# Exploring Memory Persistency Models for GPUs

Zhen Lin*, Mohammad Alshboul*, Yan Solihin†, Huiyang Zhou*
*North Carolina State University
{zlin4, maalshbo, hzhou}@ncsu.edu
†University of Central Florida
yan.solihin@ucf.edu

***Abstract*—Given its high integration density, high speed, byte addressability, and low standby power, non-volatile or persistent memory is expected to supplement/replace DRAM as main memory. Through persistency programming models (which define durability ordering of stores) and durable transaction constructs, the programmer can provide recoverable data structure (RDS) which allows programs to recover to a consistent state after a failure. While persistency models have been well studied for CPUs, they have been neglected for graphics processing units (GPUs). Considering the importance of GPUs as a dominant accelerator for high performance computing, we investigate persistency models for GPUs.**

**GPU applications exhibit substantial differences with CPUs applications, hence in this paper we adapt, re-architect, and optimize CPU persistency models for GPUs. We design a pragma-based compiler scheme to express persistency models for GPUs. We identify that the thread hierarchy in GPUs offers intuitive scopes to form epochs and durable transactions. We find that undo logging produces significant performance overheads. We propose to use idempotency analysis to reduce both logging frequency and the size of logs. Through both real-system and simulation evaluations, we show low overheads of our proposed architecture support.**

## I. Introduction

Non-volatile memory (NVM) or Persistent Memory (PM) is here [1] and is expected to supplement/replace DRAM as main memory due to its high integration density, comparably high read speed, byte addressability, and low leakage power. An example PM is Intel Optane DC Persistent Memory [2], which is a DDR4-connected device supporting 3TB/socket main memory. The non-volatility makes it possible to host data persistently in main memory, blurring the boundary between main memory and storage, thereby challenging the classical computer system design.

Persistent data storage in main memory provides an opportunity to achieve *recoverable data structures* (RDS), which allows programs to recover from crashes just by using data in main memory instead of a checkpoint. Recent research [3], [4] showed that by relying on RDS instead of checkpoints, highly significant performance and write endurance improvements can be obtained. Achieving RDS requires *persistency models* along with instruction support, and a crash recovery technique such as logging. Various memory persistency models have been proposed for CPUs, defining the order in which stores become durable in main memory, often in relation to ordering defined in memory consistency models regarding when stores become visible to other threads in a parallel program [5]. In addition to store durability ordering, ensuring a consistent data at any given point in time is required, typically supported through durable transactions and their associated logging mechanisms [6]–[8].

While persistency models in CPUs have been well explored, they have been neglected in GPUs. We envision GPU will make use of persistent memory in the near future for the following key reasons besides the increased memory capacity over volatile DRAM. First, in current systems, persistent data is kept in files, and must be read to build data structures at process start, written to files at process termination. The conversion between persistent and temporary is expensive and unnecessary with NVM. One such use case is in-memory databases, especially the GPU accelerated ones including Mega-KV [9], GPU B-Tree [10], Kinetica [11], etc. Second, long-running GPU applications, including training deep neural networks, computing proof of work in blockchain applications, scientific computation using iterative approaches, etc., would benefit from fault tolerance with RDS. Having recoverable persistent data in NVM allows the processor to recover from soft errors. This relegates system checkpointing for more serious faults, hence the checkpointing frequency can be reduced [3]. Third, in fusion-like architecture, CPU and GPU share PM. Therefore, we argue that GPU needs to support memory persistency models.

In our study, we assume discrete GPU systems shown in Figure 1, since discrete GPUs have high memory bandwidth and are most commonly used in HPC (High-Performance Computing) systems although the same support can be adopted for fusion-like architectures. Recent GPUs support both unified and non-unified memory [12]. With non-unified memory, the programmer needs to explicitly copy the data between host-side system memory and device memory while with unified memory, the memory pages can be migrated on-demand, reducing the programmer's complexity. In this work, we consider both unified and non-unified memory models and assume that the persistency of the host-side system memory is properly handled with existing approaches [5]–[7], [13].

CPU memory persistency needs to be re-architected for GPUs, because of the following key differences: **Workloads**: CPU benchmarks utilizing PM, such as Whisper [14], mainly focus on database applications. In comparison, long-running GPU tasks also include scientific computations, deep neural network training, large graph processing, and blockchain mining. They exhibit different characteristics and execution behavior. **Bandwidth vs. latency**: memory-intensive kernels in

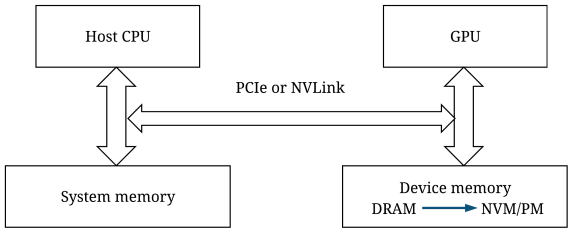


Fig. 1: The system architecture with a discrete GPU, for which NVM/PM replaces DRAM as device memory.

GPUs are typically bandwidth- (instead of latency-) sensitive. Latency-oriented optimizations for CPUs such as durable write-pending queues (WPQs) at the memory controller have limited impact on GPUs. Furthermore, creating/updating logs introduce additional write traffic/bandwidth, which affect GPUs much more than CPUs. **Multiple Memory Partitions/Controllers**: a GPU is equipped with multiple memory partitions and memory controllers for high bandwidth. The pcommit instruction, which makes pending writes durable, needs to be broadcasted to all MCs to flush all their WPQs. **Scratchpad memory**: a typical server/desktop GPU memory hierarchy has scratchpad memory (aka shared memory), which needs to be considered for GPU memory persistency.

In this paper, we *adapt, re-architect, and optimize CPU persistency models for GPUs.* The paper makes the following contributions:

First, to provide both simplicity and flexibility to use the persistency models, we propose pragmas for programmers to specify their choice of persistency model and code region, allowing the compiler to automatically generate GPU persistency code.

Second, we propose GPU-friendly implementation of strict persistency and relaxed (epoch) persistency. Whereas CPUs rely only on clwb (cache-line-write-back)/clflushopt (cache-line-flush-optimized) [6] instructions, we consider and compare GPU alternatives: store write-through (store.wt) and l2wb to flush all dirty blocks in the L2 cache. We found that a mixture of them is warranted: store.wt is profitable for writing undo logs without write allocation (as temporal locality is absent in failure-free execution) and for implementing strict persistency. We also found that l2wb is profitable in cases where it is difficult/expensive to re-generate addresses required for clwb. We also use the membar instruction as a persist barrier between epochs. Since GPU memory controller WPQs are not in the non-volatile domain, we evaluate the effect of using pcommit instruction to flush blocks in WPQs to PM.

Third, for the epoch persistency model, we make an important observation that the thread hierarchy in the GPU programming model provides intuitive choices for epoch granularity. Based on the characteristics of a kernel, we propose three epoch granularities: kernel level, CTA (cooperative thread array) level, and loop level. We show that epoch selection is crucial to performance, and different workloads require different epoch scopes for optimal tradeoff between performance and recoverability from crashes.

Fourth, due to GPU application performance being bandwidth-sensitive and logging adds to bandwidth pressure, we need to reduce the reliance on the logging, both in frequency and the size of logs. We propose to leverage idempotency analysis [15], [16] at different GPU thread hierarchy granularities and found that this analysis not only helps remove the need to create undo logging, but also reduces the size of logs when the epoch is not idempotent. We also show that many GPU kernels can benefit from this optimization.

Finally, we show evaluation results on two platforms, real GPUs (NVIDIA GTX1080) and simulated GPUs, that demonstrate data persistency and recoverability on GPUs can be achieved at low performance overheads.

