Scalable and Fast Lazy Persistency on GPUs

Ardhi Wiratama Baskara Yudha
University of Central Florida
yudha@knights.ucf.edu

Keiji Kimura
Waseda University
keiji@waseda.jp

Huiyang Zhou
North Carolina State University
hzhou@ncsu.edu

Yan Solihin
University of Central Florida
Yan.Solihin@ucf.edu

***Abstract*—GPUs applications, including many scientific and machine learning applications, increasingly demand larger memory capacity. NVM is promising higher density compared to DRAM and better future scaling potentials. Long running GPU applications can benefit from NVM by exploiting its persistency, allowing crash recovery of data in memory.**

In this paper, we propose mapping Lazy Persistency (LP) to GPUs and identify the design space of such mapping. We then characterize LP performance on GPUs, varying the checksum type, reduction method, use of locking, and hash table designs. Armed with insights into the performance bottlenecks, we propose a hash table-less method that performs well on hundreds and thousands of threads, achieving persistency with nearly negligible (2.1%) slowdown for a variety of representative benchmarks. We also propose a directive-based programming language support to simplify programming effort for adding LP to GPU applications.

I. Introduction

Graphics Processing Units (GPU) is increasingly used in high-performance computing (HPC) which requires a large memory capacity. Cutting edge machine learning (ML) applications, such as multi-camera activity recognition [1], also requires much larger memory capacity than provided in current GPU systems. As DRAM scaling is facing difficulties, non-volatile memory (NVM), such as Intel Optane DC persistent memory [2], has stepped in to provide the higher density and better scaling potential for future main memory. Furthermore, NVM offers additional benefits such as byte addressability, low leakage power, and non-volatility, while providing read latency that is closer to DRAM than to SSD. As an increasing number of GPU applications demand larger memory capacity, NVM becomes more attractive in future GPUs. A challenge of current NVM is the bandwidth (especially write bandwidth) that is lower than what GPUs demand. Hence, it is likely that DRAM will be used either as a front end buffer for the NVM, or needed pages may be brought to DRAM on demand from NVM [3].

NVM provides a unique opportunity to store data persistently in memory data structures instead of in files, bypassing the substantial overheads of interacting with the file system. To achieve that, data must be crash recoverable to a consistent state upon crash recovery, and computation must be able to recompute or resume execution from when crash occurs. To aid programmers in reasoning about crash recovery, a typical system provides a *persistency model* that provides primitives to push data into the persistency domain, and to specify the relative ordering of such pushes. In the Intel PMEM, the former is provided through cache write back and flush instructions (clwb, clflush/clflushopt), while the latter is provided through persist barrier instruction (sfence). Similarly, other persistency models in literature include strict persistency, epoch persistency, and buffered epoch persistency [4]–[8]. To use them, the programmer must actively orchestrate these instructions to maintain a redo or undo log and specify atomic durable regions, hence these approaches are referred to as *Eager Persistency* (EP).

More recently, another approach referred to as *Lazy Persistency* (LP) was proposed [9]. Utilizing LP does not require the programmer to insert any persistency instructions at all. Instead, the programmer defines regions in code, and protect each region with a checksum that can detect persistency failure of the region. Upon a crash, the checksum of each region is validated, and any regions for which the checksum validation fails must be re-executed as part of crash recovery. There are trade offs between EP vs. LP. EP incurs a large overhead during normal execution, including maintenance of logs, loss of locality due to cache line flushing, and processor stalls due to persist barriers. 20-40% slowdowns are typical for EP. LP, on the other hand, has none of such overheads, hence LP on CPUs have been reported to have negligible performance overheads of only 1% [9]. As a trade off, crash recovery is slower in LP, and its applicability is limited to associative code regions.

In contrast to CPUs, persistency on GPU systems is only just beginning, and only EP and logging has been explored in GPUs [10], [11]. A class of emerging GPU applications require crash recovery to run correctly or efficiently. Examples include GPU-accelerated memory databases, such as Mega-KV [12], GPU B-Tree [13], Kinetica [14], etc. In addition, many emerging GPU applications are also long-running, including training deep neural networks, computing proof of work in blockchain applications, scientific computation using iterative approaches, etc. Finally, supporting crash recovery also improves checkpointing performance by relegating system checkpointing for more serious faults, hence the checkpointing frequency can be reduced [15].

On the surface, LP is a promising technique to apply on GPUs for several reasons. First, it was demonstrated to achieve nearly negligible performance overheads on CPUs; if this translates equally well to GPUs, it will be the first technique that achieves persistency at very low overheads. Second, considering NVM has limited write endurance and EP techniques rely on logging and cache line flushing that incur substantial write amplification, LP is attractive as it was demonstrated to produce very small write amplification in CPUs. However, GPUs involve thousands of threads to do the computation, orders of magnitude higher than in CPUs, hence it is unclear if the performance and write amplification characteristics of LP shown in CPUs will be preserved in GPUs, especially with GPU high thread counts.

In this paper, we characterize the performance and scalability of Lazy Persistency (LP) on GPUs. The central questions we would like to answer are: what regions can be used as LP regions in the context of GPUs? What performance overheads will LP incur on GPUs? What scalability bottlenecks may present a challenge to LP on GPUs? Based on the characterizations, we identify aspects of LP that need to be redesigned on GPUs. Then, we present a design for LP on GPUs that avoids the bottlenecks, utilizing GPU-specific optimizations. Finally, we present a directive-based programming language support that can express LP in GPUs well.

To summarize, this paper makes the following contributions:

- To our knowledge, this paper is the first to propose Lazy Persistency for GPUs. We identify the design space for mapping LP on GPUs.
- We present characterization of LP performance on GPUs, looking into the design space explorations based on (1) hash table choices, (2) the use of locks vs. lock-free, and the use of (3) sequential vs. parallel reduction. The characterization leads to insights into the performance bottlenecks of LP on GPUs.
- Based on the insights of performance bottlenecks, we present a new hash-table-less design that produces very low performance overheads.
- Finally, we propose and show that a directive-based programming language support can express LP in GPUs well, allowing the programmer to achieve LP with low programming complexity. Such a support has not been demonstrated, even in CPUs.

The remainder of the paper is organized as follows. Section II discusses background and related work. Section III discusses the evaluation testbed. Section IV describes LP design space exploration on GPUs and performance characterization results. Section V shows our hash table-less LP design on GPUs. Section VI discusses our directive-based programming language support. Section VII discusses evaluation results of the new design. Finally, Section VIII concludes the paper.

