Architectural Support for NVRAM Persistence in GPUs

Sui Chen, Lei Liu, Weihua Zhang, and Lu Peng, *Senior Member, IEEE*

Abstract—Non-volatile Random Access Memories (NVRAM) have emerged in recent years to bridge the performance gap between the main memory and external storage devices, such as Solid State Drives (SSD). In addition to higher storage density, NVRAM provides byte-addressability, higher bandwidth, near-DRAM latency, and easier access compared to block devices such as traditional SSDs. This enables new programming paradigms taking advantage of durability and larger memory footprint. With the range and size of GPU workloads expanding, NVRAM will present itself as a promising addition to GPU's memory hierarchy. To utilize the non-volatility of NVRAMs, programs should allow durable stores, maintaining consistency through a power loss event. This is usually done through a logging mechanism that works in tandem with a transaction execution layer which can consist of a transactional memory or a locking mechanism. Together, this results in a transaction processing system that preserves the ACID properties. GPUs are designed with high throughput in mind, leveraging high degrees of parallelism. Transactional memory proposals enable fine-grained transactions at the GPU thread-level. However, with lower write bandwidths compared to that of DRAMs, using NVRAM as-is may yield sub-optimal overall system performance when threads experience long latency. To address this problem, we propose using Helper Warps to move persistence out of the critical path of transaction execution, alleviating the impact of latencies. Our mechanism achieves a speedup of 4.4 and 1.5 under bandwidth limits of 1.6 GB/s and 12 GB/s and is projected to maintain speed advantage even when NVRAM bandwidth gets as high as hundreds of GB/s in certain cases. Due to the speedup, our proposed method also results in reduction in overall energy consumption.

Index Terms—NVRAM, persistence, GPUs, helper warps

1 Introduction

Non-volatile Random Access Memory (NVRAM) has emerged as promising storage for computer systems, providing larger storage density and space compared with conventional DRAM; more importantly, its durability property can bring a new life to existing parallel programming techniques such as lock-free data structures, as well as transaction processing and storage applications.

As a byte-addressable, durable storage, the NVRAM may be used in a few different ways. The simplest way is to use an NVRAM as a large-capacity, drop-in replacement for the DRAM or the cache, resulting in hybrid main memory [1]. Hybrid refers to the mixture of volatility and non-volatility, and does not necessarily leverage the durability property of NVRAM.

More sophisticated approaches utilize the NVRAM as a persistent data store of a transaction processing system (TPS). A TPS usually involves two parts: a concurrency protocol layer that is responsible for detecting and resolving conflicts between transactions and may be embodied as a transactional memory or locking mechanism; and a logging layer that persists write sets of committed transactions in a certain order to ensure durable writes are consistent and can be restored during a power loss event. Although it is intriguing for computer systems, supporting persistence requires additional hardware and bandwidth costs [2], [3].

With the appearance of NVRAM, the boundary between storage and memory has been blurred. Memory operations can now persist individual bytes; disk-sized swap space, which was previously slow, can now enjoy almost DRAM-like read latency. NVRAM leads to a change in the existing memory and storage hierarchy, so that techniques originally considering volatile and non-volatile memory separately may now need to consider persistence across the stack. Persistence means any operation, including incorrect ones, will be saved on the device for its entire life time. Therefore, for persistence to be useful, consistency must be ensured so the system state is always valid. Various applications and algorithms that already address consistency, such as databases that rely on lock-free data structures including skip lists, B-trees and hash tables, may require software or hardware techniques to fully take advantage of NVRAM. Such techniques include multi-word compare-and-swap [4] and transactional memory, which enable sophisticated operations and provide consistent durability beyond the reach of single-word compare-and-swap. Most NVRAM persistence designs ensure consistency by enforcing an order in which write sets are persisted to the NVRAM. The mechanisms used to maintain the order are referred to as "epoch barriers" or "persistence barriers" [5].

- *S. Chen and L. Peng are with the Division of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. E-mail: {csui1, lpeng}@lsu.edu.*
- *L. Liu is with SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100864, China. E-mail: liulei2010@ict.ac.cn.*
- *W. Zhang is with Software School, Fudan University, Shanghai 201203, China. E-mail: zhangweihua@fudan.edu.cn.*

Manuscript received 3 May 2019; revised 24 Oct. 2019; accepted 2 Dec. 2019. Date of publication 17 Dec. 2019; date of current version 20 Jan. 2020. (Corresponding authors: Lu Peng and Weihua Zhang.) Recommended for acceptance by S. Chen. Digital Object Identifier no. 10.1109/TPDS.2019.2960233

1045-9219 © 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

In the mean time, Graphics Processing Units (GPUs) have been widely adopted as standard accelerators for high-performance computing. GPUs are designed for throughput-oriented computing using large numbers of light-weight parallel threads [6]. For this purpose, they are constructed with compute units that can house large numbers of resident threads along with their register states and a deeply pipelined memory subsystem that can handle a large number of parallel memory accesses. While the usage and persistence of NVRAM on CPUs have been investigated intensively, how to maintain persistence on GPUs are still in the early stage. In this paper, we will investigate an efficient approach to utilize the NVRAM's persistence property on GPU systems.

In combining the three components of NVRAM, GPU, and transaction processing, the NVRAM's asymmetric and relatively lower bandwidth need to be taken into account in the design of a system to achieve good performance. Bandwidth-induced latency needs to be managed well using software and/or hardware methods to avoid performance degradation and achieve full efficiency. It will be exacerbated on the GPU due to a large number of concurrently running threads competing for the limited NVRAM bandwidth. Our paper is targeted exactly at this issue: to alleviate the impact of limited NVRAM bandwidth on a GPU transaction processing system utilizing NVRAM.

In this paper, we first prepare a GPU transaction processing system that emulates NVRAM persistence behavior, and use it to run several benchmarks. We then measure the impact of bandwidth limits on those benchmarks. After that, we focus on the performance bottleneck caused by persistence operations, then propose the Helper Warps method to mitigate the performance impact. With this, we discuss the various tuneable knobs in the design of the Helper Warps, as well as a bandwidth emulation method for getting a more comprehensive understanding of the performance characteristic of the Helper Warps. Finally, we show the experimental results and make a comparison against the baseline.

In summary, we make the following contributions in this paper:

- To the best of our knowledge, we propose the first efficient and easy-to-use transaction processing system that uses NVRAM storage on GPUs;
- We propose the use of Helper Warps that utilize spare compute power on the GPU to alleviate the impact of limited write bandwidth;
- We establish a mechanism that can adaptively enable the Helper Warps to achieve the best performance under different program behaviors.

2 Background

2.1 Transaction Processing Systems and Logging

Logging, or more specifically, Write-Ahead Logging (WAL), characterized by ARIES [7], is a classic technique for achieving consistent durability in databases. In WAL, changes are written to the log ahead of the actual operation. The logs usually contain redo and undo operations so that when a power loss event occurs, the system is still able to check the log and then recover to a valid state by either discarding or committing the operation. Logging is also implemented in journaling file systems to maintain consistency for block-based devices.

On multi-core and many-core systems, contention incurred by Compare-And-Swap (CAS) operations required may greatly impact performance. Distributed logging algorithms improve performance by reducing such contention and adapting to the layout. Distributed logging mechanisms differ in hardware considerations (which cores contend over which addresses) and access patterns (the physical layout of the index) and many techniques have been proposed based on different design choices.

CPU Transaction Processing Systems. Transaction processing systems (TPS) [8] are best exemplified by database systems [9], [10]. The physical layout of the data stored in a database may be record-oriented, attribute-oriented or graph-based. The operations allowed on stored data may be online analytical processing (OLAP, concurrent read) [11], [12] or transactional (OLTP, concurrent write) [13], [14]. When concurrent writes occur, a concurrency control mechanism resolves conflicts between them and stores the data. Data is stored onto persistent storage devices such as disks and NVRAM, usually with logging techniques to ensure consistency, and may be cached in the main memory for better performance. The operations together maintain the properties of ACID (atomicity, consistency, isolation and durability) for transactions.

GPU Transaction Processing Systems. Many GPU-based transactional processing systems have been proposed, such as [15], [16], [17], [18], [19], [20]. Most systems proposed use GPUs as accelerators only for read-only, OLAP workloads, and leverage the GPU's capability to perform certain operations very quickly such as parallel reduction. In those designs, the GPU does not independently execute transactions and requires the CPU to step in for certain steps such as query plan generation. OLTP workloads, on the other hand, are not as wide-spread as OLAP systems yet, and most of the OLTP systems do writes completely on the CPU. Incidentally, transaction processing is in an early stage on the GPU as well.

Transactional memory (TM) [21] is a programming construct that originated from the database community, which simplifies the usage of fine-grained parallelism. Alternatives to TM include locking (coarse-grained and fine-grained) and lock-free data structures. Fine-grained and lock-free data structures provide high performance, but their concurrency control is difficult to implement and must be implemented on an case-by-case basis, while coarse-grained locking is easy to implement but much slower due to serialization. TM is aimed at combining the performance of fine-grained locking while being easy to use like coarse-grained locking by automatically managing concurrency control for programmers.

TM and TPS systems were traditionally applied in systems with either main memory or a combination of main memory and disks. With NVRAM, these use cases could see new possibilities.

