A Case for Scoped Persist Barriers in GPUs

Dibakar Gope*
ARM Research
dibakar.gope@arm.com

Arkaprava Basu
AMD Research
arkaprava.basu@amd.com

Sooraj Puthoor
AMD Research
sooraj.puthoor@amd.com

Mitesh Meswani†
ARM
mitesh.meswani@arm.com

Abstract

Two key trends in computing are evident — emergence of GPU as a first-class compute element and emergence of byte-addressable nonvolatile memory technologies (NVRAM) as DRAM-supplement. GPUs and NVRAMs are likely to coexist in future systems. However, previous works have either focused on GPUs or on NVRAMs, in isolation. In this work, we investigate the enhancements necessary for a GPU to efficiently and correctly manipulate NVRAM-resident persistent data structures.

Specifically, we find that previously proposed CPU-centric persist barriers fall short for GPUs. We thus introduce the concept of *scoped persist barriers* that aligns with the hierarchical programming framework of GPUs. Scoped persist barriers enable GPU programmers to express which execution group (a.k.a., *scope*) a given persist barrier applies to. We demonstrate that: (1) use of *narrower* scope than algorithmically-required can lead to inconsistency of persistent data structure, and (2) use of *wider* scope than necessary leads to significant performance loss (e.g., 25% or more). Therefore, a future GPU can benefit from persist barriers with different scopes.

CCS Concepts • **Computer systems organization → Single instruction, multiple data**; • **Hardware → Non-volatile memory**;

ACM Reference Format:
Dibakar Gope, Arkaprava Basu, Sooraj Puthoor, and Mitesh Meswani. 2018. A Case for Scoped Persist Barriers in GPUs. In *GPGPU-11: General Purpose GPUs, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180270.3180275

1 Introduction

Emerging non-volatile random access memory (NVRAM) technologies, such as phase change memory (PCM), spin-transfer torque RAM (STT-RAM) and memristors, promise to circumvent the technology scaling challenges of DRAM [10, 23, 28, 44, 63]. These non-volatile, byte-addressable technologies are expected to have access latencies close to that of DRAM. This makes NVRAMs suitable to be attached to the memory bus and be accessed via load/store interface at the word granularity [36, 47]. This could provide a tectonic shift in the computing by potentially collapsing long-held distinction between the memory and the storage [59, 65]. In short, NVRAM is expected find usefulness across databases, big data analytics, and machine learning [18, 19, 27]. It is also expected to enable a new class of applications that can store large amounts of pointer-rich, user-defined data structures directly in a nonvolatile memory [66].

In an orthogonal trend, GPUs are being widely used across many application domains including deep learning, weather modeling, data analytics, computer-aided-design, oil and gas exploration, medical imaging and, computational finance [45]. GPUs could provide large improvement in both performance and energy efficiency for these highly parallel applications. Today, at least 66 of world's top 500 supercomputers use GPUs [58], and at the same time, GPUs are being widely deployed in clouds [5, 30]. Importantly, GPUs are emerging as a first-class compute citizen with industry-promoted standards like Heterogeneous System Architecture (HSA) [3] mandating cache-coherent shared virtual memory across CPU and GPU to ease programming.

These two trends suggest that future systems will have both GPUs and NVRAM. In fact, we already observe commercial inclination to put GPUs and non-volatile memory together. AMD recently announced Radeon™ Pro SSG GPUs that tightly integrate GPUs and non-volatile fast SSD (solid-state-disk) over a high-performance NVMe link [6, 7]. AMD expects such tighter integration will allow GPUs crunch much larger data-sets [7]. While this commercially available GPU is currently attached to a SSD (Samsung® Pro 950), and not to NVRAM, it is reasonable to expect that NVRAMs will make their way into such systems as the technology matures for commercial productions.

In such systems with both GPU and NVRAM, an important question is *how a GPU application can correctly and efficiently manipulate NVRAM-resident persistent data structures?* Specifically, what are the implications of NVRAMs on the GPU design? Do GPU applications need to change in order to manipulate persistent data structures? In this work, we investigate these questions in detail.

A key challenge in manipulating NVRAM-resident persistent data structures via a fine-grain load-store interface is how to ensure consistency of those structures across crashes and power failures [15, 59, 65, 66]. Maintaining consistency allows such data structures to be restored to meaningful states after a crash. This property is called the *recoverability*. Recoverability can be ensured by observing a set of algorithmically-required *happens-before orders* among the writes to NVRAM. For example, a program to insert new entries in a key-value store must ensure that value persists in the NVRAM before the update to the key is made persistent. Otherwise, a crash can leave the key pointing to unavailable data.

*Author contributed as an intern at AMD Research.
†Author contributed while in AMD Research.

GPGPU-11, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5647-3/18/02…$15.00
https://doi.org/10.1145/3180270.3180275

Prior research has explored how CPU applications and CPU architectures need to evolve to ensure the recoverability of persistent data structures [15, 59, 65, 66]. This body of work introduced several variants of *persist barriers* (a.k.a, epoch barriers) to enforce programmer desired happens-before orders among the writes to NVRAM (referred as *persist dependencies*) [17, 26, 31, 53, 60]. A persist barrier ensures that writes to NVRAM appearing before the barrier *persists* prior to those appearing after it. We refer the act of durably writing the value of a store to NVRAM as *persist*.

Unfortunately, such CPU-centric persist barriers do not immediately extend to GPUs. To manage the massive parallelism of a GPU, programming languages like OpenCL™ and CUDA™ decompose work in a hierarchy of execution groups such as workitems (a.k.a, GPU thread), wavefronts (a group of 32-64 workitems), workgroups (a group of wavefronts) and kernel (a group of workgroup) [1, 3, 4]. Correspondingly, the underlying GPU hardware also arranges its compute resources in a hierarchy of a lanes, SIMD units, Compute Units (CUs). However, previously proposed persist barriers carry inherent global semantics that are adequate for CPUs but are *not semantically rich enough* capture the hierarchy in GPU's execution model. For example, a traditional persist barrier does not allow a GPU programmer to *specify the execution group* whose writes to the NVRAM need to be ordered. If a persist barrier is to order persists only at the fine granularity of a workitem then it can be insufficient to express algorithmically-required persist dependencies in GPU's hierarchical programming model (Sections 4 and 5). On the other extreme, globally ordering persist dependencies at a persist barrier across thousands of concurrent workitems (threads) is often unnecessary and incurs significant, yet avoidable performance loss.

To address this fundamental shortcoming, we introduce a new class of persist barriers for GPUs, called *scoped persist barriers*. Specifically, we define three different scopes for persist barriers that are aligned with execution hierarchy of modern GPUs — workitem, workgroup, and kernel scope. A persist barrier with workgroup scope ensures that writes to NVRAM by all workitems (a.k.a. GPU threads) of a given workgroup that appear before the barrier persist before those appearing after the barrier in that workgroup. Similarly, a persist barrier with workitem scope guarantees ordering for the writes to NVRAM only from a given workitem (narrower scope than workgroup), while a kernel scope barrier enforces ordering across writes to NVRAM from all workitems in a GPU kernel (wider global scope). We demonstrate that use of a narrower scope than algorithmically required can lead to inconsistency in persistent data structures while using wider scope than necessary may ease programming but lowers performance.

In summary, we make the following contributions.

- We introduce the scoped persist barriers for GPUs to manipulate persistent data structures.
- We show that use of insufficient scope in persist barriers can introduce inconsistency in persistent data structures.
- We show that persist barriers with wider-than-necessary scope can degrade performance by 25% or more.

2 Background

A background on NVRAM technologies, proposals for CPUs to ensure recoverability of persistent data, and GPU's execution and synchronization model are helpful to appreciate this work.

2.1 Nonvolatile Memory Technology

As DRAM scaling nears its limit [10, 23, 28, 63], several byte-addressable non-volatile memory (NVRAM) technologies such as phase-change (PCM), spin-transfer torque magnetic (STT-MRAM), resistive (RRAM) and 3D NAND Flash memory are emerging as possible DRAM-supplement. These NVRAM technologies promise significantly higher density than DRAM. At the same time, they can reside on the memory bus alongside the DRAM. While jury is still out on which NVRAM technology will emerge as DRAM-supplement of choice, it is highly likely that NVRAM will be accessible through load/store interface like the DRAM in future systems.