## II. Background and Related Work

### A. Memory Persistency Models on CPUs

Byte-addressable NVM, aka persistent memory (PM), provides opportunities for high-performance in-memory recoverable data structures (RDS). However, due to write-back caches in CPUs, the durability order of writes to main memory can be different from the program order of stores. This does not present a problem for volatile memory such as DRAM. But it is not the case for PM. For example, assuming that the data fields '$p \rightarrow data$' and '$p \rightarrow state$' reside in different cache lines and the update to '$p \rightarrow data$' precedes the update to '$p \rightarrow state$'. Due to the write-back last-level cache (LLC), '$p \rightarrow state$' in memory may be updated while '$p \rightarrow data$' has not. In this case, if a fault, e.g., a power failure, happens, the persistent memory state becomes incorrect after power is restored. To deal with this issue and support correct implementations of RDS, memory persistency models [5], [13] formally specify the order of writes to PM. In particular, strict persistency means that the persistent memory order is identical to the volatile memory order, which is governed by the memory consistency model. In comparison, relaxed persistency reduces the order constraints and two such persistency models, epoch and strand persistency, have been proposed [5], [13]. Under the epoch persistency model, the execution of a thread is separated into epochs separated by persist barriers.The durability order of stores to different addresses is only enforced between epochs but not within an epoch. The order of conflicting accesses, i.e., accesses to the same address, on the other hand, is maintained, which is referred to as strong persistent atomicity. The strand persistency model relaxes the order constraints even further but requires programmers to express dependencies so as to remove unwanted memory ordering. Therefore, we do not consider strand persistency in this work.

The memory persistency models mentioned above specify the durability order of stores but ordering alone does not provide recoverability. For example, assume that with either strict or epoch persistency, '$p \rightarrow data$' and '$p \rightarrow state$' in PM are updated in the program order. But it is still possible that a fault happens after '$p \rightarrow data$' is updated but before '$p \rightarrow state$' being updated. In this scenario, the memory

state in PM is still not correct for data recovery. What is missing here is transaction-like semantics, which requires that a group of stores are made durable together or none at all. To achieve such operations for PM, durable transactions have been proposed [6]–[8]. Through either redo or undo logging, they enable data to be recovered if a failure occurs during a transaction. The overhead of logging can be significant and there are some recent works to either reduce the size of the logs using recomputation [3] or to improve the performance using hardware logging [17], [18].

### B. GPU Architecture and Programming Model

Modern GPUs employ the single-instruction multiple-thread (SIMT) architecture. A GPU consists of multiple streaming multiprocessors (SMs). On each SM, there is one or more warp schedulers to feed instructions to the ALU or memory pipelines. The GPU memory hierarchy includes register files, L1 D-caches, shared memory, constant caches, texture caches, and an unified L2 cache. The L2 cache contains multiple partitions and there is a memory controller for every one or two partitions so as to achieve high memory access bandwidth. The L1 D-caches typically do not use the write back (WB) policy (e.g., a write evict policy instead) and currently there is no coherence support among the L1 D-caches residing in multiple SMs. The L2 cache uses the WB policy while the GPU ISA may support the store instructions with an option to write through the L2 cache.

The SIMT programming model is a single program multiple data (SPMD) model and massive threads are organized in a hierarchy. A kernel is launched with a grid of collaborative thread arrays (CTAs). A CTA in turn contains many threads, which can communicate and synchronize with each other through shared memory. Each thread/CTA determines its workload using its thread/CTA id. Each SM can host one or more CTAs depending on their resource usage. The threads in a CTA form warps, each of which is executed in the SIMD manner when there is no control divergence. With divergence, sub-warps may be formed to support multi-path execution while threads in each sub-warp are executed in the SIMD manner.

Memory consistency models have not been formally defined on GPUs [19]. Until recently, Heterogeneous System Architecture (HSA) Foundation [20] and OpenCL [21] start to adopt the C11's datarace-free-0 (DRF-0) model, which guarantees sequential consistency (SC) for data-race-free code, but is undefined for the cases with data-races. A few recent works show that the overhead of SC or TSO (Total Store Order) can be significantly reduced for GPUs [19], [22], [23].

There are a few prior works on memory persistency for GPUs. Gope et al. [24] performed a case study of B+-tree on GPUs with persistency memory support. They discussed the impact of persistency barrier scopes on the GPU performance. HeteroCheckpoint [25] leverages NVM as the storage for checkpointing in the CPU-GPU heterogeneous systems.

```
1 lbm_kernel{
2 #pragma gpu_pm strict clwb
3   ... = input[loc1(tid)]; ... = input[loc2(tid)];
4   ... // compute...
5   output[loc1(tid)] = ...; output[loc2(tid)] = ...;}
```

(a)

```
1 lbm_strict{
2   ... = input[loc1(tid)]; ... = input[loc2(tid)];
3   ... // compute...
4   output[loc1(tid)] = ...;
5   clwb(&output[loc1(tid)]); {sfence; pcommit;} sfence;
6   output[loc2(tid)] = ...;
7   clwb(&output[loc2(tid)]); {sfence; pcommit;} sfence;}
```

(b)

Fig. 2: An example for strict persistency. (a) Original code with pragma, (b) the code with the compiler added instructions for strict persistency.

## III. GPU Memory Persistency

We explore how to adapt and re-architect two memory persistency models for GPUs: strict and epoch persistency. We propose a compiler approach to facilitating programmers to utilize the persistency models. In the original source code, the programmer simply inserts pragmas to annotate the desired persistency model along with the options of implementation. Different pragmas are used for different persistent models:

```
#pragma gpu_pm strict options
#pragma gpu_pm epoch epoch_scope options
```

Our compiler produces code for execution on real GPUs using existing GPU instructions, as well simulated GPUs with new instructions that we add. The compiler approach supports three scopes of an epoch, including kernel-level, CTA-level and loop-level epochs, so as to take advantage of the SIMT programming model. Epochs with different scopes from these three can also be realized with the same architectural support. For example, an user may explicitly specify a region of kernel code containing several loops or a region of host code containing multiple kernel invocations as an epoch using persistency barriers.

### A. Strict Persistency

To support strict persistency for GPUs, we can persist the data in the program order. To do so, for each store, we add a clwb (or clflush/clfushopt) instruction to write the dirty cache line to the memory controller, a pcommit instruction to write the data in the WPQ to persistent memory if the WPQ in the memory controller is not durable, and sfence instructions to ensure the order of memory operations. One such example based on the benchmark lbm (see our methodology in Section V) is shown in Figure 2. Figure 2(a) is the original code with a gpu_pm pragma to direct the compiler to generate code for strict persistency. Figure 2(b) is the generated code. The instruction pair {sfence, pcommit} is not needed if the WPQs are durable.

Besides the clwb instruction, strict persistency can also be implemented using store.wt instructions. A user can choose this option by specifying 'wt' in the pragma, in which case the

compiler replaces all the store instructions with the store.wt instructions.

In current GPUs, membar/fence instructions enforce memory ordering in NVIDIA PTX ISA [26]. The PTX manual states that "*The membar instruction guarantees that prior memory accesses requested by this thread are performed at the specified level, before later memory operations requested by this thread following the membar instruction. The level qualifier specifies the set of threads that may observe the ordering effect of this operation.*" For the evaluation on real GPUs, we choose membar at the 'gl' level, i.e., the GPU level, equivalent to the fence.gpu instruction. For the evaluation on simulated GPUs, we implement the instruction with the semantics that all its prior memory operations from the same warp receive their acknowledgements.

The clwb instruction does not exist in current GPUs, hence we use store.wt in its place for evaluation on real GPUs, and implement it on the simulated GPUs. There is no existing support for the pcommit instruction either in current GPUs. Therefore, we introduce this instruction. As there are multiple memory partitions and memory controllers, stores from the same warp with different addresses may be mapped to different memory controllers. As a result, in order to correctly implement the pcommit instruction, we need to drain the WPQs in all the memory controllers. We model such semantics in our simulator for the case when the WPQs are not durable.

### *B. Epoch Persistency*

As discussed in Section II-B, the GPU/SIMT programming model requires programmers to explicitly specify the thread hierarchy. For HPC workloads, it is a common practice that each thread is used to compute one or few elements in the output domain and the threads in one CTA compute a tile/subblock of output elements. For example, in matrix multiplication, a thread is used to compute one or few elements in the product matrix. A CTA computes a tile of elements in the product matrix. For complex applications, the overall computation can be decoupled into multiple kernels.

