Efficient GPU NVRAM Persistence with Helper Warps

Sui Chen[*], Faen Zhang[†], Lei Liu[‡], Lu Peng[*]
[*]Louisiana State University, [†]Ainnovation Technology Ltd.
[‡]SKLCA, Institute of Computing Technology, Chinese Academy of Sciences
{csui1, lpeng}lsu.edu, zhangfaen@ainnovation.com, liulei2010@ict.ac.cn

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. To utilize the non-volatility of NVRAMs, programs should allow durable stores, meaning consistency must be maintained during a power loss event. GPUs are designed with high throughput, leveraging high degrees of parallelism. However, with lower NVRAM write bandwidths compared to that of DRAMs, using NVRAM as is may yield sub-optimal overall system performance. 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.

KEYWORDS

Non-volatile Random-Access Memories, GPU, Persistence, Help Warps.

1. Introduction

Non-volatile Random-Access Memory (NVRAM) has emerged and has been maturing in the past few years as a promising replacement for the DRAM. With its large capacity and durability, NVRAMs can enable and justify new programming paradigms such as transactional memory.

A byte-addressable, durable storage device such as NVRAM may be used in a few different ways. In the simplest form, it may be used as a large-capacity, drop-in replacement for the DRAM or the cache. This type of system has been discussed on both the CPU and GPU [1] but does not leverage its durability property. Another more sophisticated approach is to use the NVRAM as a persistent data store, making it an integral part of a transaction processing system (TPS). The architecture of a TPS usually involves two layers: a *concurrency protocol layer*, which may be embodied as a transactional memory or a locking mechanism, is responsible for detecting and resolving conflicts between transactions; and a *logging layer* that performs writes in the form of journal logs to achieve durability that can maintain data integrity during a power loss event. On the CPU, such TPS systems can involve hardware and software; the GPU is steps behind the CPU, as there exists works on transactional memory but not TPS systems based on NVRAM at the current moment.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317810

Despite its larger storage density, NVRAM provides less bandwidth than DRAM and the cache. Therefore, bandwidth-induced latency needs to be managed well to avoid performance degradation. Software and hardware approaches are needed to alleviate the penalty induced by the bandwidth gap.

In this paper, we make the following contributions:

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

2. Related Work

Transactional Memory (TM) [2] 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 [3, 4] utilize various GPU-centric optimizations that to utilize the GPU's parallel processing power to perform basic tasks in the TM system in parallel, such as lock management, and coalesced read and write log access, and using GPU-friendly data layout such as structures-of-arrays instead of arrays-of-structures. With these efforts combined, the GPU-based STM systems can rival or even outperform CPU-based STM systems.

For hardware approaches [5, 6, 7], these proposals feature hardware-implemented TM algorithms, as well as new hardware architectures that provide new versioning techniques and new conflict detection mechanisms.

For efficient use of NVRAM on the CPU, DudeTM [8] and ATOM [9] achieve high write/persistence performance by decoupling, i.e. by performing the persistence step outside the critical path. These systems mainly focused on CPUs. NV-Heaps [10] discusses the interoperation between the NVRAM and DRAM are used simultaneously. Kiln [11] proposes to add non-volatile caches and form a multi-versioned persistent system.

NVRAM has also been considered for enhancing existing GPU cache and memory subsystem. A recent work [12] discerns access patterns and manages hybrid DRAM and NVRAM accordingly. Due to 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, 13]. The existing work do not utilize persistence. In contrast, we will support persistence for NVRAM using in GPUs in this paper.