2.2 NVRAM

NVRAM for the CPU. NVRAMs may be used as drop-in replacement of caches, memory, or storage devices. Alternatively, they may be used as a byte-addressable persistent store. The simplest approach to utilize NVRAM as persistent

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

store is to apply a file system layer on top of it. While replacing hard drives with NVMs can greatly benefit applications [22], [23], directly replacing memory and cache with NVRAM may slow down applications [24]. In addition, using NVRAM as a drop-in replacement of DRAM misses its durability property. To take advantage of the persistence properties would require persistence to be taken into account during the design of the system. Aside from an application that requires persistence such as a TPS, a carefully designed transaction processing mechanism is also needed to guarantee the ACID properties. This line of research started in the 1990s with the appearance of Flash memory technologies, as exemplified by eNVy [25], and continually evolved with the newer iterations of NVRAM.

Recent hardware and software proposals such as Memory Persistency [2], DudeTM [26], and ATOM [27] achieve high performance by decoupling, i.e., performing persistence out of the critical path; Study of NVRAM applications in WHISPER [28] quantifies the use of NVRAM in a real-life system. NV-Heaps [29] discusses the interoperation between the NVRAM and DRAM which are used simultaneously. Kiln [3] proposes adding non-volatile caches to form a multi-versioned persistent system. In all use cases, traditional techniques such as speculative execution [30] can benefit NVRAM-based systems.

GPU and Transactional Memory. Transactional Memory (TM) is a key technique for enabling OLTP workloads involving concurrent reads and writes on the GPU. Both hardware-based (HTM) and software-based (STM) systems have been proposed on the GPU. Software systems [31], [32], [33] utilize the GPU's parallel processing power to perform basic tasks in the TM system in parallel, such as lock management, coalesced read and write log access, and using GPU-friendly data layouts such as structs-of-arrays instead of arrays-of-structs. With these efforts combined, the GPU-based STM systems are able to rival or even outperform CPU-based STM systems.

HTM systems have also been proposed for GPUs [34], [35], [36], [37]. As hardware approaches, these proposals feature TM algorithms implemented in hardware, as well as new hardware architectures that are aimed at providing new versioning techniques (such as snapshot isolation) and new conflict detection mechanisms (such as various eager conflict detection techniques).

NVRAM on the GPU. For the GPU, NVRAM has been considered for enhancing existing cache and memory subsystems. Due to the investigated NVRAM having only a fraction of the bandwidth of that of the DRAM, hybrid designs need to be adopted to alleviate the bandwidth gap [1], [38]. However, these works do not utilize persistence. Our recent paper [39] is the first effort to support NVRAM persistence for GPUs. Here, we extend it with details in sections including background, design, and experimental results.

NVRAM and Bandwidth. Currently, NVRAM devices feature higher storage density, but their write bandwidths are lower than that of the DRAM. Because of this, some NVRAM products include a small amount of DRAM buffer to provide better performance, just like hard disk drives. These optimizations improve the NVRAM's bandwidth performance to reasonable levels, but they still lag behind that of the DRAM. For instance, there is a performance gap in 4 KB block operations: contemporary high-end NVRAM devices deliver 500 K write I/O operations per second (IOPS) [40], which is lower than the 3,000 K write IOPS of RAM disks over the same PCI-E interface [41].

In recent works, researchers have assumed aggregate bandwidth ranging from several GB/s to tens of GB/s [42] for NVRAM. In particular, the Intel PMFS offers two choices for aggregate NVRAM bandwidth: 9.5 and 37 GB/s. This is comparable to that of the existing 3D XPoint commercial products. With specially-designed buffering and caching designs as well as software changes, an OLTP system that utilizes a combination of NVRAM and DRAM can bring the performance of immediate consistency to the level of DRAM-only devices [43].

Bandwidth is mentioned as a first-order concern for DRAM by a study that addresses the latency divergence issue in GPU [44] using hardware approaches, targeting the memory controller. The bandwidth issue affects both the software side and the hardware side, and compared with their solution, the TPS performance issue in our study may be resolved at the transaction processing level using a software Helper Warp approach.

2.3 Helper Warps/Threads

Continually-running helper threads/warps has seen its usage on the GPU and CPU alike. Vijaykumar *et al.* utilized GPU helper warps for memory compression [45]. Huang *et al.* proposed using helper threads in the HPC environment to backup critical data structures, with the intent of alleviating the cost of checkpointing [46].

Modern GPUs contain copy engines [47] which utilizes Direct Memory Access (DMA) to copy data between the main memory and GPU memory, allowing data transfers to overlap with kernel execution, resembling a special kind of helper warp dedicated to performing copy operations. The copy engines are usually more optimized for copying large blocks of data than for scattered, small blocks of data such as the transactional write logs. Should the copy engine be used for logging, to make transfer efficient, the logs need first be consolidated into a single continuous block, and *reproduce*d in the NVRAM after the transfer. The Helper Warps proposed in this work are a perfect fit for completing both jobs.

3 Latency and Bandwidth Impact on the GPU

The GPU is designed to tolerate long access latency by running a large number of threads concurrently. With large numbers of hardware thread contexts and scratch registers accessible to the thread scheduler, the scheduler can select a group of threads to run at a much lower cost compared to CPU context switching. This will incur high latency observed from the thread's perspective, but overall system throughput will be much higher than when optimizing for single-thread latency.

Due to this design principle, we are interested in understanding how limited bandwidth and throughput will affect the performance of programs. We run the transactional benchmarks used in this study with memory bandwidth enforced on the persistence step using the method described in Section 6.1. The bandwidth limits range from 484 GB/s (the native DRAM bandwidth) to 1.6 GB/s, covering the bandwidth range of the NVRAM devices available currently and in the future.

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

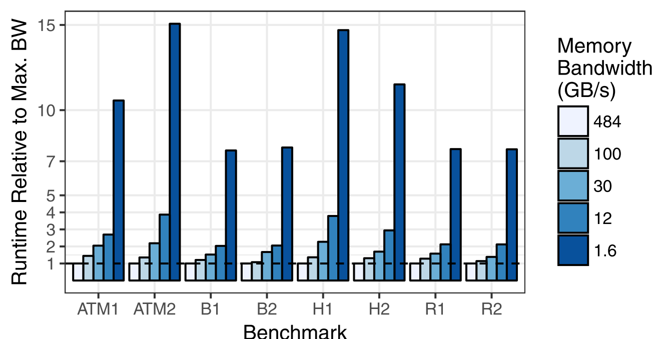


Fig. 1. Impact of bandwidth on execution time.

The impact of latency on bandwidth-bound transactional workloads is immediately visible from Fig. 1. Total workload run time gets longer as bandwidth gets more limited.

From individual workloads, we can see that slowdown increases sub-linearly to the reduction in bandwidth; Specifically, when bandwidth is limited to 100 GB/s, about 1/5 of the original bandwidth, program slows down at a factor of 1.5x; at 1.6 GB/s, which is as low as about 1/300 of the original bandwidth, slowdown increases to only as much as 15x. The relative difference is partly due to the transaction programs executing in three stages, *execute*, *commit* and *persist*, and only the *persist* step is affected. The persist step is more memory-constrained than the other two stages, where many threads may be idling, waiting for memory accesses. Fortunately, the idling compute resources are an opportunity for performance improvement: by completing the *persist* step asynchronously and overlapping it with the other stages, we can alleviate the impact of bandwidth limit.

4 Transaction Processing and Persistence

To make persistence useful, the data being persisted must remain consistent. For block devices, a file system layer, which usually incorporates logging, is responsible for maintaining consistency. Logging may also be applied to NVRAM to guarantee consistency. With logging, written data is first buffered into logs ("persist") and used to advance the system state stored in the NVRAM ("reproduce").

Log generation usually requires some concurrency control mechanism to resolve conflicts between committing transactions so that the data being written will be free of data hazards. Usually a software/hardware transactional memory (TM) or an equivalent of two-phase locking mechanism is used. A TM system may include logging for version-control, which may be extended to work with NVRAM.

We use a metadata-based software TM (STM) as the transaction execution layer for this study (although the proposed method is not limited to this TM implementation.) The TM utilizes redo-logging for multi-versioning as it provides better performance compared to undo-logging for our study. The conflict detection and logging granularity are both 32-bit words.

The metadata used in this study include a series of access locks that resolve conflicts between transactional writes and reads. In addition to the locks, a transaction ID is stored for each write and is used by the conflict resolution rule. The lock and the ID are combined to become an *ownership record*.

```
Lock Acquisition Procedure
 1: Transaction T_A reads/writes 32-bit word at address X
 2: Compute lock index i
 3: T_B = lock[i]
 4: if T_B is empty then
 5:    compare and swap lock[i] to self
 6:    if succeeded: return OK
 7:    else: abort T_A
 8: else
 9:    if T_B.id > T_A.id then
10:       compare and swap lock[i] to T_A
11:       if succeeded: return OK
12:       else: abort T_A
13:    else if T_B == T_A then
14:       return OK
15:    else
16:       abort T_A
17:    end if
18: end if

Commit Procedure
19: T_A reaches txcommit call
20: Check T_A.status
21: if not aborted: flush and persist write log
22: else: discard write log
```

Fig. 2. Metadata-based TM algorithm used in this article.

The structure of the STM, from the transaction's point of view, is listed in Fig. 2. When a transaction T_A performs a read or a write (Line 1), it first attempts to take the ownership record that corresponds to the word being read/written. When an ownership record is not held by any transaction (Line 4), the current transaction takes exclusive ownership with an atomic compare-and-swap (CAS) (Line 5). CAS guarantees when there are multiple transactions trying to acquire the ownership record, only one will succeed. When an ownership record is held by a transaction T_B, a conflict is said to have happened (line 3). A conflict must be resolved by aborting either transaction: if T_A's thread ID is smaller T_B's, T_A can preempt this ownership record and signal T_B to abort (Line 10) using a similar CAS operation. If T_B equals T_A, nothing needs to be done (Line 14). If T_A's thread ID is larger T_B's, T_A must abort itself (Line 16). This global ordering of priority based on thread IDs prevents deadlock and ensures system-wide progress.