2.2 Persistency Models and Persist Barrier in CPU

Ensuring recoverability of data structures residing in the NVRAM across crashes requires observing a set of happens-before ordering among the persist operations (also referred as *persist dependencies*). However, writes to the NVRAM, like those to the DRAM, are cached and are *not* necessarily written back from the cache in the execution order of corresponding stores. To reason about the ordering of persist operations, researchers have proposed various *persistency models* [31, 36, 47], similar in spirit to the memory consistency models that govern the visibility of data in multi-threaded systems. For example, the *strict persistency model* mandates a store to persist as soon as it becomes visible as per the memory consistency model. Since the latency to persist a data in the NVRAM can be significant such stringent ordering of persists can severely limit performance [31, 47]. *Epoch persistency model* relaxes this to order persists only at the granularity of an *epoch* which is a contiguous group of instructions demarcated by a *persist barrier* (a.k.a *epoch barrier*) [17, 26, 31, 53]. This model requires stores to NVRAM appearing before a persist barrier to persist before instructions appearing after the barrier execute [31]. A variation of this model, called *buffered epoch persistency*, relaxes the need to persist stores at the barrier and instead, only requires that stores to NVRAM appearing before a persist barrier to persist before those appearing after [1]. We omit discussion on *strand persistency* and *buffered strict persistency* model for space constraints [31, 47].

2.3 GPU and Scoped Synchronization

A compute unit (CU) is the basic computational block of a GPU. Each CU consists of multiple SIMD execution units that executes instructions in lock-step fashion. The SIMD width is vendor dependent and often ranges from 32 to 64. A GPU itself consists of multiple such CUs (typically between 8 to 64).

To keep such parallelism tractable, GPU programming languages such as OpenCL [4] and CUDA [1] divide the threads in a hierarchy of execution groups. A workitem in this hierarchy is akin to a CPU thread and is the smallest execution entity that runs on a single lane of a SIMD unit. A group of workitems that executes in lock-step fashion forms a wavefront. Wavefront is the smallest hardware-scheduled unit of work in a GPU. The number of workitems in a wavefront is determined by the hardware's SIMD width. A group of wavefronts forms a workgroup and is programmer-visible, unlike wavefront. All workitems in a given workgroup executes in the

[1] Slightly different naming conventions are used by Joshi et al. [31] and by Pelley et al. and Kolli et al. [36, 47]. The definition of *buffered epoch persistency* by Joshi et al. [31] is similar to that of *epoch persistency* by Pelley et al. and Kolli et al. [36, 47]. The *eager sync* with cache block flush in Kolli et al.'s work is similar to Joshi et al.'s *epoch persistency*. In this work, we generally follow Joshi et al.'s naming convention

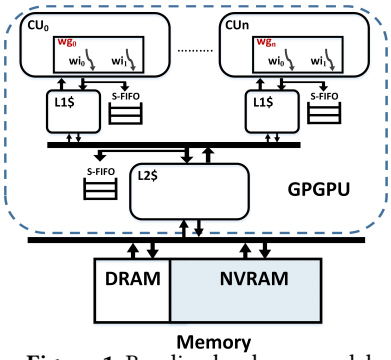


Figure 1. Baseline hardware model.

same CU and thus can communicate among them via local caches and/or scratchpad in a given CU. A GPU kernel (a.k.a. GPU program) consists of many such workgroups.

Global synchronization across thousands of workitems in a GPU is costly. It is also often unnecessary in a hierarchical programming framework. GPU programming languages thus introduced scoped synchronization operations that affect workitems only within a given *scope* or an execution group [3, 4]. For example, a synchronization operation with workgroup scope does not necessarily make an update globally visible but only guarantees visibility across the workitems belonging to that workgroup. Sub-group, kernel, and device scopes are other common examples of such scopes found in programing specifications [3]. Scoped synchronization enables a programmer to write a correct program without always paying the cost of global synchronization across all workitems.

3 Baseline System Model

Hardware model: As shown in Figure 1 the GPU contains multiple CUs. Each SIMD unit in a CU can execute 64 GPU threads or workitems (wi_n) in a lock-step fashion. All workitems belonging to a workgroup (wg_n) execute in the same CU. Each CU has its own private L1 cache. L2 cache is shared across CUs and is inclusive of the contents of L1 caches. Both DRAM and NVRAM are accessible through load/store interface at granularity of a word. Caches contain data from both DRAM and NVRAM. L1 cache is write-combining and L2 cache is write-back. There are separate memory controllers (not shown) for DRAM and NVRAM. Table 2 details the configuration of our baseline system.

The GPU supports scoped synchronizations as specified in the Heterogeneous System Architecture specification [3] and enforces memory consistency model similar to heterogeneous-race-free (HRF) model [24]. To make GPU synchronization operation faster and realistic, the baseline is equipped with synchronization FIFO (S-FIFO) proposed by Hechtman et al. [22]. On a release operation, GPU flushes dirty cache lines to the next level of caches. S-FIFOs make such scoped release operations faster by tracking outstanding writes between two consecutive release operations and avoids cache walks to find dirty blocks.

Software model: The physical address space of the system is comprised of two contiguous address ranges – one for the DRAM and the other for the NVRAM. All memory allocations from an application are satisfied from the DRAM *unless* instructed otherwise. Specifically, an application invokes special *pmalloc* to request allocation on the NVRAM. This software model is similar to that proposed by Volos et al. [60].

4 Motivation

GPUs have revolutionized a wide array of application domains – from databases, deep learning to medical imaging [45]. Previous research has also explored how NVRAM can significantly benefit applications across broad spectrum that needs persistence (e.g., [27, 59]). A significant section of overlapping application domains can benefit from both GPU and NVRAM.

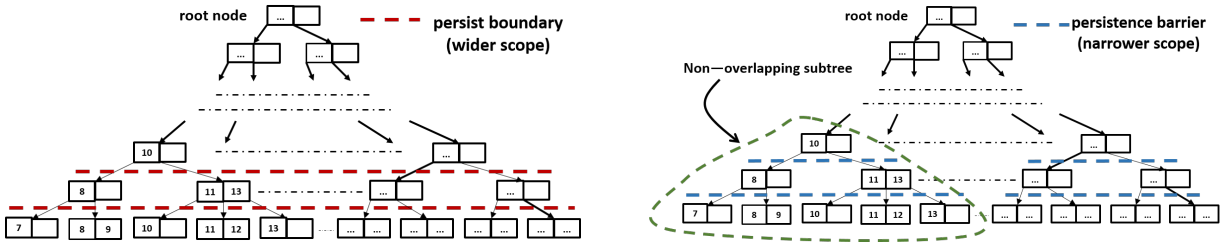
As an example, let us consider databases. GPUs can accelerate database's query processing by 20-260× [11, 34, 57]. Orthogonally, databases care about persistence and past research has demonstrated ability of NVRAM to revolutionize databases by blurring the distinction between memory and storage [9, 33, 48]. The ability to simultaneously leverage both GPUs and the NVRAM can be a game-changer for databases. However, how GPUs can leverage NVRAMs is still an open question.

A key design aspect of a system with NVRAM is how to efficiently enforce persist dependencies to ensure consistency of NVRAM-resident data structures across failures. Past work introduced persist barrier instructions to enforce such persist dependencies in a CPU [17, 31]. Unfortunately, such CPU-centric persist barriers fall short for GPUs. Unlike CPUs, GPUs deploy thousands of concurrently executing workitems (threads) that are organized in a hierarchy of execution groups. The realization that global communication and synchronization in such architecture is often neither necessary nor efficient has led GPU programming languages to introduce scoped synchronization operations [1, 3, 4, 24]. A similar reasoning suggests that tracking and ordering of writes to NVRAM across thousands of workitems to globally enforce a persist barrier is inefficient and often unnecessary.

