Scoped Buffered Persistency Model for GPUs

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

Aditya K Kamath†*
akkamath@uw.edu
University of Washington
Seattle, USA

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

ABSTRACT

While the implications of persistent memory (PM) on CPU hardware and software are well-explored, the same is not true for GPUs (Graphics Processing Units). A recent work, GPM, demonstrated how GPU programs can benefit from the fine-grain persistence of PM. However, in the absence of a persistency model, one cannot reason about the correctness of PM-aware GPU programs. Persistency models define the order in which writes to PM are persisted. We explore persistency models for GPUs.

We explore persistency models for GPUs. We demonstrate that CPU persistency models fall short for GPUs. We qualitatively and quantitatively argue that GPU persistency models should support scopes and buffering of writes to PM to leverage parallelism while adapting to higher NVM latencies. We formally specify a GPU persistency model that supports both scopes and buffers. We detail how GPU architecture can efficiently realize such a model. Finally, we quantitatively demonstrate the usefulness of scopes and buffers for PM-aware GPU programs.

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. 2023. Scoped Buffered Persistency Model for GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3575749

1 INTRODUCTION

Non-volatile memory (NVM) technologies such as Intel's Optane DC memory [24] and upcoming CXL-based NVMs [57] blur the long-held distinction between memory and storage by enabling persistent memory (PM). We refer to NVM accessed via loads and stores at byte granularity as PM. Justifiably, there has been significant research on CPU hardware and software for PM over the last decade [7, 29, 31, 36, 37, 45, 54].

In our previous work, GPM [52], we demonstrated several classes of applications that can benefit from both GPU's parallelism and PM's fine-grain persistence. Consider a persistent key-value store (pKVS), such as Meta's RocksDB-pmem, that forms the backbone of many reliable web services by enabling recoverability [56]. Prior works explored how PM can speed up pKVS [25, 38, 56, 62]. Researchers have also demonstrated that GPUs can improve the throughput of KVS [63]. GPM puts both together to create GPU-accelerated pKVS (gpKVS) by enabling fine-grain persistence to GPU kernels [52]. Consequently, the throughput improves by ~ 3× over state-of-art CPU-only pKVS [52]. Other important use cases include GPU-accelerated persistent databases and GPU-accelerated processing of PM-resident graphs. Further, long-running GPU kernels, such as DNN training, that checkpoints partial results for recoverability and early termination also benefit from PM's fast persistence.

While GPM demonstrated benefits of PM for GPU programs, in the absence of a specified *persistency model*, programmers *cannot* reason whether a GPU program can correctly recover from a crash by ensuring consistency of PM-resident data structures (i.e., recoverability). Writes to PM-resident data structures can be cached in volatile caches. These writes can then drain to the PM and, thus, become durable in an arbitrary order. A crash or a power failure can then leave PM-resident data structures in an inconsistent state.

Ensuring recoverability of data structures is a fundamental requirement of any PM-aware program – be it for a CPU or a GPU. Pelley et al. [54] were the first to show the need for specifying a persistency model for programmers to reason about the recoverability of PM-aware CPU programs. Persistency models define the ordering in which writes to PM (*persists*) should become persistent (durable). We refer to this order as persist memory order (PMO). A model determines how programmers can express PMOs, both within a thread (intra-thread PMO) and across threads (inter-thread PMO), as needed for the recoverability of a program. Researchers have explored various CPU persistency models [10, 13, 29, 36, 37, 45, 54, 55].

We explore persistency models for GPUs in this work. An obvious question is whether CPU persistency models are good enough for GPUs. We, thus, first analyze if CPU models suit GPU architecture and the needs of GPU applications.

Unfortunately, CPU persistency models fall short for GPUs due to multiple reasons. CPU models allow programmers to express inter-thread PMOs that affect *all* threads of a program – *global* in nature. However, global ordering is ill-suited for hundreds of thousands of concurrent GPU threads. Further, GPU's hierarchical programming model requires one to arrange threads in groups.

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

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575749

Thus, global ordering across different groups is often unnecessary and can limit parallelism in GPU programs.

We observe that GPU memory consistency models address a similar challenge in scaling synchronizations with *scopes*[22, 43, 49]. A scope qualifier allows a programmer to specify the subset of threads that should participate in synchronization, eschewing global visibility. We posit that GPU persistency models should similarly support scopes to specify and enforce inter-thread PMO.

Next, we observe that buffering of persists is crucial for GPUs to hide their latency. Efficient buffering can help bandwidth-hungry GPU kernels to adapt to NVM's limited bandwidth [61]. However, straightforward extensions of buffered CPU persistency models [29, 37, 45] are impractical for GPUs. In a CPU, persists are typically held at per-core buffers at L1 caches. Inter-thread PMO among buffered persists are inferred using cache coherence messages. However, in GPUs, the hardware is not responsible for keeping its L1 caches coherent [49]. It is impractical to support coherence for thousands of concurrent loads/stores in a GPU. Consequently, buffering of persists that rely on coherence messages is inapplicable to GPUs.

We draw inspiration from GPU memory consistency models in addressing the challenge. We note that GPU consistency models require programmers to *explicitly* demarcate the dependencies across threads through acquire/release patterns [22, 43, 49]. We posit that GPU persistency models should also allow acquire/release semantics to declare inter-thread PMO.

Driven by these observations, we propose and formally define a GPU persistency model named Scoped Buffered Release Persistency (SBRP). It supports scopes to express inter-thread PMO using acquire and release patterns. SBRP also enables GPU-optimized buffering of persists for hiding latency.

SBRP decouples intra-thread PMO from inter-thread PMO. It introduces an oFence operation for intra-thread PMO. The oFence ensures that persists before the fence are made durable before any later persists from the issuing thread.

SBRP introduces two *scoped* operations – persist acquire (pAcq) and release (pRel) – for expressing inter-thread PMO. If a thread T_0 needs its persists to be ordered before persists from another thread T_n, then T_0 should execute pRel after its persists, while T_n should issue pAcq before its persists. The programmer should choose the narrowest scope (i.e., the smallest subset of threads) that encompasses both T_0 and T_n for good performance. We formally specify SBRP in Section 4. Further, SBRP introduces a durability fence (dFence) [45] that guarantees immediate durability of persists from the issuing thread. A dFence is not necessary to reason about recoverability but enables greater control over the durability of partial results. Note, all persist operations only affect writes to PM.

A key implementation goal of SBRP is to enforce ordering amongst the minimum number of persists from different threads as required by oFence, pAcq, and pRel operations. This is a challenge with thousands of threads, as many persists that can be in flight at any time. We carefully design buffers to track persists at the granularity of individual warps, along with three associated hardware masks to maximize parallelism among the persist operations.

While GPM did not specify its persistency model, we find that it implicitly follows a straightforward extension of the CPU *epoch* persistency model [8, 54]. GPM uses a system-scoped fence (__threadfence_sys) that acts as a persist barrier (global) for both intra- and inter-thread PMO. Barriers divide the execution into epochs where persists within an epoch can be freely reordered but not those from different epochs. In short, GPM's design fails to embody the desirable qualities of a GPU persistency model. We evaluate SBRP against GPM to quantify the benefits.

To demonstrate broader applicability, we evaluate persistency models on two possible system designs – PM-far and PM-near. In the former, the NVM is attached to the CPU and is accessed by the GPU over PCIe, as in GPM [52]. In PM-near, we project a future hardware design where the NVM is placed onboard the GPU, avoiding the need to cross PCIe for accessing NVM.

We quantitatively demonstrate the benefits of SBRP with six PM-aware GPU applications. We show that applications speed up by up to 3.45x with SBRP over GPM, thanks to its ability to express fine-grain inter-thread PMO using scopes and an innovative persist buffer design.

In summary, we make the following contributions.

- We demonstrate how CPU-centric persistency models fall short for GPUs and make a case for scopes and buffering for GPU persistency models.
- We specify a GPU persistency model, SBRP, that supports scopes and buffers and leverages acquire/release.
- We detail a GPU architecture to support such a model and quantitatively show its benefits.

2 BACKGROUND

Persistent memory: Intel's Optane Persistent Memory [24] is the first commercially available NVM technology that enabled PM. Optane's access latencies are 3-10× of DRAM [28]. A key challenge for applications using PM is ensuring recoverability, i.e., maintaining consistency of PM-resident data structures in the face of a crash [6]. Persistency alone does not guarantee recoverability. Updates to PM can be cached in a processor's volatile caches. Thus, the order in which data is written to the cache may differ from the order it reaches the PM. For example, a failure during insertion into a doubly-linked list could lead to dangling pointers if the pointer update reaches PM before the data. We will soon discuss how a well-specified persistency model can help programmers ensure such scenarios are avoided.

CPUs that support PM enable a feature called Asynchronous DRAM Refresh (ADR) to hide the higher write latency of NVM. Under ADR, memory controllers buffer writes to NVM in a capacitor-backed write pending queue (WPQ). On a power failure, WPQ's contents are guaranteed to reach NVM. Thus, persistence is guaranteed as soon as the memory controller accepts a write to PM.