In terms of implementation, due to the difficulty to send signals between individual threads on the GPU, aborts are handled by having each thread check if their own status flags have been set to "aborted". This technique also exists in languages utilizing coroutines such as Go. The check happens at the commit stage (Line 20). If a transaction sees it has been aborted by another transaction, it will relinquish all the ownership records it has taken so far, and discard its write log. Otherwise, the commit procedure will start, where the transaction flushes its write log to both the volatile memory and the NVRAM.

When implementing this TM algorithm, standard GPU programming optimization techniques are applied. For example, the locks and thread IDs in the metadata are organized as structs-of-arrays so access can be coalesced.

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

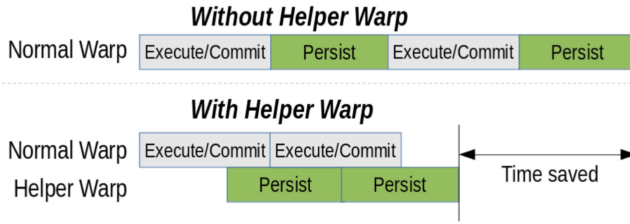


Fig. 3. Transaction behavior in the proposed architecture.

In the TM algorithm, writes to NVRAM happen during a successful commit. With baseline settings, the persist operation is executed on the thread executing the transaction, so the thread will not be able to execute a new transaction until it finishes persisting the current one. This will add NVRAM write latency onto the critical path of transaction execution, resulting in the overhead illustrated in Fig. 1.

We observe that for the ACID properties, it is possible that the first three properties be completed asynchronously from durability/persistence. In this study, we propose a commit procedure utilizing Helper Warps which is aimed at this very purpose: move the latency away from transactional critical path by allowing asynchronous persistence operations.

4.1 Helper Warps

Our approach is aimed at allowing asynchronous persistence operations by having a separate set of threads, called *Helper Warps*, do the persistence operations for the normal warps that execute transactions. Fig. 3 shows the overall transaction execution process with the Helper Warps added: committing transactions return after writing to the persistence buffer instead of writing to the NVRAM, saving execution time.

The Helper Warps and the normal warps reside in the same thread block and communicate with each other via the per-thread-block shared memory, where a persistence buffer is located as shown in Fig. 4. Each streaming multiprocessor (SM) that executes the thread blocks maintains a bandwidth monitoring window that keeps track of the instantaneous persistence bandwidth during runtime. Fig. 5 shows the overall structure of the proposed architecture, including the memory topology. In this architecture, the SMs have direct access to the DRAM and the NVRAM. Writes sets of committed transactions will be written into the NVRAM as persistence logs and then reproduced.

The persistence buffer is conceptually a FIFO queue physically implemented in a ring buffer. Address-Value pairs are enqueued by normal warps committing transactions and are

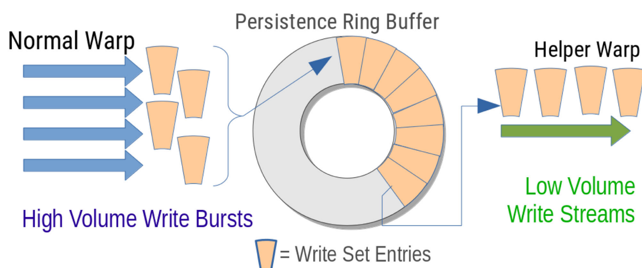


Fig. 4. Persistence buffer located in each shading multiprocessor (SM).

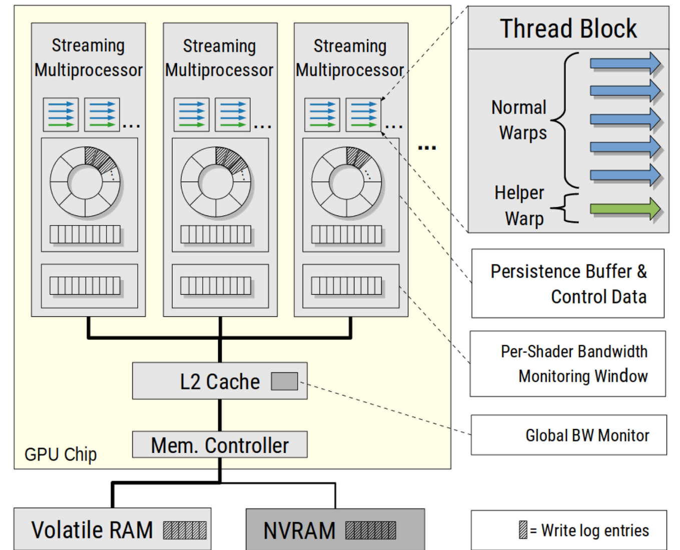


Fig. 5. Overall system architecture.

drained by the Helper Warps. Buffer spaces are acquired by committing transactions by incrementing an atomic counter that keeps track of the start location of free buffer space. Thus, multiple transactions can try to allocate space and write into the buffer simultaneously. To make sure the buffer entries are drained in-order, every entry in the queue has a “dirty bit” associated with it indicating whether the values have been written. Threads in normal warps and Helper Warps access the persistence buffer in parallel and access the entries in a coalesced fashion.

The Helper Warps mechanism comes with two performance implications: 1) The shared memory needed by Helper Warps may limit the number of concurrent thread blocks. This may slow down low-contention workloads, but may speed up high-contention workloads since less concurrency leads to less conflicts and thrashing when contention rate is high. 2) The speedup brought by the Helper Warp must overcome the overhead in maintaining the buffers to achieve overall speedup, and 3) If the buffers are frequently full and cause persisting transactions to stall, the buffers themselves become a bottleneck. These performance implications can be mostly resolved by tuning the Helper Warps mechanism, and we will discuss the tuning process in Section 5.

4.2 Correctness in Recovery

In this study, we assume the working set of the program fits in the DRAM, transactions write to both the DRAM and the NVRAM, and read from the DRAM. Because of the writes to the DRAM, read operations can immediately see recently-committed data, and write correctness can be guaranteed by the STM layer.

In case of a power loss event, the system will be restored to a valid state using the persistence logs. We assume all data including logs in the DRAM is lost and are not used for recovery. As such, the system will only check remaining log entries in the NVRAM. During this process, decisions are made per write set, from the oldest transaction to the most recent one:

- If a write set is completely *persisted*, but only partially *reproduced*, it will be reproduced again, no matter

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

```
writetest(int* scratch, int nwrites, int nrep) {
  int tid = GetThreadID(),
  my_idx = tid * nwrites;
  for (int rep=0; rep<nrep; rep++) {
    TX_START;
    for (int i=0; i<nwrites; i++) {
      scratch[my_idx + i] = tid;
    }
    TX_COMMIT;
  }
}
```

Fig. 6. Transactional micro-benchmark used to determine the Helper Warp switching threshold.

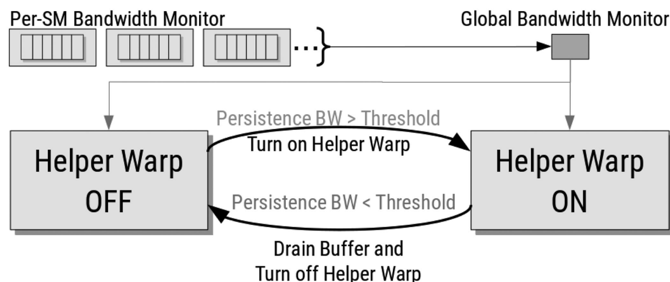


Fig. 7. Run-time Helper Warp adaptation process.

how much of it has been already reproduced prior to the crash, to make sure all its updates are reflected on the system states.

- When we encounter a transaction whose write set has not been completely *persisted*, we will discard this transaction and all transactions that happened after it without reproducing any of them. This is because the write sets must be reproduced in the correct order for the system state to be advanced from a valid state to another.

In extreme cases such as a power failure in the process of a recovery, the same recovery process may be applied. Since only fully-persisted transactions may be reproduced, we will just reproduce it again to maintain correctness.

5 Performance Tuning

5.1 Persistence Bandwidth Monitoring

For profiling and performance tuning, we keep track of the instantaneous persistence bandwidth. The bandwidth is computed locally by each SM and aggregated globally. This approach follows the GPU thread hierarchy. To track the local persistence bandwidth usage, the amount of data persisted in different time slices are logged. When a time slice passes, a delta between the bandwidth measurements of the last time slices are sent and added to the global bandwidth monitor. This process is illustrated in Figs. 5 and 7.

5.2 Adaptively Enabling Helper Warps

As mentioned earlier, the time saved by the Helper Warp mechanism must outweigh the overhead it incurs to achieve speedup. The cost of the Helper Warp mechanism mainly comes from allocating persistence buffer space. In addition, when the buffer is full, the normal warps also need to wait for Helper Warps to drain the buffer entries. For certain transactions, such as ones with fewer writes, performance may be better when the Helper Warps are disabled. Again, this is because the Helper Warps can only outperform the baseline when the persistence bandwidth becomes the bottleneck in the first place.

Thus, for a program to execute efficiently, it should be able to automatically determine when the Helper Warp should be enabled or disabled according to program behavior.

