Only Buffer When You Need To: Reducing On-chip GPU Traffic with Reconfigurable Local Atomic Buffers

Preyesh Dalmia
University of Wisconsin-Madison
Email: pdalmia@wisc.edu

Rohan Mahapatra
University of California, San Diego
Email: rohan@ucsd.edu

Matthew D. Sinclair
University of Wisconsin-Madison, AMD Research
Email: sinclair@cs.wisc.edu

Abstract

In recent years, due to their wide availability and ease of programming, GPUs have emerged as the accelerator of choice for a wide variety of applications including graph analytics and machine learning training. These applications use atomics to update shared global variables. However, since GPUs do not efficiently support atomics, this limits scalability. We propose to use hardware-software co-design to address this bottleneck and improve scalability. At the software level, we leverage recently proposed extensions to the GPU memory consistency model to identify atomic updates where the ordering can be relaxed. For example, in these algorithms the updates are commutative. At the hardware level, we propose a buffering mechanism that extends the reconfigurable local SRAM per SM. By buffering partial updates of these atomics locally, our design increases reuse, reduces atomic serialization cost, and minimizes overhead. Thus, our mechanism alleviates the impact of global atomic updates and improves performance by 28%, energy by 19%, and network traffic by 19% on average and outperforms hLRC and PHI.

***Keywords*-GPGPU; Relaxed Atomics; Machine Learning; Graph Analytics; Buffering**

I. Introduction

Traditionally, GPUs were used for streaming, data parallel applications with limited data reuse and sharing, and coarse-grained synchronization. Accordingly, GPUs utilize a simple, software-driven coherence protocol [4], [40], [57], [75], [80], [84], [93], [95] and push complexity into the memory consistency model [31], [42], [65], [94], [103]. As a result, infrequent synchronization, which is often implemented using atomics, is expensive and must be performed at a shared ordering point. The memory accesses' scope impacts the ordering point: threads in the same thread block (TB) share a *local* scope and can synchronize locally (e.g., at the L1 cache) [31], [42], [43], [60]. However threads in different TBs must use *device*-scoped atomics that are performed at the shared last level cache (LLC, usually the L2). Thus global, inter-TB synchronization is expensive, but reasonable for throughput-oriented graphics and general-purpose GPU (GPGPU) applications [8], [12], [13], [20], [35], [97], which use coarse-grained synchronization.

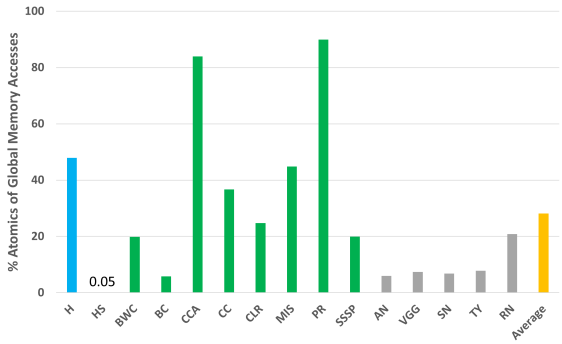

Figure 1. Percent of device-scope commutative atomics for histograms (blue), graph analytics (green), and ML training (gray) on a Titan V GPU.

However, as GPUs become increasingly general-purpose, they are also used for applications that require efficient support for more access patterns. For example, modern GPUs are widely used for graph analytics and machine learning (ML) training, both of which utilize device-scope atomics to update shared global variables. Although many memory accesses in ML training are regular data loads and stores, ML training algorithms that use techniques such as stochastic gradient descent (SGD) also use atomics to update the shared weights [110]. Similarly, graph analytics algorithms have irregular, input dependent parallelism [45], [64], [90], [102] that often use atomics to perform edge-propagated updates (discussed further in Section II-B).

Recent work has shown that these atomic updates are a large source of inefficiency in ML training [59] and graph analytics [1], [102]. To validate these claims, we profiled histograms, graph analytics, and ML training applications (described further in Section IV). Figure 1 shows that device-scoped atomics make up a significant fraction (29%) of their global memory accesses. However, in these programs all of these atomics do not imply ordering on surrounding accesses because thy are commutatively updating shared variables. Thus, they can safely use lower overhead relaxed atomics (discussed further in Section II-A). Nevertheless, relaxed atomics are still expensive. Moreover, although applications like BC have fewer atomics, these accesses are often a bottleneck because they are serialized [26] – multiple threads from the same TB concurrently, atomically update the same address, which serializes the accesses (discussed further in Section III-A).

Given the importance of ML, prior work designed ML architectures and libraries, especially for ML inference [3], [15], [21], [29], [37], [38], [44], [46], [77], [82], [91], [92], [106]. However, these solutions primarily focus on inference whereas we focus on training. Although more customized solutions also exist for graph analytics [36], [105] and ML training, GPUs are still widely used due to their availability and ease of programming. Thus, subsequent work further optimized ML training on GPUs by reducing the width and/or number of memory accesses [44], [101], [113], utilizing compression [86], or rewriting code to frequently perform memory accesses in the register file or shared memory [24], [27], [51], [71], [111] (discussed further in Section VII). However, Figure 1 shows that many of the remaining memory requests are atomics that update shared locations. Accordingly, these atomics are a significant overhead.

To overcome this inefficiency we propose a hardware-software co-design approach that reduces atomic latency, data movement, and energy. At the software level, we exploit algorithmic properties; recent work identified that graph analytics algorithms often use *commutative* relaxed atomics – i.e., although the accesses must be performed atomically to ensure correctness, the order of the atomics does not matter since the program does not view the updated values until all updates have completed [2], [10], [94], [107]. We find that this property also holds for ML training weight updates: the updated weights are not used until subsequent layers. Moreover, other, non-commutative relaxed atomics with similar properties can benefit from our approach.