Recently, Intel announced that it would discontinue Optane PM [19]. The announcement, likely driven by economics, while disappointing, is unlikely to be the end of persistent memory. Samsung has started shipping memory-semantic SSDs that can effectively act as persistent memory [58]. The memory-semantic SSD consists of an SSD along with a large DRAM cache that caches write to SSD for high throughput random reads/writes like memory while a proprietary controller pushes the updates to the SSD. It shows that the behavior of persistent memory can be conjured with different memory technologies and techniques. Samsung expects this new product to be valuable for emerging AI/ML applications. The CXL

(Compute Express Link) 2.0 specification incorporated a *global persistent flush* operation to support manipulation of PM-resident data structures on future CXL-attached PM devices [9]. In summary, while Intel's Optane is the first commercially available PM, it is unlikely to be the last.

Memory persistency models: For correct recoverability of PM-resident data structures across crashes or power cycles, programmers must dictate the order in which writes to PM become durable (persist) as per the program's semantics. A persistency model defines the permitted set of orderings of persists [54]. It also defines how programmers can express these orderings. Persistency models are thus a necessity to argue about the correctness of a recoverable program – be it on a CPU or a GPU. We term the order of persists enforced by a persistency model as the persist memory order (PMO). This is analogous to the volatile memory order (VMO) enforced by memory consistency models that define the order in which loads/stores become visible to other threads.

Many CPU persistency models have been defined with varied performance and programmability tradeoffs [13, 29, 36, 37, 45, 54]. These are broadly classified along two axes – (1) strict versus relaxed and (2) unbuffered versus buffered. In strict models, PMO follows VMO, whereas they may disagree with relaxed models [54]. The epoch persistency model is an example of relaxed model. Here, the epochs are demarcated by epoch barriers. Persists prior to a barrier are ordered before those following the barrier; writes from within an epoch may persist in any order. However, persists from different epochs are ordered. In an unbuffered model, PMOs are enforced immediately by waiting for earlier persists to complete. Buffered models allow persists to be held in volatile buffers and later drained following the PMO. This helps in hiding the latency of persists.

GPU hierarchy: GPU's hardware resources are organized in a hierarchy to scale to thousands of concurrent threads. The basic execution block of a GPU is a Streaming Multiprocessor (SM). Modern GPUs contain more than a hundred SMs. SMs contain multiple SIMD units, each with several lanes. All lanes of a SIMD unit execute the same instruction on different data items concurrently. The SIMD units of an SM share an L1 cache. L1 caches are private to each SM, while all SMs share an L2 cache. The GPU's global memory is accessible to all threads and includes the GPU's onboard HBM/GDDR. L1 and L2 cache global memory contents.

GPU programming languages, e.g., CUDA, require programmers to arrange threads in a hierarchy that mimics the hardware. The smallest execution entity is a thread. In CUDA, 32 threads make up a warp, the smallest hardware scheduled unit of work. Threads in a warp typically execute in a lock-step fashion. A threadblock is a collection of warps guaranteed to execute in the same SM, while the grid is the largest unit of execution, comprising multiple threadblocks that execute a GPU kernel together.

Scoped synchronization: Global synchronization across thousands of GPU threads is slow. It is also often unnecessary in a GPU's hierarchical programming paradigm. GPUs, thus, provide means to synchronize amongst only the threads at a given level of the hierarchy, i.e., *scope*. CUDA provides three scopes – block, device, and system. The effect of synchronization is guaranteed only for the threads in its scope. For example, a fence with the block scope guarantees visibility to threads in the threadblock of the calling thread. While the device scope affects all threads in a GPU, the system scope is necessary to synchronize across multiple GPUs or the CPU. Inter-thread communication in GPU is accomplished via acquire/release patterns. A release makes earlier writes from a thread visible to others. An acquire makes earlier writes from other threads visible to the calling thread. In the absence of explicit acquire/release operations, such as in CUDA, fences with load/stores can be used to create acquire/release patterns [43, 50]. A load operation followed by a fence creates an acquire, while a fence followed by a store creates a release. The scope of an acquire/release pattern is the narrowest scope of its constituent instructions.

3 SYSTEM DESIGNS FOR GPU AND NVM

We consider two system designs to ensure the generality of the analysis. ① NVM is attached to the CPU (host) alongside DRAM and is accessed across the PCIe interconnect by the GPU (Figure 1a). We call it PM-far. ② NVM is placed onboard the GPU alongside GDDR/HBM (Figure 1b). We call it PM-near.

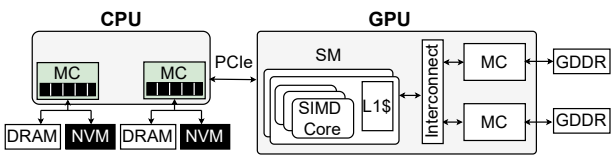
PM-far mimics the system proposed in GPM [52]. The work demonstrates that PM-far is realizable with commercial hardware. However, the PCIe latency (~ 300ns) is in the critical path of accesses to PM from the GPU. Also, the PCIe bandwidth (~ 28 GBps with PCIe 4.0) can eclipse the PM bandwidth.

Alternatively, future GPUs can place NVM onboard the GPU, i.e., PM-near. In PM-near, both DRAM and Optane NVM are connected to memory controllers, as in today's Xeon CPUs. We expect an ADR-enabled WPQ to guarantee persistence when data reaches the memory controller (MC) as in today's CPUs. The persistent domain (marked in black) comprises the MC and NVM.

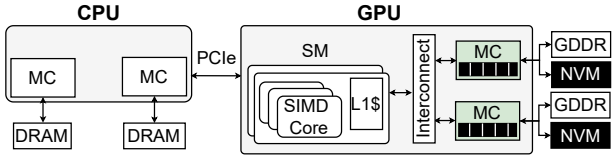
The write bandwidth of the current Optane DIMM is ~ $\frac{1}{8}^{\text{th}}$ that of DRAM [28, 61]. Extrapolating from GPUs, we expect the write bandwidth of NVM on GPU to be $\frac{1}{8}^{\text{th}}$ of GDDR. GDDR uses similar technology as DDR but with many more channels, buffers, etc., to achieve higher bandwidth. We posit that similar scaling is possible for the NVM onboard the GPU. However, note that our work on persistency models is not contingent on NVM's bandwidth. In Figure 10, we quantitatively demonstrate that our proposed persistency model retains its value even when the NVM bandwidth is scaled up or down.

Software model: Both the NVM and volatile memory are accessible via loads and stores at byte granularity from GPU in the system designs we consider. Applications choose where to place data as per their needs. This is akin to Intel's app direct mode for Optane [23] where both types of memory are part of the physical address space.

We enable APIs to allocate (de-allocate) memory on PM and map it onto GPU's virtual address space using Unified Virtual Address (UVA) [46] as in GPM [52]. These mappings should survive across power-cycles for recoverability. On PM-far, we follow GPM whereby memory is allocated out of files on PM and are mapped to GPU's virtual address space. In PM-near, we avoid higher overheads of files. Instead, we maintain a (persistent) namespace table, mapping the names (address) of allocated contiguous memory regions to respective physical addresses. The table tracks the sizes of allocated regions along with the names. A name is used to access persistently stored data after a crash. Upon recovery, previously allocated data structures are re-mapped using an open routine that takes a name as a parameter. The GPU driver manages this metadata.



(a) PM-far: GPU accesses NVM across the PCIe.



(b) PM-near: GPU accesses on-board NVM.

Figure 1: System designs with GPU and NVM. Memory controllers are ADR enabled.

Programmers must order the writes reaching the NVM (persists) for recoverability. The persistency models allows programmers to reason about the orderings that need to explicitly enforced for correctness and how they can be expressed. Much of the discussion for the rest of the paper will revolve around this topic.

4 DESIGNING A GPU PERSISTENCY MODEL

We explore GPU persistency models that will enable programmers to reason about the correctness of recoverable GPU kernels. We first discuss how CPU persistency models fall short for GPUs and detail the desirable qualities for GPU models.

A persistency model should enable programmers to order persists *within* a thread. For example, a GPU thread in gpKVS may log a key-value pair before inserting a new one in its place for recoverability. The log entry must persist before the new pair. GPU programmers may specify intra-thread persist memory order (PMO) using fences, like CPU models [45]. Persists before the fence should be made durable before those appearing after.

However, *inter-thread* PMO in GPUs is challenging. CPU models are designed for a few threads, and thus, they globally order persists across threads [37]. However, it is ill-suited for the hundreds of thousands of GPU threads and its execution hierarchy. A global ordering of persists across many threads inevitably leads to poor performance and is often unnecessary given the GPU's programming hierarchy. However, if a model only allows PMO within a threadblock, programmers will fail to order persists across threadblocks when needed by program semantics.

Consider a reduction kernel that is repeatedly invoked by applications like MapReduce [20] to find the sum of an input array. The array is divided among threadblocks, where each finds the sum of its subarray. In the last iteration, one threadblock computes the final sum from all threadblocks' partial sums. Figure 2 shows the operations performed by a threadblock. Each thread (T_i) is responsible for a single element (i) in the subarray. The threads of a threadblock add the value of an element in the second half of the array to those in the first half. The problem then reduces to finding the sum of elements in the first half and repeats itself until a single sum is obtained.