II. BACKGROUND AND RELATED WORK

A. Lazy Persistency on CPU

Lazy Persistency (LP) requires that programmers organize their algorithm into *LP regions*, where each region is a *unit of recovery*. LP regions must be *associative*, meaning that the order in which the regions are persisted must not affect the correctness of the output. For example, iterations of a loop that perform reduction, such as a loop that finds the maximum value element in an array, is associative because whatever the order the iterations is, the maximum value will be the same. Associativity is important for LP regions because with LP, there is no guarantee which iterations will be persisted before others; any regions that were found not fully persisted will be recovered through crash recovery code, while other regions that have persisted fully are left alone. The crash recovery code is specific to the code structure in the region. A specific special case is when an LP region is *idempotent*. An idempotent region is one that can be executed multiple times without changing the result. An idempotent LP region makes crash recovery simple as the region can simply be re-executed upon a crash. However, idempotency is a small subset of what LP regions can be.

An LP region is protected by a checksum, which is selected to enable detection of whether all stores in the region persisted successfully. Example checksums may be parity, modular checksum, and Adler-32 [9]. With any checksum, there is a small probability of false positive, i.e. the checksum indicates no persistency failure when in fact there is a failure. To avoid this, more than one checksum can be used simultaneously to protect each region.

With LP, none of stores in a region, including the checksum store itself, need to be flushed from the cache to memory. They will be naturally evicted and written back from the cache, possibly long after the stores were performed.

Listing 1 shows an example CPU LP code for tiled matrix multiplication. In the example, the LP region is an *ii* iteration. In each *ii* iteration, a checksum is initialized (typically assigned NaN value). Each store in the region that needs to be persistent updates the checksum (line 12). At the end of the LP region, a hash table is accessed and the checksum is inserted (lines 16-17).

Listing 1: CPU Lazy Persistency code for tiled matrix multiplication [9].

```
1  for (kk=starting_kk; kk<n; kk+=bsize) {
2      for (ii=starting_ii; ii<n; ii+=bsize) {
3          // Each ii iteration is an LP region
4          ResetCheckSum();
5          for (jj=0; jj<n; jj+=bsize) {
6              for (i=ii; i<(ii+bsize); i++) {
7                  for(j=jj; j<(jj+bsize); j++) {
8                      sum = c[i][j];
9                      for(k=kk; k<(kk+bsize); k++)
10                         sum += a[i][k]*b[k][j];
11                     c[i][j] = sum;
12                     UpdateCheckSum(c[i][j]);
13                  }
14              }
15          } // for jj
16          hashIndex = GetHashIndex(ii,kk);
17          HashTable[hashIndex] = GetCheckSum();
18      }
19  }
```

The programmer needs to determine the LP region in the code. Since LP regions must be associative, there should not be data dependencies between regions. Another aspect of choosing the LP region is the granularity. A smaller LP region incurs a higher relative overhead in computing the checksum and keeping a larger hash table for checksums. A larger LP region, on the other hand, incurs a longer recovery time since more work is lost upon a crash, but reduces the overheads of checksum computation and hash table maintenance. Thus, the programmer needs to carefully consider this trade-off when choosing the LP region.

After determining the LP region, checksum needs to be computed (line 12 in the figure). Since multiple threads will compute a value that is aggregated into the checksum, in the LP on CPUs, the checksum update must be protected by a lock. Hence, not only the LP checksum calculation is done sequentially for each LP region assigned to a thread, the checksum update also incurs critical section overheads. Such an approach is feasible for a low thread count in CPUs, but not for GPUs with thousands of threads.

LP on CPUs also rely on using a hash table to organize the checksums (lines 16-17) [9]. Each checksum for an LP region is computed, it inserts the checksum into the hash table. A hash table works by hashing a value to a hash table index, and the value is then inserted into the indexed entry. The insertion is protected by a lock due to possible multiple concurrent insertions by multiple threads. The insertion also may result in a hash table collision. To handle a collision, different strategies may be adopted. With CPU, chaining is a feasible approach, where each hash table entry points to a linked list, and collision is handled by adding the new value into the linked list. Since CPUs only having a small number of cores compared to GPUs, hash table insertion and chaining for handling collision are a feasible strategy. For GPUs with thousands of threads, a more scalable alternative is needed.

The recovery process after a failure is dependent on the code. The recovery is needed to restore the program to the consistent state upon recovering from a failure. There are two strategies for recovery in LP. The first type is eager recovery, which ensures forward progress on recovery. Alternatively, lazy recovery may be used, but increases the risk of not making forward progress if a crash occurs during recovery. The eager recovery is more appropriate since although it is expensive, since it guarantees forward progress and we only use it in on occasional time where recovery from failure is needed.

B. GPU Architecture and Programming

A GPU uses hundreds or thousands of cores to compute. The cores are organized into a hierarchical structure, starting from a single core packed together with dozens of other cores to form a Streaming Multiprocessor (SM). A GPU consists of multiple SMs. In Volta architecture, a single GPU contains 84 Volta SMs, each SM contains 64 FP32 cores, 64 INT32 cores, 32 FP64 cores, 8 tensor cores, and 4 texture units [16]. A Single Instruction Multiple Thread (SIMT) paradigm is used in GPUs to perform massively parallel execution of threads on many cores. A group of 32 threads form a warp. The threads within a warp will execute the same instruction. Multiple warps will be organized into a thread block. An SM will execute several thread blocks. Threads within a thread block share memory resources including the L1 cache and shared memory. The L1 cache is managed by the hardware and is transparent to the program. Shared memory is visible to the program and is extensively used to optimize GPU applications. Moreover, starting from Kepler Architecture [17], threads within a warp could communicate directly with one another at the register level without going down to shared memory. We will exploit this feature to minimize the performance overhead of LP in our proposed design.

On the Kepler Architecture, NVIDIA introduces instruction support for parallel reduction. Prior to the Kepler architecture, parallel reduction is done using shared memory. In order to perform reduction, data is stored to shared memory, then after synchronization with another thread, it is fetched back into the register from shared memory. The new *shuffle* instruction will efficiently achieve parallel reduction by exchanging data between threads in the same warp.

The presence of shared memory enables GPU application developers to exploit data locality explicitly. On a GPU application, threads within a thread block will work together while communicating with each other through shared memory. The typical usage of shared memory has the following pattern. First, each thread fetches data from the (slow) global memory to the (fast) shared memory. Then, during computation, threads in a thread block read from and write to data in shared memory instead of going to global memory. At the end of computation, the computation results from each thread block are then stored back to global memory.