Determining the Threshold. To determine the threshold for a specific GPU system, we use a transactional micro-benchmark listed in Fig. 6. It varies the write pressure with different numbers of threads, and finds the point at which the memory is saturated and how much latency may be inserted at that point to overlap with the wait on write operations. This micro-benchmark involves a minimal transaction that performs writes to non-overlapping addresses in the same pattern the persistence logs are written. The persistence bandwidth is measured using the global bandwidth monitor described in Section 4.1. We drop the warm-up and ending phases of the micro-benchmarks and only consider the steady phases.

The method for determining the Helper Warp enabling/disabling threshold is independent of the NVRAM hardware connected to the GPU. During run time, per-SM-level and global bandwidth measurements are updated as transactions commit and persist. The global bandwidth is fetched from the Global Persistence Bandwidth Monitor by transactions and are used by the SM running the transactions to decide when to enable/disable Helper Warps.

Since the decision on enabling and disabling Helper Warps affects the entire system, only one thread across the entire system may modify it at a given time. Considering the cost involved, the decision to switch off Helper Warps is made more conservatively than switching on: When the instantaneous bandwidth exceeds the threshold determined in Section 5.1, the Helper Warp is instantly turned on; when the bandwidth is seen below the threshold for an extended time (we choose to use 5 slices according to our experiments), the Helper Warps are turned off.

With this mechanism, a program can adaptively turn on or off Helper Warps according to the workload's behavior, achieving better performance than statically enabling/disabling Helper Warps throughout the entire workload.

6 Methodology

6.1 Bandwidth Emulation

In this paper, we study bandwidth limits ranging from 1.6 to 484 GB/s (the latter is the original bandwidth of the GPU used in the study.) To obtain desired bandwidth limits, we need to first find the relationship between the delays and bandwidth limits. Certain bandwidth limit numbers may be obtained on certain hardware configurations. An example configuration is when a block of memory is allocated as pinned memory, GPU memory writes will be propagated to the main memory through the PCI-E bus and will thus peak at the PCI-E bandwidth, which is 12 GB/s on a PCI-E 3.0 interface. With a known pair of latency and bandwidth limit, we could vary the artificially-inserted latencies to obtain the bandwidth limits that are not directly available on real hardware. The length of latency will be calibrated against the real hardware measurements. An analytical model will be used to estimate the trend latency changes with bandwidth limits.

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

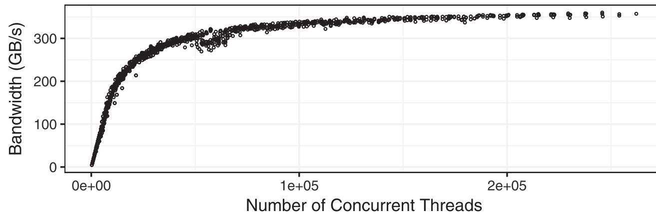


Fig. 8. Number of concurrent threads and overall write bandwidth in the non-transactional write pressure test.

Bandwidth and Concurrency. As a throughput-oriented device, the GPU encourages maximizing concurrency using many parallel threads. As the thread count increases, more requests may be sent out simultaneously, utilizing more bandwidth. For our study, this means more concurrent threads will also lead to higher latency per thread.

To illustrate this, we run a write test program listed in Fig. 9. The program is run on an NVIDIA GTX 1,080 Ti with a theoretical memory bandwidth of 484 GB/s. Each thread performs a `float4` write, the data type that maximizes write bandwidth on the device. As shown in Fig. 8, the overall bandwidth increases with the thread count, with a maximum bandwidth around 350 GB/s. This result is close to the `bandwidthTest` example in the CUDA SDK that is utilizing the CUDA built-in function `cudaMemcpy`. This suggests our write pressure test program is showing expected behavior.

We can also see from Fig. 8 that as long as the total number of threads is fixed, the number of thread blocks does not affect bandwidth usage. For example, 16 blocks of 512 threads and 512 blocks of 16 threads result in very close bandwidth measurements.

Bandwidth and Latency. Because the number of actually running threads may vary when a program is running due to factors including divergence and contention, we need to be able to find out the corresponding latency for given pairs of thread count and bandwidth limit. From Fig. 8, the relationship between the thread number and the final resultant write bandwidth is not linear. We expect the relationship between latency and thread number to be also non-linear.

To find out their relationship, we use an empirical approach: given a bandwidth limit, we find the per-thread latency that corresponds to the thread count. We vary the thread count and then accumulate the data to form an empirical distribution of latency against thread count. For this goal, we use the same program in Fig. 9, add a no-op loop between consecutive writes, and then find the latency that resulted in the bandwidth usage, and repeat this process over different thread block dimensions. The number of

```
__global__ void WriteTest(int delay) {
  const int tid=GetTID(), N=GetThreadCount(),
            mydata=0xA0000000 | tid;
    float4 payload={ float(mydata) };
    for (int i=tid; i < 128*N; i += N) {
        Latency(delay);         // delay
        g_log_out[i]=payload;   // global write
    }
}
```

Fig. 9. Source code for the write pressure test.

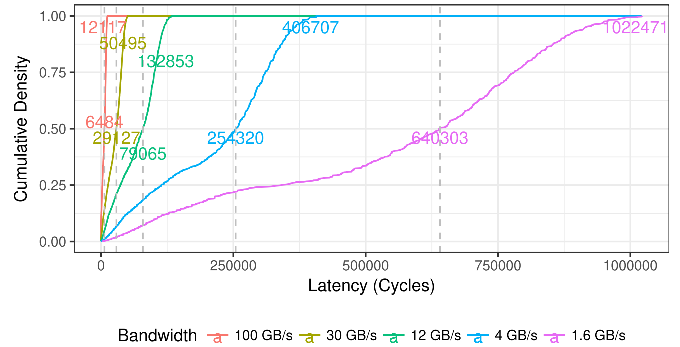


Fig. 10. Cumulative density function of latency between consecutive writes under various bandwidth limits, assuming thread block dimension is uniformly distributed.

blocks and the number of threads in a block are uniformly chosen between a range of [16, 512], and the latency numbers are aggregated into a cumulative density function map in Fig. 10. This results in a distribution of thread count with a median of 53,248 threads, a maximum of 262,144 and a minimum of 256 threads.

It can be seen from Fig. 9 that bandwidth limits of 100 GB/s, the median latency of the thread block dimensions is as high as 6,000 cycles. When the bandwidth continues to shrink, the latency increases roughly exponentially. The curves corresponding to different bandwidth limits show similar shapes. In fact, the mean latency and bandwidth matches well with the relationship $log(mean_latency) = -1.105 \cdot log(BW) + 6.063$, with $R^2 = 0.9981$. This shows us the variables in this system exhibit logarithmic relationships, which we will use to fit the running time as a function of bandwidth for the experiments.

Non-Transactional versus Transactional Persistence Bandwidth. On one hand, adding the transactional layer gives the GPU more work that does not involve non-volatile writes, thus spreading its write bandwidth pressure over a longer period of time. On the other hand, the transactional layer stretches the critical path longer, amplifying the impact of persistence latency; plus, conflicts and ownership records will propagate the impact to other threads. The result is when the system is saturated, we will see that write bandwidth measurements are much smaller compared to the available bandwidth limits.

For example, when NVRAM bandwidth is set to 12 GB/s with Helper Warps disabled, persistence bandwidth peaks at around only 200 MB/s. (This value is also the threshold for adaptive switching we used in this paper). With Helper Warps enabled, persistence bandwidth may peak at 5~6 GB/s. These values demonstrate how the transactional layer affects the achievable bandwidth: while non-transactional runs can reach $350/484 = 72\%$ of the theoretical bandwidth, with transactional workloads, that ratio becomes lower than $0.5/12 = 2\%$ without Helper Warps or around $6/12 = 50\%$ with Helper Warps. We expect the gap between these two benchmarks to depend on the pattern of conflicts, a result of the characteristics of the workloads being investigated.

For the purpose of bandwidth emulation, the observations suggest a chain of dependencies between the variables in a transaction processing system with limited persistence bandwidth, visualized in Fig. 11:

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

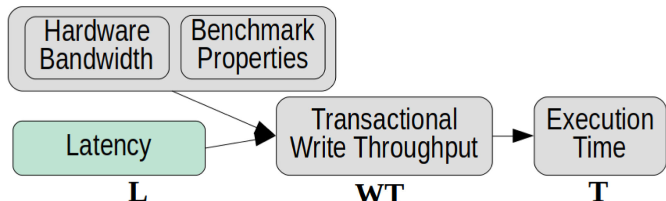


Fig. 11. Interaction of the critical variables in the system.

- The peak bandwidth and benchmark properties determine the range of observed throughput. For experimentation, we can artificially introduce a latency between writes in an individual thread to affect the throughput.
- The higher the throughput is, the faster the benchmark runs.

In this dependency chain, the latency may substitute the effects caused by the combination of peak bandwidth and benchmark properties, to yield the same throughput and total execution time. As such, we seek to find the latency that correspond to a certain bandwidth, including ones that cannot be obtained directly.

Complete Workflow. Fig. 12 shows the steps we use to emulate bandwidth limits between 1.6 and 484 GB/s for each benchmark to capture the dependency chain in Fig. 11.