3. Limited NVRAM Bandwidth on GPUs

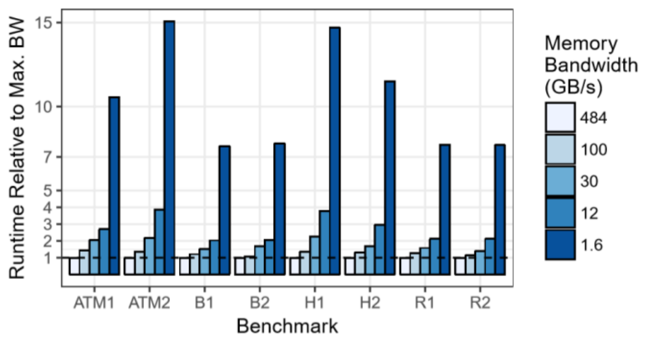


Figure 1: Impact of bandwidth on execution time.

Today's NVRAM devices deliver lower bandwidth than that of DRAMs. This performance gap may be observed in 4K block operations utilizing either technology, with one of today's high-end NVRAM devices delivering 500K write I/O operations per second (IOPS) [17], significantly lower than the 3,000K write IOPS of RAM disks in the same PCI-E interface [18]. Recent researches achieved aggregate bandwidths ranging from several GB/s to tens of GB/s [19]. Especially, the Intel PMFS offers two options for aggregate NVRAM bandwidth: 9.5 GB/s and 37 GB/s. This is in the same ballpark as that of the current 3D XPoint commercial products [15] which have a write bandwidth of 6 GB/s.

The bandwidths illustrated above are both lower than a state-of-the-art GPU, the NVIDIA GTX 1080Ti or AMD Vega 64, both at 484 GB/s. As a motivating example, we run a few benchmarks with bandwidth limits enforced on their persistent store. The impact of bandwidth limits on bandwidth-bound transaction processing workloads is easily visible: the total program running time will become several times (up to 15x) longer compared with higher bandwidth system, as shown in Figure 1.

We have observed that a transactional program consists of three stages, *execute*, *commit* and *persist*. The *persist* step is more memory-constrained than the other two stages, with some compute resources rendered idle while waiting for memory accesses. Fortunately, by performing the *persist* step asynchronously, overlapping it with the other stages, the impact of bandwidth limit and the latency may be greatly mitigated or even eliminated. In this paper, we design and implement a software-based approach for this purpose. The idea can also be expanded to hardware-based design.

4. Efficient GPU NVRAM Persistence Support

NVRAM provides data durability, and is usually used with a logging mechanism for consistency. Just like a file system that is set up on top of a disk or SSD, logging enables error-correction and maintains data consistency in power outage events. In various logging implementations, writes are first buffered into logs (called "persist") and then used to evolve the system state stored in the NVRAM (called "reproduce"). Log generation usually requires a certain concurrency control mechanism in place to resolve conflicts between participating transactions. For this purpose, a software/hardware transactional memory (STM/HTM), or an equivalent (such as two-phase locking) is usually used to work with the logging system and combined as a transaction processing system.

4.1 Transaction Processing

```
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
```

Figure 2: The STM algorithm used in this paper.

Transaction processing usually consists of two parts, concurrency control and persistence logging. The system we investigate in this paper uses Software Transactional Memory (STM) for concurrency control. Our STM algorithm employs eager conflict detection and redo-logging and resolves conflicts with global ownership records. The granularity of write/read set tracking is a 32-bit machine word. Accesses to data of larger sizes are treated as multiple 32-bit machine words. This algorithm does not distinguish between read and write. Conflicts are resolved by favoring transactions with a lower thread ID.

The structure of the TM algorithm, from the transaction's point of view, is described in Figure 2. When a transaction T_A performs a transactional read or a write (Line 1), it first attempts to take the ownership record for the corresponding machine word. When there are no other transactions holding this ownership record (Line 4), the current transaction takes ownership with an atomic compare-and-swap (CAS) (Line 5). The usage of CAS guarantees that when there are multiple transactions attempting to take ownership, only one would succeed. The situation when another transaction T_B already owns this ownership record (Line 3) is called a conflict, which must be resolved by aborting one of the transactions: if T_A's thread ID is smaller than T_B's, T_A may preempt this ownership record and signal T_B to abort (Line 10), which is also done with an atomic CAS. If T_B and T_A are the same transaction nothing needs to be done (Line 14). Otherwise, T_A must abort itself (Line 16). This global ordering of priority prevents deadlock and ensures system-wide progress. Implementation-wise, because it's difficult to directly send signals to individual threads on the GPU, aborts are handled by having threads check if their own status flags have been modified transactions as "abort". The check happens at commit stage (Line 20). An aborted transaction will relinquish all ownership records it has taken so far and discard its write log. A successfully committed transaction will flush its write log to both the volatile memory and the NVRAM.

