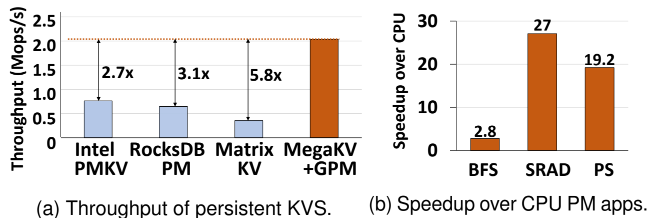
GPM: Leveraging Persistent Memory from a GPU

Aditya K Kamath, Shweta Pandey, Arkaprava Basu



(a) Throughput of persistent KVS. (b) Speedup over CPU PM apps.

Figure 1: Benefits of GPU with PM.

Non-volatile memory (NVM) technologies promise to blur the long-held distinction between memory and storage by enabling durability at latencies comparable to DRAM [7]. NVM enables fine-grain byte addressable persistence accessible via loads and stores. We define Persistent Memory (PM) as NVM accessed via load/stores at byte granularities [5]. Thanks to decade-long research into CPU's software and hardware stack for PM (*e.g.,* [4, 9]), and with the recent commercialization of Intel Optane NVM, PM's promise of revolutionizing computing seems closer to the reality.

Motivation: Unfortunately, while Graphics Processing Units (GPUs) are key computing platform today, they are deprived of direct access to PM. We find several important applications that could have benefited from *both* GPUs and PM. Consider a persistent key-value store (KVS) that leverages PM for persistence and recoverability from crashes. Today, they are limited to CPUs [8, 2, 10]. Independently, researchers found GPUs can significantly increase KVS's throughput [11]. Fine-grain persistence to PM from GPUs could enable *both together*. Figure 1a provides a glimpse of its potential performance benefits. The first three bars show throughputs of batched SET operations (8B keys and values) on PM-optimized commercial (Intel's pmemKV [2]) and RocksDB-pmem [8]) and academic KVSs (MatrixKV [10]) on many core CPU. The fourth bar reports throughput with a GPU-enabled KVS, MegaKV [11], ported onto our system, called GPM, to use PM. GPM enables fine-grain persistence for GPU kernels. The throughput improves by 2.7×-5.8× over today's multi-threaded CPU alternatives for persistent KVS while retaining same recoverablity guarantee.

Many other applications, *e.g.,* breadth-first search (BFS), image processing (SRAD), prefix sum (PS) could benefit from both fine-grain persistence and GPU's parallelism. They speed up by 2.8-27× over multi-threaded CPU versions of the same applications leveraging PM for recoverability (Figure 1b).

Today, if an application wishes to leverage PM's persistence, it would typically perform both the computation and ensure persistence of results from the CPU. Alternatively, one can perform computations on the GPU, but then transfer results to the CPU's memory and rely on the CPU to guarantee persistence for recoverability. We call this alternative that at least uses the GPU for computation, <u>C</u>PU-<u>A</u>ssisted <u>P</u>ersistence (CAP).

Unfortunately, CAP fails to bring the full benefits of byte-grain low-latency persistence to GPU kernels. GPU's inability to efficiently *guarantee* persistence of PM-resident data structures at byte granularities impedes programmers from creating recoverable GPU kernels that can correctly restart after a crash during GPU execution. Second, GPUs accelerate computation through massive parallelism. Many benefits of parallelism are lost by relying on the CPU to write and persist results of GPU computations. Third, many times only a fraction of data is updated during computation. However, which data would be updated is not known apriori. Since the GPU cannot directly persist results while computing, extraneous data could be transferred to and persisted by the CPU.

Towards this, we set three goals. ① Design a system with a GPU having direct access to PM *without* needing new hardware. It should address all of CAP's shortcomings. ② Explore types of applications that can benefit from both GPU and fine-grained persistence. ③ Finally, create a software ecosystem (*e.g.,* library, runtime) that could help programmers easily and efficiently program PM from a GPU kernel.

Key insights: Towards the first goal, we propose <u>G</u>PU with <u>PM</u> or GPM where GPU kernels can directly manipulate PM-resident data and guarantee persistence wherever desired within the kernel, *without* needing the CPU or the OS. Currently, there exists no hardware with NVM onboard the GPU. The NVM (Intel Optane [1]) is placed alongside the DRAM, as in a typical Intel Xeon-based server, and can be accessed by a GPU over the PCIe interconnect. GPM leverages NVIDIA's Uniform Virtual Address (UVA) to map desired portions of NVM onto the virtual address space of a GPU kernel. Kernels could then manipulate PM-resident data structures at a byte granularity using GPU loads/stores.

To compose programs that are recoverable in the presence of crashes or power failures, a programmer must be able to guarantee persistence of data to PM (i.e., *persist*) wherever programs' semantics demand. A persist operation typically requires flushing the contents of volatile cache lines to PM and waiting for flushes to finish. Unfortunately, unlike CPUs, today's GPUs are not designed for PM and do not have instructions to flush cache lines [6]. However, we noticed that a fence operation with *system* scope (__threadfence_system() in CUDA) ensures all writes to host (system) memory before the fence are made visible on the host (CPU) when the fence completes. While the original purpose of that fence was to synchronize computations between the CPU and GPU, it provides semantics needed for persist operations. This is because in GPM, the NVM is a part of the host memory, alongside DRAM.