Epoch persistency requires choosing the scope of epochs. Since an epoch scope corresponds to the code region that needs to be re-executed on a failure, the scope of an epoch must correspond to a code region that the programmer finds easy to analyze and to reason about failure recovery. GPU thread hierarchy provides intuitive epoch granularities for this purpose: an entire kernel as an epoch, a CTA as an epoch, or a loop iteration as an epoch.

To help users determine the proper epoch granularity, we propose the following scheme. First, based on its runtime characteristics, we classify a GPU kernel into one of the three categories: (1) short-running kernels; (2) long-running kernels with short-running CTAs; and (3) long-running kernels with long-running CTAs. Note that determining long-running vs. short-running needs to take the failure rate/mean-time-to-failure (MTTF) and recovery cost into consideration to ensure forward progress. Then we apply three different epoch persistency models accordingly: kernel-level epoch persistency for short-running kernels, CTA-level epoch persistency for long-running kernels with short-running CTAs, and loop-level epoch persistency for long-running kernels with long-running CTAs.

```
1 ... //setup thread hierarchy, i.e., grid & block.
2 ... // prepare input array
3 #pragma gpu_pm epoch kernel scope=1
4 histo_kernel_2<<<grid, block>>>(input, output);
5 ... // consume output array
```

(a)

```
1 histo_kernel_2<<<grid, block>>>(input, output);
2 cudaDeviceSynchronize();
3 cudaL2WB();
4 cudaDeviceSynchronize(); //wait for l2wb to finish
5 ... // consume output array
```

(b)

Fig. 3: An example for kernel-level epoch persistency.(a) Original code with pragma, (b) the code with compiler added APIs for kernel-level epoch persistency.

**a. Kernel-Level Epoch Persistency**

For kernel-level epoch persistency, each kernel invocation is an epoch. At the end of the kernel execution, we persist all updated data and add a persist barrier. In current GPUs, the dirty data in the L2 cache are not written back to the device memory as the memory controller monitors the incoming data requests (e.g., from the host CPU) and feeds the most recent data from the L2 cache directly if needed. To support kernel-level epoch, we propose to add a new l2wb instruction to write back *all* dirty lines in the L2 into device PM. This instruction can be used in either the kernel code or the host code through a driver API. Figure 3 shows such an example based on the histogram benchmark. The original host code is shown in Figure 3(a). The pragma before the kernel launch is for the compiler to generate the host code with the added APIs, including the synchronization and l2wb. The scope option is used to determine how many kernel invocations to be included in an epoch. When multiple kernels are included in one epoch, only one synchronization is inserted at the end of the last kernel invocation. As a result, this option reduces the synchronization overhead among kernel invocations. In the example shown in Figure 3(b), the epoch only contains one kernel invocation and the device synchronization function, cudaDeviceSynchronize(), is used as the persist barrier between kernel invocations.

**b. CTA-Level Epoch Persistency**

In the CTA-level epoch persistency, each CTA is an epoch. Durability ordering is not enforced for stores within a CTA and we just need to persist all the updated data at the end of each CTA. Many GPU applications, especially scientific computing workloads including BLAS, stencil, FFT, etc., share a popular programming pattern that the inputs are accessed at the beginning of a kernel function and the outputs are generated at the end, and many threads in a CTA collaboratively compute a set of output data. Such a programming pattern fits nicely with the CTA-level epoch persistency model. One such an example based on the benchmark lbm is shown in Figure 4.

**Algorithm 1** Code generation for CTA-level epoch persistency with clwb

**Input:** Kernel source code
1: **function** EP-CTA-CLWB(Kernel)
2: Create a post-dominant block in the end of kernel
3: Move code generator to the created block
4: Get all global memory stores in the kernel
5: **for** each store **do**
6: Detect use-define chain of the store address
7: Replicate all statements in the chain if the address is no longer available
8: Insert $clwb$ with the address
9: **end for**
10: Insert $sfence$
11: **if** WPQ is volatile **then**
12: Insert $pcommit$ and $sfence$
13: **end if**
14: **end function**

```
1 lbm_kernel{
2 #pragma gpu_pm epoch cta clwb
3  ... = input[loc1(tid)]; ... = input[loc2(tid)];
4  ... // compute...
5  output[loc1(tid)] = ...; output[loc2(tid)] = ...;}
```

(a)

```
1 lbm_CTA{
2  ... = input[loc1(tid)]; ... = input[loc2(tid)];
3  ... // compute
4  output[loc1(tid)] = ...; output[loc2(tid)] = ...;
5  clwb(&output[loc1(tid)]); clwb(&output[loc2(tid)]);
6  {sfence; pcommit;} sfence;}
```

(b)

Fig. 4: An example for CTA-level epoch persistency.(a) Original code with pragma, (b) the code with the compiler added instructions for CTA-level epoch persistency using the clwb option.

Figure 4(a) shows the original code with a pragma to indicate the compiler to generate the CTA-level epoch persistency code using the clwb option. And the code with the compiler-inserted instructions for persistency is shown in Figure 4(b). Compared to the code using the strict persistency model in Figure 2(b), the stores and the cache-line write backs are performed in an overlapped manner instead of being sequential. The last sfence instruction also serves as a persist barrier.

The compiler algorithm for such code transformation is shown Algorithm 1. It first creates a basic block that post-dominates all statements and this basic block is used for code generation. Then it determines all global memory stores in the kernel. For each store, the compiler detects the use-define chain of the store address. In the created basic block, the whole chain is replicated to re-calculate the address if necessary. And the clwb instruction is inserted with the address to write back the cache line. After all clwb instructions have been generated, the sfence instruction is inserted to wait for the clwb instructions to be posted.

Besides the clwb option, we can also use wt and/or l2wb to implement CTA-level epoch persistency. In some kernel functions, their outputs are distributed in the code and it may be hard or too costly to re-generate the addresses at the end of the kernel, which are to be used by the clwb instructions. In such cases, we can either replace the store instructions with the stores using the WT operator (i.e., store.wt) or resort to the l2wb instruction followed by a sfence instruction to persist all the dirty cache lines in the L2 cache, even some of them are not updated by this particular CTA.

Note that we argue that there is no need for a CTA-level synchronization, i.e., syncthreads(), after the last sfence instruction. The reason is that every warp in the CTA will execute the sfence instruction as its last instruction, which guarantees that no further memory instructions will be issued from this CTA.

We choose not to support the scope option for CTA-level persistency. The reason is that in order to include multiple CTAs into one epoch, these CTAs need to be synchronized through an inter-CTA synchronization mechanism, which may lead to performance degradation due to the lack of hardware support for CTA ordering and global synchronization across CTAs.

With CTA-level epochs, there is one distinction between the epoch persistency model on CPU and GPU. As specified in the GPU programming model, CTAs are supposed to be executed in parallel without ordering constraints. As a result, we can view that with CTA-level epochs, there are multiple concurrent epochs running on a GPU. In contrast, in a sequential program, epochs are executed sequentially and there are order constraints between epochs.

**c. Loop-Level Epoch Persistency**

In the loop-level epoch persistency model, the scope of an epoch can be reduced to an iteration of a long-running loop in a kernel. Here, we use the benchmark tpacf as a case study. The simplified kernel code is shown in Figure 5(a) and contains a long-running nested loop. A pragma is inserted immediately before the outer loop to indicate the scope of the epoch. Also, because the code uses shared memory for the intermediate results in every loop iteration. The compiler creates a shadow copy of the shared memory array for each CTA and persists the shadow copy for shared memory data recovery. As the shared memory array is updated in every iteration in the innermost loop, it is very costly to reconstruct such array indices at the end of the outermost loop. Therefore, we can leverage the l2wb instruction to write back all the L2 dirty lines, which is then followed by a sfence instruction as the persist barrier to ensure that no subsequent memory operations can be issued from this warp until the write backs are finished. Note that due to the costly overhead of the l2wb instruction, we only use one thread in a CTA to execute this instruction. To reduce the overhead of l2wb and sfence instructions, we allow multiple loop iterations to be included in one epoch. In this example, the scope is set to 4, which means the l2wb and sfence are inserted every 4 loop iterations. The resulting code is shown in Figure 5(b). For reference, we also include the (commented out) code for strict persistency in lines 13 & 14.