At the hardware level, we exploit the commutativity of these atomic updates to buffer partial device-scope atomic updates locally at each SM in a small *local atomic buffer* (*LAB*), before sending coalesced updates to the shared L2 later. We propose to extend the partitioning of the unified local memory [32] to include the LAB. This enables LAB to coalesce commutative atomic updates across all TBs on an SM, and improves performance, energy, and network traffic by reducing both the latency for atomic accesses and the number of commutative atomic accesses sent to the L2.

Prior work also exploits commutativity to improve performance and reduce energy [28]. In particular, DeNovo and hLRC cache device-scoped atomics in GPU L1 caches [4], [93], [94]. However, they require significant coherence protocol or consistency model changes. Similarly, in multi-core CPUs AIM, CCache, Coup, and PHI exploit commutativity by adding an additional coherence state or in-cache buffers [2], [9], [68], [107]. Although these approaches exploit similar insights, since GPUs use lightweight, software-driven coherence protocols [57], [93], [95] and have high L1 cache contention, our results show these solutions are not ideal fits for GPUs. Instead, LAB shows that using the existing reconfigurable SRAM to separately buffer atomics improves performance and energy efficiency relative to PHI and hLRC (Section V), requires minor software changes (annotating commutative atomic accesses), and does not require coherence protocol or consistency model changes. Thus, LAB provides similar or better benefits than hLRC and PHI, without their downsides. We further discuss how LAB compares to these and other related works in Section VII.

Overall, across 15 graph analytics and ML training workloads, a small, reconfigurable, per SM LAB improves performance by 28%, reduces energy by 19%, and reduces on-chip traffic by 19% on average, respectively. Moreover, LAB improves on state-of-the-art solutions like hLRC and PHI. Additionally, our results show that reconfiguring 8 KB or less of the SM's local SRAM into a LAB is often sufficient. Finally, LAB does not affect applications that do not use commutative atomics, unlike other state-of-the-art solutions.

II. Background

A. GPU Coherence and Consistency

Since GPUs traditionally run massively data-parallel, streaming applications with coarse-grained synchronization and little to no data reuse, they use a simple, software-driven Valid-Invalid (VI)-style coherence protocol [57], [84], [93], [94], without ownership requests, downgrade requests, writer-initiated invalidations, state bits, snoopy buses or directories [95]. Thus, since synchronization is infrequent, GPU coherence protocols invalidate the cache on acquires (the start of the kernel or an acquire synchronization operation) so that subsequent reads do not read stale values. Typically, GPU L1 caches either use a write through or write no-allocate approach for global memory writes [50], [95]. To improve performance, these writes may be buffered and coalesced until the next store release [40]. When a store release occurs (the end of the kernel or a release synchronization operation), all prior stores must complete and the data is written to the next level of the memory hierarchy, which is usually shared between SMs.

Thus, fine-grained synchronization that uses load acquires or store releases provides ordering between data and atomic requests from TBs on multiple SMs. For example, sequentially consistent (SC) atomics orders both data and atomic accesses. However, since GPU coherence protocols do not obtain ownership for written data or atomics, GPUs perform all global atomics at the first shared level of the memory hierarchy (usually the L2). When fine-grained synchronization is required, it is usually implemented with atomics, which may incur expensive load acquire and store release overheads to ensure a consistent view of memory for variables being accessed by multiple TBs. However, sometimes *relaxed* atomics can be used. Relaxed atomics act neither as an acquire nor a release, and since they imply no ordering on other memory accesses, they can be reordered with other data and atomic accesses. As a result, relaxed atomics are cheaper. However, since L2 accesses in modern GPUs take over 100 cycles (Section IV), even relaxed, device-scope atomics are expensive. In contrast, CPUs often obtain

```
1  if (tid <= (N - 1)) {
2      loc = arr[tid];
3      //atomicAdd(&hist[loc], 1, mem_order_comm);
4      atomicAdd(&hist[loc], 1); } // commutative
```

Listing 1. Pseudo-code of GPU histogram kernel [67].

```
1  end = ((tid+1) < numNodes ? row[tid+1] : numEdges);
2  for (edge = row[tid]; edge < end; ++edge) {
3      nodeID = col[edge];
4      inc = pR1[tid]/(float)(end-start);
5      //atomicAdd(&pR2[nodeID], inc, mem_order_comm);
6      atomicAdd(&pR2[nodeID], inc); } // commutative
```

Listing 2. Pseudo-code from key PageRank kernel [14].

ownership for written data and atomics (e.g., in MOESI-style coherence), which makes synchronization points cheap; however, these protocols are a poor fit for GPUs [40], [95].

Instead GPU consistency models utilize scopes to reduce synchronization overhead, as part of sequentially consistent for heterogeneous-race-free (SC-for-HRF) based consistency models [31], [42], [43], [60], [65]. Programs which properly identify both memory accesses as data or synchronization and each synchronization accesses' scope are guaranteed to be SC-for-HRF. Although SC-for-HRF has multiple scopes, we focus on the two most widely used variants: local and device. Locally scoped atomics are only guaranteed to be visible to other threads in the same TB, while device-scoped atomics are visible to all threads across the GPU. Thus, locally scoped synchronization is significantly cheaper since it does not invalidate all valid L1 data on acquires nor writes through dirty data on releases.

B. Applications

We first examine a histogram microbenchmark, then extend the ideas to the larger benchmarks.

Histograms: Histograms are widely used in image processing to find the frequency of pixel values [67], [78]. However, since different threads may encounter the same pixel value, either device-scope atomics must be used, or each thread must store its per-pixel value counts separately, then sum up the per-thread counts in a separate kernel. We focus on the former, as it requires less memory and fewer kernels. Listing 1 shows a code snippet of a histogram performing device-scoped atomics. Its atomic increments are commutative – the final result will be the same regardless of the order the atomics execute. Furthermore, since the intermediate histogram results are not read, they do not affect the program's control flow. Thus it is safe to use relaxed atomics (i.e., `memory_order_relaxed` [10]) while still ensuring the final results are SC [94]. However, more optimized GPU software implementations are possible. In particular, per-TB sub-histograms that perform atomics locally in shared memory, before sending per-TB partial updates with a single device-scoped atomic per histogram bin, can significantly reduce overhead [78] if the array fits in shared memory.