The input array resides on the GDDR, while the partial sums and output arrays are on the NVM. Since reduction may operate on a large dataset, persisting the output allows for recovery. While persisting the partial sums allows computation to resume instead of restarting after a crash. The values obtained in each iteration depend on the sums from the previous iterations of the same threadblock. Ordering these persists allows a program to resume from the last unfinished iteration. A programmer needs to order persists from different threads of the same threadblock (inter-thread PMO) but

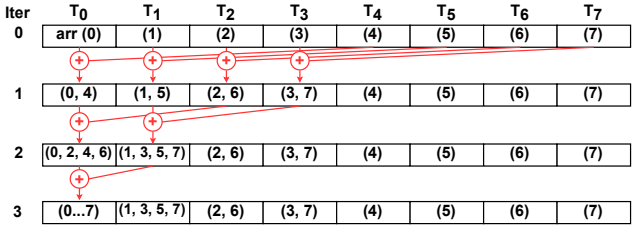


Figure 2: Reduction operation

persists from *different* threadblocks can proceed concurrently. In the last iteration, however, ordering across *all* threadblocks is needed. In short, for all iterations but the last, programmers should be able to ordered persists only within each threadblock blocks but for the last iteration ordering across threadblocks is needed. A GPU persistency model should, thus, allow the flexibility of choosing the subset of threads (here, within or across threadblocks) that should observe a given inter-thread PMO.

Next, we posit that buffering of persists is necessary for bandwidth-sensitive GPU kernels to hide the long latency of persists. However, traditional CPU persist buffers do not scale well to GPUs. For example, naively sharing a single buffer in an SM by up to 1024 threads of a threadblock results in artificial serialization among persists, e.g., those due to *intra*-thread PMO. Alternatively, a per-thread buffer would incur massive hardware overheads due to hundreds of thousands of concurrent GPU threads. It would also make it impossible to detect and enforce *inter*-thread PMO efficiently. In summary, straightforward extensions of buffers proposed in CPU persistency models are ill-suited for GPUs.

This brings the next challenge of designing a GPU persistency model: how to detect inter-thread PMO? CPU models typically snoop on cache coherence messages to infer inter-thread ordering. However, GPU's L1 caches are not kept coherent by the hardware. Thus, CPU persistency models cannot be extended to detect inter-thread PMO for GPUs.

In summary, CPU persistency models are not suitable for GPUs because: ① Global communication of persists across threads, does not scale well for GPUs. It is expensive and often unnecessary. ② Naive extensions of CPU persist buffers are impractical for GPUs. ③ In the absence of coherent L1 caches, GPU models cannot easily communicate inter-thread PMO.

We take cues from GPU's scoped synchronization operations to address these challenges. We posit that GPU persistency models should enable programmers to order persists within a chosen subset of threads in its execution hierarchy. Programmers should be able to express intra-thread, intra-, and inter-threadblock PMO. Such

a scope-aware model could allow one to write both correctly, recoverable and efficient kernels. Since scopes are already ubiquitous in GPU consistency models, a scope-aware persistency model will also be programmer-friendly.

We find that persist buffers are also necessary for realizing scoped persistency models. This is because the NVM resides after the globally shared L2 cache. SMs do not have private NVMs. Without buffers, threads would end up writing to the NVM even for intra -thread or -threadblock PMO. Instead, volatile buffers in SMs could independently track writes to PM and their PMO. This buffer could be appropriately drained to PM at a later point to enforce PMO. Hence, buffering persists at the L1 cache is needed for a scope-aware persistency model. In Section 6, we detail our GPU-optimized buffer design where PMOs are tracked per warp instead of per thread or per threadblock.

Finally, to detect inter-thread PMO in GPUs, we again take a cue from GPU consistency models that rely on acquire/release patterns for inter-thread ordering. We posit that a GPU persistency model should also rely on acquire/release semantics for inter-thread (intra- and inter-threadblock) PMO.

GPM's persistency model: Although GPM [52] did not specify its persistency model, it follows a scope-agnostic, unbuffered epoch persistency model. It employs system-scope fences to order persists that acts as an epoch barrier. The model is scope-agnostic; all PMOs are communicated globally as persists are flushed to order them. It is unbuffered since the fence stalls execution by waiting for persists to become durable. In short, GPM's model lacks the desirable properties of a GPU persistency model.

Summary: Specifying a GPU persistency model is a prerequisite to reason about the recoverability of kernels. We argue that a scoped and buffered persistency model can enable GPU programmers to harness PM.

5 SCOPED BUFFERED RELEASE MODEL

We propose a scope-aware buffered persistency model named Scoped Buffered Release Persistency (SBRP). SBRP introduces operations that dictate persist memory order (PMO). Box 1 defines the orderings that we consider. We first discuss how a programmer can specify intra- and inter-thread PMO before formalizing the model.

Box 1. Ordering definitions

Program order ($\overrightarrow{po}$): An operation (op_1) precedes another operation (op_2) in program order iff they are both executed by the same thread, and a sequential processor would execute op_1 before op_2 ($op_1 \ \overrightarrow{po} \ op_2$).

Volatile memory order ($\overrightarrow{vmo}$): A memory operation ($M1$) is said to precede another memory operation ($M2$) in volatile memory order iff the underlying memory model guarantees such an order ($M1 \ \overrightarrow{vmo} \ M2$).

Persist memory order ($\overrightarrow{pmo}$): A write to PM ($W1$) is said to precede another write to PM ($W2$) in persist memory order iff the persistency model guarantees that if $W2$ is durable, then $W1$ must be durable ($W1 \ \overrightarrow{pmo} \ W2$).

Transitivity: VMO and PMO are transitive in nature.

$$(M1 \ \overrightarrow{vmo} \ M2) \wedge (M2 \ \overrightarrow{vmo} \ M3) \implies M1 \ \overrightarrow{vmo} \ M3$$
$$(W1 \ \overrightarrow{pmo} \ W2) \wedge (W2 \ \overrightarrow{pmo} \ W3) \implies W1 \ \overrightarrow{pmo} \ W3$$

Intra-thread PMO: We introduce an ordering fence instruction (oFence) to indicate intra-thread PMO. An oFence guarantees that any persists before the oFence are ordered before later persists in program order. Box 2 formally specifies the behavior of oFence.

Box 2. SBRP definitions and guarantees

- W_i^t: A write by thread t to location i in NVM.
- OF^t: A persistent ordering fence by thread t.
- $pAcq_{i,s}^t$: An acquire by thread t on location i of scope s.
- $pRel_{i,s}^t$: A release by thread t on location i of scope s.
- DF^t: A persistent durability fence by thread t.

Intra-thread PMO: Writes from a thread separated by an oFence ensures they are ordered by intra-thread PMO.

$$W_i^t \ \overrightarrow{po} \ OF^t \ \overrightarrow{po} \ W_j^t \implies W_i^t \ \overrightarrow{pmo} \ W_j^t$$

Inter-thread PMO: If a thread performs a persist then executes a release operation, following which another thread acquires, then performs a persist, the two writes are ordered by inter-thread PMO. All operations should be of a sufficient scope that include both threads.

$$W_i^{t1} \ \overrightarrow{po} \ pRel_{X,S}^{t1} \ \overrightarrow{vmo} \ pAcq_{X,S}^{t2} \ \overrightarrow{po} \ W_j^{t2} \implies W_i^{t1} \ \overrightarrow{pmo} \ W_j^{t2}$$

Durability fence: A durability fence provides the same guarantees as an ordering fence, but also guarantees that all prior persists by the thread are durable on completion.

Inter-thread PMO: We introduce scoped persist acquire and release operations (pAcq_*scope* and pRel_*scope*). The block- and device-scope acquire/release operations, pAcq_block()/pRel_block() and pAcq_dev()/pRel_dev(), enable *intra-* and *inter*-threadblock PMO, respectively. pRel accepts two parameters, a variable and a value. It sets the variable to the given value and makes it visible in the specified scope (e.g., threads in a threadblock) *after* all persists before pRel are made durable. pAcq accepts a variable as a parameter, and ensures that the latest value of the variable is read from the specified scope. A pRel on a variable followed by pAcq to the *same* variable ensures that persists before pRel are ordered before the persists after pAcq (see Box 2). Note that the variable is an identifier for matching pRel to pAcq and can be volatile. The scope of pAcq and pRel determines if the enforced PMO is intra- or inter-threadblock.

Let us assume that a thread T_0 wishes to order its persists before another thread T_{32} from the same threadblock. T_0 should issue a pRel_block() after the persists that it wishes to order. Thread T_{32} can then issue a block-scoped pAcq. This guarantees writes from T_0 before pRel are persisted before writes from T_{32} after pAcq.

Durability: PMO does not guarantee immediate durability. While it is not a necessity to reason about recoverability, SBRP also introduces a dFence (durability fence) [45] instruction to enable the flexibility of guaranteeing immediate durability of writes to PM. A dFence ensures that all persists from the issuing thread are durable before the thread's execution proceeds. The completion of a dFence guarantees that the writes from the issuing thread have persisted. Since the NVM is shared by all threads on a GPU, dFence ensures visibility of the writes across all threads on a GPU. Thus, dFence can

```
1 __global__ void reduction(...){
2   // For recovery
3   if (pArr[tid] != EMPTY)
4     return;
5   ... // Finish iter 0
6   if (tid < 4) {
7    // Iter 1, compute local sum
8    sum += pArr[tid + 4];
9    if(tid > 2) {
10    pArr[tid] = sum;
11    // Persistent release
12    pRel_block(&flag[tid], 1);
13   }
14  }
15  // Wait for iter 1
16  if (tid < 2) {
17   // Persistent acquire, spin
18   while (pAcq_block(
19    &flag[tid + 2])  == 0) {}
20  }
21  ... // Finish all compute
22  if(tid == 0)
23  //Persistency bug if blk scope
24   pRel_dev(outSum[blkId], sum);
25 }
```

Figure 3: Reduction kernel.

```
1 __global__ void
2 insert(...){
3   insert_into_log(...);
4   ofence();
5   insert_pair(...);
6   ofence();
7   commit_log();
8 }
9 __global__ void
10 recover(...){
11   read_from_log();
12   restore_pair();
13   dfence();
14   remove_log();
15 }
```

Figure 4: gpKVS kernels.

also be used for inter-thread durability by forming acquire/release patterns with dFence (Section 2).