III. Evaluation Testbed

A. System configuration

We evaluate our characterization on an NVIDIA Tesla V100 GPU system. The GPU runs CentOS Linux release 7.4.1708 with NVIDIA driver version 440.33.01. For compilation we use the CUDA version 10.1. and GCC version 6.2.0. The system is DRAM based since there is no available commercial NVM-based GPU. Therefore, the results that we obtain should not be interpreted as absolute performance overheads that measure persistency on NVM as the read and write latencies will not be the same with using an actual NVM. Rather, our results can be interpreted for relative performance overheads between various schemes that allow us to reason about which scheme performs better than others and the reasons behind it. Furthermore, the results of LP are more closely aligned to the real NVM than Eager Persistency (EP) because with LP, we do not rely on any persistency instructions such as cache line flushes and store fences.

B. Benchmarks

We use tiled matrix multiplication [18], benchmarks from Parboil suite [19], and a real world key-value store application MEGA-KV [12]. From the Parboil suite, we selected benchmarks with differing performance bottlenecks: Two-Point Angular Correlation Function (TPACF), MRI Cartesian Gridding (MRI-Gridding), and Sparse Matrix-Dense Vector Multiplication (SpMV), Sum of Absolute Differences (SAD), Saturating Histogram (HISTO), Distance-Cutoff Coulombic Potential (CUTCP) and Magnetic Resonance Imaging - Q (MRI-Q). Table I describes these benchmarks along with their performance bottlenecks identified by a prior study [19]. We use the biggest input provided in the parboil data sets.

TABLE I: Benchmarks

Name	Input	Suite	Bottleneck
TMM	4096x4096	[18]	Inst throughput
TPACF	Biggest Input	Parboil [19]	Inst throughput
MRI-GRIDDING	Biggest Input	Parboil	Inst throughput
SPMV	Biggest Input	Parboil	Bandwidth
SAD	Biggest Input	Parboil	Bandwidth
HISTO	Biggest Input	Parboil	Bandwidth
CUTCP	Biggest Input	Parboil	Inst throughput
MRI-Q	Biggest Input	Parboil	Inst throughput
MEGA-KV	insert, search, & delete 16K recs.	[12]	Unkown

IV. GPU Lazy Persistency Exploration

LP on CPUs provides an alternative to EP in achieving crash recoverability, with fast normal execution (the common case) but more complex crash recovery (the rare case). In this section we will explore the design space for mapping LP to GPUs, and explore whether the benefits of LP in CPUs also apply in GPUs. A further benefit of LP is that it is implementable in current GPUs right away because no new instructions are required. In contrast, EP requires cache line flush and durable barrier instructions which are not supported in current GPUs [10].

To map LP to GPUs, we identify the following design space aspects: (a) LP region selection; (b) checksum computation method; (c) checksum table organization; and (d) using locks or using lock-free design. These aspects must take into account that GPU applications tend to stress on interconnect and/or memory bandwidth, employ thousands of threads, and the hierarchical parallel nature of GPU programming model. We will discuss each aspect subsequently.

A. LP Region Selection

LP regions are units of crash recovery and must be associative. LP region selection must consider (a) a good balance between checksum overhead and recovery granularity and (b) simplicity of recovery code construction. We observe that the the thread hierarchy in GPU programming model actually provides a natural fit to this problem, for several reasons. First, thread blocks are *naturally associative* as GPU hardware provides no guarantee on the execution order of thread blocks. No additional manual or compiler analysis is needed to identify them, in contrast to code for CPUs. Second, they usually present a good amount of work such that the checksum computation will likely not become a major overhead. Third, they can be enlarged if needed, e.g. through thread block fusion [20]. Fourth, threads within each thread block can synchronize and communicate through on-chip shared memory, thereby providing the efficient support for checksum computation. Fifth (and finally), using thread block as the LP region also simplifies the recovery code.

The recovery code contains two parts: validation and recovery function. After a crash, the recovery kernel, which has the same thread dimension as the original kernel, first validates the checksum for each thread block by fetching the corresponding data from memory, computing their checksums, and then comparing them with the ones stored in the checksum table. For the failed thread blocks, the recovery function is then invoked. Usually a thread block is idempotent, hence the recovery function is trivially identical to the original kernel function. Such idempotency can be statically identified using compiler. For non-idempotent thread blocks, the recovery function is application dependent.

One issue with LP is that the validation and recovery may affect arbitrarily old regions due to the lack of guarantee that old regions persisted successfully. To avoid this, we can combine periodic checkpointing [21]–[23] or periodic whole-cache flushing [9]. With such mechanisms, only regions newer than the checkpoint or flushing point need to be validated and possibly recovered. The interval period can be selected based on probability of crashes and recovery time to achieve a certain MTBF or availability target.

The natural associativity and ease of crash recovery code construction make GPUs a better fit for LP than CPUs, and lend itself to a directive-based programming support, which allows the programmer to insert a small number of `pragma`'s (discussed in Section VI).

B. Checksum Computation Methods

The next design aspect to consider is the checksum computation method. For a thread block LP region, all threads in the thread block will perform their own checksum computation and then generate combined/reduced checksums collectively. Two factors must be considered: type of checksums and whether or not checksum computation should be parallel.

To enable the detection of persistency failure, the checksum is computed over all store values that must persist in an LP region. Hence, if any store (including the checksum store) did not persist, at crash recovery the checksum validation fails, i.e. the recomputed checksum value mismatches with the stored checksum. An ideal checksum has a low execution time overhead yet has very low *false negative* rate (matching checksum despite some store not persisted). In CPUs, three checksums were considered: *parity checksum* (store values are XORed together), *modular checksum* (store values are added), and *Adler-32* [24], [25] (used in compression libraries). Through random error injection, modular and Adler-32 checksums both provide false negative rates of less than one in two billion ($2{\times}10^{-9}$) [9]. However, Adler-32 is significantly more expensive than modular checksum. Considering these, we select two simultaneous checksums: modular and parity checksums. The false negative rate using these two checksums simultaneously falls to less than one in a trillion (10^{-12}), which is more than good enough for our purpose.

We illustrate our LP GPU support with a matrix multiplication example shown in Listing 2. Lines 21-30 show the LP support: lines 21-24 compute the checksum and lines 26-30 store the checksum in the checksum hash table.