Graph Analytics: Graph analytics algorithms perform edge-propagated updates, where a source vertex can either pull changes from its neighbor vertices or push changes to its neighbors [14], [53]. We focus on the push variants since the pull variants, similar to Histogram, tradeoff additional kernels and memory for avoiding atomics. Additionally, recent work has shown neither push nor pull algorithms are always best on GPUs [88], [96]. Nasre et al. also reduced atomics in irregular programs by using global barriers or other software-based approaches such as idempotence [69]. However, global barriers limit the number of TBs (most of our applications use too many TBs to use this approach), and their other software-based approaches cannot be applied to our applications. Others use vertex duplication and out-of-core execution [58] to optimize pull-based graphs that cannot fit in memory [66]. However, we focus on push-based graphs that fit in memory.

We examine seven graph analytics algorithms: Betweenness Centrality (BC), Connected Components Analysis (CCA), Cuda Cuts (CC), Graph Coloring (CLR), Maximal Independent Set (MIS), PageRank (PR), and Single-Source Shortest Path (SSSP) [11], [14], [88], [100]. Since multiple vertices may be connected to each other, device-scoped atomics must be used for updates to avoid data races (e.g., PR in Listing 2). Like Histogram, the atomics are commutative: their order does not affect the final result and the intermediate results do not affect control flow. However, unlike Histogram, it is difficult to apply software optimizations because each TB accesses input dependent locations. Thus, to use shared memory, a TB must allocate enough shared memory space for all of the graph's vertices (discussed further in Section VI).

ML Training: To study the impact of atomics on ML training algorithms, we examined several popular Convolutional Neural Networks (CNNs), as discussed in Section IV. In deep neural network (DNN) training, ML algorithms improve accuracy by propagating the loss function gradient backwards through the network and updating the weights, also known as backpropagation [87]. In GPUs, weight gradients and input gradients are calculated across multiple SMs. Thus, device-scope atomics are needed to add these partial updates together and ensure subsequent layers see the updated weights. Calculating these gradient updates is a significant part of the total training execution time [59]. However, these updates are commutative and are only used in subsequent kernels.

Like Histogram, software optimizations could perform many of these atomics locally before sending a few device-scoped atomic updates. However, since weight matrices are large for modern DNNs, storing the weights in shared memory may hurt performance by limiting SM utilization (Section VI). Thus, cuDNN and other libraries often use device-scoped atomics for weight updates instead.

III. Proposed Design

A. LAB Hardware Support

Figure 2a shows our proposed addition of the LAB to the GPU memory hierarchy. Physically, the LAB is similar to the L1 data cache; as shown in Figure 2b we exploit the reconfigurability of the unified local memory [32] to

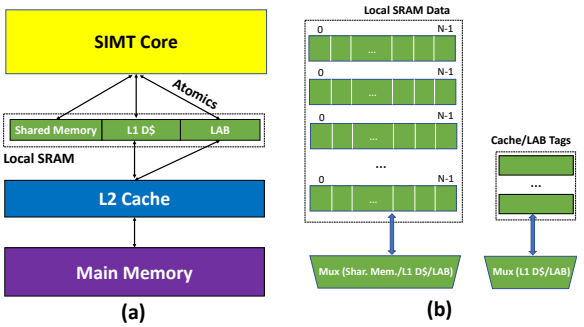

Figure 2. Proposed design (a) including LAB (in green) and (b) local SRAM.

partition it into L1 data cache, shared memory, and LAB. This requires an additional 2:1 mux for the tag array to avoid duplication and determine if a tag belongs to the cache or LAB; the data array already has a 4:1 mux which previously had one unused input [32].[1] Like some GPU L1 caches, the LAB has 128 byte lines, broken into four 32-byte sectors. Since LAB is intended to be small, we make it associative.[2] LAB also utilizes a portion of the local SRAM's data and tag arrays (via the muxes in Figure 2b). The data array holds partial values for a given address, while the metadata holds address, replacement, and atomic function information. Next, we discuss LAB's operation:

Steady-State Behavior: Since atomics entering the LAB are all commutatively updating shared global variables, we use an allocate on fill write miss policy.

Evictions: When the LAB is full, we use an LRU replacement policy to determine which entry to evict. However, evictions can be done off the critical path, since the commutative atomics do not imply any ordering on other memory accesses. Accordingly, as soon as the message is sent to the L2, we reuse the entry. When the evicted atomic request reaches the L2, it updates the appropriate addresses' value.[3]

Coalescing: Like L1 cache and shared memory accesses, the GPU coalescer coalesces accesses before sending them to the LAB. Thus, threads within the same warp are coalesced and LAB can use the same number of read and write ports as the L1 cache.

Handling Uncoalesced Accesses: Some programs have divergent, uncoalesced memory accesses where every thread may access a unique cache line. Although we only observed up to 38% divergence (12 unique cache lines/warp), LAB still supports these accesses. On an uncoalesced access, the GPU (and LAB) treats each requested line as a unique request. Consequently, LAB handles them as they arrive. If the number of requests exceed the LAB's size, then requests evict entries from the same uncoalesced access.

Behavior at Ordering Points: Although atomics using the LAB are relaxed and thus do not imply ordering on other memory requests, at ordering points we flush all LAB entries to the L2. Thus, at all kernel boundaries or at software enforced ordering points (e.g., CUDA's `threadfence` or barriers, mutexes, and semaphores) all LAB entries are evicted. None of our benchmarks had software ordering points, so flushes only happened at kernel boundaries.

Atomics with Ordering Requirements: Since LAB stores partial atomic updates, it cannot be used for atomics with ordering requirements (e.g., SC atomics) without causing significant delays. Thus, our software requirements (Section III-B) ensure that only commutative atomics use the LAB. Non-commutative atomics must be performed at the appropriate level defined by their scope. We discuss LAB's implications on GPU coherence and consistency further in Section III-C.