First, we measure execution time by having the benchmarks persist into the GPU RAM (with 484 GB/s bandwidth), as well as on the zero-copy pinned memory accessible through the PCI-E bus (for 1.6 and 12 GB/s). We call these measurements as *reference points*. Second, we add artificial latency in the persist operation to emulate limited bandwidth on the NVRAM between the measurements in Step 1. When varying the artificial latency, for each latency value, we get a corresponding memory bandwidth, called the *proxy bandwidth*, by running the write pressure test in the absence of the transactional processing layer. The obtained bandwidths are then mapped linearly to the reference points. The linear function considers the maximum sustained bandwidth and the maximum theoretical bandwidth, such as the 350 GB/s versus 484 GB/s as measured with the program in Fig. 9. In the end of this procedure, the running time of an application given a memory bandwidth between 1.6 and 484 GB/s may be obtained by first computing the proxy bandwidth, and then performing a lookup from the proxy bandwidth to expected execution time.

In our experiments, we have found that the proxy bandwidth WB_x and running time T_x may fit against a function $T_x \sim a \cdot exp(-b \cdot WB_x)$, where a and b are constant parameters. The exponential component matches our observation in Fig. 10.

6.2 Hardware and Software Platform

We use real-system evaluation because it allows us to take into account system-wide factors that may impact performance. A real-system scenario is also much closer to our vision of an easy-to-use NVRAM transaction processing use case, where the user just needs to install a set of libraries in their GPU programs without having to worry about changes in other parts of the system.

```
Bandwidth Emulation Workflow
```

1: Measure running time T_{484}, T_{12} and $T_{1.6}$ with no artificial latency, as reference points.
2: Find the artificial persistence latency $L_{1.6}$, L_{12} and L_{484} that results in running time $T_{1.6}$, T_{12} and T_{484}.
3: For x in (L_{12}, $L_{1.6}$ with step size 1000):
4: Measure run time of this benchmark T_x;
5: Measure the memory write throughput WT_x;
6: Record the pair (WT_x, T_x).
7: Find the mapping between the range (1.6, 12) and the range covered by the recorded proxy variable WT_x
8: Repeat steps 2 and 7 for (0, L_{12}) to create a mapping for the bandwidth range between 12 and 484 GB/s. In the end of the algorithm, the running time given a bandwidth is found by first look up its proxy variable using the linear mapping, and then look up the running time associated with it.

Fig. 12. Approach for emulating bandwidths between 1.6 GB/s and 484 GB/s with artificial latency.

Experiments in this paper are run on an NVIDIA Pascal GPU, the GTX 1,080 Ti, which has 56 streaming multiprocessors (SMs) operating at a processor clock of 1,582 MHz, and has access to 11 GB of GDDR5X memory. There are 64 CUDA cores in each of the SMs. The total memory bandwidth of the GDDR5X memory is 484 GB/s.

In this study, we implement STM libraries in CUDA for running transactions on the GPU, using the algorithms in Fig. 2.

6.3 Benchmarks

We use a series of transactional benchmarks to evaluate the Helper Warp mechanism proposed in this paper. The benchmarks are built on our STM library implementing the algorithm in Fig. 2. They are listed as follows:

ATM is a program that performs bank transfers between two accounts. We use two different configurations. Configuration `A1` involves 1 M transactions between 100 K accounts and `A2` involves 1 M transactions and 1 M accounts. `A1` is expected to experience more conflicts than `A2`. The source and destination accounts of transfers performed are randomly generated.

Hashtable is an implementation of a key-value store, which resolves conflicts using linear probing. Inserted keys are mapped to the base entry table, backed by an extended entry table, which also handles collision resolution. Parallel insertions into the hash table are realized through transactions. This benchmark involves two configurations: `H1` performs 500 K insertions into a hash table with 15 M base entries and 35 M extended table entries, while `H2` performs 900 K insertions into a hash table with 1 M base entries and 3 M extended entries. Inserted entries are generated randomly.

Bounding Volume Hierarchy [48] is a program that builds a bounding volume hierarchy (BVH) for a 3D mesh model. This benchmark involves two kernels: 1) *BVH Construction*, that finds the parents of the nodes according to the sorted Morton Codes assigned to each of the primitives, and 2) *BVH Reduction*, which sets the bounding volume of every node by computing the union of its children. The first kernel

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

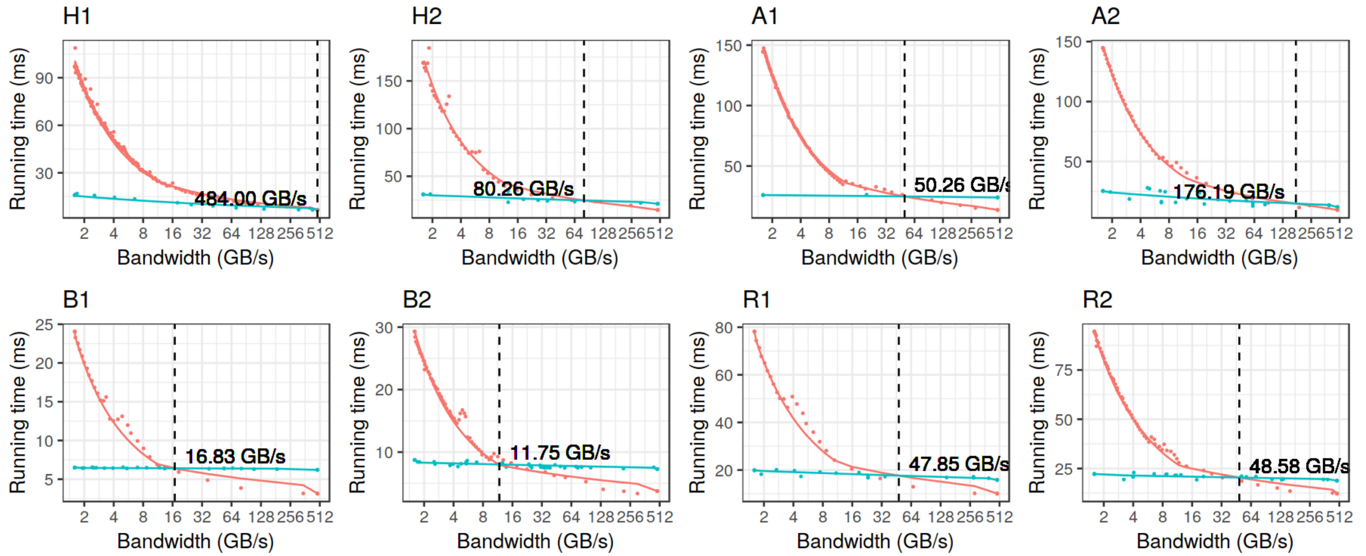


Fig. 13. Overall running time of the benchmarks, with helper warps enabled (green) and disabled (red). The "crossover point bandwidths" are marked with vertical dashed lines.

has no write/read conflicts and therefore serves as a lightweight write pressure test for the proposed Helper Warp system in a GPU-optimized algorithm context. The second kernel performs reduction from the leaf to the root of the BVH tree, which requires each node to be touched exactly once as a performance optimization technique. This constraint requires the updates to nodes be performed in transactions which keep track of both the bounding volume associated with a node as well as a flag that indicates whether this node has been touched. The benchmark has two configurations, each taking a model with 100,000 faces and 187,854 faces respectively as input. The two kernels in these configurations are denoted `B1`, `R1` and `B2`, `R2`, respectively.

All benchmarks are run with a thread dimension of 512 blocks of thread, with each block containing 256 threads. This will give 131,072 concurrent threads.

7 Evaluation

7.1 Overall Results

Fig. 13 shows the run time of the benchmarks with and without Helper Warps, using the Metadata-based TM implementation. The lines denote the trend in which the run time is changing according to NVRAM bandwidth limit. Green and red lines and dots denote the running time with the Helper Warps enabled and disabled, respectively. As the bandwidth decreases, running time for both configurations tend to increase. However, the speeds at which the running times increase are not the same in the two configurations, with the running time with Helper Warps disabled eventually increasing faster and eventually surpassing the time with Helper Warps enabled. The point when these two running time curves cross each other is referred to as the crossover point. Given a benchmark, if the NVRAM bandwidth is below the crossover point, the Helper Warps approach will result in shorter running time than the baseline (no Helper Warps).

The crossover point varies in different benchmarks: The benchmarks with the highest crossover point to the ones with the lowest crossover point are H1, A2, H2, A1, R1, R2, B1 and B2, in decreasing order as listed in Fig. 14. Overall, the order matches a decrease in persistence intensity in those benchmarks; a benchmark with more writes are more likely to be bottlenecked by a bandwidth limit. To compare the different configurations of the same workload, H1 and A2 show less contention than H2 and A1 and therefore have a higher crossover points. The rest of the benchmarks, BVH construction and reduction, all have lower cross over point values than Hashtable and ATM. The reason is in those benchmarks each transaction write is preceded by more work than Hashtable and ATM. In other words, they do not have as high a write pressure as Hashtable and ATM. The write pressure may be seen in the persistence buffer utilization as well, which we will discuss in Section 7.2.

7.2 Persistence Buffer Utilization

The utilization of the Persistence Buffer is the number of write set entries that are in the Persistence Buffer, which reflects how "full" the Persistence Buffer is during run time. Utilization is affected by how frequent benchmark persists during run time.