SBRP in action: We demonstrate inter-thread PMO using the reduction kernel (Figure 2). Recall that the partial sum and output reside on NVM for recovery after a crash.

Figure 3 shows how the reduction kernel is written under SBRP. All elements of pArr are initially EMPTY. First, each thread (say T_0) reads the element kept by the thread 4 indices to its right (T_0 reads $pArr[4]$), and adds that to its current sum ($sum + pArr[4]$) kept in local memory (line 8). If the threads have finished computing their sum (e.g., T_2, T_3) and are not a part of iteration 2, (line 9), they will persist $pArr[tid]$ with value sum (line 10). The threads (T_2, T_3) then release a flag (lines 12) indicating completion of iteration 1. Meanwhile, the threads participating in the next iteration (e.g., T_0, T_1, line 16) will spin on the flag using an acquire operation (lines 18-19). The flag having a value '1' indicates that a release operation was performed on it. This guarantees that persists from the next iteration are always ordered after persists from the prior iteration.

Once all threadblocks have completed their reduction, the first thread of each threadblock writes the sum using a device-scoped release (line 22-24). Afterward, a single threadblock performs the final reduction (not shown). It does a device-scope acquire to obtain each threadblock's partial sum. This ensures that persists during the final reduction are ordered after all previous persists.

5.1 Recovery under SBRP

The recovery mechanism depends on the program semantics, as is the case for Optane's [12] and GPM's [52] programming model. The role of the persistency model is to enable programmers to effectively express the PMOs needed for recovery and enable one to reason about its correctness. For example, in gpKVS, recovery is achieved through write-ahead undo logging to PM. Top half of Figure 4 shows how oFence is used (lines 4 and 6) to ensure that old key-value pair is persisted before the new pair overwrites it and the new pair is persisted before the log is committed. The bottom half of Figure 4 shows the recovery kernel that reads the PM-resident log and inserts the logged pairs into the KVS after a crash. The dFence in line 13 ensures that the restored KVS is persisted before the log is discarded.

For kernels such as reduction, no separate recovery kernel is needed as the recovery logic is embedded within itself (i.e., native) as detailed previously (Figure 3). Here, the execution can simply resume from the last unfinished (persisted) iteration and will return immediately if the thread's index in the intermediate array is non-empty (line 3). For example, assume a crash happens during the second iteration, and the output of iteration 1 has persisted. Then threads responsible for iteration 1 will simply return after checking line 3. We list the recovery method for each application in Table 2 of Section 7.

5.2 Interaction between VMO and PMO

In SBRP, conventional scoped fences (e.g., __threadfence in CUDA) affect both volatile and persistent writes. This is to ensure PMO follows the VMO. However, pAcq, pRel, and oFence operations only impact writes to PM and thus, allows one to specify the desired PMO without impacting writes to the volatile memory.

To avoid the possibility of a cyclic order between VMO and PMO between a set of persists, we ensure that any ordering guaranteed in PMO is reflected in VMO too. For example, a persist to a location A separated from a persist to location B by an oFence ensures that the write to A persists before B. It also ensures the write to A is ordered in VMO before the write to B. However, if A or B are volatile, the oFence has no effect. As the PMO agrees with VMO, SBRP is a strict persistency model (Section 2).

5.3 Possibility of Scoped Persistency Bugs

A persistency model cannot guarantee correct recovery by itself. It enables the framework to reason about the correctness of a recoverable GPU kernel. Inadequate use of oFence pAcq, pRel can lead to bugs. For example, under SBRP, if programmers use narrower scope than needed by program semantics in pAcq/pRel operations for inter-thread PMO, it can lead to persistency bugs. That is, PM-resident data structures may not be recoverable to a consistent state after a crash. We call such programming errors scoped persistency bugs. Such bugs, in spirit, are similar to scoped synchronization bugs for VMO [32, 33]. To observe how such persistency bugs can lead to incorrect recovery, let us refer to the example of the reduction kernel (Figure 3). On line 24, a device-scoped pRel is used to order persists across threadblocks. Instead, if this was block-scoped, a persistency bug could manifest. During the final reduction by the chosen threadblock, it could acquire incorrect values when computing the final sum. After a crash, threads could see incorrect values (neither EMPTY nor correct partial sum) in the array (line 3). This will result in incorrect final computation. However, note that since pAcq/pRel affects only writes to PM, the correctness of programs that deal with only volatile memory is unaffected.

Summary: We propose a scope-aware buffered GPU persistency model SBRP which relies on acquire/release semantics for specifying inter-thread PMO. We shall next demonstrate how the model can be efficiently realized in GPU architecture.

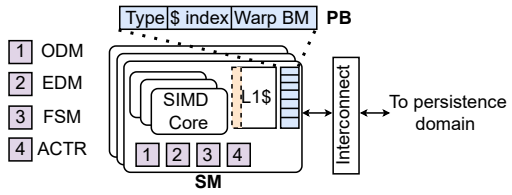


Figure 5: Design of scoped buffered persistency model.

6 IMPLEMENTATION

We first discuss hardware modifications, detailing fundamental challenges in designing GPU buffers for persists. Afterward, we discuss how the operations in SBRP are implemented and highlight opportunities for optimizations.

Hardware modifications: Figure 5 shows the new hardware components added to *each* SM of the GPU. First, we add a FIFO persist buffer (PB), which is itself volatile. Three 'Type' bits indicate whether an entry corresponds to a persist or an ordering point (oFence/dFence/pAcq/pRel). If the entry is a persist, it holds the index of the dirty L1 cache line containing the data. A 32-bit bitmask (Warp BM) is maintained to track which warps issued the store or fence operation. The typical size of an entry is 44 bits.

We add three more bitmasks to each SM – the order delay mask (ODM), the eviction delay mask (EDM) and the flush status mask (FSM). The number of bits in each mask is equal to the maximum resident warps in an SM (here, 32). Threads may stall when enforcing ordering (e.g., dFence) or when eviction of a cache line violates ordering constraints. The ODM tracks warps that are stalled due to the former, while the EDM tracks warps that are stalled due to the latter. The FSM tracks the warps whose persists are currently being flushed. We shall later see the role of these structures. Each L1 cache line is extended with a bit to indicate whether the data is for PM (marked in orange). Eight bits are added per cache line to index into the corresponding PB entry. A hardware acknowledgement counter (ACTR), tracks pending persists. ACTR is initially zero and is incremented on eviction of a persist from the L1 cache. When the write reaches the persistence domain, an acknowledgement is sent to the SM to decrement ACTR.

The PB's design plays a key role in the models' implementation. The PB should enable efficient enforcement of intra-thread, intra- and inter-threadblock PMO. Simultaneously, it should maximize concurrency of persists and coalescing to benefit from GPU parallelism and minimize writes to PM.

The first challenge is deciding the granularity at which the PB should track persists. One option is to track persists from each thread separately, as is done in CPUs. This would add a large hardware overhead, given the number of GPU threads. Alternatively, one may track only the cache lines updated, without noting the threads updating it. However, this leads to false ordering. Consider threads T_0 and T_{32} belong to a threadblock. Thread T_0 writes to pX, after which thread T_{32} executes an oFence and writes to pY, where pX and pY are on PM. If we track just the updates to cache lines by any thread in a threadblock, the PB will have the following entries: $pX \rightarrow$ oFence $\rightarrow pY$. It would then unnecessarily order pY after pX, although they were issued by different threads and could have persisted concurrently.

We find that tracking persists, and PMOs *per warp* provide a sweet spot. We use the Warp BM in PB entries for this purpose. The storage overhead of per-warp tracking is 32× lesser than per-thread. Also, threads in a warp typically execute in lock-step. The writes from a warp often fall in the same cache line, allowing the hardware to coalesce them into a single write [49]. Consequently, it introduces minimal false ordering for GPU kernels.

We do not maintain a persist buffer at the L2 cache. Instead, we let the persists write through the L2. A buffer shared across all SMs in a GPU is not scalable as modern GPUs have up to 108 SMs. If we track persists and PMO at a warp level in an L2 buffer, one will need ~ 3400 bits per buffer entry for the Warp BM. If we instead track persists and PMO per threadblock, we again end up with false ordering. Further, the L2 caches in GPUs are in several MBs. To maintain a cache coverage similar to L1 PB, the L2 buffer must be hundreds of times bigger than the L1 PB. Maintaining such a large, centralized buffer is infeasible. Note that most buffered CPU persistency models also do not use buffers at the shared LLC [37, 45].