Serialization of Atomics: Atomic serialization occurs due to two primary factors: atomic collisions and centralized resources. Intra-warp atomic collisions occur when multiple threads in a warp attempt to update the same memory location concurrently, while inter-warp and inter-TB atomic collisions occur when multiple threads from different warps (or different TBs) attempt to update the same address concurrently. When this happens, one request must be issued before the other. Atomic serialization can also occur at a centralized destination such as the L2 cache. Since device-scoped atomics are performed at the L2 (which is ≈6X more expensive than GPU L1 accesses, Section IV), this serialization can be very expensive, especially when combined with L2 queuing delay and potentially backpressure from interconnect stalls. Since LAB performs all commutative, device-scope atomics locally, LAB reduces the serialization penalty due to atomic collisions. Moreover, LAB's decentralized updates (one LAB per SM) also reduces atomic serialization from centralized resources, since only the combined requests are sent to the L2.

Identifying Atomic Function: CUDA supports multiple types of atomics, some of which are not commutative with each other. Thus, the LAB must track what atomic function is being performed for each line. Since CUDA has fewer than 16 atomic functions [72], we use 4 bits per LAB line to identify the atomic operation. If a LAB hit occurs but the atomic function does not match, we flush the entry.

Benefits Over Software Solutions: Since the atomic operations are commutative and are not read before all updates complete, programmers could either use per-TB shared memory to accumulate updates locally or use L1 data cache with private variables. In this situation, a per-TB global atomic performs the global memory final update. However, for most of our applications this would require large, sparsely

[1]It is also possible to implement LAB inside the L1 cache using techniques such as way partitioning [18]. However, way partitioning also requires additional muxes and may increase conflict misses. Thus, we focus on partitioning the SRAM, which also decouples data and atomic accesses.

[2]We examined various LAB associativities including 8-way and fully associative; the performance difference was small. Thus, similar to modern GPU L1 caches, we assume a 8-way associative LAB to meet timing (although Figure 2b does not show associativity for simplicity).

[3]Like prior work [4], [57], [93], we assume the GPU has an ALU co-located with the L1 cache. If this is not the case, then Figure 2 and the overall area (Section V-G) would need to include this.

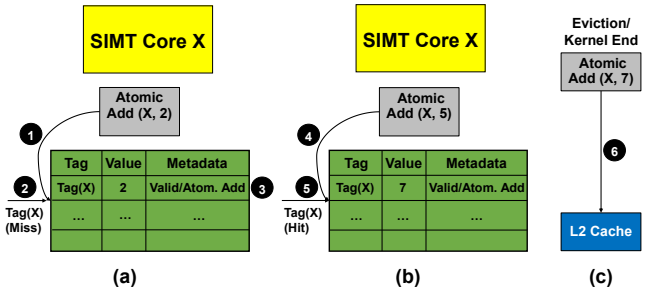

Figure 3. Example LAB access sequence with reuse.

accessed shared memory allocations that limit the number of TBs per SM (discussed further in Section VI). In contrast, LAB only holds frequently accessed atomic addresses and their values, without requiring large allocations.

Overall, LAB effectively enables per-SM intra-warp, inter-warp, and inter-TB reuse in the same kernel invocation. For example, when threads in Listings 1 and 2 on the same SM (inter- or intra-TB) access the same address atomically, they can be reused in the LAB. Moreover, when atomic reuse is minimal, LAB's decentralized design still reduces the serialization penalty (**Serialization of Atomics**). Although the LAB may increase the burstiness of the atomic traffic in the worst case, we have not observed significant increases in queuing delay due to LAB's coalescing benefits and because SMs usually flush at different times (as shown in Section V). Moreover, the atomics sent to the L2 by the LAB will be performed off the critical path, except at kernel boundaries where the LAB is flushed in parallel with the caches, since instructions complete upon reaching the LAB. This lessens observed impact of burstiness (Figure 9).

Figure 3 shows two LAB operations from the same SM:

1) An *atomicAdd(X, 2)* (Figure 3a) is issued.
2) We use the tag to index into the LAB and check if it hits. Since this is the first access to *X* on this SM, it misses.
3) Next, we allocate an LAB entry and update the value of the entry to the value of the first update (here, 2).
4) Another atomic is issued to the same address, *atomicAdd(X, 5)* (Figure 3b).
5) We use the tag to index into the LAB; this access hits and updates the partial value at the LAB index (here, 7).
6) Later, when the LAB evicts the entry (Figure 3c), a partial atomic update is sent to the L2 (*atomicAdd(X, 7)*). Since the L2 already supports atomics, LAB simply sends its an atomic to update the global value.

B. Software Support: Distinguishing Atomic Operations

LAB relies on identifying which atomic accesses can be buffered locally (e.g., commutative atomics). To identify which accesses are commutative atomics, we leverage recent work that proposed additional memory orderings: SC-for-Data Race Free Relaxed (or SC-for-DRFrlx) introduced a new memory ordering, `memory_order_comm`, to identify commutative accesses [94]. However, since we do not need the additional complexity for other, non-commutative relaxed atomics that SC-for-DRFrlx proposes, we use a SC-for-HRF consistency model with the additional commutative memory ordering. Programmers or compilers can instrument software to use this new memory ordering to indicate commutative atomics to the hardware, analogous to how C, C++, HSA, and OpenCL specify other memory orderings [10], [43], [60]. For example, Listing 1 and 2, show how the programmer would label the `atomicAdd`'s as commutative atomics. Non-commutative atomics bypass LAB.