In the STM algorithm shown in Figure 2, writes to the NVRAM occur during a successful commit. Under the default Strict Persistency model [14], a transaction must wait for the persist operation to complete before declaring a successful commit. This adds the

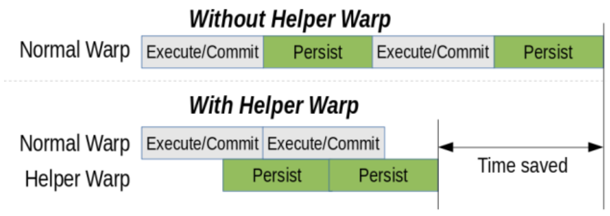


Figure 3: Transaction timeline in the proposed architecture.

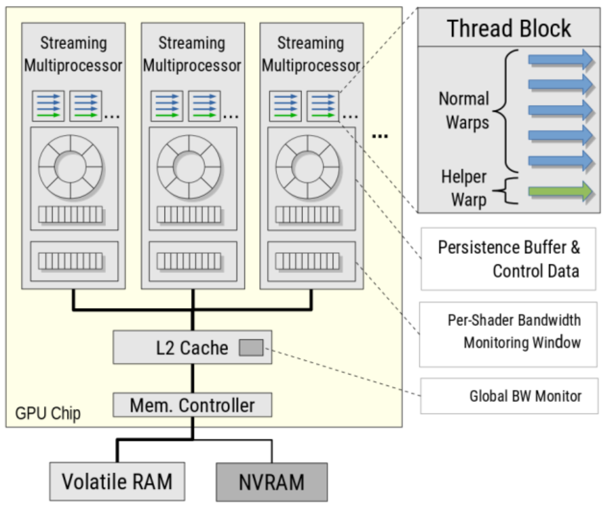


Figure 4: Overall system architecture.

NVRAM write latency onto the critical path of transaction execution, resulting in the overhead seen in Figure 1. To address this issue, we propose a commit procedure utilizing Helper Warps to move the delay away from the critical path.

4.2 Efficient Logging System with Helper Warp

Our proposed method separates the commit and persist steps of transactions using Helper Warps. The Helper Warps are responsible for handling the persistence portion of transactions, enabling persist operations to complete asynchronously with the rest of the transaction. Figure 3 shows the overall commit protocol with the Helper Warps added.

There is a Helper Warp residing in each thread-block, and it communicates with the normal warps via the per-thread-block shared memory, where a *persistence buffer* is located. In addition, each streaming multiprocessor (SM) has a bandwidth monitoring window which is used to keep track of the instantaneous persistence bandwidth during run time. Figure 4 illustrates the proposed architecture, including the memory topology and added parts. The connection between the volatile RAM and NVRAM is like the one found in the recent AMD Vega architecture [17] which is designed to support heterogeneous memory architecture, such as SSD and DRAM.

The persistence buffer is conceptually a FIFO queue physically implemented as a ring buffer. Address-Value pairs are enqueued by committing transactions and are drained by the Helper Warp. Because there can be many more normal warps than logging warps, the incoming write addresses are expected to be high-volume bursts of writes. The Helper Warp drains the ring buffer steadily and shapes the traffic into steady low-volume writes.