However, a system-scoped fence alone is insufficient to create persist operations in GPUs. When Intel's Xeon processor's Data Direct IO (DDIO) feature is enabled (default), the writes

to system memory by devices, *e.g.,* NIC, GPU, are cached in the CPU's volatile last level caches (LLCs) [3]. Consequently, the fence completes as soon as writes reach LLC, and not PM. Therefore, GPM *selectively* disables DDIO when persistence is desired. Consequently, the system-scoped fence completes only when persistence of the writes to PM is guaranteed.

In short, we use UVA to map PM to the GPU's address space, and system-scoped fences with selective disabling of DDIO to create GPM on Xeon servers with Optane NVM and NVIDIA GPUs. GPM mitigates CAP's shortcomings. GPM's ability to guarantee persistence from the GPU enables programmers to write recoverable kernels. Applications benefit from both the GPU's parallelism and fine-grained persistence without CPU or OS involvement. Finally, on GPM, kernels can persist only the necessary parts of (intermediate) results while computing.

Towards the second goal of exploring applications that benefit from both GPUs and fine-grained persistence, we find three categories of applications. Transactions in GPU-accelerated persistent KVS and relational databases benefit from fine-grained logging to PM. Next, long-running applications that iteratively invoke GPU kernels, *e.g.,* DNN training, benefit from faster checkpointing to PM for fault tolerance. Finally, GPM enables a new class of GPU kernels that embed the logic to perform in-place byte-grained updates to PM-resident data structures while ensuring they remain recoverable (consistent) after a crash. These kernels can then resume, rather than restart computation upon recovery from a crash. GPU-accelerated BFS on PM-resident graphs is an example. In the process, we created a workload suite, named GPMbench, with 9 GPU workloads that leverage PM's persistence.

Our third contribution is a CUDA library (libGPM) that implements GPU-optimized write-ahead logging to implement transactions, checkpointing to PM, and ordering persist operations. A key innovation in libGPM is Hierarchical Coalesced Logging (HCL) with two *GPU-specific* optimizations.

To scale logging to GPU, where each of its tens of thousands of threads may attempt to insert entries into the log concurrently, HCL mimics the GPU's execution hierarchy in the structure of its log. On a NVIDIA GPU, a group of 32 threads, called a warp, typically execute in lockstep. Tens of warps are organized into a threadblock, and many threadblocks constitute a grid. In HCL, each GPU thread computes a unique offset in the log to insert its entry based on its thread ID, warp ID, and threadblock ID. Since every thread, warp, threadbock has a fixed offset(s) in the log for inserting its entry, the logging can proceed in a data-parallel manner without requiring locks.

Another key innovation in HCL is how it leverages the GPU's hardware coalescer. HCL ensures that log entries written by threads of a warp in SIMD fashion are packed into a single GPU cache line. Thus, concurrent writes to the log by a warp are merged into a single write by the hardware coalescer. Since a typical GPU cache block is 128-byte long and there are 32 threads in a warp, HCL ensures each thread writes its log in 4-bytes chunks. HCL stripes larger log entries into 4-byte chunks

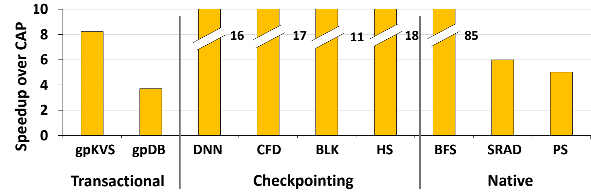


Figure 2: Speedup of GPM normalized to CAP.

over multiple cache lines. HCL speeds up logging by $\sim 3.6\times$ over traditional distributed logging on average.

Besides logging, libGPM provides primitives for persistence by enabling means for (de-)allocating GPU-accessible memory on PM, flushing and draining data to PM and also supports checkpoint/restoration from the GPU.

Figure 2 shows GPM provides multi-fold speedup over CAP. gpKVS speeds up by 7-8$\times$ over CAP, due to GPM's ability to persist only the updated/new entries in pKVS, as GPM provides in-kernel selective persistence. Checkpointing workloads, like DNN, benefit from GPU's massive parallelism when checkpointing to PM. Native workloads, like BFS speedup from GPM's potential to directly write and persist data.

Citation of original paper: Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. *GPM: Leveraging Persistent Memory from a GPU.* In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022).
DOI: https://doi.org/10.1145/3503222.3507758

References

[1] Intel. Intel optane persistent memory. https://www.intel.in/content/www/in/en/architecture-and-technology/optane-dc-persistent-memory.html, 2021.

[2] Intel. Intel pmemkv, 2021. https://github.com/pmem/pmemkv.

[3] A. Kalia, D. Andersen, and M. Kaminsky. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 105–119, New York, NY, USA, 2020. ACM.

[4] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, New York, NY, USA, 2019. ACM.

[5] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with whisper. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 135–148, New York, NY, USA, 2017. ACM.

[6] NVIDIA. Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2019.

[7] I. B. Peng, M. B. Gokhale, and E. W. Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 304–315, New York, NY, USA, 2019. ACM.

[8] RocksDB. Rocksdb. 2021. https://rocksdb.org/.

[9] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, New York, NY, USA, 2011. ACM.

[10] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He. Matrixkv: Reducing write stalls and write amplification in lsm-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31. USENIX Association, July 2020.

[11] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.*, 8(11):1226–1237, July 2015.