With loop-level epoch persistency, the order constraints are between the epochs (i.e., loop iterations) in a single warp while epochs in different warps and CTAs can be executed in parallel.

```
tpacf_kernel(g_hists, data) {
 __shared__ int s_hists[N_BINS][N_THD];
 ... // Initialization
 // Long nested loop
#pragma gpu_pm epoch loop l2wb scope=4
 for (i = 0; i < N_ELEMS; i += CTA_SIZE) {
  for (k = 0; k < CTA_SIZE; k++) {
    ...
    bin_idx = ...
    s_hists[bin_idx][tid] += 1;
  } ... }
```

(a)

```
__device__ int shadow[N_CTA][N_BINS][N_THD]
tpacf_kernel_loop(g_hists, data) {
 __shared__ int s_hists[N_BINS][N_THD];
 ... // Initialization
 // Long nested loop
 for (i = 0; i < N_ELEMS; i += CTA_SIZE) {
  for (k = 0; k < CTA_SIZE; k++) {
    ...
    bin_idx = ...
    s_hists[bin_idx][tid] += 1;
    shadow[cta_id][bin_idx][tid] = s_hist[bin_idx][tid];
    //The next two lines are for strict persistency
    //clwb(&shadow[cta_id][bin_idx][tid]);
    //{sfence; pcommit;} sfence;
  } //end of the inner loop
  ...
  __syncthreads();
  if (i/CTA_SIZE % 4 == 3) {
   if(tid == 0) l2wb; // write back L2 dirty cache lines
   {sfence; pcommit;} sfence;}
 } //end of the outer loop
...}
```

(b)

Fig. 5: An example for loop-level epoch persistency. The scratchpad memory is made persistent through a shadow copy in global memory. (a) Original code with pragma, (b) the code with the compiler added instructions for loop-level epoch persistency using the l2wb option.

**d. Summary**

We explore both strict and epoch persistency models for GPUs. A summary of their architectural support and their targeted GPU kernels is presented in Table I. Note that different models treat shared memory data (i.e., the data in the scratchpad memory) differently. Among the epoch persistency models, only the ones with the scope less than a CTA, e.g., the loop-level, need to construct a shadow copy in the global memory and persist the data at the end of an epoch. For the CTA- and kernel-level, the shared memory data are no longer live at the end of an epoch, therefore there is no need for the data to be persisted.

## IV. Durable Transactions for GPUs

The memory persistency models (Section III) specify the durability ordering of stores, which is necessary but insufficient for guaranteeing a fail-safe state as discussed in Section II-A. Durable transactions are often required to persist a group of stores together or none at all. In this section, we discuss turning an epoch into a durable transaction with undo logging, or in some cases omit logging altogether by exploiting the idempotency property of an epoch.

Software-based undo logging contains the following steps. First, before a transaction starts, an undo log is created by making a copy of the data to be updated and this undo log is persisted. Second, we set and persist a flag to indicate that the transaction is running. Third, during the transaction, data are updated and at the end of the transaction, the updated data are persisted. Fourth, we mark the transaction complete and release the undo log.

With undo logging, the recovery code checks the flags to see whether there is a transaction is interrupted. If so, it uses the undo log to restore the data.

```
enum flag {initial, inTx, complete};
... //setup thread hierarchy, i.e., grid & block.
... // prepare input array
cudaMemcpy(undo_log, output,
       size, cudaMemcpyDeviceToDevice); // create undo log
flag = inTx;
clwb(&flag); sfence; //persist the flag in host memory
cudaDeviceSynchronize(); // wait for cudaMemcpy to finish
histo_kernel_2<<<grid, block>>>(input, output);
cudaDeviceSynchronize(); // wait for the kernel to finish

cudaL2WB();
cudaDeviceSynchronize(); //wait for l2wb to finish
flag = complete;
clwb(&flag); sfence; //persist the flag in host memory
... // consume output array
```

Fig. 6: A code example for kernel-level durable transaction.

```
enum FLAG {initial, inTx, complete}; FLAG flag[NUM_CTA];
lbm_CTA_log{
 ... = input[loc1(tid)]; ... = input[loc2(tid)];
 //log[loc1[tid]] = output[loc1[tid]]; clwb(&log[loc1[tid]]);
 st.wt &log[loc1(tid)], output[loc1(tid)];
 st.wt &log[loc2(tid)], output[loc2(tid)];
 // compute...
 {sfence; pcommit;} sfence;
 __syncthreads(); // logs are durable
 if (tid == 0) {
  st.wt &flag[ctaid], inTx; // inside tx
  {sfence; pcommit;} sfence; }
 output[loc1(tid)] = ...; output[loc2(tid)] = ...;
 clwb(&output[loc1(tid)]); clwb(&output[loc2(tid)]);
 {sfence; pcommit;} sfence;
 __syncthreads(); // CTA is done
 if (tid == 0) {
  st.wt &flag[ctaid], complete; // committed
  {sfence; pcommit;} sfence;}}
```

(a)

```
lbm_CTA_idem{
 ... = input[loc1(tid)]; ... = input[loc2(tid)];
 st.wt &flag[ctaid], inTx; // inside tx
 {sfence; pcommit;} sfence;
 // compute...
 output[loc1(tid)] = ...; output[loc2(tid)] = ...;
 clwb(&output[loc1(tid)]); clwb(&output[loc2(tid)]);
 {sfence; pcommit;} sfence;
 __syncthreads(); // CTA is done
 if (tid == 0) {
  st.wt &flag[ctaid], complete; // committed
  {sfence; pcommit;} sfence; }}
```

(b)

Fig. 7: An example for CTA-level durable transaction. (a) The code with undo logging, (b) the optimized code using idempotency analysis.

### A. Kernel-Level Durable Transactions

As GPUs are used as accelerators, their input data are prepared at the host side and copied to the device. Then, the kernel is invoked by the host. Therefore, we propose to implement kernel-level durable transactions in the host code. We also assume that the host side memory is persistent. The

TABLE I: A summary of memory persistency models, the architectural support, and the targeted kernels.

| Persistency Models | Strict Persistency | Relaxed Persistency | | |
|---|---|---|---|---|
| | | Kernel-Level Epoch | CTA-Level Epoch | Loop-Level Epoch |
| Architectural Support | clwb/clflush(opt)/store.wt; sfence; pcommit | l2wb; DeviceSynchronization | clwb/clflush(opt); store.wt; sfence; pcommit; l2wb | clwb/clflush(opt); store.wt; sfence; pcommit; l2wb |
| Suitable Kernel | All | Short-running kernels | Long-running kernels with short-running CTAs | Long-running kernels with long-running CTAs |

resulting code based on the histogram benchmark is shown in Figure 6. We define a flag to show whether a transaction is running ('inTx') or completed. A copy of the data to be updated (i.e., output) is persisted in the device memory using the 'cudaMemCpy' function. Then, the flag is set to be inside a transaction ('inTx') and persisted in host PM. After the undo log is persisted, the kernel is launched. Next, after the kernel completes and persists its results using the l2wb instruction, the flag is set to 'complete' and persisted in host PM. With this transaction-style execution, the recovery code at the host side checks the flag. If it is 'inTx', the potentially corrupted data (i.e., output) is restored using the undo log.

The kernel 'histo_kernel_2' in Figure 6 computes a histogram of the input and does not change the input. Also, there are no side effects during kernel execution. Therefore, the kernel is *idempotent*, meaning that it can be executed multiple time without changing the result. We propose to leverage re-execution to recover from failure rather than using the undo log. As a result, we can completely eliminate the code for undo logging (i.e., 'cudaMemCpy' and the first 'cudaDeviceSyncronize') in Figure 6.