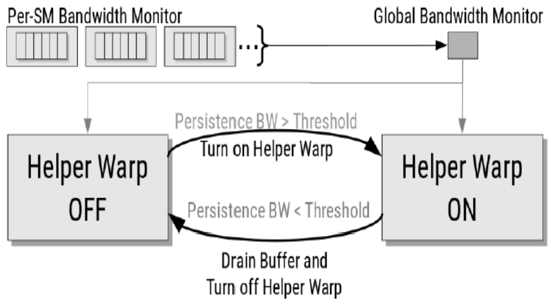


Figure 5. Run-time Helper Warp Adaptation Process.

Once a committing transaction finishes flushing its write set into the buffer, the transaction can safely update the volatile memory, release its ownership records, and allow its written values to be read by other transactions. When needed, the notification completion of persistence can be made by the Helper Warps.

The per-SM persistence buffers consist of an array of entries, each consisting of a 40-bit address (for a memory space of 4TB); and a 32-bit write value. Each buffer also contains control data including buffer head and tail pointers (one per SM) and dirty bits (one bit per entry) indicating whether each entry is updated or not. Therefore, the size cost of one entry is 40+32+1 = 73 bits. The size is small enough so it's possible to use part of the per-SM shared memory for the persistence buffers. Current-generation GPUs provide users 64KB~96KB of shared memory that can be accessed by all the warps running on the same SM. In practice, a buffer with size around 1000 would suffice (around 10 KB), which is acceptable compared with the size of the shared memory. We use a size of 1300 which gives the best balance of concurrency (maximum thread blocks allowed) and buffer size.

4.3 Correctness

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. The memory hierarchy layers in this paper involves the DRAM, the volatile write logs, non-volatile write logs, and the NVRAM system state. Each layer has its own mechanism of guaranteeing its own correctness and the correctness of the next layer for overall system correctness.

The STM layer guarantees execution correctness in the DRAM. This means the volatile and non-volatile write logs of each transaction will be conflict-free, so we only need to consider potential conflicts between transactions committed at different times. This can be handled by labeling each write log with a transaction ID, so a transaction committed later will always overwrite one committed earlier. The persist step checks the completeness of write logs before evolving the NVRAM system state. During recovery from a power loss event: partially persisted logs will be discarded, while the logs that have completed the persist step and are partially reproduced will be simply be reproduced again, overwriting the partially-reproduced data.

The Helper Warps will only affect how the volatile and non-volatile write logs are ordered against each other, and have no direct effect on the other layers. Correctness in the persist step will be handled in the same way as without Helper Warps. To put the proposed method under the Memory Persistency [15] perspective, the Helper Warps changes the Strict Persistency in the baseline to *Buffered Strict Persistency*, from the non-volatile write log point-of-view.