Listing 2: Matrix multiplication kernel with LP code.

```
1  __global__ void tiledMatrixMul(int *a, int *b, int
       *c, int n,
2  int tile_size) {
3    __shared__ int A[SHMEM_SIZE];
4    __shared__ int B[SHMEM_SIZE];
5    int row = by * tile_size + ty;
6    int col = bx * tile_size + tx;
7    int temp_val = 0;
8
9    // Sweep tiles over entire matrix
10   for (int i = 0; i < (n / tile_size); i++) {
11     A[(ty * tile_size) + tx] = a[row * n + (i *
       tile_size + tx)];
12     B[(ty * tile_size) + tx] = b[(i * tile_size *
       n + ty * n) + col];
13     __syncthreads();
14     for (int j = 0; j < tile_size; j++) {
15       temp_val += A[(ty * tile_size) + j] * B[(j *
        tile_size) + tx];
16     }
17     __syncthreads();
18   }
19   c[(row * n) + col] = temp_val;
20
21  int lane = ((ty * tile_size) + tx) % warpSize;
22  int warpId = ((ty * tile_size) + tx) / warpSize;
23
24  int* reduced_vals = blockReduceSum(temp_val, n,
        tile_size, lane, warpId);
25
26  if (lane == 0 && warpId == 0) {
27      int key = by*gridDim.y + bx + 1; //blockId+1
28      int key2 = by*gridDim.y + bx + gridDim.x *
       gridDim.y + 1;
29          cuckooSingleInsert(_tableState, key,
        reduced_vals[0]);
30        cuckooSingleInsert(_tableState, key2,
        reduced_vals[1]);
31 }
```

A second factor to consider is how to compute the checksum: sequentially or in parallel. In CPU, it is computed sequentially. In GPUs, we can leverage parallel reduction through the *shuffle down* instruction. Parallel reduction lowers the number of iterations vs. sequential from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$, where N is the number of checksums to be combined/reduced. Listing 3 shows the two steps. First, each warp performs a reduction for all the threads within each warp (warpReduceSum function on line 5) and the reduction results are stored in shared memory with an array indexed by the warp id. Second, after all the warps in the thread block finish their checksum computation, which is ensured with a barrier on line 9, the items in the array are reduced by warp 0 using the same warpReduceSum function (line 12).

Listing 3: Parallel reduction at the thread block level.

```
1  __inline__ __device__
2  int  blockReduceSum(int val, int lane, int warpId,
        unsigned mask) {
3    int ix=blockIdx.x*blockDim.x+threadIdx.x;
4    static __shared__ int shared[32]; //32 part sums
5      val = warpReduceSum(val, mask); //part reduce
6    if (lane == 0) {
7      shared[warpId] = val; //write reduced value
8    }
9    __syncthreads();  //wait for all part reduce
10   val = (ix < blockDim.x / warpSize) ? shared[lane
       ] : 0;
11   if (warpId == 0) {
12     val = warpReduceSum(val, mask); //final reduce
13   }
14   return val;
15 }
```

The warp-level reduction, warpReduceSum, is accelerated using the shuffle down instruction, as shown in Listing 4 and illustrated in Figure 1. As discussed earlier, to keep the false negative rate low, multiple checksums are computed in the warpReduceSum function.

Listing 4: Parallel reduction at the warp level with two checksums.

```
1  __device__  int* warpReduce(int val, int val2,
       unsigned mask, int *results)
2  {
3    int offset;
4    for (offset=warpSize/2; offset>0; offset/=2) {
5      val += __shfl_down_sync(mask, val, offset);
6      val2 ^= __shfl_down_sync(mask, val2, offset);
7    }
8    results[0] = val;
9    results[1] = val2;
10   return results;
11 }
```

XOR cannot be applied to floating point data in CUDA. Hence, to compute the checksums, all floating point numbers that are the elements of the matrices are converted into ordered

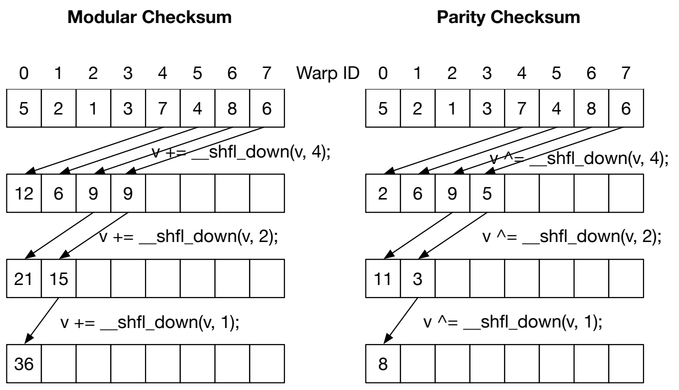

Fig. 1: An example of two checksum calculations using the `shfl_down` instruction, with 8 warps. For parity checksum, the 4-bit binary representation of each number is used, e.g. 0010 (2) XOR 1001 (9) = 1011 (11).

integer prior to applying bitwise XOR. The conversion includes both the exponent and mantissa, allowing the detection of persistency failure for either of them. Figure 2 illustrates an example binary representation of a floating point data with a value of 3.5, with a single sign bit, 8-bit exponent, and 23-bit mantissa. The conversion concatenates all the bits, resulting in an integer value of 1080033280.

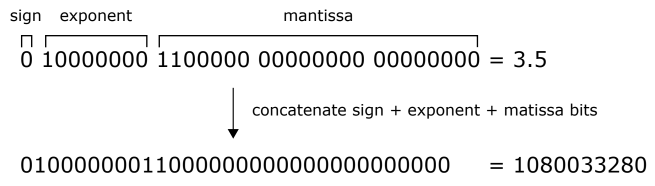

Fig. 2: floating point data to integer conversion

C. Hash Table Organizations

After a checksum is calculated, a thread block stores it into the modified data structure or a hash table. In the former, the main data structure of the original benchmark is modified to embed the checksums. In the latter, a separate data structure (i.e., a hash table) keeps the checksums. The former approach requires a substantial change to the original program and increases the complexity of the programming. It is also not compatible with a directive based programming language support. Hence, the latter is more attractive.

In CPUs, hash table data structures can be more sophisticated when handling collisions, e.g. collision by chaining (Figure 3 (left)). In the example, if a key is hashed into a non-empty index, the key is inserted into a linked list chained to the entry. With chaining, the hash table never fills up and the performance is less sensitive to the *load factor*, i.e., the ratio of number of keys hashed to number of hash table entries. However, it requires pointer chasing and lock synchronization on the shared entry.