In some kernel functions, the input and the output may be altered. In this case, the undo log needs to include the input as well. If we use the non-unified memory model, i.e., the host code explicitly copies the data to the device memory, a copy of the input should already exist in host memory. Therefore, we do not need to make a redundant copy of the input data in device memory. On the other hand, if the unified memory model is used, we need to explicitly make a copy of the input, either in host or device PM.

### B. CTA-Level Durable Transactions

With a CTA as a transaction, undo logging is implemented at the device side. Using the benchmark lbm as an example, the kernel function with undo logging at the CTA level is shown in Figure 7(a). We first create an undo log for the output elements that are to be updated by the CTA. Here, we use the store.wt option as an alternative to the regular store instruction followed by clwb since the log has no reuse in failure-free execution. After ensuring that all the threads persist their log using the sfence followed by the 'syncthreads()' function, we set the flag to be 'inTx' and make it durable. Then at the end of the CTA, we ensure that all the outputs have been persisted using another syncthreads() and set the flag to be 'complete'.

When a CTA is idempotent, i.e., the kernel function has no anti data dependency and is re-executable, the undo log can be eliminated. [1] We only need to set the flag of each CTA as shown in Figure 7(b). In this case, the recovery code simply re-executes those CTAs with their flag being 'inTx' and does not need to undo the changes using the undo log. Note that in Figure 7(b), when we set the flag to 'inTx' (i.e., 1), all the threads in a CTA will execute the code instead of using only thread 0 as in Figure 7(a). The reason is that there is no syncthreads() function right before it. As a result, if only thread 0 updates this flag, there may be a chance that some threads/warps in the CTA change the output before thread 0 sets the flag, violating the transaction semantics. By allowing all the threads to set the flag, it ensures that the flag is set before any thread can change the output. Since all the threads set the same value to the flag, this data race is benign.

```
1 __device__ int log[N_CTA][N_BINS][N_HISTS];
2 __device__ int flag[N_CTA];
3 __device__ int last_iter[CTA]; // last persisted iteration
4 __device__ int last_log_iter[N_CTA]; // last logged iteration
5 tpacf_kernel_loop_log(g_hists, data) {
6   __shared__ int s_hists[N_BINS][N_THD];
7   ... // Initiation
8   // Long nested loop
9   for (i = 0; i < N_ELEMS; i += CTA_SIZE) {
10    // Create the log
11    for (b = 0; b < N_BINS; b++)
12      st.wt &log[cta_id][b][tid], shadow[cta_id][b][tid];
13    st.wt &last_log_iter[cta_id], last_iter[cta_id];
14    {sfence; pcommit;} sfence;
15    __syncthreads(); // log is durable for the CTA
16    if (tid == 0)
17      st.wt &flag[cta_id], inTx; // inside Tx
18    {sfence; pcommit;} sfence;
19    for (k = 0; k < CTA_SIZE; k++) {
20      bin_idx = calculate(data[k]);
21      s_hists[bin_idx][tid] += 1;
22      shadow[cta_id][bin_idx][tid] = s_hist[bin_idx][tid];
23    } //end of the inner loop
24    ...
25    __syncthreads();
26    if(tid == 0) l2wb;
27    st.wt &last_iter[cta_id], i;
28    {sfence; pcommit;} sfence;
29    __syncthreads(); // the results are durable
30    if (tid == 0) st.wt &flag[cta_id], complete;
31    {sfence; pcommit;} sfence; // committed
32  } //end of the outer loop
33  ...}
```

Fig. 8: A code example for loop-level durable transaction.

### C. Loop-Level Durable Transactions

To achieve loop-level durable transactions, we need to analyze the long-running loops in a kernel. The simplified kernel code of the benchmark tpacf is used a case study. The long-running outer loop uses scratchpad memory for data communication among threads within a CTA and is not

[1] Even when a CTA is not idempotent, idempotency analysis shows which stores can be safely repeated, hence we still apply it in order to reduce the size of undo logs.

idempotent. As discussed in Section III-Bc, we use a shadow copy of shared memory variables in global memory and this shadow copy is updated in each iteration. Therefore, we need to create an undo log for this shadow copy. Moreover, besides the flag, we need to record the meta data such as the loop iterator value to indicate which iteration is being executed. The resulting code is shown in Figure 8. As the loop is not idempotent, the log cannot be eliminated and its overhead due to the increased memory traffic can be significant.

In comparison, for the kernels without data communication among threads in a CTA, i.e., each thread works on its private data, loop-level undo logging is relatively simple. Each thread backs up & persists its private data to be changed at the beginning of the loop iteration and sets the flag to be 'inTx'. Then, at the end of the loop iteration, the updated data are persisted and the flag is set to be 'complete'. As there is no shared data, there is also no need for the syncthreads() barrier.

## V. Experimental Methodology

We evaluate our proposed schemes on both an NVIDIA GTX1080 GPU and the GPGPU-Sim [27], a cycle-accurate GPU microarchitecture simulator. The GTX1080 GPU is hosted on a Red Hat 7.4 Linux machine and we use the CUDA 9.0 in our experiments. The simulation configurations of GPGPU-Sim are shown in Table II.

Our experiments use all the benchmarks in the Parboil GPU benchmark suite [28]. The kernels are listed in Table III. As each benchmark may have multiple kernels, a number followed by a benchmark name is used to denote the order of the kernel in the benchmark. For each kernel, we also report whether the kernel function is idempotent in Table III. We classify all the kernels into one of the three categories according to their execution time. If a kernel's execution time is less than 100us, it is categorized as a short-running kernel and is labelled 'S'. The long-running kernels with short-running CTAs are labelled 'LS' and they have the kernel execution time longer than 100us while the average execution time of their CTAs is less than 100us. When a kernel has CTAs that have an average execution time longer than 100us, it is categorized as a long-running kernel with long running CTAs or 'LL'. Note that this classification is ad hoc and should take MTTF and the recovery cost into consideration. We use this setting for two reasons. The first is that it enables us to examine the performance impacts of different persistency models on a variety of kernels and evaluate the effects of our proposed optimizations. With a more realistic setting (e.g., in the order of seconds or minutes), all the kernels would be classified as short-running ones. Second, the 100us threshold used in our classification criteria implies an unreliable system as we need to achieve durable transactions with similar latency. Considering such an unreliable system with volatile memory, in order to make forward progress, we may resort to periodical checkpointing and recovery. A checkpoint would consist of the GPU context and the memory content, and would need to be persisted in host memory to ensure reliability, for which the latency would be much higher than 100us considering the PCIe bandwidth and the cost of GPU context switching. In other words, while volatile memory would not be able to support such fine-grain checkpoints, a GPU with PM along our proposed architectural support can achieve such level of durable transactions with relatively low performance overhead. With a coarser-grain epoch/durable transaction, the performance overhead would be further reduced. In our experiments, for kernel- and loop-level epoch persistency models and durable transactions, the scope option is set to the largest number which satisfies the condition that the execution time of an epoch is smaller than 100us.