We motivate the need to extend semantics of a persist barrier with scopes (a.k.a. execution group in GPU's programming hierarchy) for GPUs using an example of parallel insertion of keys into a NVRAM-resident `B+tree`. A `B+tree` is a height-balanced tree that is widely used for indexing in databases and file systems [62]. In a `B+tree`, all keys are stored in the leaves while non-leaf nodes act as the index. Height-balancing ensures that the number of non-leaf nodes along any path from the root to any leaf node remain equal. This guarantees a bounded number of lookups to find any key in a tree of a given size. However, after insertion of a new key, non-leaf nodes *may* need to be updated to keep the tree height-balanced.

In our example, millions of new keys are concurrently inserted into a persistent `B+tree` from the GPU. Each workitem is responsible for inserting one key. The insertion starts with inserting keys at the leaf nodes concurrently by workitems. Updates are then propagated towards the root, as necessary, to re-balance the tree. *Ensuring consistency of a persistent B+tree under such operation requires that the update to a child node persist before persisting updates to its parent.* These persist dependencies result in *persist boundaries* across which persist operations cannot be reordered.

One possible set of persist boundaries are shown by the red lines (thick dashed) in Figure 2a. At each such persist boundary, all workitems across all workgroups in a GPU kernel tasked to insert the keys need to first synchronize for *functional correctness*. It then needs to ensure that *updates to all nodes at a given level of B+tree persists before persisting any updates to nodes at a higher level* (*closer to root*). The synchronization is required even in a system with only DRAM to ensure correct updates but the ordering of persist operations is necessary only for a persistent `B+tree`. This



(a) Insertion with coarse-grain (kernel) persist boundaries. **(b)** Insertion with finer-grain (workgroup) persist boundaries.

Figure 2. Example: Two possible parallel insertion of keys into B+tree from GPU.

implementation of the insertion requires one to track all stores (writes) to NVRAM by all workitems in the kernel and ensure persist dependencies are observed across all such writes. It is easy to realize that ensuring such persist dependencies across stores from thousands of GPU workitems adds significant overhead.

The above-mentioned implementation is correct but conservative. Figure 2b shows an alternative implementation with a finer-grained persist boundaries (blue thick-dashed lines). The key idea here is to distribute the keys to be inserted among different workgroups such that each workgroup updates non-overlapping subtrees. In this way, updates to a subtree by a given workgroup can persist independent of persist operations from other concurrent workgroups. Within a given workgroup, updates to a given level of the tree should persist before updates to the higher level persist. Here, only the persist dependencies across writes to NVRAM from workitems within *the same workgroup* need to be tracked and enforced. This incurs lesser overhead and enables more concurrency.

While enforcing persist dependencies at finer granularity may improve performance, a granularity finer than that is required by the algorithm can break recoverability. For example, enforcing persist dependencies at workitem granularity instead, will not cover all workitems that concurrently update child nodes of a common parent node and thus, can leave B+tree inconsistent in the NVRAM.

Summary: Enforcing ordering among persist operations *globally* across thousands of concurrent workitems in a GPU may be unnecessary. A programmer could more efficiently enforce ordering among persist operations at a finer granularity in GPU's programming hierarchy (e.g., a workgroup). This however, requires the GPU programming framework and the hardware to expose ability to enforce persist dependencies at different levels of GPU's programming hierarchy (a.k.a. scopes).

5 Scoped persist barriers

Driven by above observations, we introduce the *scoped persist barriers* to enable GPU programmers to efficiently and correctly manipulate persistent data structures. Simply put, scoped persist barriers allow programmers to specify which execution group or scope a persist barrier applies to. In other words, a scoped persist barrier defines persist dependencies among *writes to NVRAM from which execution group* should be respected. Specifically, we define three different scoped persist barriers as follows.

Workitem persist barrier (p_barrier_wi): A workitem persist barrier ensures that stores to NVRAM from a given workitem that appears before the barrier in the program order [2] persists before the stores after the barrier. Stores to persistent memory by other workitems may not be ordered by this barrier. More formally, we define a p_barrier_wi as:

$$M_a^{ijk} \leq_{prog} p\_barrier\_wi \leq_{prog} M_b^{mno} \rightarrow M_a^{ijk} \leq_{persist} M_b^{mno} \text{ if } i{=}m,\ j{=}n \text{ and } k{=}o \quad (1)$$

Here, M_a^{ijk} is a load or store from a workitem i of workgroup j and kernel k for an address a. $M_a^{ijk} \leq_{prog} p\_barrier\_wi \leq_{prog} M_b^{mno}$ means that M_a^{ijk} appears before M_b^{mno} in the program order of a GPU kernel and $M_a^{ijk} \leq_{persist} M_b^{mno}$ means that M_a^{ijk} persists no later than M_b^{mno}.

As an example use case, if each workitem on the GPU inserts non-overlapping key-value pair in a persistent key-value store then ordering between persistent updates to the value and the key can be enforced by workitem persist barrier.

Workgroup persist barrier (p_barrier_wg): A workgroup persist barrier ensures that no stores from any workitem in a given workgroup past the barrier in the program order can persist until all prior stores from all workitems of that workgroup persist. Following the same notation as above, p_barrier_wg is specified as:

$$M_a^{ijk} \leq_{prog} p\_barrier\_wg \leq_{prog} M_b^{mno} \rightarrow M_a^{ijk} \leq_{persist} M_b^{mno}\ \forall i \text{ and } m, \text{ if } j{=}n \text{ and } k{=}o \quad (2)$$

Persists operations from *distinct* workgroups are not necessarily ordered by this barrier. Workgroup-scoped persist barrier is required when workitems within a given workgroup cooperate to perform an update to a persistent data structure but updates by different workgroups are independent (e.g., updates within independent subtrees).

Kernel persist barrier (p_barrier_kr): A kernel persist barrier ensures that no stores from any workgroup in a given kernel past the barrier in program order can persist until all stores appearing before the barrier from that kernel are made persistent. Following the same notations, p_barrier_kr can be specified as follows:

$$M_a^{ijk} \leq_{prog} p\_barrier\_kr \leq_{prog} M_b^{mno} \rightarrow M_a^{ijk} \leq_{persist} M_b^{mno}\ \forall i \text{ and } m,\ j \text{ and } n, \text{ if } k{=}o \quad (3)$$

Kernel barrier is useful when work of multiple workgroups semantically overlap. Kernel-scoped persist barriers are useful for *irregular applications* where the producer and consumer of a data are determined.

5.1 Example use: Key insertion into B+tree

Next, we show different scoped persist barriers can be used to implement the above-mentioned parallel insertion of keys into a

[2] The program order in a GPU program follows the same semantics as that in a CPU program. Specifically, in a GPU program (kernel) if an instruction i appears before another instruction j then the instruction i precedes instruction j in the program order

Algorithm 1 Parallel B+tree Insertion (Enforcing correct persist ordering using kernel-scoped p_barrier)

Input: *B+tree, keys_to_insert*

```
1:  sortAscending(keys_to_insert)
2:  /* Launch B+treeInsert kernel */
3:  for i ← 0 .. n_keys do in parallel
4:      leaf_node ←findLeafNode(keys_to_insert[i])/* traverse to
        leaf node for keys_to_insert[i] */
5:      leaf_node.new_keys.append(keys_to_insert[i])
6:  /*new_keys stores the keys to be added to a node*/
7:  end for
8:  for cur_depth ← depth_leaf_nodes to root do
9:      nodes_cur_depth ←set of nodes at cur_depth
10:     for each node in nodes_cur_depth do in parallel
11:         if node.new_keys.size() ≤ node.availableSpace()
        then
12:             updateNode()/* update keys/pointers */
13:         else
14:             splitAndUpdateNode()
15:             node.parent.new_keys.append(new generated keys
        from splitting child nodes)
16:             node.parent.new_child_nodes.append(pointers to
        added child nodes)
17:         end if
18:     end for
19:     sync_barrier_kr
20:     p_barrier_kr
21: end for
```

persistent `B+tree`. We then qualitatively demonstrate the trade-offs between different scopes.