When the Persistence Buffer is full, newly-committed transactions must wait for free space on the Buffer before it can persist. Thus, the buffer utilization can also reflect whether the Persistence Buffer itself has become the

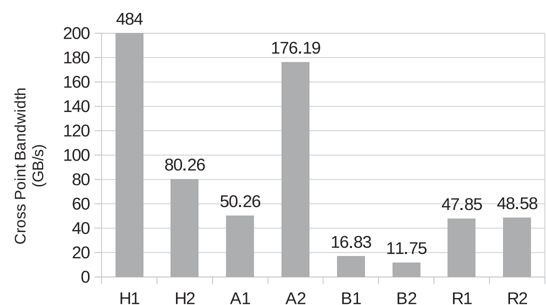


Fig. 14. The crossover point bandwidths.

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

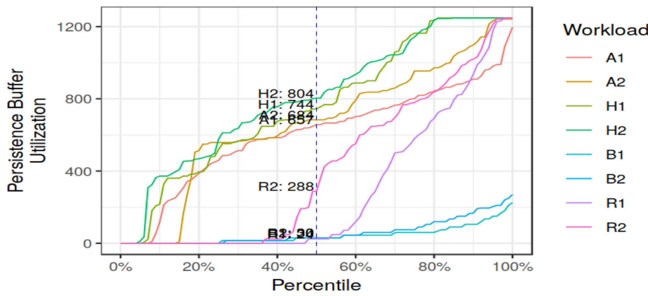


Fig. 15. Percentiles of persistence buffer utilization.

bottleneck. With a constantly-full buffer, the system throughput will be limited by how fast the Helper Warp can drain the Persistence Buffer. In extreme cases where all writes are coalesced and no contentions happen, Helper Warps may not be able to write as fast as normal warps simply because there are much fewer Helper Warps. For this case, disabling Helper Warps may yield better performance than enabling Helper Warps.

For a high-level overview, We aggregate the per-benchmark utilization measurements into a percentile map in Fig. 15. This figure presents an overview of the cumulative density function of Persistence Buffer utilization, throughout the life time of the workloads. The points on the curves with an X value of 50 percent correspond to the median of the utilization, while the values on the right end of the curves correspond to the maximum utilization observed through each benchmark. From the figure, we can see the benchmarks that stress the buffer the most, namely A1, A2, H1, H2, has a median utilization of 650~800 entries out of 1,280, or 50~63 percent full. In other words, for these benchmarks, the buffer is less than 50~63 percent full for half of the run time. This suggests the buffer is not being thrashed to become a bottleneck. Buffer utilization is benchmark-specific: benchmarks with high write pressure (H1, H2, A1 and A2) showing higher utilization than benchmarks with lower write pressure (B1, B2, R1 and R2).

Fig. 16 shows the details of how the buffer utilization changes over time. For the sake of brevity, only B1 and H1 are shown as representations of benchmarks with high and low utilization. From the figure, we can see that the buffers are drained almost as quickly as they are filled, despite there being only one Helper Warp serving multiple normal warps in a thread block. As a result, although the buffer may sometimes become full, it can be drained quickly enough such that the benchmark does not get stalled for too long. The buffers can be drained quickly partly because the

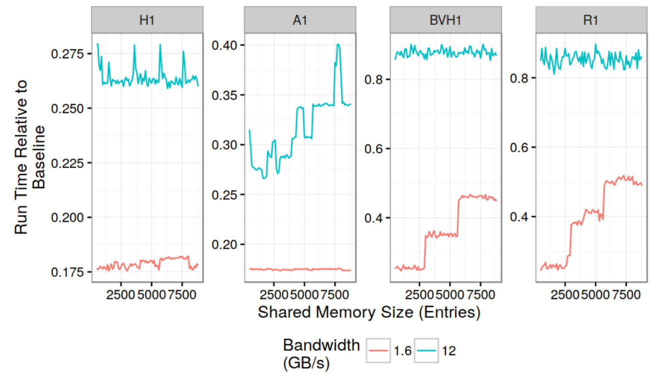


Fig. 17. Relative running time as a function of the shared persistence buffer size.

Helper Warp's write into the NVRAM is less scattered than writes in the baseline, where each transaction writes to different locations without coalescing. This, combined with adequate buffer space, results in the overall speedup of the method proposed in this paper.

7.3 Discussion

Determining the Shared Persistence Buffer Size. We ran the benchmarks across a range of persistence buffer between 500 entries to 9,000 entries. Fig. 17 shows the relative run time with respect to the baseline (i.e., without Helper Warps) for different persistence buffer sizes under two selected bandwidths for 4 of the benchmarks running with the metadata-based TM. For benchmark `H1`, the shared buffer size does not significantly affect the overall performance, but a staircase pattern is observed in benchmark `A1` with 12 GB/s bandwidth and `BVH` and `R1` with 1.6 GB/s bandwidth. The staircase pattern is a direct result of the limited concurrency permitted by the fixed L1 cache size in the SMs. From this figure we pick 1,300 entries as the persistence buffer size for all of the benchmarks.

Analysis of Transaction Timeline. Fig. 18 shows the commit timeline of transactions in block 0 for benchmark `A1`. The maximum commit count per clock cycle is equal to the warp size of 32. It can be observed that when persistence bandwidth is limited to 1.6 GB/s, a big gap appears between consecutive commits. Since behavior of different blocks will be similar, the gap will directly translate to longer overall running time. With Helper Warps, the gap is noticeably reduced, resulting in a much shorter running time for the benchmarks.

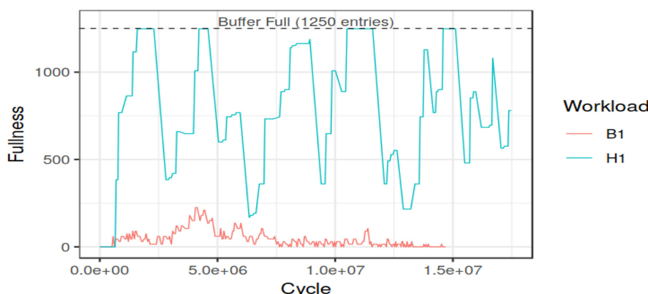


Fig. 16. Change of persistence buffer utilization over time.

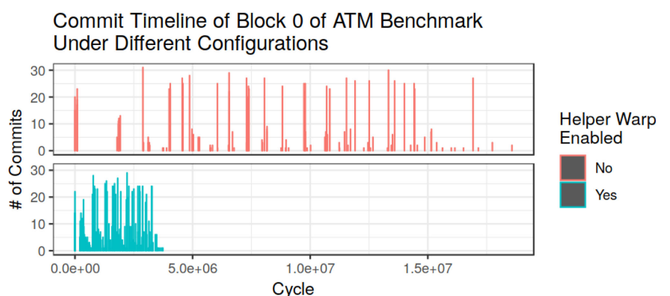


Fig. 18. Block-level transaction commit timeline for `A1` with metadata-based TM.

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

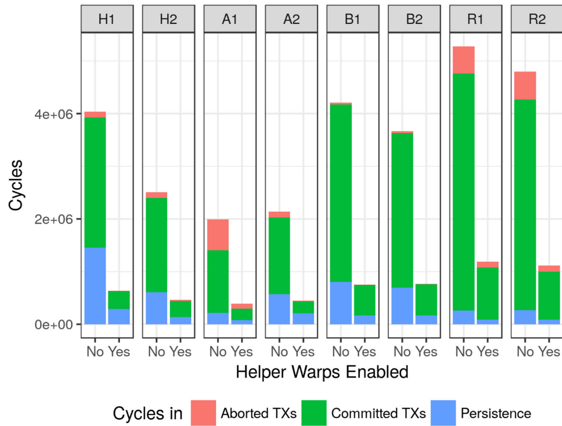


Fig. 19. Breakdown of the average execution time of transactions for the metadata-based TM used in this study.

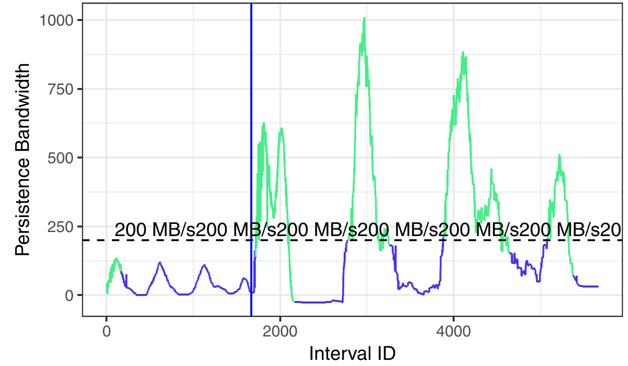


Fig. 21. Adaptive switching of Helper Warps and running time in the BVH benchmark (green=on, blue=off) (with two kernels, `B1` and `R1`, on either side of the blue line).

Transaction Execution Time Breakdown. Fig. 19 shows the breakdown of transaction execution time with Helper Warps either statically enabled and disabled. From the figure, the latency in the persistence phase introduced by limited bandwidth causes other committing transactions to run for significantly longer time than with Helper Warps. This is due to warp-level divergence, where transactions in the same warp as a committing transaction will have to wait for the lengthy persistence operation to complete. In addition, transactions holding the ownership records for a longer time also increase abort rates. By enabling Helper Warps, both the time spent in persistence and in executing transactions are reduced, leading to overall speedup.

7.4 Dynamic Switching of Helper Warps

We applied the switching mechanism described in Section 5.2 to various bandwidth limits. In extreme cases, the optimal choice is to either turn on Helper Warps throughout the benchmarks (when the bandwidth is set to the minimal value of 1.6 GB/s) or turn off Helper Warps (when set to the maximal value of 484 GB/s bandwidth.) For these two cases, dynamic switching does not yield noticeable benefits. With intermediate bandwidth limits, effects of dynamically switching starts to become noticeable. in while benchmarks are running. The effectiveness depends on the characteristics of the benchmarks: In Fig. 20, benchmarks `A1`, `B2`, `B2`, `R1` and `R2` see speedup of up to around 6 percent with dynamic switching compared to always enabling Helper Warps, while the rest of the benchmark showed slight slowdown due to switching overheads. We now take a closer look at a few of the benchmarks in detail to see why they react to dynamic switching differently.