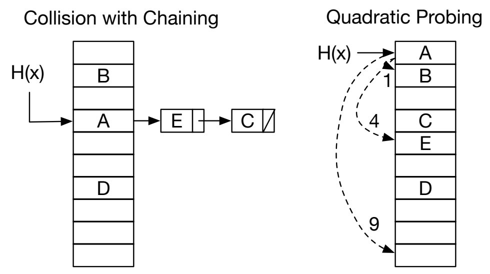

Fig. 3: Illustration of the hash table with chaining for LP implementation on CPU (left) and quadratic probing hash table (right)

In GPU, we need a more scalable hash table even if it is less flexible. A GPU hash table ideally does not use pointers, is lock free, generates few collisions, scales well with large thread counts, has low average insertion latencies and limited worst-case insertion latencies. The key insertion latency is in the critical path of execution time when the system is running normally, hence a slow insertion can be a performance bottleneck for the system. However, the lookup time is not in the critical path, as it is performed only on crash recovery, which is the rare case. Furthermore, we know the number of unique keys in advance, since the number of thread blocks is known. So we can size the hash table in a way to avoid the hash table becoming full and keep the load factor in the range that produces low probability of collisions. This makes chaining unnecessary and unattractive. Finally, in order to avoid use of pointers, we consider only *open addressing*, where all keys are placed in the hash table itself.

Considering the characteristics, we use two hash tables: quadratic probing and cuckoo hash tables. The quadratic probing hash table uses a simple mechanism when dealing with collision, as illustrated in Figure 3 (right). It first checks the entry the hash produces. If the entry is full, it calculates the next index by adding the successive value of the power of two to the original hash function result, i.e. adding i^2 to the original index in the i^{th} iteration. This process is repeated until an empty entry is found. In the figure, key X initially collides in the entry occupied by A, hence the second index is calculated by adding $1^2 = 1$ to the first index. Unfortunately, the second entry is occupied by key B, so $2^2 = 4$ is added to calculate for the third index, which also collides with E. Finally, $3^2 = 9$ is added to calculate the fourth index and the empty entry is found. Quadratic probing is simple to implement, and has been shown to produce superior performance to linear probing. However, it has two disadvantages: high worst case insertion time due to possibly high number of collisions, and it works well only if the load factor is 70% or less.

The next hash table that we consider is Cuckoo hash table. Compared to quadratic probing, the cuckoo hash table theoretically provides an amortized constant worst case insertion time [26], [27]. The standard cuckoo hash table uses two tables

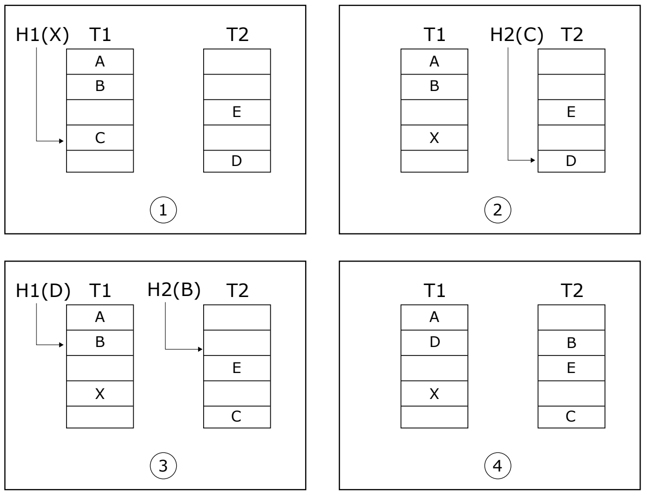

Fig. 4: Illustration of key insertion with collision on Cuckoo hashing.

and two hash functions to index the table. Let us denote the two tables as T_1 and T_2 and their hash functions H_1 and H_2, respectively. Figure 4 illustrates an example insertion step. Initially there are three keys A, B, and C in table T_1 and there are two keys D and E in table T_2. Suppose now a key X needs to be inserted into the hash table. In Step (1), $H_1(X)$ is computed to index T_1. Because the entry is not empty, X is inserted into the table, evicting key C. In Step (2), a new place for key C is calculated using H_2 to index T_2, i.e., $T_2(H_2(C))$. Unfortunately, the entry in table T_2 is occupied by key D, hence C is inserted and D is evicted. In Step (3), we find a new place for key D in table T_1, which results in inserting key D to table T_1 and evicting key B. Finally, in Step (4) key B is inserted into an empty entry in table T_2. An unlikely but possible case is when successive evictions end up with a cycle. When a cycle is detected, the cuckoo hash table performs hash tables rehashing with new tables and new functions. The lookup process of a Cuckoo hash table is to simply find all possible indices pointed by all hashing functions. For example, to lookup key X, we will find on indices $T_1(H_1(X))$ and $T_2(H_2(X))$. Cuckoo hashing was theoretically proven to have constant-time worst-case insertion time, constant-time worst-case lookup, and constant-time worst-case deletion time, which is attractive for our LP GPU use case. There are drawbacks, however. First, there are always as many lookups as the number of tables. Fortunately, lookups are not in the critical path of execution time, as they are only needed for crash recovery. Second, the load factor should be kept at less than 50% [28], beyond that the performance dramatically degrades.

1) Using or not Using Locks: In any of the hash table, we can avoid using locks by relying on atomic instructions. The atomic instruction is required for inserting a checksum into the hash table in order to make sure there is no race condition. The atomic instruction that we use is atomic compare and swap (**atomicCAS()**) for quadratic probing, which allows the table to make sure that the hash table entry is empty, before inserting a new key.

For cuckoo hashing, we use atomic exchange (**atomicExch()**) for exchanging a key to be inserted with a key that may already be present in the hash table entry. The **atomicExch()** instruction allows a lock-free implementation to guard against race condition due to simultaneous insertions into the same hash table entry. In this implementation, we do not use atomic compare and swap since in cuckoo hashing we will always insert the entry to a hash table entry no matter whether it is occupied or empty.

D. Performance Characterization Results

To evaluate the design space for LP on GPUs, we characterize the performance over the design space.

1) Performance Overhead for Naive LP Implementation: We compare the performance from the two different hash tables: Cuckoo hash table (Cuckoo) and quadratic probing (Quad). Figure 5 shows the execution time overheads for all of the benchmarks and their geometric mean. All hash tables use parallel reduction. The figure shows that in general, Cuckoo is slower (31.7% mean overhead), compared to Quad (29.4%). For MRI-GRIDDING, Quad's overhead is truncated because it is larger than the y axes scale (218.6%) and for SAD, Cuckoo's overhead is truncated (232.79%). The reason is that MRI-GRIDDING and SAD uses a lot of thread blocks, thus there is a high amount of contention while inserting checksums into to the hash table. For TMM, TPACF, and SPMV, HISTO, CUTCP, and MRI-Q the difference of the overheads between the two mechanisms are much smaller. The reason behind this is that the number of the thread blocks used is much smaller. Since the number of the thread blocks is small, the degree of memory contention is smaller during insertion process. We hypothesize that big performance overhead is primarily due to the huge number of collision in the table.