Algorithm 1 depicts pseudo-code for implementing parallel `B+tree` insertion that enforces persist dependencies at the coarse kernel granularity as shown in Figure 2a. The algorithm takes two arguments – the B+ tree data structure and the set of keys to insert. For easier insertion we first sort the keys in ascending order on the host CPU (line 1). Then, a GPU kernel is launched to find (line 3-7) the correct leaf node where a given key is to be inserted. Lines 8-21 depict the process of inserting keys concurrently from a GPU. Insertion starts at the leaf level and iteratively propagates updates for re-balancing the tree towards the root node. Each iteration of the outer loop performs necessary updates at a given level of the tree (line 8-21). At the leaf level, each workitem is responsible for updating one node with newly inserted keys. The non-leaf nodes may split if the number of children of a node is greater than order of the B+tree(line 14-16). After updating nodes at a given level all workitems synchronize for functional correctness (line 19) [3]. Then persist operations for these updates are ordered before the updates from next level using a kernel-scoped persist barrier (*p_barrier_kr*).

As discussed in Section 4, the above-mentioned implementation is correct but highly conservative in enforcing persist orderings. Algorithm 2 describes an alternative implementation that enforces ordering among persist operations at a finer granularity of work-groups as depicted in Figure 2b. It distributes keys such that each workgroup updates non-overlapping subtrees.

In Algorithm 2, the number of workgroups is determined by taking into account the number of keys to insert and the size of each workgroup (line 2-4). It then determines a level (*d*) where

[3]Kernel synchronization barrier instruction is not directly available in current hardware but can be enabled through preemption in modern GPUs. Details in Section 6.

Algorithm 2 Parallel B+tree Insertion (Enforcing correct persist ordering using mostly wg-scoped p_barrier)

Input: *B+tree, keys_to_insert*

```
1:  sortAscending(keys_to_insert)
2:  n_wgs_est ← n_keys/wg_size/*n_keys = #keys_to_insert,
    wg_size = #work-items in a work-group*/
3:  d ←findReqdDepth(n_wgs_est)/*Find depth of B+tree where
    n_internal_nodes >= n_wgs_est*/
4:  n_wgs_decided ← n_internal_nodes
5:  keys_allotted_per_wg[n_wgs_decided] ←
    distributeKeysAmongSubtrees(keys_to_insert)
6:  /* Launch B+treeInsert kernel */
7:  wg_id ← get_global_id(0)/wg_size
8:  for each i in keys_allotted_per_wg[wg_id] do in parallel
9:      leaf_node ←findLeafNode(keys_to_insert[i])
10:     leaf_node.new_keys.append(keys_to_insert[i])
11: end foreach
12: for cur_depth ← depth_leaf_nodes to d do
13:     nodes_cur_depth ←set of nodes at cur_depth/*nodes at
    cur_depth of the sub-tree allotted to work-group wg_id*/
14:     for each node in nodes_cur_depth do in parallel
15:         if node.new_keys.size() ≤ node.availableSpace()
        then
16:             updateNode()
17:         else
18:             splitAndUpdateNode()
19:             node.parent.new_keys.append(new generated keys
        from splitting child nodes)
20:             node.parent.new_child_nodes.append(pointers to
        added child nodes)
21:         end if
22:     end for
23:     sync_barrier_wg
24:     p_barrier_wg
25: end for
26: for cur_depth ← d to root do
27:     nodes_cur_depth ←set of nodes at cur_depth
28:     for each node in nodes_cur_depth do in parallel
29:         /* same as line 15 to line 21 above */
30:     end for
31:     sync_barrier_kr
32:     p_barrier_kr
33: end for
```

the independent subtrees are rooted. The value of *d* is determined based on the number of workgroups and height of the tree. Keys are then distributed to workgroups (line 5). Each workgroup then independently inserts its subset of keys into its subtree and iterates to re-balance the subtree (line 7-22), in a manner similar to that in Algorithm 1. It then uses workgroup scoped synchronization for functional correctness and uses *workgroup scoped persist barrier* to enforce ordering among persists by workitems within a workgroup at each level of its subtree (line 24). Persists from different work-groups are not ordered. Finally, closer to root, above the level *d*, enough parallelism to employ multiple workgroups cease to exist. Thus, beyond level *d*, a kernel-scoped persist barrier is used to enforce needed ordering (line 31-32).

It is easy to notice that Algorithm 2 imposes less persist dependencies by using finer grain persist barriers. However, it is also visibly more complex compared to Algorithm 2. Our quantitative evaluation shows that an implementation using such finer grain persist barrier provides significant performance benefit relative to

Table 1. Trade-offs between scoped p_barriers

p_barrier	Performance	Programm--ability	Recover--ability
wi-scoped	Good (regular) Poor (irregular)	High effort	Fine-grain
wg-scoped	Good	Moderate effort	Fine-grain
kr-scoped	Poor	Less effort	Coarse-grain

an implementation using coarse-grain persist barrier (Section 7). This demonstrates the trade-off between programmability and performance of using different scopes.

Note that, there exists no meaningful implementation of a parallel B+tree insertion using only workitem-scoped persist barriers. Like any other irregular GPU kernels, interactions between workitems here are unpredictable. If p_barrier_wi used, each insertion will need to hold locks and thus not meaningful on a GPU. However, regular GPU kernel like updating a simple hash-table could make use of workitem scoped persist barriers.

Table 1 qualitatively captures the tradeoffs among differently scoped persist barriers. A programmer can use all three scoped persist barriers in an application.

6 Hardware support for persist barriers

Figure 3 shows a modified GPU hardware for *one possible implementation* of the scoped persist barriers. Here the only additional hardware is Non-Volatile-Write FIFOs (NVW-FIFOs). NVW-FIFOs are themselves made out of *volatile* SRAM. Their purpose is to track and order writes (committed stores) to NVRAM (henceforth, referred as "NV writes") that are held in a cache and are yet to be persisted. Every GPU cache is thus augmented with a NVW-FIFO.

A NVW-FIFO typically holds between 16 to 64 eight-byte entries for NV writes held in its corresponding cache. Each time a cache block containing data from the NVRAM is updated, an entry is added in the corresponding NVW-FIFO. Each entry contains the physical address of the updated data. When a scoped persist barrier is executed, a *p_flush* marker is appended to the NVW-FIFO(s) corresponding to the cache(s) that may contain NV writes from the scope of that persist barrier. This p_flush marker then flows through NVW-FIFO(s), ordering any NV writes in its path to the NVRAM. In short, NVW-FIFOs help to order NV writes for a scoped persist barrier without necessitating full cache lookups to find which cache blocks may need to be flushed to the NVRAM. Next, we describe how NVW-FIFOs are used to implement scoped persist barriers.

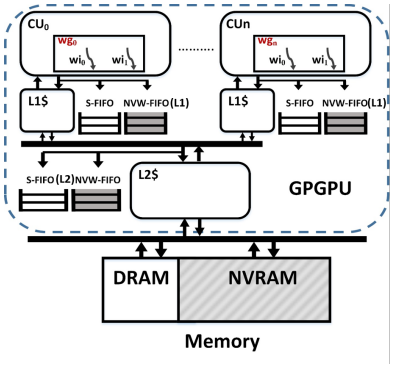


Figure 3. GPU hardware with proposed additions (shaded).