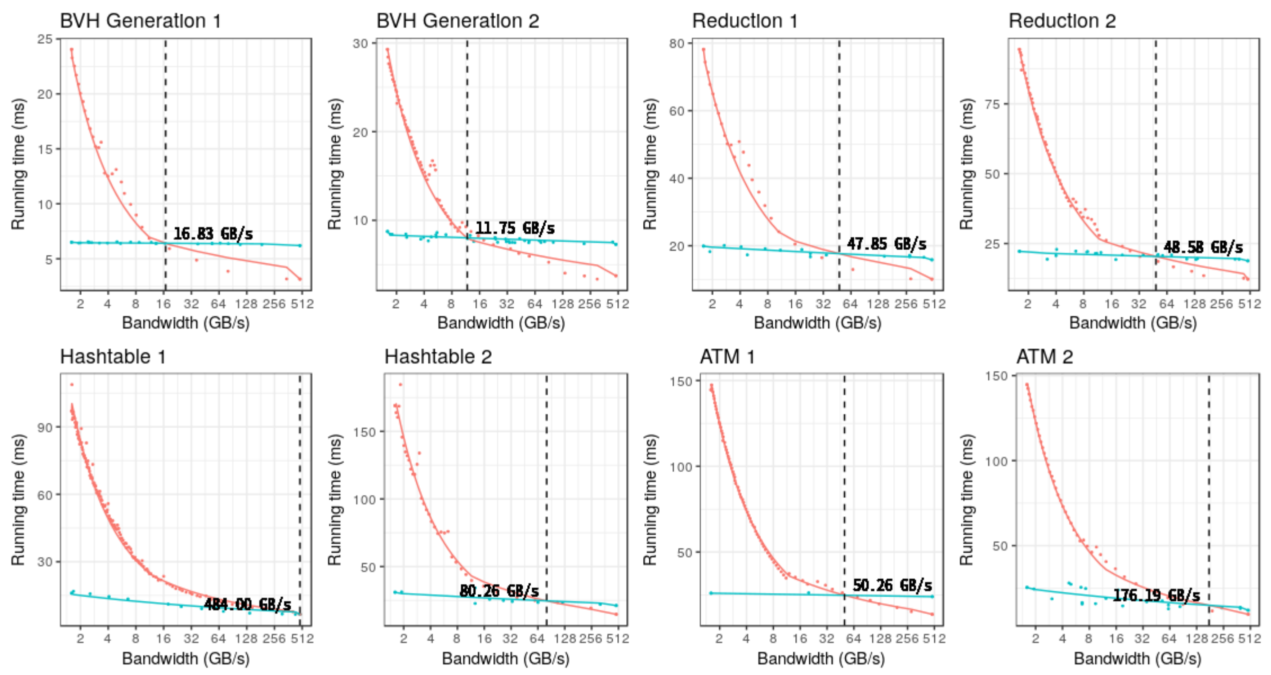


Figure 6: Overall running time of the benchmarks, with helper warps enabled (green) and disabled (red).

5. Performance Tuning

5.1 Persistence Bandwidth Monitoring

For profiling and performance tuning, we keep track of the instantaneous persistence bandwidth. Due to the distributed architecture of GPU, the bandwidth is computed by each SM and accumulated globally. In each SM, 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 Figure 4. Per-SM data are accumulated at the *global bandwidth monitor* which can be consulted by individual SMs to decide when to turn on/off Helper Warps.

5.2 Adaptively Enabling Helper Warps

Although Helper Warps could reduce the persistence delay of transactions, speedup can only be observed when the persistence time saved outweighs the overhead. The overhead includes the cost to allocate entries in the persistence queue and waiting for the queue to be drained by the Helper Warps at high write volumes. For a program to achieve maximum efficiency, it should be able to automatically determine when the Helper Warp should be enabled or disabled according to program behavior.

We use microbenchmark-based profiling to determine the threshold for turning on/off Helper Warps. This is independent of the NVRAM hardware connected to the GPU and can be used when a new NVRAM device is connected to the GPU or during device initialization. When turning off the Helper Warps, transactions on a SM will wait for the persistence queue to be drained such that strict buffered strict persistency may be maintained. The whole run-time adaptation process is illustrated in Figure 5.

6. Experimental Setup

6.1 Hardware and Software Platform

Table 1: Benchmarks used in this paper

A1	ATM Bank Transfer, 1M transfers, 100K accounts
A2	ATM Bank Transfer, 1M transfers, 1M accounts
H1	Hash Table, 500K inserts, 15M base entries
H2	Hash Table, 900K inserts, 1M base entries
B1	BVH [20] generation, 100,000-face model
R1	BVH reduction, 100,000-face model
B2	BVH generation, 187,854-face model
R2	BVH reduction, 187,854-face model

We use real-system evaluation because it allows us factor in system-wide factors such as software stack and OS overheads. Experiments in this paper are run on an NVIDIA Pascal GPU, the GTX 1080 Ti, which has 56 streaming multiprocessors (SM), operating at a processor clock of 1582 MHz, and 11 GB of GDDR5X memory. There exist 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 Figures 2.