Storage overheads: Each entry in the PB needs 44 bits. The Warp BM occupies 32 bits; the cache line index needs 9 bits. Type information is 3 bits long. Each L1 cache line is extended with a bit to indicate whether it contains PM data and 8 bits to index into the PB. Ninety-six bits are maintained per SM for the ODM, EDM, and FSM combined and 8 bits for the ACTR. If we maintain half the number of PB entries as there are L1 cache lines, the additional storage requirement is ~ 2 KB per-SM. The overhead is ~ 3.1% for a 64 KB L1 cache. This overhead is less (~ 1%) if we include the L2 cache. Section 7.3, shows that reducing the PB entries does not lead to a significant performance loss.

6.1 Operation of SBRP

Persist operation: We now discuss how the PB is designed to maximize coalescing of persists, while also guaranteeing correct ordering between them. When a persist is issued, the data may not be in the L1 cache. On a miss, a PB entry is allocated along with a cache line. The PB entry contains the index of the cache line. The Warp BM is set to the warp issuing the persist. The cache line is set to point to the PB entry. On a cache hit, however, we find the corresponding PB entry (say, PB_k) from the index stored in the cache line. We check the PB for any ordering operations issued after PB_k by the same warp(s). If there are none, the write is performed to the cache line and the warp's bit is set in the Warp BM for the PB entry. This *coalesces* unordered writes to the same cache line.

If there is an ordering entry after PB_k, the store is not allowed to proceed as it will violate the required PMO. The issuing warp is stalled and marked in the EDM until PB_k is persisted. Only after that the persist is allowed to proceed. For example, assume a thread: ① sets $pX = a$, ② sets $pY = b$, ③ issues an oFence, then ④ sets $pX = c$. This creates a PMO between $pY = b$ and $pX = c$. To guarantee this order, the second persist on pX (operation ④) is neither allowed to modify the current cache line nor can it insert a new entry into the PB until the first persist on pX (operation ①) is durable. A persist at the head of the PB is removed and the corresponding cache line is evicted. Simultaneously, the ACTR is incremented to track if it has been made durable.

Intra-thread PMO: The oFence operation creates intra-thread PMO. While oFence was proposed for CPUs [45], it is new for GPUs. Buffered implementation of oFence on GPU presents unique challenges. The per-thread tracking of PMO as proposed for CPUs is impractical for GPU's thousands of threads. Hence, the PB tracks these ordering requirements at the granularity of a warp. We discuss the GPU-specific implementation of ofence below.

When oFence is issued, we add an entry into the PB and mark its 'Type'. We allow oFence entries to coalesce by setting the respective Warp BM bits in the entry for each warp issuing the oFence. If an oFence is issued by $Warp_0$ followed by another by $Warp_1$ without intervening any operations, we maintain a single PB entry with bits 0 and 1 set in the Warp BM for $Warp_0$ and $Warp_1$.

When the oFence entry reaches the head of the PB, the FSM bits are set corresponding to the Warp BM of the PB entry (bitwise OR) and the entry is removed from the PB. Later, when a persist (say pX) reaches the PB head or is evicted, its Warp BM is compared with the FSM bits (bitwise AND). If there are any common bits, it indicates that an unacknowledged, flushed cache line is ordered before pX. Flushing pX is delayed to preserve the PMO. Once all outstanding persists are acknowledged, i.e., ACTR reaches zero, the FSM bits are reset and pX can be flushed. In the absence of FSM, warps that issued an oFence can not be distinguished from warps that did not. We will have to assume that all warps issued oFence, introducing false ordering amongst persists from different warps.

Intra-threadblock PMO: The intra-threadblock PMO is expressed through pAcq_block() and pRel_block(). For pAcq and pRel, a PB entry is added with 'Type' set as block-scope acquire or release along with the warp issuing it. Persists after pAcq can be made durable only after pAcq exits PB and those before pRel must be made durable before pRel reaches the PB head. When the pAcq and pRel entries reach the PB head, FSM is set for associated warps. Note, pAcq cannot complete until pRel finishes. Hence, warps waiting for pAcq to complete can only persist their writes after writes from warps performing pRel persists. This is how the FSM avoids stalling unrelated warps.

Inter-threadblock PMO: The inter-threadblock PMO is expressed using device-scoped pAcq and pRel. On a device-scoped pAcq/pRel, we add a new entry in the PB with the appropriate 'Type' and note the warps that issued it in the Warp BM. In current GPUs, device-scoped threadfences force writes from L1 cache to be flushed to the shared L2 cache. This makes the writes visible to all the threadblocks. For inter-threadblock PMO, we do the same, in the absence of an L2 buffer. Unlike block-scoped, the device-scoped pAcq invalidates the cache line to avoid reading stale data.

The ODM bits are set to track warps that issued the operations. The warps executing pRel are delayed, while others not marked in the ODM continue execution. Once the bitmask is set, we flush the persists. The ACTR keeps a count of the flushed, unacknowledged persists. When the pRel entry reaches the buffer head its ODM bits are reset, and the same bits are set in the EDM. Once all persists are acknowledged (ACTR hits zero), EDM bits are reset and the participating warps can resume. This ensures that only the warps whose persists have been made durable resume execution. Similar to FSM, ODM also helps to avoid false ordering.

Durability: The dFence forces all writes to PM from a thread to be durable before later writes from that thread. The issuing thread stalls until the dFence completes. We add an entry into the PB marking the warps that issued the dFence. While executing the dFence, we set the bits in the ODM for all the warps that issued it. The persists from those warps are flushed and counted by the ACTR. These warps are not allowed to proceed until the ACTR hits zero, indicating the dFence operation completed. The use of ODM allows unrelated warps to continue execution. We note that dFence was proposed for CPUs [45]. Although its implementation for GPUs is new, including tracking at warp granularity, it is not our key contribution.

Eviction: If a read/write attempts to evict a dirty PM cache line (say pX), we consult its PB entry using the PB index kept with pX. We check if evicting pX violates PMO by looking for an ordering entry in the PB before pX's. If none exist, pX is evicted. Otherwise, the warp causing the eviction is stalled by setting its bit in the EDM. Once the outstanding flushes are complete (ACTR is zero), the warp can retry eviction.

Table 1: Simulated Hardware configuration

# of SMs	30	**Window size**	6
Clock speed	1365 MHz	**Threads/block**	1024
L1 cache	64 KB/SM	**L2 cache**	3 MB
GDDR BW	336 GBPS	**GDDR latency**	100 ns
NVM BW	84 GBPS read, 42 GBPS write	**NVM latency**	300 ns
PCIe BW	28 GBPS	**PCIe latency**	300 ns

6.2 Hardware Optimization

A key performance consideration while designing the buffers is deciding when to flush dirty PM cache lines. Prior CPU models [37] typically perform this eagerly by flushing as soon as ordering constraints allow. While this ensures that the NVM bandwidth is utilised well, it reduces the coalescing opportunities in the cache. For example, a thread may perform two consecutive stores to a cache line without intervening ordering. In an eager policy, this would create two persists. However, a single persist after the second store would have been sufficient.

One can also lazily flush data only at ordering operations. This allows unordered stores to the same cache line to coalesce. However, it leads to poor utilization of NVM bandwidth. If ordering is infrequent, it induces long periods with no persists followed by a burst, creating contention. Instead, we propose a window-based policy to get best of both worlds. Each SM tries to maintain a fixed number of outstanding persists at a time. This creates a stream of persists, while allowing coalescing opportunity. In Section 7.3 we quantitatively compare these different policies.

7 EVALUATION

We evaluated SBRP by extending GPGPU-Sim 4.0 [35] running CUDA 11.4 [48]. Table 1 lists the simulated configuration. We evaluate memory persistency models on both system designs discussed in Section 3 – PM-far and PM-near. The PCIe latency and the attainable bandwidth were set to mimic that of PCIe 4.0 [44]. In PM-near, PCIe does not play a role since the NVM placed on board the GPU.

We evaluated SBRP against GPM [52], which implicitly assumes the epoch persistency model (Section 4) ('GPM' in figures). GPM was implemented on real hardware. We implemented and validated

Table 2: Applications used in evaluation

Applications	Params	Scoped PMO	Recovery
gpKVS	~ 64K pairs	Intrathread	Logging
Hashmap (HM)	~ 50K entries	Intrathread	Logging
SRAD	512 sq. matrix	Intrathread	Native
Reduction (Red)	~ 4M ints	Blk/dev-interthread	Native
Multiqueue (MQ)	2K batches	Intra/blk-interthread	Logging
Scan	~ 120K ints	Blk-interthread	Native

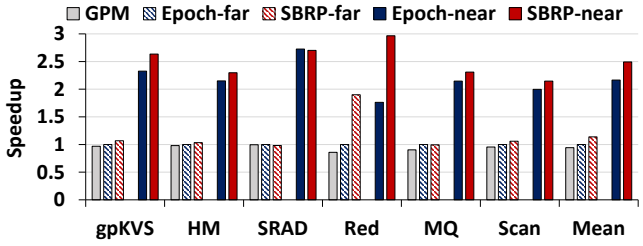


Figure 6: Speedup over epoch-far of different models.

GPM on the simulator for it to be compared with our proposed persistency model – SBRP. Since SBRP needs modifications to the GPU hardware, it cannot be evaluated on current hardware. The original implementation of GPM used a system-scoped fence as an epoch barrier to avoid modifications to GPU hardware. Such fence flushes writes to both PM and volatile memory. We thus also simulated an enhanced version of GPM's implementation of the epoch persistency model where an epoch barrier *only* impacts writes to PM ('Epoch' in figures).