C. Impact on Consistency and Coherence

The key ideas that allow the LAB to batch atomic updates without affecting consistency and coherence guarantees are:
Commutativity: The order of commutative Read-Modify-Write updates to a shared variable can be arbitrary without affecting correctness. Since these updates race, they must use atomics to conform to the SC-for-HRF consistency model (Section II-A). However, since reordering updates still produces correct results, programs often use relaxed atomics for the updates [94]. Thus, caching relaxed commutative atomics in the LAB should not affect the final results. By perturbing the order of atomics, LAB may cause small rounding errors for floating point commutative atomics, but this is already a problem on real GPUs, where the atomic order is not deterministic [16], [17], [22]. Although we only observed minor impacts on the final results, if complete determinism is desired we could adopt DAB [16] or Reproducible FP [22] at the cost of additional area.
Interaction with Data Accesses: Like C++ [10], HSA [43], and OpenCL [52], by default we assume that atomically updated shared variables are always accessed atomically. Thus, if a program uses properly labeled and synchronized commutative atomics, there will never be data accesses to the same variable and SC results are guaranteed (which also ensures that the program does not view the updated values until all updates have completed) [94]. As a result, data accesses and non-commutative atomics do not need to check the LAB, since the commutative atomics do not order other accesses and all accesses to a commutative address are atomic. However, although CUDA is moving towards similar requirements [73], currently it allows a variable to be accessed by both atomic and data accesses in the same kernel. If this happens for accesses using `memory_order_comm`, a commutative race may occur [94] and, like other DRF-based consistency models, threads may access a stale value and SC results are not guaranteed (e.g., since data accesses do not check the LAB). However, if the data accesses occur in separate kernels, since the LAB is flushed at the end of each kernel, the data accesses will see any previously buffered LAB updates.
Coherence: As discussed in Section II-A, GPU coherence relies on data-race-freedom and software invalidations to

GPU Feature	Configuration (Size, Access Latency)
SMs	80
# Registers / SM	64 KB
LI Instruction Cache / SM	128 KB
LI Data Cache / SM	32 KB (max 128 KB), 28 cycles
L2 Cache	4.6 MB, 148 cycles
MSHR	256 (L1) and 192 (L2) Entries
Shared Memory Size / SM	96 KB (max 128 KB), 19 cycles
Memory	16 GB HBM2, 248 cycles

Table I
SIMULATED BASELINE GPU PARAMETERS

Operation	Energy (pJ)
Non-Memory Operation	3.7
L1D (32 KB) Read/Write	1.4097, 1.7044
L1I (132 KB) Read/Write	5.6387, 6.8177
L2 (4.6 MB) Read/Write	193.59, 234.0675
LAB (Size 8) Read/Write	0.0881, 0.1065
LAB (Size 16) Read/Write	0.1762, 0.2131
LAB (Size 64) Read/Write	0.3524, 0.4261
LAB (Size 128) Read/Write	0.7048, 0.8522
LAB (Size 256) Read/Write	1.4097, 1.7044
LAB (Size Inf) Read/Write	45.1097, 54.5417
NOC	254
Main Memory	501

Table II
PER-ACCESS ENERGIES USED [19], [37], [76].

ensure that there is no stale data in the local caches. Adding the LAB does not impact the GPU coherence protocol, since it only aggregates updates for commutative, atomically accessed global addresses. Accordingly, LAB does not require any changes to the existing coherence protocol and additional fences are not required because the commutative updates do not need to be ordered with one another.

IV. METHODOLOGY

A. Simulation Environment & Parameters

To evaluate LAB's impact, we added LAB to GPGPU-Sim v4.0 [8], [50], [62], [63], [81], which has been shown in previous work to provide high accuracy for modern NVIDIA GPUs, including when running ML workloads [50], [62], [63]. Table I summarizes the key system parameters, which is based on a NVIDIA Titan V GPU [74]. Additionally, we assume support for performing atomics at the LAB. We use CUDA 8 and cuDNN v7.1.3 for the ML training benchmarks, because these are the latest versions of CUDA and cuDNN that embed the PTX in the libraries – which is necessary to run cuDNN in GPGPU-Sim [63]. For all other benchmarks (Table III), we use CUDA 11.2. Although GPGPU-Sim has an integrated energy model [61], it has not been validated for post-Fermi architectures and is not representative for modern GPUs.[4] Thus, we use a per-access energy model (Table II) based on recent work [19], [37], [76]. To label commutative atomics (Section III-B), like prior work [94] we use software flags to find and simulate these accesses.

[4]Accel-Wattch now provides a validated GPU energy model for modern GPUs, but was not available until after this work was submitted [47].

B. Configurations

Since the LAB is physically located in the unified local SRAM, configuring part of the SRAM to be LAB reduces the size of the L1 data cache or shared memory. Hence, we examine varying the size of the L1 data cache and shared memory. Overall, we use the following configurations:

Baseline: The baseline GPU configuration without an LAB, with a 32 KB L1 data cache and 96 KB shared memory, and which performs all device-scoped atomics at the shared L2.

Cache-8KB: Models the *Baseline* configuration with 8 KB less cache, which is representative of a 64-entry LAB.

Cache+8KB: *Baseline* configuration with 8 KB more cache instead of using LAB.

*Cache*2*: Like *Cache+8KB*, except doubles the cache size.

hLRC: hLRC [4] obtains ownership for atomics, enabling it to cache them locally. Since hLRC has not been publicly released, we implemented and validated it in GPGPU-Sim.

PHI: PHI [68] buffers atomics in write allocate L1 caches (fetch on write); we implemented and extended PHI for GPUs (only cache lines with commutative atomics are buffered updates). We also extended PHI to use a lazy fetch on read scheme, which has a 6% difference but did not affect the takeaways. Although PHI was designed for MESI-like CPU coherence, we optimistically ignore invalidation and downgrade overheads – otherwise PHI is worse.

LAB i: We vary LAB's size to examine its sensitivity: LAB *i* represents *i* LAB entries per SM: 8, 16, 32, 64, 256, and Infinite. Each statically reconfigures the cache, shared memory, and LAB proportions based on LAB size before kernel launch, similar to CUDA's existing cache/shared memory flag. Although some configurations require significant SRAM, we include them to examine larger LAB performance. For context, a 64-entry LAB uses approximately 8 KB of local SRAM. For all LABs except infinite, we take space from the cache since the applications were less sensitive to cache size and changing shared memory size affected utilization.