6.1 Implementing scoped persist barriers

We detail *one possible implementation* of different scoped persist barriers introduced in Section 5. While persist barriers require to only order NV writes in its scope, the following implementation (henceforth, referred as "proposed implementation") immediately persists those NV writes to NVRAM, as well. This is correct but conservative implementation that we empirically found minimally affects performance but greatly simplifies the hardware.

workitem persist barrier(p_barrier_wi): (1) On execution of a p_barrier_wi instruction a p_flush marker is appended to the L1 NVW-FIFO corresponding to the SIMD unit where the issuing workitem is executing. This also stalls the execution of that workitem. (2) This p_flush marker starts to dequeue entries from the head of the L1 NVW-FIFO and flush corresponding cache blocks from the L1 cache to the L2 cache. Correspondingly, the dequeued L1 NVW-FIFO entries are appended to the L2 NVW-FIFO. (3) When the p_flush marker reaches the head of the L1 NVW-FIFO it is dequeued and appended to the L2 NVW-FIFO. (4) The L2 NVW-FIFO entries are then dequeued in similar fashion and corresponding dirty L2 cache blocks are written back to the NVRAM. (5) The L2 cache controller then waits for acknowledgement of the completion of all writebacks to the NVRAM. (6) Once the L2 cache controller receives the acknowledgement, it forwards it to the L1 cache controller which in turn forwards to the concerned SIMD unit. This un-stalls the workitem. Since smallest unit of scheduling in a GPU is an wavefront, the entire wavefront stalls till p_barrier_wi completes. However, other ready wavefronts can continue execution.

workgroup persist barrier (p_barrier_wg): (1) In this implementation, a workgroup scoped persist barrier first requires all workitems of a given workgroup to reach the barrier. Alternative implementation described later relaxes this. (2) A p_flush marker is appended to the L1 NVW-FIFO corresponding to the CU that executes the issuing workgroup and stalls execution of that workgroup. Steps (3) - (6) are same as steps (2) - (5) of workitem scoped persist barrier. (7) The L2 cache controller forwards completion acknowledgement to L1 cache controller and then to the concerned CU. The stalled workgroup then resumes execution.

kernel persist barrier (p_barrier_kr): (1) In this implementation, a kernel scoped persist barrier requires all workitems in the kernel to reach the barrier and then persists NV writes from all those workitems [4]. (2) p_flush markers are appended to every L1 NVW-FIFO since any L1 cache can contain NV writes form the kernel. Steps (3) - (6) are similar as to steps (2) - (5) of workitem scoped persist barrier but repeated for every L1 NVW-FIFO. (7) L2 cache controller(s) forwards completion acknowledgments to L1 cache controllers and then finally on to the GPU's hardware scheduler (called Command Processor or CP) that schedules kernels on the GPU. Finally, the kernel is un-stalled.

Alternative implementation: The above-mentioned implementation of scoped persist barriers persists NV writes at the barrier

[4]There is no instruction immediately available in the current GPUs to wait for all workitems in a kernel due to possibility of livelock. However, it can be realized through preemptions of work in the GPU. In fact, the GPU in AMD "Carrizo" heterogeneous processor supports preemption [8] and industry-promoted HSA specification mandates support for context-switching and preemption in HSA-compliant GPUs [25]. We implemented the wait for all workitems in a kernel to reach the p_barrier_kr using the ability to preempt and context switch.

```
1:__global atomic<int> A = {0}; __global atomic<int> B = {0};
2:
3: if (current wi == wi1 in workgroup wgX) {
4:  X = 1; Y = 1; // X and Y are writes to volatile and NV memory respectively
5:  p_barrier_wg;
6:  A.store(1, mo_sc, wg_scope);    // release
7:}
8:              else if (current wi == wi2 in wgX) {
9:                 while(!A.load(mo_sc, wg_scope));    // acquire
10:                   int R2 = Y;                      // R2 will get value
11:                   B.store(1, mo_sc, dev_scope);    // release
12:             }
13:                                  else if (current wi == wi3 in wgY (Note !wgX)) {
14:                                    while(!B.load(mo_sc, dev_scope));  // acquire
15:                                    int R3 = X;                        // R3 is undefined
16:                                    int R4 = Y;                        // R4 is undefined
17:                                  }
```

(abbreviation) wi → workitem, dev → device, wg → workgroup, mo_sc → memory_order_seq_cst

Figure 4. An example to show scoped synchronization and scoped persist barrier interacts.

while *only* ordering is required by definition. It is a correct but conservative implementation of persist barriers defined in Section 5.

Alternatively, it is possible to enforce only the ordering among persist operations by extending GPU caches in a similar manner as implemented in *BPFS* by Condit et al. (Section 4.4, [17]). Specifically, each cache block would need to be extended with a persist barrier identifier and its scope. The cache replacement policy then needs to be extended such that eviction of a cache block first triggers evictions of other blocks containing updates that need to be persisted before the given cache block as per the semantics of the persist barrier (identifiable by comparing the identifier of persist barrier and its scope). Then it would not require to wait for workitems in a given scope to first reach the persist barrier, as in step (1) for workgroup and kernel persist barriers.

However, our evaluation suggests that performance overhead of conservatively persisting data to NVRAM is limited to 5% due to superior latency tolerance of GPUs. Thus, significant additional hardware for the alternative implementation could not be justified.

6.2 Interactions between scoped synchronizations and scoped persist barriers

Both scoped synchronizations (discussed in Section 2) and the proposed scoped persist barriers, leverage GPU's execution hierarchy to avoid unnecessary global communications and orderings. However, they are complementary to each other and both can exist in the same program without altering semantics of each other. Scoped synchronizations provide same visibility guarantees for a data in the cache hierarchy irrespective of whether it is from DRAM or from NVRAM. Scoped persist barriers, in contrast, provide ordering guarantees for writes to NVRAM.

We use an example pseudo-code for a GPU kernel in Figure 4 to show that (1) the visibility guarantees provided by scoped synchronization operations (acquire/release) are not altered by an intervening persist barrier, (2) a *racey* program remains racey even after use of a persist barrier. Lines 3 -7 in Figure 4 are executed only by the workitem 'wi1' of workgroup 'wgX'. It writes to one variable in DRAM ('X') and one in NVRAM ('Y'). This is followed by a workgroup scoped persist barrier. It then performs a workgroup scoped release (line 6) which is paired with a scoped acquire operation (line 9) in workitem 'wi2' of the same workgroup (synchronization). Because of this scoped synchronization the 'R2' is guaranteed to see the updated value. Intervening persist barrier in 'wi1' (line 5) does not alter this guarantee. Next, 'wi3' does *not* synchronize with the same scope with 'wi1' and as per HRF-direct memory consistency model there is no guarantee if R3 and R4 will observe updated values [24]. The guarantees provided by definition of p_barrier_wg (Section 5) do not affect that as well. Note that a workgroup-scoped persist barrier in our proposed example implementation will make update to 'Y' globally visible while persisting it in NVRAM. The alternative implementation discussed in Section 6 may not make update to 'Y' visible though. However, the act of pro-actively making a store visible does not alter semantics of scoped synchronization since a release operation defines the latest point in the execution by which an update must be visible.

In summary, scoped synchronization and scoped persist barriers serve different purposes and they are not substitute for one another.

Table 2. Simulation Configuration

Compute Units	4 CUs running at 1GHz Max. 40 wavefronts (64-wide SIMD) per CU
Memory Hierarchy	Data Cache: 16-way, 16KB, 64B line, Instr. Cache: 8-way, 32KB, 64B line L2 Cache: 16-way, 512KB, 64B line, DRAM read/write latency: 25 ns, NVRAM read latency: 75 ns, write latency 250 ns
S-FIFO and NVW-FIFO	L1/L2 S-FIFO: 16 and 64 entries L1/L2 NVW-FIFO: 16 and 64 entries

7 Evaluation

We present our methodology of evaluation and then discuss detailed results in this section.

7.1 Methodology

We implemented scoped persist barriers in the gem5 simulator [2]. The applications are written in OpenCL©and are compiled to HSAIL (HSA intermediate language). The simulator runs this HSAIL code on the top of HSA [3] runtime. The simulator models a detailed memory hierarchy including coherent shared virtual memory across the CPU and GPU. Hardware cache coherence protocol keeps the GPU and CPU caches coherent. Table 2 details the simulated system configuration.

Workload: To the best of our knowledge, this is the first work to explore how a GPU program can directly manipulate a persistent data structure. As expected, we face the classic "chicken-and-egg" problem – no suitable benchmark suite exists.

Since databases could benefit significantly from both GPU and NVRAM, we focused on B+tree – a data structure widely used for indexing in databases. For example, popular database management implementation like Kyoto Cabinet [21], H-store [32, 64], SAP HANA [54, 64], MongoDB [41] uses B+tree for indexing. Moreover, Linux's widely used BRTFS filesystem uses B-tree as the primary data structure [52]. In short, any improvement in B+tree manipulation algorithm impacts wide range of commercial software. Besides, our observations are generally applicable to applications that use pointer-based data structures like red-black trees, hashmaps, graphs.