6.2 Bandwidth Emulation

In this paper, we study bandwidth limits ranging from 1.6 GB/s to 484 GB/s (the latter is the original bandwidth of the GPU used in the study.) The bandwidth limits are achieved using artificial delays. We establish the correlation between the delays and the underlying bandwidth limits by using the program-observed bandwidth as a proxy variable. The detailed steps are as follows:

First, we measure execution time by having the benchmarks persist into the GPU RAM (whose bandwidth is 484 GB/s), as well as on the zero-copy pinned memory accessible through the PCI-E bus (for 1.6 GB/s and 12 GB/s). We call these measurements as *reference points*. Second, we add artificial latency in the

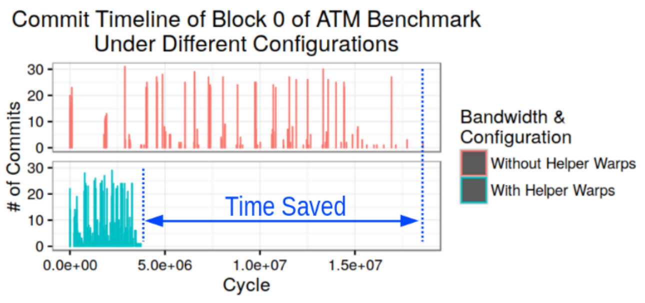


Figure 7: Block-level transaction commit timeline for benchmark A1.

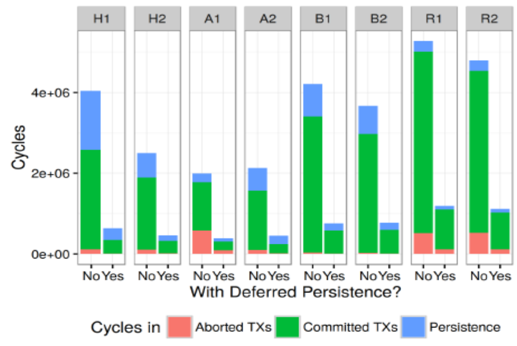


Figure 8: Breakdown of the average execution time of transactions for metadata-based TM.

persist operation to emulate limited bandwidth on the NVRAM between the measurements in step 1. We vary the artificial latency. For each latency value, we run a write pressure test (transactions that only perform a non-conflicting write) and get a corresponding program-observed memory bandwidth, called the *proxy bandwidth*. We then map the program-observed bandwidths to the delay values using linear interpolation. Thus, the delay value for a desired memory bandwidth limit can be obtained simply by performing a lookup.

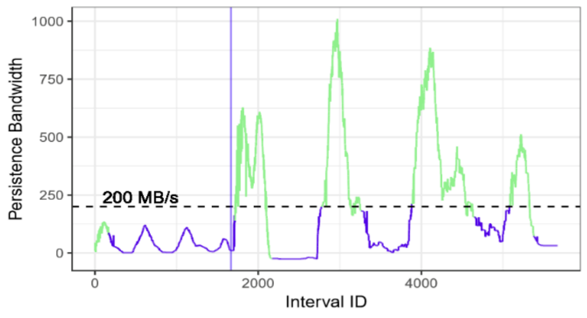
6.3 Benchmarks

We use a series of transactional benchmarks to evaluate the Helper Warp mechanism proposed in this paper. The details are listed in Table 1.

7. Evaluation

7.1 Overall Results

Figure 6 shows the run time of the benchmarks with and without Helper Warps, using the experimental setup. 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 increase. However, the running time without Helper Warps will eventually increases faster and surpasses the time with Helper Warps. There exists a point when these two running time curves cross each other, which we refer to as the cross-point. The cross-point is as high as 484 GB/s for H1 (meaning Helper Warps perform better even at volatile RAM bandwidth) or as low as 16.83 GB/s (11.75GB/s) for BVH1 (BVH2). For other cases, the cross-point hovers between tens of GB/s to around 100 GB/s, which we



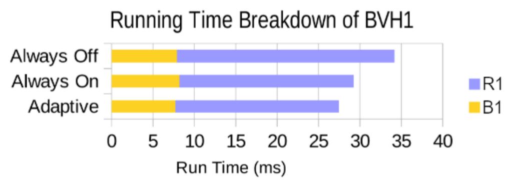


Figure 9: Persistence bandwidth trend for benchmark B1+R1 with adaptive switching of Helper Warps (top) and running time breakdown for 3 configurations (bottom).

expect to be in the range of achievable NVRAM bandwidths given existing technologies.