We also increased shared memory size per SM, but it had no effect (discussed further in Section VI). Moreover, although weights currently do not use shared memory, cuDNN uses shared memory for other arrays (from inspecting disassembled binaries [99]). However, since TBs per SM is limited by register file size [79], increasing shared memory per SM did not increase the TBs per SM. Since cuDNN is closed source, we tried modifying cuTLASS [48]. However, cuTLASS lacked the corresponding kernels. Finally, to isolate LAB's serialization and coalescing benefits, we implemented a LAB variant that performs atomics locally but does not store data in the LAB, preventing reuse and separating the coalescing and serialization benefits. Since it is difficult to isolate serialization from reducing backpressure and interconnect stalls, this provides a minimum bound on LAB's serialization reductions. We also isolated the burstiness overheads by turning off the end-of-kernel LAB flushes and measuring its impact.

Benchmark	Input
Microbenchmarks	
Histogram [67] (H)	256K (30720x17280 pixels)
Histogram_Shared [67] (HS)	256K (30720x17280 pixels)
Backward Conv [25] (BWC)	NCHW = 128,3,256,256
Graph Analytics ('_' denotes different utilization levels)	
BC[_100, _75, _50, _25] [14]	CT, NH, VT, AK [23]
CCA[_100, _75, _50, _25] [88]	amazon, olesnik0, wing, emailEnron [88]
CC [88], [100]	flower.txt [88]
CLR [88], MIS [88], PR [14], SSSP [88] [_100, _75, _50, _25]	or2010, nd2010, nv2010, nh2010 [7]
ML Training	
AlexNet [25] (AN)	NCHW = 16,3,227,227
VGG-19 [25] (VGG)	NCHW = 16,3,112,112
SqueezeNet [25] (SN)	NCHW = 16,3,224,224
Tiny YOLO [83] (TY)	NCHW = 16,3,416,416
ResNet [39] (RN)	NCHW = 16,3,256,256
GPGPU	
Backprop [12]	64K
B+Tree [12]	mil.txt
BFS [12]	graph1MW_6.txt
DWT2D [12]	1Kx1K
gaussian [12]	1Kx1K
Heartwall [12]	test.avi
Hotspot [12]	512x512
huffman [12]	1024
HybridSort [12]	2^{18}
KMeans [12]	kdd_cup
LavaMD [12]	boxes1D
Leukocyte [12]	testfile.avi
LUD [12]	512
MummerGPU [12]	NC_003997
Myocyte [12]	100
NN [12]	lat 30, long 90
NW [12]	8Kx8K
Pathfinder [12]	1Mx100x20
ParticleFilter [12]	128x128, 4K particles
SRAD [12]	2Kx2K
Streamcluster [12]	8K

Table III
BENCHMARKS AND INPUTS.

C. Benchmarks

Table III summarizes the workloads we use. To study performance for GPGPU applications, we analyze Rodinia [12], [13] with inputs sized to fully utilize the GPU.[5] We analyze two histogram [67] variants each with 256 bins: an ideal use case for LAB where most accesses are device-scoped atomics, similar to Listing 1, and another that uses shared memory to bin updates locally before sending a single atomic update per bin to global memory [94]. We also use popular graph analytics and ML training workloads: BC, CCA, CC, CLR, MIS, SSSP, PR, AlexNet, VGG-19, SqueezeNet, Tiny YOLO, and ResNet. We selected these benchmarks because they cover a wide variety of use cases, and have been shown to be high performance in prior work [88], [96]. As discussed in Section VI, the larger benchmarks are unable use similar software optimizations to the histograms. Since the graph analytics algorithms are input dependent, we focus on input graphs that fully utilize the GPU. However, we also studied three other utilization levels for all graph analytics algorithms except CC to study how performance is impacted.[6] For all DNNs we extend DNNMark [25] to model the networks and run them for one iteration since CNN iterations have similar behavior [109], [112]. Moreover, to study the training kernels that use atomics in isolation, we also run micro-benchmarks such as BWC and ResNet (1 Layer).

V. RESULTS

Figures 4-7 show the performance improvement, interconnect traffic reduction, miss rate, and energy consumption, respectively, for all micro-benchmarks and benchmarks, across the configurations described in Section IV. We divide energy into multiple components based on source: ALU, shared memory, L1, L2, interconnect, LAB, and main memory. Broadly, the smaller and larger cache configurations show no appreciable change in performance due to the mostly streaming, read only nature of the application's data loads. Thus, we do not show the additional cache configurations in Figure 7, as the energy impact follows a similar trend. In comparison, LAB yields significant benefits. With LAB, an application's performance is closely tied to the ratio of global atomic requests to the total number of global memory requests (ATGR), the application's spatial and temporal locality for atomic transactions, and the application's ratio of LAB size to atomic's working set size. Since these properties vary per application, LAB's benefits vary. LAB's coalescing benefits tap into the locality that exists within atomic transactions, while the serialization benefits (Section III) result from performing atomics locally at the LAB rather than at the L2. Without LAB, overlapping, device-scoped relaxed atomics are sent to the L2, increasing queuing delay and interconnect buffer stalls. Coalescing these atomics in the LAB reduces these overheads, and helps LAB rival Figure 1 (since Figure 1 profiles real GPUs, it cannot include LAB reuse). Overall, across all non-infinite LAB sizes, on average LAB improves performance by 28%, reduces energy by 19%, and reduces interconnect traffic by 19%, while also improving on state-of-the-art techniques like PHI and hLRC.

A. Microbenchmarks

Histogram vs. Histogram Shared: *Histogram*'s device-scope atomics are ideal for LAB. Furthermore, the histogram completely fits in a size 8 LAB. Thus, LAB perfectly coalesces all partial updates from each SM, significantly improving performance (64%), energy (82%), and interconnect traffic (77%). Moreover, since the array fits in a size 8 LAB, increasing LAB size does not help. Thus, LAB can provide significant benefits when applications have many commutative atomics with significant reuse.