In our implementation, our compiler uses inline assembly to insert the new instructions including clwb, pcommit, and l2wb, into the kernel code. We also modify GPGPUsim to support the semantics of these instructions. The sfence instruction is implemented using the membar instruction. In modeling the clwb or l2wb instruction in GPGPUsim, the instruction is sent through the interconnect network to the L2 cache. The l2wb instruction is implemented with the controllers in multiple partitions, which go through every cache line and write back the dirty ones. It blocks subsequent L2 accesses until all the cache lines are checked, resulting in high performance overhead. In our work, we assume non-atomic l2wb instruction, which means a failure could happen during the l2wb execution. With undo logging, if a failure happens during l2wb, the undo_log holds a clean copy of the original data. Therefore, the recoverability is not affected. The clwb instruction writes the specific dirty line in the L2 cache to a WPQ. If the target line is not dirty, no action

TABLE II: Baseline architecture configuration.

| | |
|---|---|
| # of SMs | 20, SIMD width=32, 1.8GHz |
| Per-SM warp schedulers | 4 Greedy-Then-Oldest schedulers |
| Per-SM limit | 2048 threads, 64 warps, 32 CTAs |
| Per-SM L1D-cache | 24KB, 128B line, 6-way associativity, 256 MSHRs |
| Per-SM SMEM | 96KB, 32 banks |
| Unified L2 cache | 2048 KB, 128KB/partition, 128B line, 16-way associativity, 256 MSHRs |
| L1D/L2 policies | xor-indexing, allocate-on-miss, LRU, L1D:WEWN, L2: WBWA |
| Interconnect | 16*16 crossbar, 32B flit size, 1.4GHz |
| Memory Controller | 8 channels, 2 L2 banks/channel, FR-FCFS scheduler, 1.2GHz, BW: 307GB/s |
| NVMM latency | Read: 160ns, write: 480ns |
| DRAM Latency | Read: 160ns, write: 160 ns |

TABLE III: Benchmarks

| Kernel | Type | Idempotent | Kernel | Type | Idempotent |
|---|---|---|---|---|---|
| bfs-1 | S | No | sad-1 | LS | Yes |
| bfs-2 | LL | No | sad-2 | S | Yes |
| cutcp | LS | Yes | stencil | LS | Yes |
| grid-1 | LS | No | tpacf | LL | No |
| grid-2 | LS | Yes | histo-1 | S | No |
| grid-3 | LS | No | histo-2 | S | Yes |
| grid-4 | LS | Yes | histo-3 | LS | No |
| grid-5 | S | No | histo-4 | S | Yes |
| gird-6 | S | No | lbm | LS | Yes |
| mriq-1 | S | Yes | spmv | LS | Yes |
| mriq-2 | LS | No | sgemm | LS | No |

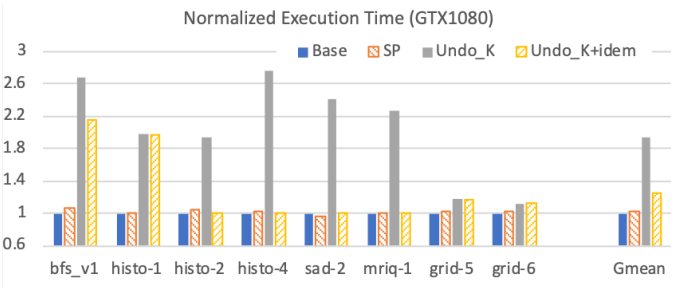


Fig. 9: Normalized execution time of short-running kernels with different persistency models on a GTX 1080 GPU (the lower, the better).

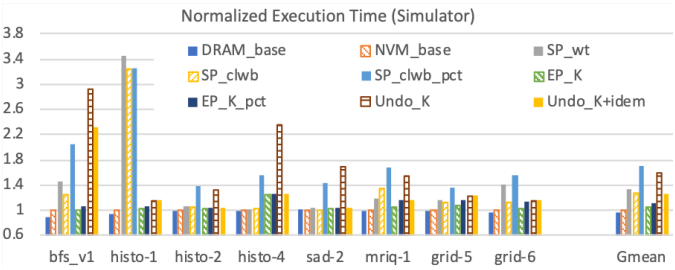


Fig. 10: Normalized execution time for short-running kernels with different persistency models on GPGPUsim.

will be taken except sending back an acknowledgement. If the target is dirty, an acknowledgement is sent back from a memory controller when the dirty cache line reaches the WPQ. The pcommit instruction needs acknowledgements from all the memory controllers when their WPQs are completely drained. For store.wt, our simulator models that its acknowledgment is sent by the memory controller once the data is written to device memory. Therefore, store.wt has higher latency than a clwb instruction in our simulation and uses volatile WPQs by default.

On the GTX1080 GPU, we use store.wt to enforce the store data to be written to memory and the membar.gl instruction as the sfence instruction. As the instructions l2wb and pcommit are not supported on the real GPU, we do not include them in the benchmark code, i.e., ignoring their overheads. Also, as DRAM is used as device memory, the read and write speeds do not reflect the characteristics of NVM. Nevertheless, we run our benchmarks on real GPUs to verify the functional correctness of our compiler generated code and compare the performance trend with our simulation results.

## VI. Experimental Results

In our evaluation, we use the following the naming convention. In the results on a GTX 1080 GPU, 'Base' denotes the baseline execution. We use 'NVM_base'/'DRAM_base' to denote the baseline using NVM/DRAM without persistency support in the simulation results. Among the persistency models, 'SP' denotes strict persistency while 'EP_scope' denotes epoch persistency with a particular scope, which can be 'K' (kernel level), 'C' (CTA-level), or 'L' (loop level). Among the durable transaction models, 'Undo_scope' denotes the undo logging with a particular scope, which adopts the same scope

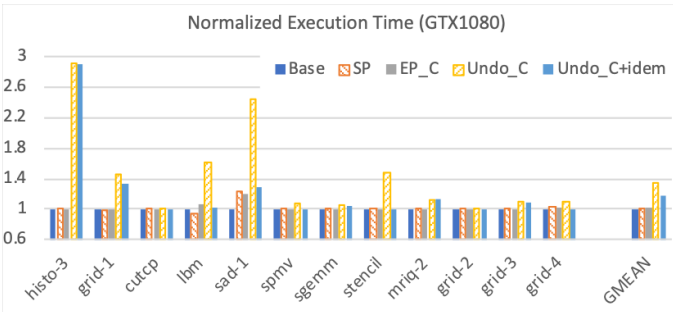


Fig. 11: Normalized execution time of long-running kernels with short-running CTAs using different persistency models on a GTX1080 GPU.

notation as in epoch persistency. We use '+idem' following by 'Undo_scope' to indicate that idempotency analysis is used to optimize undo logging. On the real GPU, store.wt is the only option to persist the memory stores. In the simulation results, we include the 'wt', 'clwb' and 'l2wb' options to denote that the store.wt, clwb and l2wb instructions are used to persist the data, respectively. The label 'pct' is included when pcommit instructions are used for volatile WPQs.

### A. Short-Running Kernels

We first report the performance results of the short-running kernels on the GTX1080 GPU in Figure 9. For each kernel, we show the normalized execution time to the baseline. The performance of kernel-level epoch persistency is the same as the baseline as we cannot include the overhead of l2wb on the real GPU. Several observations can be made from the figure. First, the performance impact of the WT operator and the membar instruction is rather limited, 1.5% on average. Second, undo logging has high overhead although we use the high bandwidth device memory rather than host-side system memory. Third, idempotency analysis eliminates the undo logging overhead if a kernel is idempotent. Fourth, even if a kernel is not completely idempotent, idempotency analysis may still reduce the logging overhead. For example, the kernel bfs-1 shows the high logging overhead, which is due to its short kernel execution time compared to the memory copy time (which in turn indicates that the scope of this kernel is too small as an epoch). Although the kernel is not idempotent, it does not change all its inputs. Idempotency analysis discovers the opportunity and reduces the size of the undo log, thereby reducing the overhead.