7.2 Discussion

Analysis of transaction timeline. Figure 7 shows the commit timeline of transactions in block 0 in 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.

Transaction execution time breakdown. Figure 8 shows the breakdown of transaction execution time in thread block 0, with Helper Warps statically turned on and off. The latency in the persistence phase introduced by limited bandwidth causes a "cascade" effect, making other committing transactions longer time than with Helper Warps. This is due to warp-level divergence and the holding of ownership records making committing transaction wait for the lengthy persistence operation to complete. This also increases abort rates. By enabling Helper Warps, persistence completes faster, and the "cascade" effect is mitigated.

7.3 Dynamic Switching of Helper Warps

We applied the switching mechanism described in Section 5.2 to various bandwidth limits. In extreme cases such as 1.6 GB/s and 484 GB/s, the method decides to either turn on Helper Warps in all situations (for the minimal, 1.6 GB/s bandwidth) or turn off in all situations (for 484 GB/s bandwidth.)

However, with intermediate bandwidth limits, the bandwidth is set such that the switch happens within the program run and gives better performance than either statically turning on or off the Helper Warps.

Figure 9 (top) shows the switching of Helper Warps in action in response to changing persistence bandwidth. Kernel B1 is run and immediately 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; in comparison, transactions in

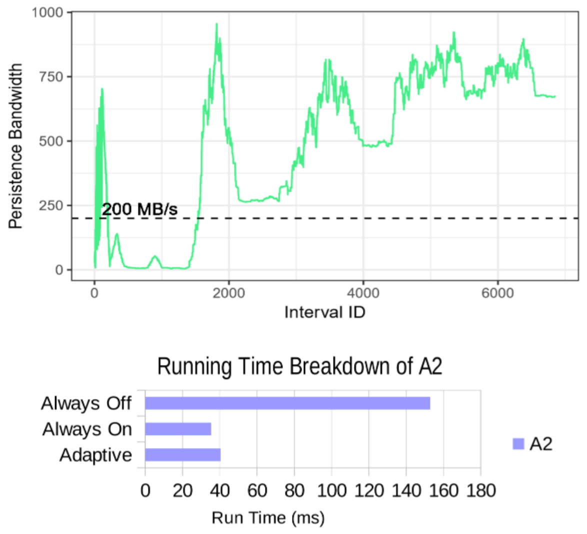


Figure 10: Persistence bandwidth trend for benchmark A2 with Helper Warp turned off (top) and running time breakdown for 3 configurations (bottom).

R1 start from the leaf nodes of the tree and may travel all the way to the root node, thus the number of writes performed can be as many as the depth of the tree. As a result, R1 writes more data than B1: While the persistence bandwidth of B1 mostly stays below 125 MB/s, the bandwidth of R1 spikes to nearly 1000 MB/s. Overall, the switching significantly reduced the time in R1 kernel and results in an improvement in running time of 20% compared to always turning off Helper Warps, or 6% compared to always turning on Helper Warps, as shown in Figure 9 (bottom).

In contrast to the BVH benchmark, some other benchmarks will observe a committing bandwidth that is higher than the threshold for most of the program execution, such as A2. Its persistence bandwidth trend may be observed in Figure 10 (top). Turning on or off the Helper Warps statically throughout for this benchmark results in a slightly performance loss as shown in Figure 10 (bottom), due to the overhead involved in switching.

8. Conclusion

In this paper, we observed the performance penalty for transactional GPU programs, resulting from the bandwidth limit of NVRAMs, which causes long persistence latency. 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 commits will be redirected to. This removes the time overhead from the critical path, making the persistence operation faster. We also proposed a method to enable the Helper Warps only when necessary for best performance. 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 for today and the near future.