By optimizing software to perform most atomics in shared memory, *Histogram Shared* significantly reduces LAB's

[5]We do not use CFD because it has issues with GPGPU-Sim 4.0 [49].

[6]CC only had one input size.

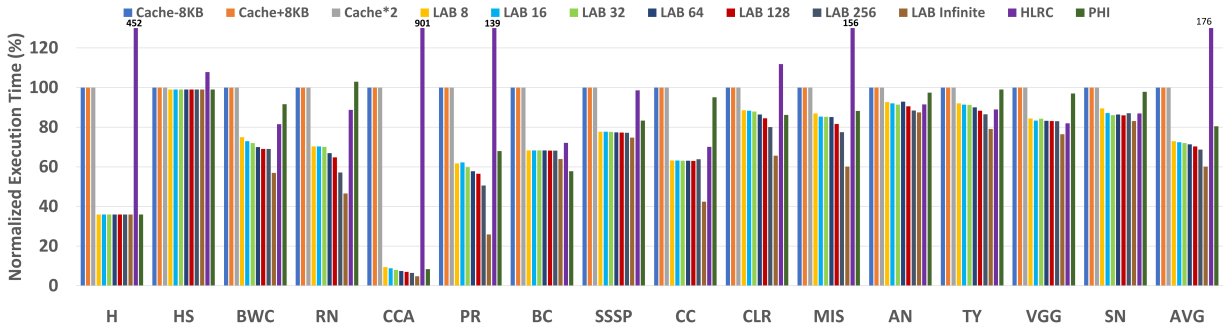

Figure 4. Execution time for different cache configurations, LAB sizes, hLRC, and PHI, normalized to the baseline configuration without LAB from Table I.

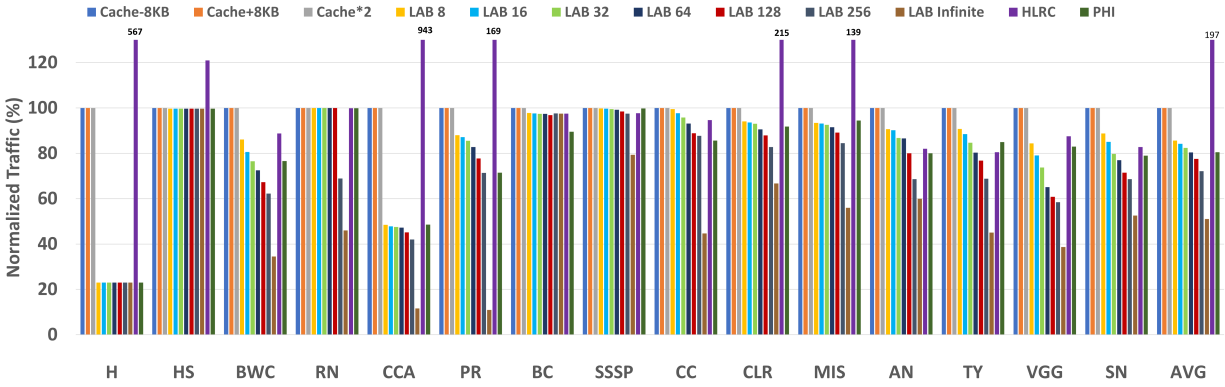

Figure 5. Interconnect traffic reduction for different cache configurations, LAB sizes, hLRC, and PHI, normalized to the baseline configuration without LAB from Table I.

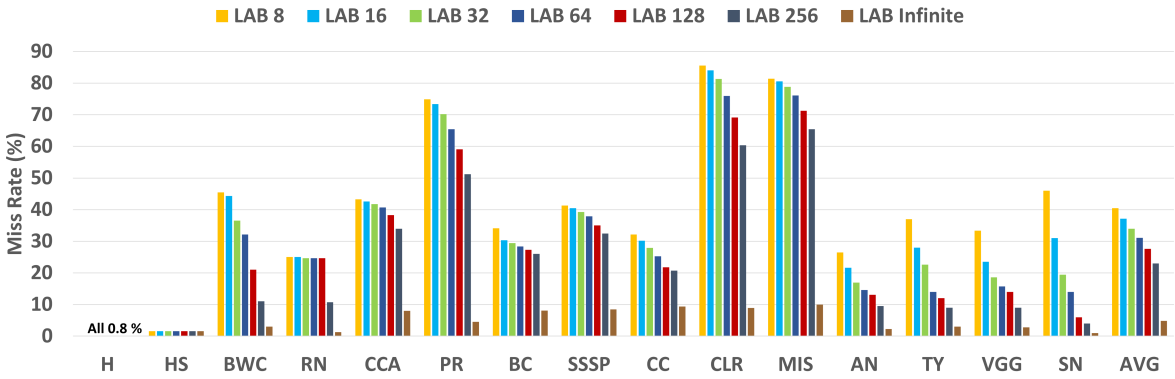

Figure 6. LAB miss rate for different LAB sizes and cache configurations, normalized to the baseline configuration without LAB from Table I.