If a model is evaluated on PM-far (PM-near), the name of the persistency model is appended with '-far' ('-near'). For example, SBRP-near represents SBRP model on a PM-near system.

7.1 Applications

Table 2 lists the GPU-accelerated applications used in the evaluation. Each of these applications places its key data structures in the PM for recoverability. These applications have different types of PMO – intra-thread, intra-, and inter-threadblock and different crash recovery models.

GPU-accelerated Persistent key-value store (gpKVS): A batch of key-value pairs are inserted into a PM-resident KVS in parallel. The application uses PM-resident write-ahead undo log for recovery. Section 5.1 and Figure 4 detail the recovery for gpKVS.

Hashmap (HM): Cuckoo hashing is used to insert batches of values into a hashmap in parallel. Before inserting a new value, the old value is logged to PM, necessitating an intra-thread PMO. As in gpKVS, upon recovering from a crash, a recovery kernel would read from the log restoring the hashmap to a consistent state [1].

SRAD (SRAD): This application is used to remove noise from an input image to produce an output image. Each thread is responsible for processing a pixel of the input image. The processing for each pixel happens in two steps. After the first step, the intermediate noise values and the image (pixels) are persisted for one to resume processing even if a system crashes during the second part of the processing. For recovery to a consistent state, each pixel should be persisted only after a thread persists the corresponding noise values, requiring intra-thread PMO. After power-up, each thread resumes processing from the persisted values and the partially processed image [18, 52].

Reduction (Red): Section 4 details Reduction. It requires both intra-threadblock and inter-threadblock PMO [47].

Multiqueue (MQ): In this kernel, each threadblock in a kernel maintains a single persistent queue [4]. A batch of entries is inserted or removed from the queue in a transaction. Each thread is responsible for processing a subset of entries in the batch. Recovery requires either all entries in a batch to be inserted or none. For this purpose, a queue's head and tail indices are logged to PM. Once all threads finish inserting or removing entries, the tail/head is updated by a leader thread, creating an intra-threadblock PMO. The leader also marks a transaction complete after logging the tail/head, creating an intra-thread PMO. During recovery, queue heads and tails are restored from the log for in-progress transactions to ensure a consistent state of the queues.

Scan: This kernel computes the scan of many data arrays. As in reduction, the scan of an element is computed iteratively. A thread uses the output from the previous iteration of another thread in the threadblock. This needs intra-threadblock PMO. During recovery, the computation resumes from the persisted array contents [47].

7.2 Performance Analysis

Figure 6 shows the performance of the models (higher is better). We normalize the performance to epoch-far (higher is better). Applications on the left half have only intra-thread PMO. The right half has those with inter-thread PMO. There are five bars for each application. Besides GPM [52], we evaluate the epoch persistency model on two system designs – PM-far and PM-near. Note that GPM only works for PM-far as it avoids hardware changes.

First, note that epoch-far outperforms GPM by 6% on average, and by up to 16%. While both follow epoch persistency, in GPM writes to both volatile and PM are flushed on an epoch barrier, unlike for epoch-far. Since epoch-far strictly outperforms GPM, we do not focus our analysis on GPM for the latter parts of the section.

SBRP-far outperforms GPM and epoch-far by 21% and 14%, on average, respectively. The difference can be up to 121% and 90%, respectively. SBRP-near outperforms epoch-near by 15% on average and up to 68%. This establishes the importance of a scoped, buffered persistence model for a GPU, irrespective of the system design.

We also observe that epoch-near outperforms epoch-far by 116%, SBRP-near outperforms SBRP-far by 119% and GPM by 165%. The removal of the PCIe bottleneck is crucial to the improved performance of PM-near. This demonstrates the value of placing NVM onboard the GPU.

The epoch model expresses both inter- and intra-thread PMO using global barriers. At a barrier, threads flush persists to PM and invalidate them from L1 cache. Invalidation is needed to ensure that threads executing the barrier for inter-threadblock PMO do not read stale data. In SBRP, L1 cache contents remain unaffected on intra-thread (oFence) and intra-threadblock (block-scope) PMO as the accessing threads share the SM's L1 cache. The use of oFence for intra-thread PMO in gpKVS and HM avoids invalidating L1 cache, unlike in epoch, explaining their speedups. Cached PM data is invalidated *only* on inter-threadblock PMO and dFence.

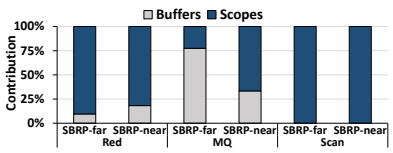


Figure 7: Speedup breakdown.

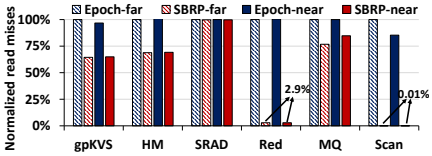


Figure 8: L1 read misses for NVM data.

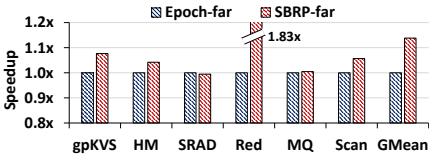
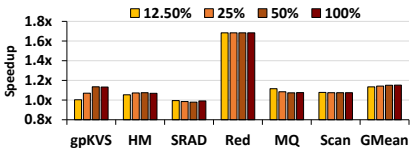
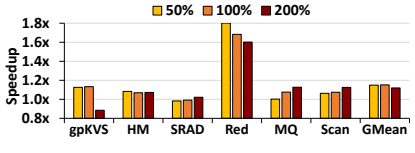


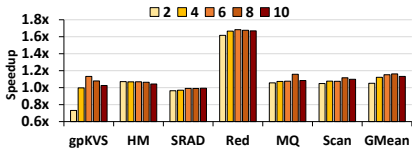
Figure 9: SBRP speedup with eADR.



(a) Varying L1 coverage.



(b) Varying bandwidth.



(c) Varying window sizes.

Figure 10: SBRP sensitivity studies. Speedups of SBRP over epoch.

SRAD performs all calculations, then persists its results. Due to bursty writes to the PM while persisting, all models behave similarly. For these applications, SBRP-far outperforms epoch-far by 3% and SBRP-near outperforms epoch-near by 6%. SRAD requires only intra-thread PMO; it benefits from buffering but not from scopes.

Reduction greatly benefits from block-scoped pAcq and pRel, providing 90% speedup for SBRP-far over epoch-far and 68% for SBRP-near over epoch-near. Scopes allow SBRP-far to overcome the bandwidth limitations of PCIe. The process of reducing a subarray by a threadblock can be performed within the L1 cache, avoiding traversing the PCIe. As writes under SBRP-near remain in L1 cache without being flushed to the PM, it speeds up over epoch-near. MQ benefits from inter-threadblock PMO. However, intra-thread PMO during logging causes frequent flushes, limiting speedups for SBRP to 8% under PM-near. In scan, every iteration has many accesses to bandwidth-limited NVM, limiting its speedup (6-7%).

Importance of scopes: To quantify the impact of scopes for applications with intra-threadblock PMO, we converted all block-scope operations to device-scope. Figure 7 shows the relative contribution of buffers and scopes to SBRP's speedups. As gpKVS, HM, SRAD use only intra-thread PMO, scopes are irrelevant for them, and are excluded. For the rest, scopes are key except for MQ with SBRP-far. MQ is bottlenecked while persisting data during logging, further exacerbated by the poor bandwidth in SBRP-far. On average, scopes account for 77% of the speedups for applications with intra-threadblock PMO.

Impact on caching: Figure 8 shows the number of L1 cache read misses (normalized to epoch-far) to quantify SBRP's impact on cache utilization (lower is better). gpKVS and HM witness lower L1 read misses with SBRP since oFence does not invalidate the L1 cache. SRAD's persists happen at the end of the computation, limiting benefits from improved L1 utilization. The block-scope operations lower L1 read misses for reduction and scan under SBRP as data is cached longer. Due to logging, benefits from block-scope operations are limited in MQ. In general, SBRP improves read latency over the epoch model, as it caches data longer by avoiding invalidating cache lines for intra-thread and intra-threadblock PMO. This helps relatively read-heavy workloads like reduction.

Impact of eADR: Intel recently announced enhanced ADR (eADR) feature that allows battery-backed servers to drain the CPU cache contents to PM on power failure [59]. It obviates the need to explicitly flush lines from CPU caches for persistence. Although, Intel acknowledges that enabling eADR is challenging [26]. We study the impact of eADR on GPU persistency models. Note it affects only PM-far systems. Under eADR, persists from a GPU need to only reach the CPU's LLC to become durable. Figure 9 shows SBRP-far's performance normalized to epoch-far in the presence of eADR. SBRP-far speeds up over epoch-far by 14%, on average. This is similar to that without eADR since eADR doesn't remove the PCIe bandwidth bottleneck. It only reduces the latency of persisting. Scopes and buffering in SBRP help by limiting the PCIe traversals.