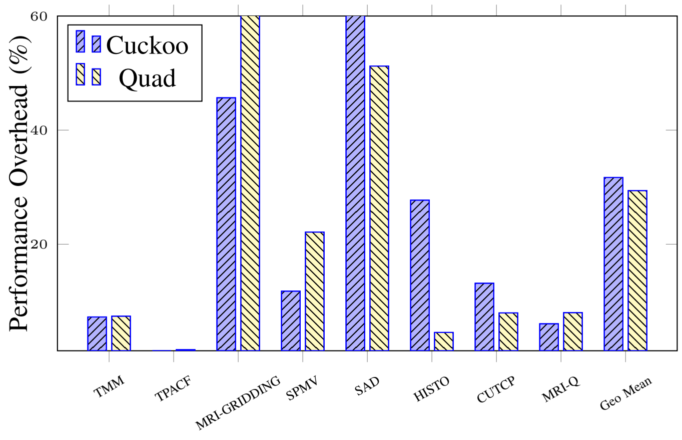

Fig. 5: Overhead compared to baseline for different hash table

2) Impact of hash table collisions: To test the hypothesis, we collect the number of hash table collisions for Quad and Cuckoo (Table II). The table confirms a large number of collisions for TMM, MRI-GRIDDING, and SAD, with Cuckoo achieving lower number of collisions for TMM and MRI-GRIDDING that correlate with its lower performance overheads, especially for MRI-GRIDDING. While for SAD, Quad achieving lower number of collision that leads to lower performance overhead. We further look at MRI-GRIDDING using the quadratic probing and cuckoo hashing, and modify the code to remove collision by making sure that the entry lookup for the first time during insertion is always empty. We evaluate this and the result confirmed the hypothesis, with the overheads dropping dramatically to only 0.1% and 0.8% for cuckoo hashing and quadratic probing, respectively. Therefore, much of the slowdown comes from hash table collision.

TABLE II: Number of hash table collisions.

Name	Quadratic Probing	Cuckoo Hashing
TMM	60443	38951
TPACF	532	483
MRI-GRIDDING	172978	26351
SPMV	57	39
SAD	31971	44566
HISTO	26	54
CUTCP	550	562
MRI-Q	120	112

3) Impact of atomic instruction: Another aspect we try to investigate is the usage of atomicExch() instruction in Cuckoo hashing and atomicCAS() instruction is Quadratic probing. For Cuckoo hashing we replace the atomicExch() with the code to do swap between two variable using a temporary variable. For Quadratic probing, we remove the atomicCAS() and replace it with *if* condition to comparison and swap. It turns out that without using atomic instructions, the overheads increase to 41.9% and more than 16$\times$ for Cuckoo hashing and Quadratic probing, respectively. Thus, using atomic instructions do not degrade performance. Instead, it improves performance.

4) Impact of using or not using locks: We investigate two hash table implementations based on whether locks are used or not. We anticipate that this aspect is more important in GPUs due to high thread counts. Table III shows the overhead results of lock-based and lock-free implementations. The last column shows the number of total thread blocks used. From the table, we can observe that the lock-free version always performs better than the lock-based one: the lock-based version performs $32-37\times$ worse on average for Quad and Cuckoo respectively compared to the baseline. For MRI-GRIDDING and SAD, the overhead difference between lock-based and lock-free implementation is extremely high. This is correlated to the huge number of threads blocks: 65,536 in MRI-GRIDDING and 128,640 in SAD. From this, we learn that lock-free implementation is crucial for LP design and implementation on GPUs.

5) Impact of parallel reduction: In contrast to CPUs, GPUs provide parallel reduction (`shfl_down` instruction) to exchange data directly from register to register between threads inside the same warp. We compare all benchmarks with and without the sfhl_down instruction. To implement a version without parallel reduction, we rely on shared memory and global memory to calculate the checksum. We store data to these memories and calculate checksums sequentially. Table IV shows the performance overheads of the two hash tables (Quad and Cuckoo) with and without parallel reduction. The table shows substantial increase in overheads when shuffle down is not used: from 29.4% to 63.3% (Quad) and from 31.7% to 65.8% (Cuckoo). All bandwidth bound benchmarks e.g. SPMV, SAD, and HISTO (Table I) suffer to a larger degree. The overheads for SPMV go from just 22.1% to 437.6% (Quad) and from 11.78% to 431.18% (Cuckoo). Without parallel reduction, the reduction must go through memory and this increases memory bandwidth pressure.

V. Scalable LP on GPU

From the previous section, it is clear that the hash table is the last performance bottleneck, after using parallel reduction and removing the use of locks. Even though Quad and Cuckoo hash tables perform reasonably well, the hash tables must still deal with collisions.

After rethinking the LP design on GPUs, we observe that because our LP region consists of a thread block, and each thread block has its own unique ID, we can completely avoid collisions in hash table by utilizing thread block IDs. Here, we propose a global array to replace the hash tables. Every thread block produces a single checksum and we use the thread block ID to index this table, and store the checksum in the entry. With this, we remove collisions completely and we can also keep a 100% load factor, reducing the memory overheads. Moreover, this mechanism will eliminate the race condition since each thread block will access different memory address to store the checksum. We refer to this solution as *checksum global array*. The global array scales well, achieves minimum required space, and is both collision and race free.

VI. Directive-Based Programming Support

Due to the fit of GPU programming model to lazy persistency (LP), LP can be integrated into an existing program without much programming complexity. At the heart of this is the use of directives that the programmer can annotate their code. The compiler uses the directives to insert appropriate code to implement LP. Older compilers that do not support these directives simply ignore them. For the rest of the section, we assume that the directives are used in conjunction with CUDA.