The performance results of the short-running kernels on the simulator are shown in Figure 10. The following observations can be made. First, the results correlate well those obtained from the real GPU, thereby confirming the observations made from Figure 9. Second, there is very little performance difference (2.8% on average) between the baseline with DRAM and the baseline with NVM, meaning that the additional write latency has small performance impact in the baseline. Third, when clwb instructions are used to send cache lines to WPQs, durable WPQs show significant performance improvement on average due to the reduced latency and the

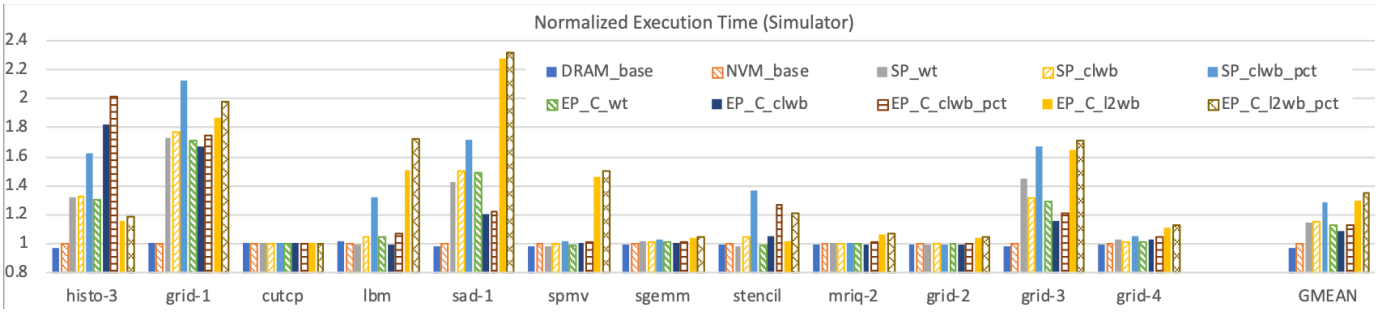


Fig. 12: Normalized execution time of long-running kernels with short-running CTAs using various persistency models on GPGPUsim.

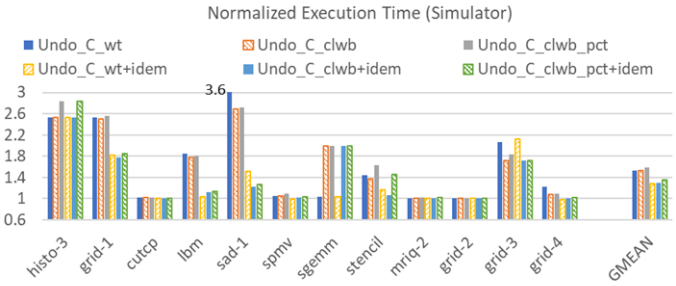


Fig. 13: Normalized execution time of long-running kernels with short-running CTAs using various durable transaction models on GPGPUsim.

removal of pcommit+sfence instructions. Fourth, SP_wt has close performance to SP_clwb, meaning that SP_wt achieves good performance without durable WPQs. The reason is that the clwb instructions introduce additional traffic through the interconnect network whereas the wt operator is part of the store instruction and does not incur additional traffic. Fifth, the kernel-level epoch persistency models have high parallelism due to the l2wb instruction. Therefore, the overhead of the kernel-level epoch persistency model over the baseline is low, 5.5% and 11.6% on average for EP_K (epoch model with durable WPQs) and EP_K_pct (with volatile WPQs), respectively. With undo logging, the performance overhead becomes 25.4% with idempotency analysis.

### *B. Long-Running Kernels with Short-Running CTAs*

We first report the performance of long-running kernels with short-running CTAs on the GTX 1080 GPU in Figure 11. Among the kernels, histo-3 and grid-1 make use of atomic operations on global memory variables. As a result, the CTA-level durable transaction model is not feasible for these two kernels. Therefore, we resort to kernel-level durable transactions for them. From the figure, we can observe: (a) minor overhead of the strict persistency and CTA-level epoch persistency models, (b) relatively high overhead due to undo logging, and (c) significant reduction in the undo-log sizes and performance overhead (from 35.2% to 17.0%) through idempotency analysis.

The simulation results of the persistency models and durable transaction models are shown in Figure 12 and Figure 13, respectively. Compared to the GTX1080 results, Figure 12 and 13 confirms that strict persistency can be supported with relatively low overhead using either the in-place store.wt with volatile WPQs or clwb with durable WPQs. Also, the idempotency analysis effectively reduces the performance overhead of logging as observed on both the real GPUs and the simulator.

Figure 12 also shows that the CTA-level epoch persistency models have lower performance overhead than strict persistency models, especially when clwb instructions are used with volatile WPQs. The reason is that overlapping multiple memory writes as in the CTA-level epoch models saves more clock cycles than the sequential updates as in the strict persistency models. Due to such overlapping, the performance impact of the durable WPQs is also limited in the CTA-level epoch persistency models (i.e., EP_C_clwb vs. EP_C_clwb_pct).

Between the epoch models, EP_C_clwb_pct and EP_C_wt, some kernels show interesting behavior although both models use volatile WPQs. The kernel sad-1 shows better performance with EP_C_clwb_pct while the kernel stencil shows better performance with EP_C_wt. The reason is that sad-1 has streaming-like memory updates but has poor memory coalescing. As a result, write-back caches can leverage spatial locality to reduce the number of memory updates, thereby achieving better performance using EP_C_clwb_pct. On the other hand, the stencil kernel only has one store at the end of the kernel. Therefore, the CTA-level epoch persistency model is the same as the strict persistency model. As the store always misses the L2 cache, the write-not-allocate policy used with store.wt has lower latency than the write-back write-allocate policy. The overhead of the clwb & pcommit instructions also contributes to the lower performance in EP_C_clwb_pct than EP_C_wt.

### *C. Long-Running Kernels with Long-Running CTAs*

Using our criterion in Section V, 2 kernels, tpacf and bfs-2, are classified in the category of long-running kernels with long-running CTAs. The kernel bfs-2 uses atomic operations on global memory variables. So, we choose to use the kernel-level durable transaction model rather than the loop-level model for bfs-2. The performance results on the GTX1080 GPU is shown in Figure 14. As both kernels use shared memory variables, persisting their shadow copies incurs relatively

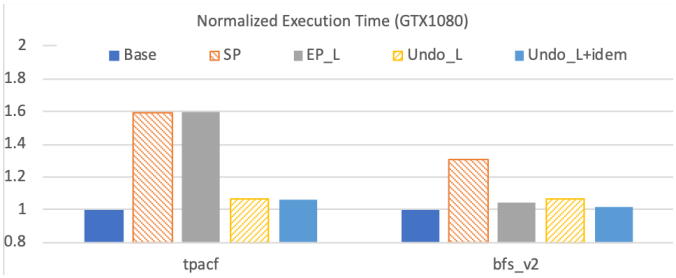


Fig. 14: Normalized execution time of long-running kernels with long-running CTAs using different persistency models on a GTX1080 GPU.

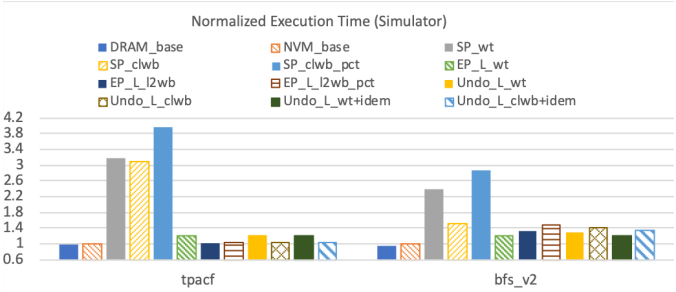


Fig. 15: Normalized execution time of long-running kernels with long-running CTAs using various persistency models on GPGPUsim.

high performance overhead. The loop-level epoch persistency model reduces the overhead for bfs-2 as it enables parallel updates with a single fence at the end of each iteration. For the tpacf kernel, which has higher shared memory usage, such benefit is not clear on GTX1080 but more visible on our simulator (see Figure 15). As the shadow copy of the shared memory variables also serves the role of the undo log, the loop-level durable transaction model for tpacf has the similar performance to the loop-level epoch persistency model. The kernel-level durable transaction model for bfs-2 has small overhead as we do not need to back up shared memory at the kernel level. Since neither kernel is idempotent, the impact of idempotency analysis is limited.

The performance results of the two kernels on the simulator are shown in Figure 15.We can see that the loop-level epoch persistency models have better performance than the strict persistency models. As tpacf uses a high amount of shared memory data, the l2wb instruction at the end of the loop (i.e., EP_L_l2wb_pct) achieves better performance than in-place store.wt instructions (i.e., EP_L_wt) as it enables more overlapping among the updates. The bfs-2 kernel, however, shows the opposite behavior due to its few shared memory updates.