7.3 Sensitivity Studies and Recovery Cost

Persist buffer: Figure 10 (a) shows the speedup of SBRP-near over epoch-near with varying the PB size. The different bars indicate the percentage of L1 cache that the PB covers, e.g., if the L1 cache holds 512 cache lines, then a 50% buffer (default) holds 256 entries. If threads try to bring more PM data to the L1 cache than the PB can hold, it starts draining cache lines. This can lead to anomalies, as in HM, where the 50% buffer outperforms 100%, as eagerly flushing reduces the amount of data that needs to be persisted in the critical path. On average, the 50% buffer performs within 1% of the 100% buffer, showing that buffer size is not a key factor. As expected, the performance drops if the buffer is too small, seen in gpKVS.

NVM bandwidth: Figure 10 (b) shows the speedup of SBRP-near over epoch-near with varying NVM bandwidth. The 100% bar is the baseline we evaluate on (84 GBPS read and, 42 GBPS write bandwidth). The 200% bar represents when we double those bandwidths, while the 50% bar represents when bandwidths are halved. For gpKVS and HM, benefits of buffering log writes reduces with higher bandwidth. Similarly, for reduction, epoch's performance loss due to flushing persists that could've been retained in the L1 cache moderates. Conversely, an increased bandwidth helps SRAD, MQ, and scan perform better as the bursty persists are lesser concern. On average, SBRP-near provides noticeable speedups regardless of NVM bandwidth with 15%, 15% and 12% performance improvement for 50%, 100%, and 200% bandwidth.

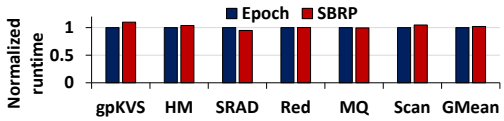


Figure 11: Normalized runtime of recovery kernel.

Window size: Figure 10 (c) shows speedup of SBRP-near with varying window sizes. The best speedup is obtained with a window size of 6 (default), i.e., each SM maintains 6 outstanding persists. If the window is too small, PM is underutilized, while larger window leads to congestion.

Recovery time: Figure 11 compares the recovery time under epoch-near and SBRP-near (normalized to epoch-near; lower is better). We evaluated scenarios with highest recovery time. For example, we crash gpKVS just before the transaction completes. This creates the largest log size. Upon recovery, all logged pairs are read and re-inserted to KVS. We observe that the average recovery time for both models are similar – within 3%. For gpKVS, however, SBRP's recovery is 10% slower. Here, the recovery kernel reads the entire log from PM, re-inserts old pairs into the PM-resident KVS, and then all updates are persisted using epoch barrier in epoch-near and dFence in SBRP, before clearing the log. While a barrier flushes eagerly, dFence uses buffering that takes longer to drain writes to PM. While buffering speeds up the common case of crash-free execution (Figure 6), it slightly slows down recovery. Scan and HM also slows down slightly (< 5%) for the same reason. For others, recovery does not need bulk persists. Thus, recovery time is identical.

We also noticed that the (worst case) recovery time as the percentage of crash-free execution time varies between 0.7-42% across applications. The highest was for gpKVS as it involves reading all old pairs from the log, re-inserting and persisting them. In contrast, SRAD, Red, MQ need negligible recovery time. In general, crash-free execution is the common case and recovery is important but uncommon. Thus, runtimes of recovery is relatively less important than crash-free execution.

8 RELATED WORK

GPU persistency models: There are relatively limited number of works exploring NVMs for GPUs [3, 5, 14, 34, 39, 52]. Lin et al. [39] proposed a pragma-based compiler and transactional API to adopt strict and epoch persistency. The authors do not propose any new model for GPUs. Gope et al. [2, 14] proposed new instructions for GPU persistency called scoped persist barriers. It provides inter-thread PMO by forcing all threads in the specified scope to wait until prior writes have persisted (made durable). Their model is actually *not* scope-aware, as it globally communicates all PMO requirements. The model uses a buffer architecture inspired by quick release [21], but does not distinguish between intra-thread or inter-thread PMO requirements,. A persist barrier simply stalls the issuing thread, drains the buffer, and waits for the writes to reach PM. In SBRP, the buffers allow intra- and inter-thread PMO to proceed without global synchronization. In SBRP, acquire/release for inter-thread PMO and limits the impact of PMOs only to the warps of the participating threads. Lazy persistency for GPUs [3] proposed a software optimization for recoverable applications where idempotent code regions can persist writes without any strict ordering. In contrast, we propose a hardware persistency model that supports all classes of GPU applications.

CPU persistency models: There exists a rich literature on hardware persistency models for CPUs. Some were discussed in Sections 2 and 4. DPO [37] proposed a relaxed consistency, buffered strict persistency model that maintains buffers to store and order persists separately from the caches. HOPS [45] extends the cache hierarchy to enforce ordering and durability constraints for persists separately. LLP [36] provides language-level persistency semantics to programmers. Gogte et al. [13] proposed a hardware implementation of strand persistency which divides thread execution into strands. PMO is enforced within strands but not across them.

An orthogonal set of work [11, 15, 16, 40–42] also focuses on detecting bugs in persistent memory programs. These bugs are caused by programmer mistakes, such as missing fences. To reason about whether PM-aware programs contain bugs, a formalization of the persistency model is required – the focus of this work.

Several works proposed optimizing CPU's logging to PM [17, 27, 30, 51, 60]. Some of the optimizations, e.g., leveraging multi-versioning in the memory hierarchy to avoid logging overheads [17], can speedup half of our applications that use logging. However, optimizing logging or even eliminating it does not obviate the need to have a well-specified, efficient GPU persistency model.

9 CONCLUSION

Important applications, such as persistent KVS, can benefit from both GPU's parallelism and PM's fine-grain persistence. However, a well-specified GPU persistency model is a pre-requisite for GPU programs to reason about recoverability. We propose a scope-aware, buffered persistency model that allows GPU programmers to express PMO through acquire/release semantics. We show that our new model can provide a speedup of up to 90% across various applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We thank Ashish Panwar and Ajay Nayak 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. Shweta is supported by Google PhD Fellowship.

10 DATA AVAILABILITY STATEMENT

We make the artifact for the work available [53]. The artifact contains scripts to reproduce all important figures from section 7.

A ARTIFACT APPENDIX

A.1 Abstract

We provide the source code and setup for our GPU persistency model, Scoped Buffered Release Persistency (SBRP). SBRP is a scope-aware, buffered persistency model that provides high performance to GPU applications that wish to persist data on Non-Volatile Memory (NVM). SBRP modifies the GPU hardware and has been implemented using GPGPU-Sim, a GPU simulator.

This artifact consists of the source code of the simulator, benchmarks used for evaluation and all scripts needed to replicate the figures in the paper.

A.2 Artifact check-list (meta-information)

- **Compilation:** CUDA 11.4, GCC 9.4.0.
- **Binary:** Included for x86-64.
- **Data set:** Scripts are provided to download/generate datasets.
- **Run-time environment:** Workloads can be run on the bare machine. To run the workloads on bare machine, CUDA 11.4 is required on a Linux installation of Ubuntu 20.04.
- **Hardware:** A machine with atleast 12 threads and 50 GiBs of memory is required.
- **Execution:** Compiled CUDA binaries are run on our hardware simulator. Makefiles and python scripts are provided.
- **Output:** We generate csv files and bar graphs for each experiment. The graphs and csv files can be found in outputs folder. Raw numbers for each figure can be found in the results folder associated with them.
- **How much disk space required (approximately)?** Atleast 10 GiBs of storage space.
- **How much time is needed to prepare workflow (approximately)?** 10 minutes.
- **How much time is needed to complete experiments (approximately)?** 20 hours.
- **Publicly available?** Yes.
- **Archived (provide DOI)?** Yes, https://doi.org/10.5281/zenodo.7306303

A.3 Description

The artifacts contains the source of SBRP model along with the evaluated benchmarks. It allows to reproduce the following results:

- Figure 6: Speedup over epoch-far of different models.
- Figure 8: L1 read misses for NVM data.
- Figure 9: SBRP speedup with eADR.
- Figure 10: SBRP sensitivity studies. Speedups of SBRP over epoch.
  - (a) Varying L1 coverage.
  - (b) Varying bandwidth.
  - (c) Varying window sizes.
- Figure 11: Normalized runtime of recovery kernel.

A.3.1 How to access. The artifact is made available in the GitHub repository https://github.com/csl-iisc/SBRP-ASPLOS23 and also at https://doi.org/10.5281/zenodo.7306303.

A.3.2 Hardware dependencies. The scripts require a processor supporting at least 12 threads and at least 50 GiBs of memory.

A.3.3 Software dependencies. The artifact needs CUDA-11.4 on a bare machine. We also provide a Docker container with all included dependencies. To install CUDA 11.4 on a bare machine follow the instructions mentioned in the README or download from NVIDIA. To run the simulator, dependencies for GPGPU-Sim also have to be downloaded. Follow the README or the instruction manual for GPGPU-Sim [35]. The dependencies can be installed as follows:

```
$ sudo apt-get install build-essential xutils-dev bison \
zlib1g-dev flex libglu1-mesa-dev
$ sudo apt-get install libxi-dev libxmu-dev libglut3-dev
```

Also ensure that CUDA_PATH variables are properly set before running the artifact.