Benchmarks B1 and R1. Fig. 21 shows the switching of Helper Warps in action in response to changing persistence bandwidth. This program consists of two kernels, `B1` followed by `R1`. In `B1`, each thread only writes one element, since in this kernel every transaction performs one operation on one node of the BVH tree; to compare, transactions in `R1` start from the leaf nodes of the tree and may go all the way up to the root node, thus the number of writes performed can be as many as the height of the tree. The result is `R1`'s higher persistence bandwidth than that of `B1`: While the persistence bandwidth of `B1` mostly stays below 125 MB/s, the bandwidth of `R1` spikes to nearly 1,000 MB/s. The spikes of both kernels correspond to the waves of commits at the different levels of the tree, with the spikes in `R1` taller than those in `B1`. Overall, dynamic switching reduced running time in both kernels (around 6 percent in `B1` and 0.8 percent in `R1`), resulting in an overall improvement of 20 percent compared to always turning off Helper Warps, or around 3 percent compared to always turning on Helper Warps, as shown in Fig. 22.

Benchmark A2. For some benchmarks, the persistence bandwidth observed exceeds the switching threshold threshold for most of the program execution, and `A2` is one of these benchmarks. Its persistence bandwidth trend is shown in Fig. 23. Dynamic switching for this benchmark results in the Helper Warps being turned on for the most of the duration of the benchmark; indeed, there is not much low-bandwidth regions that may benefit from Helper Warps being turned off. Overall, the running time is on par with always enabling Helper Warps, with a slight performance loss of around 0.3 percent as a result of the switching overhead. The results are shown in Fig. 24.

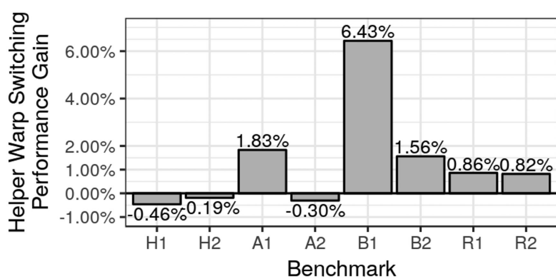


Fig. 20. Comparison of dynamic switching of Helper Warps versus always enable helper warps with 12 GB/s persistence bandwidth limit.

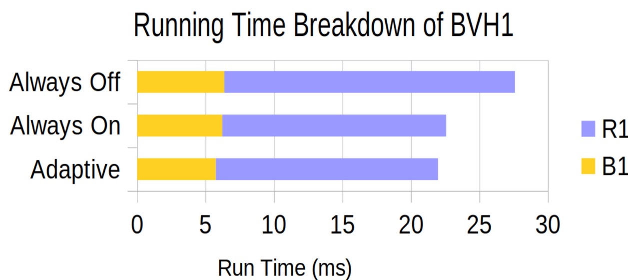


Fig. 22. Running time of `BVH1` with three different Helper Warp configurations.

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

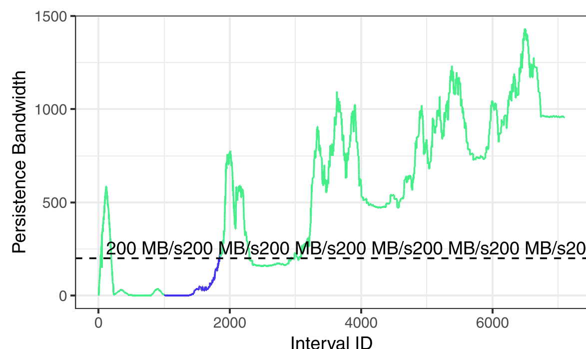


Fig. 23. Adaptive switching of Helper Warps (green=on, blue=off) and instantaneous bandwidth of A2.

Overall, the comparison between Figs. 22 and 24 suggests the performance gain mainly results from regions with a mix of low and high persistence bandwidths. With a correct choice of the switching threshold bandwidth, the time saved by dynamic switching can outweigh the switching overhead and result in overall performance gain.

7.5 Power and Energy Consumption

We measure power consumption using a Yokogawa WT210 [49] power meter. Because the power meter measures power draw from the wall socket, we derive GPU power consumption by subtracting the system idle power without GPU installed. Each benchmark is repeated 1,000 times so that the time duration is long enough to obtain a stable reading. Fig. 25 shows power, running time and energy consumption results (one row per metric). Energy consumption data obtained by multiplying average power reading with the average run time for each of the benchmarks.

Effects of helper warps on power consumption may be seen from the line pairs in the first column that correspond to the PCIe emulation method. From the line pair in the first row, we can see that the Helper Warps introduce additional power consumption, as the data points for all benchmarks appear higher than those without Helper Warps. The power consumption increase could also be due to the fact that most of the GPU time is spent in executing transactions rather than waiting for persistence operation, with executing transaction being more power-intensive. Despite higher power consumption, with reduced running time (line pair in the second row), the overall energy consumption is actually lower with Helper Warps enabled. For example, with benchmark A2 which has a big power increase from 125 to 187 W, a steep increase of about 50 percent, yet run time gets reduced by an even larger margin of about 62 percent (from 40 to 15 milliseconds), resulting in a net energy consumption reduction of about 44 percent.

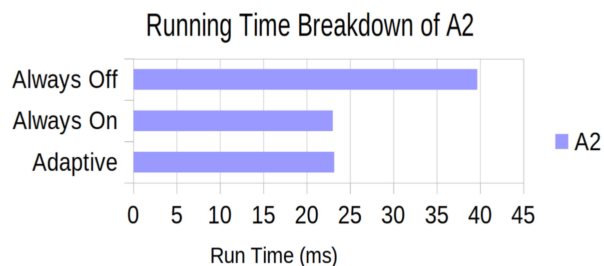


Fig. 24. Running time of A2 with 3 different Helper Warp configurations.

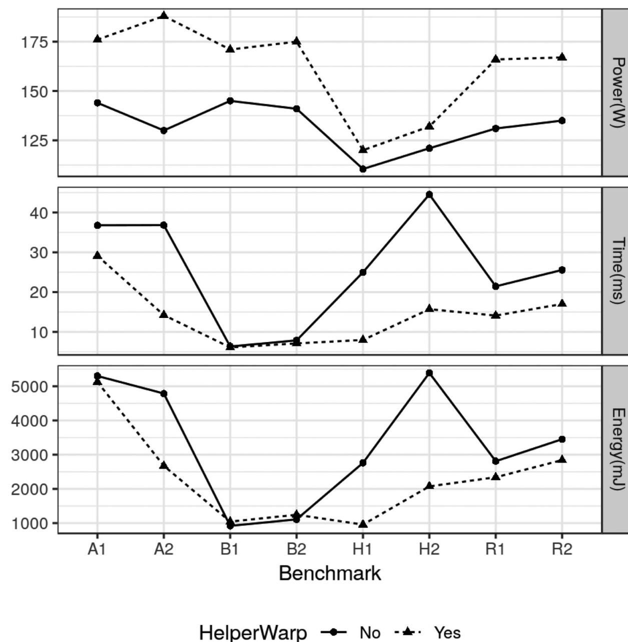


Fig. 25. Power and energy consumption with Helper Warps enabled and disabled and two bandwidth emulation methods.

8 Conclusion

In this paper, we have identified that the bandwidth limit of NVRAMs can result in longer persistence latency, due to the massive parallelism that exist on GPUs. When the NVRAM is used as a drop-in replacement of the main memory, the latency will be directly added onto the critical path of transactions, causing transactions to run longer. Further, this latency can affect other threads located in the same warp, which turns into even more running time overhead for entire benchmarks.

We have proposed Helper Warps, which consists of a persistence buffer located in the on-chip shared memory, where transaction persistence will be redirected to. This removes the time overhead on the critical path of transactions and makes the persistence operation faster. We also identified an optimal size for the shared buffer, which makes a tradeoff between larger buffer size and more concurrent thread blocks. Overall our proposed Helper Warps method yields better performance when the NVRAM write bandwidth does not exceed a threshold value, which can be up to hundreds of gigabytes per second in certain cases. This covers the range of NVRAM bandwidth available today and in the near future.

Acknowledgments

The authors would like to appreciate the invaluable comments from the anonymous reviewers. This work was supported in part by The US National Science Foundation (NSF) Grants CCF-1422408 and CNS-1527318. The authors would also like acknowledge the computing resources provided by the Louisiana Optical Network Initiative (LONI) HPC team.

References

[1] J. Zhao and Y. Xie, "Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2012, pp. 81–87.

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

[2] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit.*, 2014, pp. 265–276.

[3] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 421–432.

[4] T. Wang, J. Levandoski, and P. Larson, "Easy lock-free indexing in non-volatile memory," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 461–472.

[5] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proc. 48th Int. Symp. Microarchit.*, 2015, pp. 660–671.

[6] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep./Oct. 2011.

[7] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," in *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162.

[8] W. H. Highleyman, *Performance Analysis of Transaction Processing Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1989.

[9] P. M. Gray, D. S. Moffat, and N. W. Paton, "A prolog interface to a functional data model database," in *Proc. Int. Conf. Extending Database Technol.*, 1988, pp. 34–48.