For the purposes of evaluation, we implemented three versions of parallel B+ tree insertion in OpenCL as follows – (a) Algorithm 1 that uses kernel scoped persist barrier (Section 5.1), (b) Algorithm 2 that uses workgroup scoped persist barrier (Section 5.1), and (c) the baseline implementation that runs on a *DRAM-only* system and is similar to the above workgroup scoped implementation but without persist barriers. Note that we do not implement B+tree insertion using workitem scoped persist barriers as it is insufficient to enforce necessary persist dependency in any meaningful way on the GPU (Section 4).

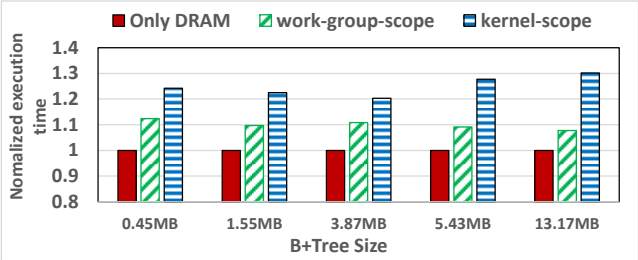


Figure 5. Normalized execution time for different scoped persistence barriers. NVRAM latency: 75ns read, 250ns write [14, 43].

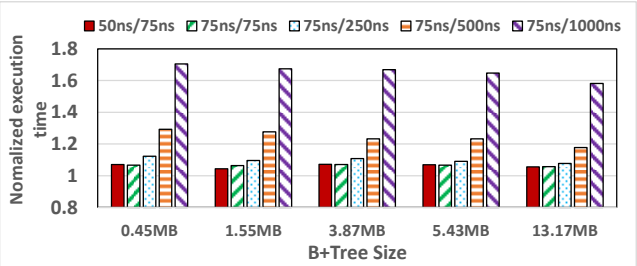


Figure 6. Sensitivity to NVRAM latency with workgroup-scoped persist barrier. Execution time normalized to DRAM only system.

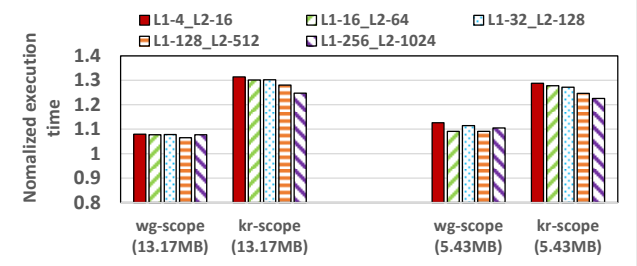


Figure 7. Sensitivity to NVW-FIFO sizes. Execution time normalized to DRAM only system. NVRAM read/write latency: 75ns/250ns. L1-*i*_L2-*j* stands for L1 NVW-FIFO with 'i' entries and L2 NVW-FIFO with 'j' entries.

7.2 Results

We evaluate four aspects of scoped persist barriers. Subsection 7.2.1 shows how different scopes for persist barrier impact performance. Subsection 7.2.2 shows the sensitivity to different latencies of the persist operation. Subsection 7.2.3 describes sensitivity to sizes of NVW-FIFOs of our design. Finally, subsection 7.2.4 quantitatively compares the recoverability offered with different scopes.

7.2.1 Performance of scoped persist barriers

Figure 5 shows the execution time of Algorithm1 that uses kernel scoped persist barrier ("kernel-scope") and Algorithm 2 that uses workgroup scoped persist barrier ("workgroup-scope"). The execution times are relative to the baseline (Section 7.1) running on a DRAM-only system. On the x-axis we vary the size of the B+tree to show scalability of the results. We assume PCM-like latencies (75ns read, 250ns write [14, 43]) with reads and writes to NVRAM being 3x and 10x slower than DRAM, respectively.

We observe that workgroup scoped persist barriers incur about 8–10% overhead across different B+tree sizes whereas kernel scoped implementation incurs about 25% overhead, relative to DRAM-only baseline[5]. Furthermore, the overhead gradually increases with the size of the B+tree for the kernel-scope algorithm, while that for the workgroup-scope remains flat. This is expected since finer-grain workgroup persist barrier enforces much fewer persist dependencies than coarse-grain kernel barriers. Furthermore, as the size of the B+tree increases, the number of workitems employed to perform the task increases proportionally. Therefore, a kernel scoped persist barrier would require enforcing persist dependencies across larger number of workitems. However, the number of workitems per workgroup does not change with workload size. Only the number of workgroups increases. The workgroup persist barrier thus scaled better since it enforces persist dependencies *only within* a workgroup and not across the workgroups.

In summary, we observe that a fine-grained persist barrier both performs and scales better than a coarse-grained barrier. However, as shown in Section 5.1, use of workgroup scoped barrier could require more programming effort than using kernel scoped barrier.

7.2.2 Sensitivity to Persist Latency

Figure 6 shows how performance overhead of workgroup scoped persist barrier changes with varying persist latencies (75ns read(R)/250ns write(W)[43], 75ns(R)/500ns(W)[50],75ns(R)/1000ns(W)[61]). As expected, the *relative* overhead generally goes up with increase in persist latencies but remains mostly reasonable (20%) up to the configuration with 75ns read and 500ns write latencies. Beyond that, we observe significant increase in performance overhead. The same analysis for kernel scoped barrier shows similar trend but with relatively larger runtime overhead as expected (figure omitted to avoid repetition).

[5]The workgroup scoped persist barrier incurs $< 3-4\%$ overhead across different B+tree sizes when DRAM latency assumptions are applied to the NV memory region.

Table 3. Recoverability comparison

B+tree depth	NV writes(wg-scope)	NV writes(kr-scope)
Leaf	5734	329359
Leaf - 1	4748	80829
Leaf - 2	2806	32076
Leaf - 4	1198	10299
Leaf - 5	582	2095

7.2.3 Sensitivity to NVW-FIFO sizes

The size of the NVW-FIFOs in our proposed implementation may affect how often cache blocks need to be persisted to NVRAM. In Figure 7, we thus sweep the size of NVW-FIFO. Although the persist overhead drops marginally with very large sizes of NVW-FIFO such as a 256 entry L1 and a 1024 entry L2, the configurations with smaller sizes show little or no improvement. Hence, the L1 and L2 NVW-FIFO sizes are set to 16 and 64 in our simulation.

7.2.4 Recoverability Comparison

Persist barriers are needed to ensure persistent data structures remain recoverable across failures. Different scopes determine amount of work needed to restart the computation. With the use of p_barrier_wg, NV writes are persisted in finer granularity of workgroups relative to p_barrier_kr. Thus, in the event of a system crash, writes from a subset of workgroups from the kernel might have persisted before the crash. Hence, one can recover from this crash by only re-executing the set of work-groups that could not persist at the time of that crash. However, in case of p_barrier_kr, unless writes from all the workgroups of a kernel have persisted, the entire kernel should be re-executed for correct recovery.

Table 3 compares the recoverability of workgroup and kernel scope persistence with a B+tree size of 13.17MB at various levels of the tree. *Leaf - n* in the table denotes 'n' levels above the leaf node. Columns for NV writes give the number of NV writes that are to be persisted by the given scope at different depths of the tree. Smaller number of NV writes to persist indicates finer-grain and thus possibility of faster recovery. Table 3 shows that the number of NV writes to persist for work-group scope (column 2) is significantly less than that for the kernel scope (column 3).

7.2.5 Area and Energy Overhead

NVW-FIFOs introduced to support scoped persist barrier incurs small area and energy overhead. NVW-FIFOs are implemented as SRAM circular buffers with 1 read and 1 write ports (i.e. head and tail pointers). Each entry in NVW-FIFOs is 8-byte long. NVW-FIFOs are updated in parallel to corresponding L1 or L2 cache accesses. L1 and L2 NVW-FIFOs have 16 and 64 entries respectively. Using FabMem tool [56] for 45nm technology node we find that four L1 NVW-FIFOs each of 2535.24 um^2 and one L2 NVW-FIFO of 9133.83 um^2 add trivial area overhead to a state-of-the-art GPU layout. Reading a L1(L2) NVW-FIFO entry requires only 1.44(5.00) pJ of energy, whereas writing to a L1(L2) NVW-FIFO entry requires only 2.49(6.24) pJ of energy which is negligible (about 1/200th [42]) when compared to the energy involved in updating caches.