ACKNOWLEDGEMENTS

We appreciate the invaluable comments from the anonymous reviewers. This work is supported in part by US National Science Foundation (NSF) grants CCF-1422408 and CNS-1527318. We also acknowledge the computing resources provided by the Louisiana Optical Network Initiative (LONI) HPC team. The corresponding authors are Lu Peng and Lei Liu.

REFERENCES

[1] J. Zhao and Y. Xie, “Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration,” in 2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 81–87, Nov 2012.

[2] M. Herlihy and J. E. B. Moss, Trasactional Memory: Architectural Support for Lock-free Data Structures. IEEE Computers Society Press, 1993.

[3] Y. Xu, R. Wang, N. Goswami, and T. Li, “Software Transactional Memory for GPU Architectures,” Proceedings of the International Symposium on Code Generation and Optimization (CGO), 2014.

[4] A. Holey and A. Zhai, “Lightweight Software Transactions on GPUs,” in Proceedings of the 43rd International Conference on Parallel Processing (ICPP), 2014.

[5] W. W. L. Fung and T. M. Aamodt, “Energy efficient GPU transactional memory via space-time optimizations,” in Proceedings of the 46th International Symposium on Microarchitecture (MICRO), 2013.

[6] S. Chen and L. Peng, “Efficient GPU Hardware Transactional Memory through Early Conflict Resolution,” in Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 274–284, March 2016.

[7] S. Chen, L. Peng, and S. Irving, “Accelerating GPU Hardware Transactional Memory with Snapshot Isolation,” 2017 ACM/IEEE 44th International Symposium on Computer Architecture (ISCA), vol. 45, pp. 282–294, June 2017.

[8] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, “DudeTM: Building durable transactions with decoupling for persistent memory,” in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’17, (New York, NY, USA), pp. 329–343, ACM, 2017.

[9] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, “Atom: Atomic durability in non-volatile memory through hardware logging,” in 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 361–372, Feb 2017.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories,” SIGPLAN Not., vol. 46, pp. 105–118, Mar. 2011.

[11] 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 Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, (New York, NY, USA), pp. 421–432, ACM, 2013.

[12] L. Liu, S.-J. Yang, L. Peng, and X. Li, “Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems,” IEEE Transactions on Parallel and Distributed Systems (TPDS), 2019.

[13] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter, “Exploring Hybrid Memory for GPU Energy Efficiency through Software-Hardware Co-Design,” in Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT ’13, (Piscataway, NJ, USA), pp. 93–102, IEEE Press, 2013.

[14] S. Pelley, Peter M. Chen, Thomas F. Wenisch, “Memory Persistency,” 2014 ACM/IEEE 41th International Symposium on Computer Architecture (ISCA), vol. 42, pp. 265-276, June 2014.

[15] Intel Corporation, “Intel Optane DC Persistent Memory Now Sampling.” http://www.legitreviews.com/intel-optanedc-persistent-memory-now-sampling_205757, 2018. Retrieved on 2018-11-24.

[16] Advanced Micro Devices, “Radeon’s next-generation Vega architecture”, 2017. Retrieved on 2018-11-24.

[17] F. T. Hady, A. Foong, B. Veal, and D. Williams, “Platform storage performance with 3d xpoint technology,” Proceedings of the IEEE, vol. 105, pp. 1822–1833, Sept 2017.

[18] E. Kim, “‘How Fast is Fast?’ Block IO Performance on a RAM Disk,” Proceedings of the Storage Networking Industry Association (SNIA) Data Storage Innovation Conference (DSI 2015).

[19] 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," 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Orlando, Florida, USA, 2017, pp. 1021-1028.

[20] T. Karras, “Maximizing parallelism in the construction of bvhs, octrees, and k-d trees,” in Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics, EGGH-HPG’12, Aire-la-Ville, Switzerland, Switzerland, pp. 33–37, Eurographics Association, 2012.