We have also containerized our setup and provide a docker image for ease of use. To install Docker on an Ubuntu machine, use the following command.

```
$ sudo apt install docker.io
```

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/SBRP-ASPLOS23
$ cd SBRP-ASPLOS23
```

A.5 Experiment workflow

The outermost directory consists of two important folders: models, benchmarks. models/ contains all the models needed for evaluating all the figures. benchmarks/ contains all the benchmarks we evaluate on. Apart from that, the scripts/ folder contains the scripts needed for generating all the figures.

The experiments can be run in parallel. However it would require more compute and memory. We provide a Makefile that compiles, executes and generates a comma-separated file and bar-graph reports for the figures 6, 8, 9, 10a, 10b, 10c and 11.

The experiments can be run within a docker container or a bare machine. To run experiments within the container, first build the container in the main folder using –

```
$ docker build . -t sbrp:v1
```

Run the Docker container in interactive mode using the following command. This command opens a bash shell in the Docker image.

```
$ docker run -it sbrp:v1
```

Individual experiments can now be run as follows: Note, these experiments can also be run outside the docker container with the correct setup required for GPGPU-Sim.

```
$ make run_figure6 #To run figure 6
$ make run_figure9 #To run figure 9
$ make run_figure10_a #To run figure 10(a)
$ make run_figure10_b #To run figure 10(b)
$ make run_figure10_c #To run figure 10(c)
$ make run_figure11 #To run figure 11
```

To run all figures, run the following command:

```
$ make run_all #To run all figures
```

The raw numbers and figures can be obtained from the outputs folder present in the outermost directory.

A.6 Evaluation and expected results

For each key result, a comma separated file and a graph are generated. The outputs folder contains all generated reports and graphs. The reports can be matched against figures reported in the paper.

The generated reports and output folders are given in Table 3. To obtain the reports and graphs, use the following command:

```
$ make output_figure6
$ make output_figure8
$ make output_figure9
$ make output_figure10_a
$ make output_figure10_b
$ make output_figure10_c
$ make output_figure11
```

Run the following commands to make all outputs:

```
$ make output_all
```

Table 3: Output folders for each figure

Fig	Raw numbers	Graphs	CSV files
6	figure6_results/	figure6_graph.pdf	figure6_output.txt
8	figure6_results/	figure8_graph.pdf	figure8_output.txt
9	figure9_results/	figure9_graph.pdf	figure9_output.txt
10(a)	figure10_a_results/	figure10_a_graph.pdf	figure10_a_output .txt
10(b)	figure10_b_results/	figure10_b_graph.pdf	figure10_b_output .txt
10(c)	figure10_c_results/	figure10_c_graph.pdf	figure10_c_output .txt
11	figure11_results/	figure11_graph.pdf	figure11_output.txt

REFERENCES

[1] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. 2009. Real-Time Parallel Hashing on the GPU. In *ACM SIGGRAPH Asia 2009 Papers* (Yokohama, Japan) *(SIGGRAPH Asia '09)*. Association for Computing Machinery, New York, NY, USA, Article 154, 9 pages. https://doi.org/10.1145/1661412.1618500

[2] Arkaprava Basu, Mitesh R Meswani, Dibakar Gope, Sooraj Puthoor. 2019. Scoped persistence barriers for non-volatile memories. https://patentimages.storage.googleapis.com/8e/40/ee/6b238c91b5ffcf/US10324650.pdf.

[3] Ardhi Wiratama Baskara Yudha, Keiji Kimura, Huiyang Zhou, and Yan Solihin. 2020. Scalable and Fast Lazy Persistency on GPUs. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 252–263. https://doi.org/10.1109/IISWC50251.2020.00032

[4] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R. Gao. 2010. Dynamic load balancing on single- and multi-GPU systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470413

[5] Sui Chen, Lei Liu, Weihua Zhang, and Lu Peng. 2020. Architectural Support for NVRAM Persistence in GPUs. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 31. 1107–1120. https://doi.org/10.1109/TPDS.2019.2960233

[6] 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

[7] 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

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[9] CXL. November 2020. CXL Consortium. Compute Express Link Specification Revision 2.0. https://www.computeexpresslink.org/download-the-specification.

[10] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. 2020. Lazy Release Persistency. 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, 1173–1186. https://doi.org/10.1145/3373376.3378481

[11] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs. 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, 503–516. https://doi.org/10.1145/3445814.3446744

[12] Eduardo Berrocal Garcia De Carellan. 2018. Recovery and Fault-Tolerance for Persistent Memory Pools Using Persistent Memory Development Kit (PMDK). https://www.intel.com/content/www/us/en/developer/articles/technical/recovery-and-fault-tolerance-for-persistent-memory-pools-using-persistent-memory.html.

[13] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 652–665. https://doi.org/10.1109/ISCA45697.2020.00060

[14] Dibakar Gope, Arkaprava Basu, Sooraj Puthoor, and Mitesh Meswani. 2018. A Case for Scoped Persist Barriers in GPUs. In *Proceedings of the 11th Workshop on General Purpose GPUs* (Vienna, Austria) *(GPGPU-11)*. Association for Computing Machinery, New York, NY, USA, 2–12. https://doi.org/10.1145/3180270.3180275

[15] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: Efficiently Model Checking Persistent Memory Programs. 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, 415–428. https://doi.org/10.1145/3445814.3446735

[16] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2022. Yashme: Detecting Persistency Races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 830–845. https://doi.org/10.1145/3503222.3507766

[17] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. 2019. Distributed Logless Atomic Durability with Persistent Memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 466–478. https://doi.org/10.1145/3352460.3358321

[18] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Víctor García-Floreszx, Simon Garcia de Gonzalo, Thomas B. Jablin, Antonio J. Peña, and Wen-mei Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 43–54. https://doi.org/10.1109/ISPASS.2017.7975269

[19] Tom's Hardware. August 2022. Intel Kills Optane Memory Business. https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good.

[20] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. 2008. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 260–269. https://doi.org/10.1145/1454115.1454152

[21] Blake A. Hechtman, Shuai Che, Derek R. Hower, Yingying Tian, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 189–200. https://doi.org/10.1109/HPCA.2014.6835930

[22] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-Race-Free Memory Models. *SIGARCH Comput. Archit. News* 42, 1 (Feb. 2014), 427–440. https://doi.org/10.1145/2654822.2541981

[23] Intel. 2020. Why Is the Intel® Optane™ Persistent Memory in Memory Mode Not Persistent? https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf.

[24] Intel. 2021. Intel Optane Persistent Memory. https://www.intel.in/content/www/in/en/architecture-and-technology/optane-dc-persistent-memory.html. Accessed: 2021-11-15.

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

[26] Intel. 2021. Section "Power-Fail Protected Domains" of "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. Accessed: 2021-11-15.

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

[28] 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.

[29] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 660–671. https://doi.org/10.1145/2830772.2830805

[30] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 361–372. https://doi.org/10.1109/HPCA.2017.50

[31] 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

[32] Aditya K. Kamath and Arkaprava Basu. 2021. IGUARD: In-GPU Advanced Race Detection. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 49–65. https://doi.org/10.1145/3477132.3483545

[33] Aditya K. Kamath, Alvin A. George, and Arkaprava Basu. 2020. ScoRD: A Scoped Race Detector for GPUs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 1036–1049. https://doi.org/10.1109/ISCA45697.2020.00088

[34] 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

[35] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[36] 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

[37] 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.

[38] 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

[39] Zhen Lin, Mohammad Alshboul, Yan Solihin, and Huiyang Zhou. 2019. Exploring Memory Persistency Models for GPUs. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 311–323. https://doi.org/10.1109/PACT.2019.00032

[40] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. PMFuzz: Test Case Generation for Persistent Memory Programs. 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, 487–502. https://doi.org/10.1145/3445814.3446691

[41] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. *Cross-Failure Bug Detection in Persistent Memory Programs*. Association for Computing Machinery, New York, NY, USA, 1187–1202. https://doi.org/10.1145/3373376.3378452

[42] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. 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, 411–425. https://doi.org/10.1145/3297858.3304015

[43] 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

[44] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. 2020. EMOGI: Efficient Memory-Access for out-of-Memory Graph-Traversal in GPUs. *Proc. VLDB Endow.* 14, 2 (oct 2020), 114–127. https://doi.org/10.14778/3425879.3425883

[45] 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

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

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

[48] NVIDIA. 2020. CUDA Toolkit 11.2 Downloads. https://developer.nvidia.com/cuda-11.2.0-download-archive.

[49] NVIDIA. 2021. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2021-11-15.

[50] NVIDIA. 2021. Parallel Thread Execution ISA Version 7.5. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html. Accessed: 2021-11-15.

[51] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 336–349. https://doi.org/10.1109/HPCA.2018.00037

[52] 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* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 142–156. https://doi.org/10.1145/3503222.3507758

[53] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2023. "Scoped Buffered Persistency Model for GPUs". https://doi.org/10.5281/zenodo.7306303.

[54] 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.

[55] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371079

[56] RocksDB. 2021. RocksDB. https://rocksdb.org/.

[57] Andy Rudoff. 2021. Persistent Memory on CXL. https://www.snia.org/educational-library/persistent-memory-cxl-2021. Accessed: 2021-11-15.

[58] Samsung. August 2022. Samsung Memory-Semantic CXL SSD. https://tinyurl.com/3daezum2.

[59] Steve Scargall. 2020. *Programming persistent memory: A comprehensive guide for developers*. Springer Nature.

[60] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 178–190. https://doi.org/10.1145/3123939.3124539

[61] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'20)*. USENIX Association, USA, 169–182.

[62] 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, USA, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[63] 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