8 Discussion

Here we discuss a few topics related to the implementation.

8.1 Persistency models for scoped persist barriers

Memory persistency models define ordering of persists operations [31, 36, 47] (Section 2.2). The proposed scoped persist barrier instructions can be used to implement multiple variations of epoch persistency models (Section 5). Specifically, our proposed implementation in Section 6 enforces epoch persistency model (as defined in Joshi et al. [31] and in Section 2.2). The alternative implementation mentioned in Section 6.2 could enforce buffered epoch persistency model instead. However, the concurrency in GPU could hide part of persist latency and thus, extra state overhead of implementing buffered epoch persistency may not be justifiable.

8.2 Design implications of NVW-FIFOs

NVW-FIFOs in our proposed implementation help to keep track of outstanding writes to NVRAM. However, a NVW-FIFO can be shared by multiple workitems or by multiple workgroups. For example, the L1 NVW-FIFOs in our implementation are shared by all workitems running in a CU. All CUs share one L2 NVW-FIFO. Such sharing may result in persisting writes to NVRAM that may not belong to the scope of a persist barrier. However, this is conservative but correct implementation since persist barriers do not define earliest time a write can be persisted. Further, this can also be avoided in the alternative implementation described in Section 6.2.

8.3 Designing a system recovery program

The semantic of a persist barrier neither mandates nor requires any system recovery guarantee. In the event of a crash, a *recovery program* needs to restart the crashed program(s) and restores the state of data structure to an algorithmically-defined restart-able state. While design of a recovery program is a separate topic we did explore how such recovery program can be designed for our proposed implementation.

Persist barriers divide a GPU program in *epochs*, contiguous sequences stores of a program execution demarcated by persist barriers [31]. After a crash, a recovery program resumes execution from the last completely *persisted* epoch. However, at the time of a crash, an epoch can be *partially* persisted (Section 5.2 of [31]). Following previous proposals [16, 20, 39], undo logging can be used to discard effects of a partially persisted epoch. Specifically, before a data in NVRAM is updated in-place, an entry with old data is created in a NVRAM-resident undo log. Each entry in the log is tagged by a unique identifier of the corresponding epoch. Once all writes to NVRAM corresponding to a given epoch have successfully persisted, its completion is noted in a NVRAM-resident epoch status table (EST, similar to [51]) through an 8-byte atomic update. The corresponding log entries are then de-allocated.

Upon restart after a crash, a recovery program can then first restore memory locations corresponding to partially persisted epoch from the undo log. The recovery program then reads the EST to identify the last completed epoch and resumes execution from that.

9 Related Work

There has been plenty of research in ensuring correct persist ordering for recoverability while minimizing persist overhead in systems with NVRAM. These proposals offer programmers the requisite interfaces to write a program with desired happens-before persist ordering and ensure those orderings in the hardware. They broadly fall into the following categories.

Persistency models and Programming Recent works [35, 47] formally outline several memory persistency models as described in Section 2.2. Furthermore, several recent proposals [12, 13, 16] precisely define transactional interface to NV memory. Kolli et al. [37] also showed how decoupling persist ordering from volatile execution ordering can bring significant performance uplift. While all these prior proposals can correctly enforce the happens-before persist ordering required in single-threaded or multi-threaded CPU-centric systems, they fall short to ensure the same while executing in a GPU-like hierarchical execution environment. Our work seeks to introduce a framework for memory persistency operations on such hierarchical execution environment in GPU systems.

Persist concurrency There has been significant work on hardware support to reduce persist ordering constraints. [13, 17, 31] delay persist operations as far as possible by recording and buffering them while subsequently enforcing any ordering constraints among them out of the critical path. This results in reducing the persist overhead considerably. The loose-ordering consistency [40] reduces ordering constraints by exploiting a memory-backed hardware logging support. Recently Kolli et al. [36] proposed mechanisms to minimize persist ordering constraints in transactional programs. They show how a transaction system can incur many unnecessary persist dependencies by performing the commit step of the transaction while holding the locks, and how their proposal removes these dependencies by deferring commit until after the locks are released. While they focused solely on CPUs, the concept of transactions can possibly be extended to GPUs.

Chakrabarti et al. recently proposed *Atlas* that provides durability semantics to lock-based code in a system with NVRAM [13]. Joshi et al. [31] proposed mechanisms for efficiently implementing a persist barrier in a multi-core CPU. While these mechanisms work well for single-threaded or multi-threaded CPUs, it may not scale to massively-threaded GPUs because of the overheads that they

are likely to impose. Furthermore, Joshi et al. [31] also proposed a mechanism to detect stores to NVRAM that conflicts with epoch semantics provided by the persist barriers in a CPU. It can be extended to GPUs but GPU's weak memory model (e.g., no multi-store atomicity guarantee) unlikely to need this. More recently, Shin et al. proposed speculative execution past a persist barrier for hiding the latency of enforcing persist dependencies in CPUs [53].

Unlike these proposals, we focus on GPUs. Specifically, we enable programmers of GPUs to semantically specify the desired persist orderings using scoped persist barriers.

Scoped synchronization Our work is inspired by research on the scoped synchronization for GPUs [22, 24, 46]. Scoped synchronization fits well with GPUs hierarchical programming framework and help keep cost of synchronization under massive parallelism, manageable. The NVW-FIFO in our proposed implementation also draws inspiration from the idea of S-FIFO introduced by Hechtman et al. to reduce overhead of scoped synchronization [22]. Unlike these works though, we apply the concept of scopes in a very different context – ensuring desired persist dependencies.

Persist overhead KILN [67], FIRM [68], NVM Duet [38] and DP2 [55] in general, attempt to reduce persist latency or logging overhead either by using NVRAM technology in processor caches or by carefully tailoring memory controllers for persist operations. There are other recent works [29, 49, 51] that simplify log management by exploiting different memory access patterns of applications or through non-temporal write-combining streaming of log records. More recently, Arulraj et al. [9] showed how databases could simplify recovery by utilizing NVRAM. Implementations of scoped persist barrier can benefit further by adopting these proposals.

NVRAM technology and reliability It is well known that most NVRAM technologies [43, 50, 61] suffer from several issues that DRAMs do not, such as endurance and retention loss, read disturbance etc. However, these technology-specific issues and their implications on software are orthogonal to our work. We focus on programmability of NVRAM-resident data structures from GPUs.

10 Conclusion

We introduce the concept of *scoped persist barriers* for GPUs to be able to correctly and efficiently manipulate persistent data structures residing in NVRAM. We proposed three flavors of scoped persist barrier instructions – workitem, workgroup, and kernel scoped. They enable a GPU programmer to identify the execution grpup (scope) whose writes to NVRAM should be ordered. We empirically demonstarte that use of coarse-grain kernel scoped persist barrier could help ease programming but leads to significant performance overhead. If finer-grain workgroup barrier is used instead, then performance improves since less number of persist dependencies are enforced but comes with additional programming complexity. This shows trade-offs stemming from use of different scoped persist barriers to perform a given GPU task.