The directives that we propose are:

- `#pragma nvm lpcuda_init(checksum_tab_id, nelems, selem)`

TABLE III: Slowdown performance comparison between lock-based and lock-free implementation

Name	quad lock-free	quad lock-based	cuckoo lock-free	cuckoo lock-based	no. of blocks
TMM	1.07×	4.70×	1.07×	4.04×	16384
TPACF	1.01×	1.02×	1.01×	0.02×	512
MRI-GRIDDING	3.19×	6,332×	1.46×	1,868.09×	65536
SPMV	1.22×	23.78×	1.12×	18.85×	1536
SAD	1.51×	4,491.87×	3.33×	9,162.23×	128640
HISTO	1.05×	1.30×	1.28×	1.48×	42
CUTCP	1.08×	32.31×	1.13×	50.73×	128
MRI-Q	1.08×	5.50×	1.06×	4.88×	1024
Geo Mean	1.33×	36.62×	1.35×	31.73×	-

TABLE IV: Performance overheads of Quad with parallel reduction (Quad+shfl) vs. without parallel reduction (Quad+no), as well as Cuckoo with and without parallel reduction.

Name	Quad+shfl	Quad+no	Cuckoo+shfl	Cuckoo+no
TMM	8.1%	15.4%	7.25%	13.65%
TPACF	1.5%	2.6%	1.33%	1.89%
MRI-GRIDDING	218.6%	224.1%	45.67%	50.32%
SPMV	22.1%	437.6%	11.78%	431.18%
SAD	51.23%	86.34%	232.79%	242.13%
HISTO	4.54%	9.70%	27.73%	45.81%
CUTCP	7.96%	9.01%	13.16%	14.78%
MRI-Q	8.01%	9.78%	6.06%	8.03%
Geo Mean	29.4%	63.3%	31.7%	65.8%

- `#pragma nvm lpcuda_checksum(checksum_type, chechsum_tab_id, key1, ...)`

The first directive is for initializing a checksum table before calling a kernel function that contains an LP region. The directive takes three parameters: `checksum_tab_id`, `nelems`, and `selem`. The first specifies the name of checksum table, the second specifies the number of elements in the table, and the third specifies the number of checksums in an element in the table. The directive is called once for each LP region.

The second directive specifies a statement in an LP region that calculates the value to be used for checksum calculation. The directive takes at least three parameters. The first parameter specifies the types of checksums to be used, for example "+" for modular checksum (addition of values) and "^" for parity checksum (XOR of values). The second parameter specifies the checksum table ID that was initialized earlier using the `lpcuda_init` directive. The third parameter specifies a variable that is used as a key for indexing the checksum (hash) table. This directive can take more parameters as key variables after `key1`. The value calculated at the right-hand side of the statement specified by this directive is stored in the checksum table by using `key1`.

Listing 5 and Listing 6 show example codes that utilize the proposed directives at the host (host code) and at the kernel (kernel code), respectively. In Listing 5, the `lpcuda_init` directive is inserted before the kernel function invocation. The directive will be replaced by a runtime function call that initializes a checksum table named `checksumMM` with `grid.x*grid.y` number of elements. The number of checksums is 1 in this example.

Listing 5: Pragma directive sample for the host code.

```
#pragma nvm lpcuda_init(checksumMM, grid.x*grid.y,
    1)
MatrixMulCUDA<<<grid, threads, 0, stream>>>
    (d_C, d_A, d_B, dimsA.x, dimsB.x);
```

In Listing 6, the thread block consisting of the body of the function `MatrixMulCUDA` that calculates matrix C as the multiplication result of matrices A and B is implicitly defined as the LP region. Inside the LP region, the second directive "`lpcuda_checksum`" is placed right before the statement that stores the calculated value in matrix C. Here, the value of "`Csub`" on the right-hand side of the statement is stored in the checksum table by adding it to the current checksum value.

Listing 6: Pragma directive sample for the kernel code.

```
__global__ void MatrixMulCUDA(float *C,
        float *A, float *B, int wA, int wB) {
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    ...
    int c = wB*BLOCK_SIZE*by+BLOCK_SIZE*bx;
#pragma nvm lpcuda_checksum("+", checksumMM, \\
          blockIdx.x, blockIdx.y)
    C[c+wB*ty+tx] = Csub;
```

An important role of `lpcuda_checksum` is to generate a check-and-recovery code that is executed at crash recovery as well as inserting a runtime function call for calculating the checksum. In the check-and-recovery code, a checksum is calculated from the data values fetched from the NVM, and then it is compared against the value fetched from the checksum table. At this time, the elements in the NVM for calculating the checksum and their corresponding value in the checksum table with its key(s) must be specified. The relationship between the location of those elements and the key(s) depends on the source code. Therefore, a compiler has a responsibility of generating a check-and-recovery code from the source code and embedded `lpcuda_checksum` directive.

The compiler exploits the locations, or pointers, of the elements from the left-hand side of the statement specified by the `lpcuda_checksum` directive. From the source kernel code, the compiler exploits a program slice [29] that is used for the pointer calculation, then it generates a comparison code of the calculated checksum and the value in the checksum table with the checksum type and key(s) specified in the directive parameters. Similarly, it also generates the recovery code from the body of the kernel code that is defined as the LP region code.

Listing 7 shows a sample of generated check-and-recovery code for the matrix multiply program. In this sample, the statements to calculate the pointer of the element "`C[c+wB*ty+tx]`" is exploited from the source program, then the element is passed to the `validate()` function as well as the checksum table (`checksumMM`) and the keys (`blockIdx.x` and `blockIdx.y`) to calculate the checksum and compare it with the value in the table. If this check result represents a failure, the `recovery()` function generated from the source kernel code is invoked.

Listing 7: Check-and-recovery code for Matrix Multiply

```
__global__ void crMatrixMulCUDA(float *C,
        float *A, float *B, int wA, int wB)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int c = wB*BLOCK_SIZE*by+BLOCK_SIZE*bx;
    if (!validate(C[c+wB*ty+tx], checksumMM,
                blockIdx.x, blockIdx.y))
        recovery(C, A, B, wA, wB);
}
```

Though the introduced directives are explained with CUDA sample programs, they have no CUDA related specifications. Therefore, they can be also applicable to OpenCL programs.

VII. Evaluation Results

1) Performance of scalable LP (the global array method): The best performance that we obtain is when we use a combination of: parallel reduction with `shfl_down` instruction and global array (denoted as *array+shuffle*). To make false negative rates as low as possible, we recommend the simultaneous use of both modular and parity checksums. Table V shows the execution time overheads of this scheme compared to the original version of the benchmarks, which supports no crash recovery. The geometric mean is only 2.1% overhead, with individual overheads ranging from $0.6 - 6.2\%$. Compared to LP on CPUs with reported 1% slowdown [9], the slowdown of our LP on GPU is nearly as low, even when considering we use thousands of threads on GPUs vs. only 16 threads on CPUs. While geometric mean of the space overhead introduced by this scheme the is only 1.63%.