[10] N. W. Paton and P. M. Gray, "Identification of database objects by key," in *Proc. Int. Workshop Object-Oriented Database Syst.*, 1988, pp. 280–285.

[11] N. W. Alkharouf, D. C. Jamison, and B. F. Matthews, "Online analytical processing (OLAP): A fast and effective data mining tool for gene expression databases," *BioMed Res. Int.*, vol. 2005, no. 2, pp. 181–188, 2005.

[12] C. Chen, X. Yan, F. Zhu, J. Han, and S. Y. Philip, "Graph OLAP: Towards online analytical processing on graphs," in *Proc. 8th IEEE Int. Conf. Data Mining*, 2008, pp. 103–112.

[13] J. M. Andrade, M. T. Carges, and M. R. MacBlane, "Open online transaction processing with the TUXEDO system," in *Proc. Dig. Papers COMPCON Spring*, 1992, pp. 366–371.

[14] J. Gray, "Database and transaction processing performance handbook," in *The Benchmark Handbook Database Trans. Syst.*, 2nd ed., Morgan Kaufmann Publishers, San Francisco, 1993.

[15] S. Breß, "The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS," *Datenbank-Spektrum*, vol. 14, pp. 199–209, Nov. 2014.

[16] Y. Yuan, R. Lee, and X. Zhang, "The Yin and Yang of processing data warehousing queries on GPU devices," *Proc. VLDB Endowment*, vol. 6, pp. 817–828, Aug. 2013.

[17] B. He and J. X. Yu, "High-throughput transaction executions on graphics processors," *Proc. VLDB Endowment*, vol. 4, pp. 314–325, Feb. 2011.

[18] C. Root and T. Mostak, "MapD: A GPU-powered big data analytics and visualization platform," in *Proc. ACM SIGGRAPH Talks*, 2016, pp. 73:1–73:2.

[19] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl, "Hardware-oblivious parallelism for in-memory column-stores," *Proc. VLDB Endowment*, vol. 6, pp. 709–720, Jul. 2013.

[20] S. Zhang, J. He, B. He, and M. Lu, "OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures," *Proc. VLDB Endowment*, vol. 6, pp. 1374–1377, Aug. 2013.

[21] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. 20th Annu. Int. Symp. Comput. Archit.*, 1993, pp. 289–300.

[22] Samsung Semiconductor Inc., "Performance benefits of running RocksDB on Samsung NVMe SSDs." 2015. [Online]. Available: https://www.samsung.com/us/labs/pdfs/collateral/Performance-Benefits-of-Running-RocksDB-on-SSDs_Whitepaper.pdf

[23] L. Liu, S. Yang, L. Peng, and X. Li, "Hierarchical hybrid memory management in OS for tiered memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2223–2236, Oct. 2019.

[24] M. Shihab, J. Zhang, S. Gao, J. Sloan, and M. Jung, "Couture: Tailoring STT-MRAM for persistent main memory," in *Proc. 4th Workshop Interact. NVM/Flash Operating Syst. Workloads*, 2016, pp. 1–6.

[25] M. Wu and W. Zwaenepoel, "eNVy: A non-volatile, main memory storage system," *ACM SIGPLAN Notices*, vol. 29, pp. 86–97, Nov. 1994.

[26] M. Liu *et al.*, "DudeTM: Building durable transactions with decoupling for persistent memory," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017, pp. 329–343.

[27] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: Atomic durability in non-volatile memory through hardware logging," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Feb. 2017, pp. 361–372.

[28] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with WHISPER," *SIGOPS Operating Syst. Rev.*, vol. 52, pp. 135–148, Apr. 2017.

[29] J. Coburn *et al.*, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGPLAN Notices*, vol. 46, pp. 105–118, Mar. 2011.

[30] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," *ACM SIGPLAN Notices*, vol. 52, pp. 91–104, Apr. 2017.

[31] D. Cederman, P. Tsigas, and M. Chaudhry, "Towards a software transactional memory for graphics processors," in *Proc. Eurographics Symp. Parallel Graphics Vis.*, 2010, pp. 121–129.

[32] A. Holey and A. Zhai, "Lightweight software transactions on GPUs," in *Proc. 43rd Int. Conf. Parallel Process.*, 2014, pp. 461–470.

[33] S. Irving, S. Chen, L. Peng, C. Busch, M. Herlihy, and C. J. Michael, "CUDA-DTM: Distributed transactional memory for GPU clusters," in *Proc. Int. Conf. Netw. Syst.*, 2019, pp. 183–199.

[34] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware transactional memory for GPU architectures," in *Proc. 44th Int. Symp. Microarchit.*, 2011, pp. 296–307.

[35] W. W. L. Fung and T. M. Aamodt, "Energy efficient GPU transactional memory via space-time optimizations," in *Proc. 46th Int. Symp. Microarchit.*, 2013, pp. 408–420.

[36] S. Chen and L. Peng, "Efficient GPU hardware transactional memory through early conflict resolution," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 274–284.

[37] S. Chen, L. Peng, and S. Irving, "Accelerating GPU hardware transactional memory with snapshot isolation," in *Proc. ACM/IEEE 44th Int. Symp. Comput. Archit.*, 2017, pp. 282–294.

[38] B. Wang *et al.*, "Exploring hybrid memory for GPU energy efficiency through software-hardware co-design," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 93–102.

[39] S. Chen, F. Zhang, L. Liu, and L. Peng, "Efficient GPU NVRAM persistence with helper warps," in *Proc. 56th ACM/IEEE Annu. Design Autom. Conf.*, 2019, Art. no. 155.

[40] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3D Xpoint technology," *Proc. IEEE*, vol. 105, no. 9, pp. 1822–1833, Sep. 2017.

[41] E. Kim, "'How fast is fast?' Block IO performance on a RAM disk," in *Proc. Storage Netw. Ind. Assoc. Data Storage Innov. Conf.*, Art. no. 13, 2015.

[42] M. Shantharam, K. Iwabuchi, P. Cicotti, L. Carrington, M. Gokhale, and R. Pearce, "Performance evaluation of scale-free graph algorithms in low latency non-volatile memory," in *Proc. Int. Parallel Distrib. Process. Symp. Workshops*, 2017.

[43] Intel, "Intel optane DC persistent memory now sampling," 2018. [Online]. Available: http://www.legitreviews.com/intel-optane-dc-persistent-memory-now-sampling_205757. Accessed on: May 30, 2018.

[44] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonia, "Managing DRAM latency divergence in irregular GPGPU applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 128–139.

[45] N. Vijaykumar *et al.*, "A case for core-assisted bottleneck acceleration in GPUs: Enabling flexible data compression with assist warps," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 41–53.

[46] Y. Huang, K. Wu, and D. Li, "High performance data persistence in non-volatile memory for resilient high performance computing," in *CoRR*, May 2017.

[47] NVIDIA Corporation, "NVIDIA Quadro dual copy engines," 2010. [Online]. Available: https://www.nvidia.com/docs/IO/40049/Dual_copy_engines.pdf. Accessed on: Aug. 01, 2018.

[48] T. Karras, "Maximizing parallelism in the construction of BVHs, Octrees, and K-D Trees," in *Proc. 4th ACM SIGGRAPH / Eurographics Conf. High-Perform. Graphics*, 2012, pp. 33–37.

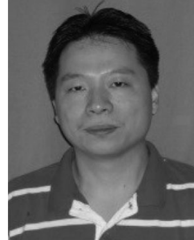
[49] Yokogawa, "WT210/WT230 digital power meters," 2009. [Online]. Available: http://www.electro-meters.com/Assets/pdf2_files/Yokogawa/Power_meters/WT200/WT210_WT230_Manual.pdf. Accessed on: Oct. 07, 2019.

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.

Sui Chen received the bachelor's degree in information security from Shanghai Jiao Tong University, Shanghai, China, in 2011, and the master's and PhD degrees from Louisiana State University, Baton Rouge, Louisiana, in 2016 and 2017, respectively. He is currently a software engineer with AMD, Santa Clara, California. His research interests include GPU performance, transactional memory, and resilience.

Lei Liu is an associate professor with the Institute of Computing Technology (ICT), CAS, where he leads the Sys-Inventor Research Group. His research interests include OS, memory architecture, and computer architecture. His efforts are published in ISCA, PACT, the *IEEE Transactions on Computers*, *ACM Transactions on Architecture and Code Optimization*, etc.

Weihua Zhang received the PhD degree in computer science from Fudan University, Shanghai, China, in 2007. He is currently a professor of Parallel Processing Institute, Fudan University. His research interests include compilers, computer architecture, and parallelization and systems software.



Lu Peng received the bachelor's and master's degrees in computer science and engineering from Shanghai Jiao Tong University, Shanghai, China, and the PhD degree in computer engineering from the University of Florida, Gainesville, Florida. He is currently the Gerard L. "Jerry" Rispone professor with the Division of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, Louisiana. His research interests include memory hierarchy system, reliability, power efficiency, and other issues in processor design. He was a recipient of the ORAU Ralph E. Power junior faculty enhancement awards in 2007 and the Best Paper Award from IEEE International Green and Sustainable Computing Conference (IGSC) in 2019 and IEEE International Conference on Computer Design (ICCD) processor architecture track in 2001. He is a senior member of the IEEE and ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Louisiana State University. Downloaded on February 26,2020 at 20:25:18 UTC from IEEE Xplore. Restrictions apply.