### D. Recommended Models

With the kernel classification criteria in Section V, we list in Table IV the recommended memory persistency model and durable transaction model for each kernel as well as the performance overheads. The average performance overhead to support the memory persistency models is 6.6%. To enable durable transaction with undo-logging, the average performance overhead is 22.2%. Note that the performance overhead is based on the kernel classification criteria in Section V to explore the performance impacts of persistency models and our architecture supports.

TABLE IV: Recommended persistency and durable transaction models and their performance overheads.

| Bench | PM Model | PM Ohd | DT Model | DT Ohd. |
|---|---|---|---|---|
| bfs-1 | EP_K | 0.8% | Undo_K | 131.2% |
| histo-1 | EP_K | 3.7% | Undo_K | 16.0% |
| histo-2 | EP_K | 3.7% | Undo_K | 4.0% |
| histo-4 | SP+wt | 0.2% | Undo_K | 26.5% |
| sad-2 | SP+clwb | 0.3% | Undo_K | 4.8% |
| mriq-1 | EP_K | 5.6% | Undo_K | 16.5% |
| grid-5 | EP_K | 3.2% | Undo_K | 23.2% |
| grid-6 | EP_K | 3.0% | Undo_K | 15.2% |
| histo-3 | EP_C+l2wb | 16.2% | Undo_K | 153.4% |
| grid-1 | EP_C+clwb | 67.6% | Undo_C_clwb | 77.6% |
| cutcp | EP_C+wt | 0.1% | Undo_C_clwb | 0.5% |
| lbm | SP+wt | -0.6% | Undo_C_wt | 3.9% |
| sad-1 | EP_C+clwb | 20.6% | Undo_C_clwb | 22.7% |
| spmv | EP_C+wt | -1.6% | Undo_C_wt | -0.3% |
| sgemm | EP_C+wt | 0.8% | Undo_C_wt | 3.3% |
| stencil | EP_C+wt | -1.2% | Undo_C_clwb | 6.6% |
| mriq-2 | EP_C+clwb | 0.0% | Undo_C_wt | 0.3% |
| grid-2 | EP_C+clwb | 0.0% | Undo_C_clwb | 0.0% |
| grid-3 | EP_C+clwb | 16.1% | Undo_C_clwb | 71.0% |
| grid-4 | EP_C+wt | 1.6% | Undo_C_wt | -1.6% |
| tpacf | EP_L+l2wb | 3.1% | Undo_L_clwb | 4.4% |
| bfs-2 | EP_L+wt | 20.4% | Undo_K | 21.0% |
| Avg. | | 6.6% | | 22.2% |

In our experiments, we also explore the performance impact of coarser epochs/durable transactions. When we adjust the criteria to 200us, 400us, and 800us, the performance overhead of supporting duration transactions with undo-logging is reduced to 17.9%, 13.5%, and 10.9%, respectively.

### E. Impact on Write Endurance

Different memory persistency models lead to different numbers of writes to PM, which may affect its write endurance. Here, we use the long-running kernels with short running CTAs to examine this effect. Figure 16 shows the number of writes for each kernel using different persistency models normalized to the strict persistency model implemented with store.wt instructions. It can be seen that the write-back policy is effective in reducing the write traffic and the CTA-level epoch persistency model further reduces the number of writes by delaying the write backs at the end of the CTAs, which enables more opportunities to combine the updates.

### F. Summary

The key results from our experiments include (1) the strict persistency model incurs higher performance overhead than the epoch persistency models; (2) among the epoch persistency models, epochs with coarser granularities have lower performance overheads and more opportunities to reduce the number of writes; (3) write-through is a good fit for strict persistency while the epoch persistency models work better with clwb; (4) undo logging may introduce significant performance overhead, especially when the execution time of a transaction/epoch


Fig. 16: Normalized numbers of writes in different memory persistency models for long-running kernels with short-running CTAs.

is low; and (5) idempotency analysis effectively reduces the overhead of undo logging for various scopes of epochs/durable transactions.

## VII. Conclusions

In this paper, we adapt, re-architect, and optimize CPU persistency models for GPUs. Besides the architectural support for different persistency models, we highlight that the thread hierarchy in the GPU programming model offers intuitive ways to define the scope of an epoch in epoch persistency. Furthermore, these epochs can serve as boundaries of durable transactions, which are supported through undo logging. We propose idempotency analysis to eliminate unnecessary undo logs, and reduce the size of undo logs when the epoch is not idempotent.

We design a pragma-based compiler approach to facilitate programmers to express the persistent models. Our experiments show that with our proposed architectural support and optimizations, different memory persistency models can be effectively achieved for GPUs to provide various granularities of recoverability at low performance overhead. Our analysis also reveals interesting difference in supporting memory persistency models in GPUs vs. CPUs.

## Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments to improve our paper. This work is supported by NSF grants 1717550, 1908406, 1908079, an AMD gift fund, and UCF.

## References

[1] L. Spelman, "Reimagining the data center memory and storage hierarchy," *Online: https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy/*, May 2018. [Online]. Available: https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy/

[2] Intel, "Intel octane technology." [Online]. Available: Online: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html

[3] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2017, pp. 318–329.

[4] J. T. Mohammad Alshboul and Y. Solihin, "Lazy persistency: a high-performing and write-efficient software persistency technique," in *Proceeding of the 45st Annual International Symposium on Computer Architecuture*, ser. ISCA '18, 2018.

[5] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency: Semantics for byte-addressable nonvolatile memory technologies," *IEEE Micro*, vol. 35, no. 3, pp. 125–131, May 2015.

[6] NVM Library Team at Intel, "Persistent memory programming," *http://pmem.io*.

[7] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 399–411. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872381

[8] 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*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 91–104. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950379

[9] 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.*, vol. 8, no. 11, pp. 1226–1237, Jul. 2015. [Online]. Available: https://doi.org/10.14778/2809974.2809984

[10] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "Engineering a high-performance gpu b-tree," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 145–157. [Online]. Available: http://doi.acm.org/10.1145/3293883.3295706

[11] kinetica. [Online]. Available: Online: https://www.kinetica.com

[12] N. Sakharnykh, "Beyond gpu memory limits with unified memory on pascal," *https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/*, 2016.

[13] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 265–276. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665712

[14] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 2, pp. 135–148, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3093315.3037730

[15] J. Menon, M. de Kruijf, and K. Sankaralingam, "igpu: Exception support and speculative execution on gpus," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 72–83.

[16] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "ido: Compiler-directed failure atomicity for nonvolatile memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 258–270.

[17] 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)*, Feb 2017, pp. 361–372.

[18] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 178–190. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124539

[19] A. Singh, S. Aga, and S. Narayanasamy, "Efficiently enforcing strong memory ordering in gpus," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 699–712.

[20] HSA Foundation, "Hsa programmer's reference manual: Hsail virtual isa and programming model, compiler writer, and object format (brig)," 2015.

[21] A. Munshi, "The opencl specification (version 2.0)," *Khronos OpenCL Working Group*, Nov. 2013.

[22] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood, "Lazy release consistency for gpus," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–14.

[23] X. Ren and M. Lis, "Efficient sequential consistency in gpus via relativistic cache coherence," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 625–636.

[24] D. Gope, A. Basu, S. Puthoor, and M. Meswani, "A case for scoped persist barriers in gpus," in *Proceedings of the 11th Workshop on General Purpose GPUs*, ser. GPGPU-11. New York, NY, USA: ACM, 2018, pp. 2–12. [Online]. Available: http://doi.acm.org/10.1145/3180270.3180275

[25] S. Kannan, N. Farooqui, A. Gavrilovska, and K. Schwan, "Heterocheckpoint: Efficient checkpointing for accelerator-based systems," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 738–743.

[26] "Nvidia ptx isa," *http://docs.nvidia.com/cuda/parallel-thread-execution/index.html*.

[27] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009, pp. 163–174.

[28] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, vLi Wen Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *IMPACT Technical Report*, 2012.