11 Acknowledgment

AMD, the AMD Arrow logo, Radeon® and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] [n. d.]. CUDA C Programming Guide. ([n. d.]). Accessed: 2015-11-20.
[2] [n. d.]. The Gem5 simulator. http://gem5.org/. ([n. d.]). Accessed: 2015-11-19.
[3] [n. d.]. Heterogenous System Architecture (HSA). http://www.hsafoundation.com/standards/. ([n. d.]). Accessed: 2015-11-19.
[4] [n. d.]. The OpenCL Specification Version 2.0. ([n. d.]). https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.
[5] AMD. 2016. AMD Collaborates with Alibaba Cloud to Deliver Cloud Computing Services Based on AMD Radeon Pro GPU Technologies. (2016). http://www.amd.com/en-us/press-releases/Pages/amd-collaborates-with-2016oct14.aspx.
[6] AMD Inc. 2016. Radeon Pro SSG launched. (2016). http://www.amd.com/en-us/press-releases/Pages/amd-radeon-pro-2016jul25.aspx.
[7] AMD Inc. 2016. Radeon Pro SSG Technical brief. (2016). https://www.amd.com/Documents/Radeon-Pro-SSG-Technical-Brief.pdf.
[8] AnandTech. [n. d.]. AMD Carrizo. ([n. d.]). http://www.anandtech.com/show/9319/amd-launches-carrizo-the-laptop-leap-of-efficiency-and-architecture-updates/7.
[9] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind Logging. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 337–348. https://doi.org/10.14778/3025111.3025116
[10] Greg Atwood. 2011. Current and Emerging Memory Technology Landscape. In *The Flash Memory Summit.*
[11] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3).* ACM, New York, NY, USA, 94–103. https://doi.org/10.1145/1735688.1735706
[12] Hans-J. Boehm and Dhruva R. Chakrabarti. 2015. Persistence Programming Models for Non-Volatile Memory. In *Technical Report HPL-2015-59, Hewlett-Packard, 2015.*
[13] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14).* ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224
[14] Y. Choi, I. Song, M. H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y. J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y. T. Lee, J. Yoo, and G. Jeong. 2012. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International.* 46–48. https://doi.org/10.1109/ISSCC.2012.6176872
[15] 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 (ASPLOS XVI).* ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380
[16] 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 (ASPLOS XVI).* ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380
[17] 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 (SOSP '09).* ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589
[18] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stan Zdonik, and Subramanya Dulloor. 2014. A Prolegomenon on OLTP Database Systems for Non-Volatile Memory. In *ADMS@VLDB.* 57–63. http://hstore.cs.brown.edu/papers/hstore-nvm.pdf
[19] Digital Ternds. 2016. Intel confirms 2016 arrival of 3D XPoint-based Optane SSD. (2016). http://www.digitaltrends.com/computing/intel-optane-ssd-2016-3dxpoint/.
[20] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14).* ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814
[21] Fallabs. 2012. Kyoto Cabinet: a straightforward implementation of DBM. (2012). http://fallabs.com/kyotocabinet/.
[22] 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 *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA '14).*
[23] Sungjoo Hong. 2010. Memory technology trend and future challenges. In *Proceedings of the 201 IEEE International Electron Devices Meeting (IEDM 2010).* San Francisco, CA, USA.
[24] 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. In *Proceedings of the 19th International Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS '14). New York, NY, USA.

[25] HSAFoundation. [n. d.]. Kernel agent context switching. ([n. d.]). http://www.hsafoundation.com/html/Content/SysArch/Topics/02_Details/req_kernel_agent_context_switching.htm.

[26] Intel. 2017. Pmem: Persistent Memory Programming. (2017). http://pmem.io.

[27] Intel Corp. 2015. Intel and Micron Produce Breakthrough Memory Technology. (July 23, 2015). Press Release from http://newsroom.intel.com/docs/DOC-6713.

[28] International Technology Roadmap for Semiconductors. 2011. Process, Integration, Devices and Structures. (2011). http://www.itrs.net/Links/2011ITRS/2011Chapters/2011PIDS.pdf.

[29] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

[30] John Barrus. 2016. Announcing GPUs for Google Cloud Platform. (2016). https://cloudplatform.googleblog.com/2016/11/announcing-GPUs-for-Google-Cloud-Platform.html.

[31] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *The 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '15)*. IEEE, New York, NY, USA.

[32] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[33] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 691–706. https://doi.org/10.1145/2723372.2746480

[34] Kinetica. 2017. GPU-accelerated analytics database. (2017). https://www.kinetica.com/.

[35] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2015. Persistency Programming 101. In *Non-Volatile Memories Workshop (NVMW '15)*.

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

[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 *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*. ACM, New York, NY, USA.

[38] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. 2014. NVM Duet: Unified Working Memory and Persistent Store Architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 455–470. https://doi.org/10.1145/2541940.2541957

[39] David E. Lowell and Peter M. Chen. 1997. Free Transactions with Rio Vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 92–101. https://doi.org/10.1145/268998.266665

[40] Youyou Lu, Jiwu Shu, Long Sun, and O. Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. 216–223. https://doi.org/10.1109/ICCD.2014.6974684

[41] MongoDB Inc. 2017. MongoDB. (2017). http://www.mongodb.org/.

[42] N. Muralimanohar, R. Balasubramonian and N. P. Jouppi. 2009. CACTI 6.0: A Tool to Model Large Caches. (2009). HP Laboratories.

[43] P. J. Nair, C. Chou, B. Rajendran, and M. K. Qureshi. 2015. Reducing read latency of phase change memory via early read and Turbo Read. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. 309–319. https://doi.org/10.1109/HPCA.2015.7056042

[44] Prashant J. Nair, Dae-Hyun Kim, and Moinuddin K. Qureshi. 2013. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 72–83. https://doi.org/10.1145/2485922.2485929

[45] NVIDIA. 2016. GPU-Accelerated Applications. (2016). http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf.

[46] Marc S. Orr, Shuai Che, Ayse Yilmazer, Bradford M. Beckmann, Mark D. Hill, and David A. Wood. 2015. Synchronization Using Remote-Scope Promotion. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 73–86. https://doi.org/10.1145/2694344.2694350

[47] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[48] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage Management in the NVRAM Era. *Proc. VLDB Endow.* 7, 2 (Oct. 2013), 121–132. https://doi.org/10.14778/2732228.2732231

[49] L. Pu, K. Doshi, E. Giles, and P. Varman. 2015. Non-intrusive Persistence with a Backend NVM Controller. *IEEE Computer Architecture Letters* PP, 99 (2015), 1–1. https://doi.org/10.1109/LCA.2015.2443105

[50] M. Qureshi, S. Gurumurthi, and B. Rajendran. 2011. *Phase Change Memory: From Devices to Systems*. Morgan and Claypool.

[51] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 672–685. https://doi.org/10.1145/2830772.2830802

[52] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage* 9, 3, Article 9 (Aug. 2013), 32 pages. https://doi.org/10.1145/2501620.2501623

[53] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 175–186. https://doi.org/10.1145/3140659.3080240

[54] Vishal Sikka, Franz Färber, Anil Goel, and Wolfgang Lehner. 2013. SAP HANA: The Evolution from a Modern Main-memory Data Platform to an Enterprise Application Platform. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1184–1185. https://doi.org/10.14778/2536222.2536251

[55] Long Sun, Youyou Lu, and Jiwu Shu. 2015. DP2: Reducing Transaction Overhead with Differential and Dual Persistency in Persistent Memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15)*. ACM, New York, NY, USA, Article 24, 8 pages. https://doi.org/10.1145/2742854.2742864

[56] T. A. Shah. 2010. FabMem: A Multiported RAM and CAM Compiler for Superscalar Design Space Exploration. (2010). Master's thesis, North Carolina State University.

[57] TheNextPlatform. 2017. MapD GPU Database Looks Forward To Heftier Iron. (2017). https://www.nextplatform.com/2016/03/30/mapd-gpu-database-looks-forward-heftier-iron/.

[58] top500.org. 2016. Top500 Supercomputer listing. (2016). https://www.top500.org/statistics/sublist/.

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

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

[61] Hao Wang, Jie Zhang, Sharmila Shridhar, Gieseo Park, Myoungsoo Jung, and Nam Sung Kim. 2016. DUANG: Lightweight page migration and adaptive asymmetry in memory systems. In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture (HPCA '16)*.

[62] Wikipedia. [n. d.]. B+ tree. ([n. d.]).

[63] Jung H. Yoon, Hillery C. Hunter, and Gary A. Tressler. 2013. Flash and DRAM Si scaling challenges, emerging non-volatile memory technology enablement - Implications to enterprise storage and server compute systems.. In *The Flash Memory Summit*.

[64] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (July 2015), 1920–1948. https://doi.org/10.1109/TKDE.2015.2427795

[65] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2694344.2694370

[66] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744

[67] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744

[68] Jishen Zhao, Onur Mutlu, and Yuan Xie. 2014. FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 153–165. https://doi.org/10.1109/MICRO.2014.47