[97] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *Proceedings of Programming Language Design and Implementation*.

[98] Jian Xu and Steven Swanson. 2016. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'16)*. USENIX Association, USA, 323–338.

[99] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[100] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[101] 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 *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, USA, 13–24. https://doi.org/10.1109/PACT.2015.43

[102] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A Case for GPUs to Maximize the Throughput of in-Memory Key-Value Stores. *Proc. VLDB Endow.* 8, 11 (July 2015), 1226–1237. https://doi.org/10.14778/2809974.2809984

[103] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A Key-Value Store for Optane Persistent Memory. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 194–209. https://doi.org/10.1145/3447786.3456237