TABLE V: Execution time overheads for array+shuffle over the original benchmarks and its space overhead

Benchmark	array+shuffle	Space overhead
TMM	6.2%	0.2%
TPACF	1.0%	0.02%
MRI-GRIDDING	2.5%	0.82%
SPMV	1.6%	0.02%
SAD	0.6%	12.27%
HISTO	0.6%	0.01%
CUTCP	2.1%	0.02%
MRI-Q	2.7%	0.25%
Geo Mean	2.1%	1.63%

2) Impact of multiple checksum: We also investigate using multiple checksum calculation in the same block, combining together the calculation for modular and parity checksums. For the latter, the overheads include converting floating point data to ordered integer. We found that simultaneously using both checksums only adds minor additional overheads compared to using just one checksum. For example, for TMM with quadratic probing, parity and modular checksums individually cause 7.6% and 7.7% overheads, respectively. When both are calculated simultaneously, the overhead increases to 8.1%, due to additional data exchange through direct register to register communication. Hence, combining modular and parity checksums with lower false negative rates is worth the small bump in performance overheads. However, we hope that GPU architects will consider adding support for other parallel reduction operators beyond just addition and XOR.

3) Write amplification results: We measure the number of writes to the main memory of our final LP design with global array, lock-free, and two checksum methods. For this, we use GPGPU-sim to model the Volta Titan V architecture. To simulate NVM, we lowered the memory bandwidth to 326.4GB/s, and set the NVM read and write latency to 160ns and 480ns, respectively. We then run the model on SPMV, MM, and SAD. Applications run until completion to measure the overhead of checksum calculation and insertion into the global array at the end of kernel execution. The number of writes increase by between 0.5% (SPMV) to 2.2% (MM). Unlike EP, LP relies on natural cache evictions without any flushing, hence the increase in writes is due to stores of checksums.

4) Evaluation on real world application: We also evaluate a real application MEGA-KV [12], an in-memory key-value store, on our final LP design. The search, delete, and insert operations incur performance overheads of 3.4%, 5.2%, and 2.1%, respectively, indicating our LP's low performance overheads.

VIII. Conclusion

In this work, we showed how lazy persistency (LP) can be mapped to GPUs and characterized its performance over their design space (LP region selection, checksum type, reduction method, use of locking, and hash table design). We identified the performance bottleneck sources, and based on those, we proposed a hash table-less design. This is the first work that shows we can provide persistency at nearly negligible performance overheads on GPUs (2.1%) and negligible write amplification. We discussed that GPU programming lends naturally to the LP model, allowing programmers to rely on a small number of directives to integrate LP into their applications.

IX. Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work is funded in part by NSF grants 1717550, 1908406, and 1908079, and by Waseda University, and by UCF.

References

[1] Y. T. Tesfaye, E. Zemene, A. Prati, M. Pelillo, and M. Shah, "Multi-target tracking in multiple non-overlapping cameras using fast-constrained dominant sets," *Int. J. Comput. Vis.*, vol. 127, no. 9, pp. 1303–1320, 2019. [Online]. Available: https://doi.org/10.1007/s11263-019-01180-6

[2] I. Cutress and B. Tallis, "Intel intel launches optane dimms up to 512gb: Apache pass is here!" https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here, Mar. 2018.

[3] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka, "Dragon: Breaking gpu memory capacity limits with direct nvm access," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 414–426.

[4] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *International symposium on Computer architecuture*, Jun. 2014.

[5] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *ACM Symposium on Operating Systems Principles*, Oct. 2009.

[6] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *International Symposium on Microarchitecture*, Dec. 2015.

[7] Intel, "Persistent memory programming," http://pmem.io, Aug. 2016.

[8] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-Ordering Consistency for persistent memory," in *Computer Design, 2014 32nd IEEE International Conference on (ICCD'14)*, October 2014.

[9] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 439–451. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00044

[10] Z. Lin, M. Alshboul, Y. Solihin, and H. Zhou, "Exploring memory persistency models for gpus," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 311–323.

[11] S. Chen, F. Zhang, L. Liu, and L. Peng, "Efficient gpu nvram persistence with helper warps," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[12] 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, p. 1226–1237, Jul. 2015. [Online]. Available: https://doi.org/10.14778/2809974.2809984

[13] 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: Association for Computing Machinery, 2019, p. 145–157. [Online]. Available: https://doi.org/10.1145/3293883.3295706

[14] K. DB, "Kinetica high performance analytics database." [Online]. Available: https://www.kinetica.com/

[15] H. Elnawawy, M. A. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*, 2017, pp. 318–329. [Online]. Available: https://doi.org/10.1109/PACT.2017.58

[16] NVIDIA, "Nvidia tesla v100 gpu architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, Aug. 2017.

[17] ——, "Nvidia's next generation cudatm compute architecture: Keplertm gk110/210," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf, Aug. 2014.

[18] Nick, "Tiled matrix multiplication in cuda," 2019. [Online]. Available: https://github.com/CoffeeBeforeArch/cuda_programming/tree/master/matrixMul/tiled

[19] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[20] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, 2010, pp. 86–97. [Online]. Available: https://doi.org/10.1145/1806596.1806606

[21] J. J. K. Park, Y. Park, and S. A. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, Eds. ACM, 2015, pp. 593–606. [Online]. Available: https://doi.org/10.1145/2694344.2694346

[22] I. Tanasic, I. Gelado, J. Cabezas, A. Ramírez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, 2014, pp. 193–204. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853208

[23] Z. Lin, L. Nyland, and H. Zhou, "Enabling efficient preemption for SIMT architectures with lightweight context switching," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, 2016, pp. 898–908. [Online]. Available: https://doi.org/10.1109/SC.2016.76

[24] P. Deutsch and J.-L.Gailly, "Zlib compressed data format specification version 3.3," *Network Working Group Request for Comments (RFC) 1950*, 1996.

[25] T. Maxino, "Revisiting fletcher and adler checksums," *DSN 2006 Student Forum*, 2006.

[26] R. Pagh, "On the cell probe complexity of membership and perfect hashing," in *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, 2001.

[27] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, p. 122–144, May 2004. [Online]. Available: https://doi.org/10.1016/j.jalgor.2003.12.002

[28] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1093–1108. [Online]. Available: https://doi.org/10.1145/3373376.3378493

[29] M. Weiser, “Program slicing,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.