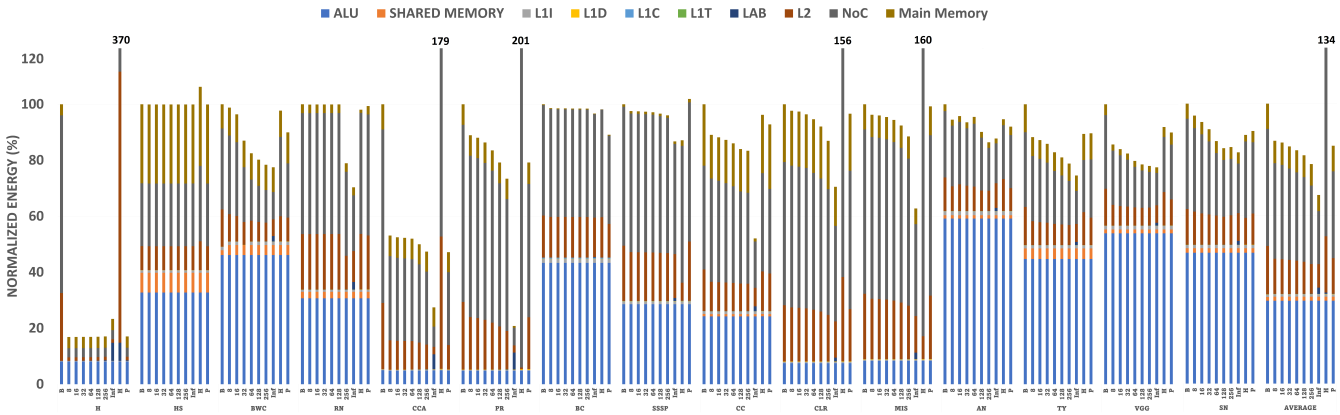

Figure 7. Energy consumption normalized to the baseline without an LAB from Table I. For each application, left to right is the baseline (*B*), LAB-8 (8), LAB-16 (16), LAB-32 (32), LAB-64 (64), LAB-128 (128), LAB-256 (256), LAB-Inf (*Inf*), hLRC (*H*), and PHI (*P*).

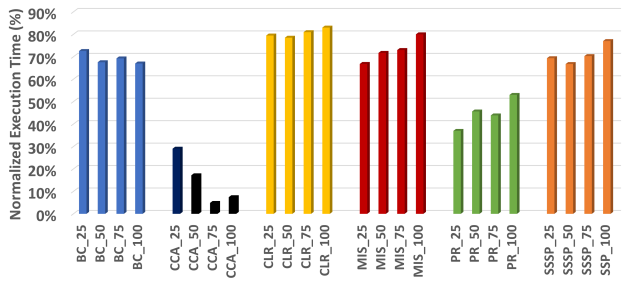

Figure 8. Execution time for the graph analytics workloads with different utilization levels, averaged across LAB-8 to LAB-256, normalized to the baseline configuration without an LAB from Table I.

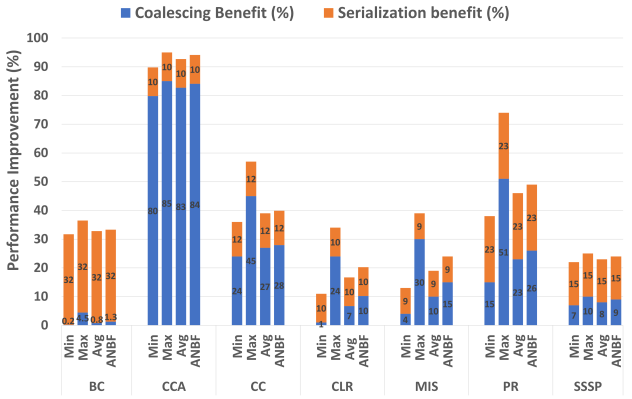

Figure 9. Isolating serialization and coalescing benefits for the graph analytics workloads. ANBF: average without bursty flush. ML workloads not included due to space constraints.

benefits: 1% better performance, with minimal differences in energy and interconnect traffic. These small benefits come from coalescing updates across TBs on the same SM. Thus, LAB may provide small benefits if significant software optimizations are possible. However, in applications where data cannot fit in the shared memory, such optimizations are less practical, as discussed in Section VI.

BWC: *BWC* has a high ATGR ratio because it is composed of atomic-heavy kernels for a single backward pass through a convolution layer. Thus, LAB significantly improves BWC: on average performance improves 28% (max 43%), energy is reduced by 14% (max 22%), and interconnect traffic is reduced by 25% (max 65%). Even with smaller LAB sizes (e.g., LAB 64), BWC obtains significant improvements from reduction in serialization penalty (discussed in further detail in Section V-C). Moreover, BWC uses a larger batch size, which demonstrates that LAB's gains improve as batch size and the number of device-scoped atomics increase.

ResNet: For *ResNet* (*RN*) we study the atomic-heavy kernels since they form major parts of training. On average LAB improves performance by 31%, reduces network traffic by 6%, and reduces energy by 8%. RN's benefits are larger than the full networks (discussed next) because RN only runs the kernels with atomics. However, since these kernels form 70-80% of training time, we expect end-to-end RN would show similar trends. RN's kernels are largely streaming, and only exhibit temporal reuse after a specific number of accesses (256-512) to unique addresses. This pattern repeats N times, where N depends on layer parameters. Thus, for LAB sizes < 256, LAB's benefits come solely from reducing the serialization penalty. Size 256 and larger LABs also benefit from coalescing; for LAB-Inf, 13% of LAB's benefits are from coalescing and 29% from reduced serialization costs.

B. Graph Analytics Workloads

GPU Utilization Study: Figure 8 shows how the graph analytics algorithms perform for different input graphs that utilize 25%, 50%, 75%, and 100% of the GPU, respectively, averaged across LAB sizes 8-256. As utilization increased, the graphs provide additional reuse opportunities, but also increased contention that may increase LAB misses. For benchmarks with High ATGR, the connectivity of the graphs played impacts performance improvement. For example, in PR_75 24% of all nodes are strongly connected, while PR_50's graph has less connectivity: only 5% of the nodes are strongly connected. As a result, LAB provides more reuse for PR_75 than PR_50, and further improves performance. For CCA, both CCA_75 and CCA_100 have a few very strongly connected nodes. Thus, LAB captures most of the reuse even for smaller LAB sizes and enables them to outperform CCA_25 and CCA_50. However, CCA_75 slightly outperforms CCA_100 because it has less contention. The lower utilization graphs also outperform by the higher utilization graphs in a few other cases. For BC and SSSP, which generally have low locality, the majority of LAB's benefits come from reducing the serialization benefit. As a result, performance is similar for all utilization levels. However, for benchmarks with Moderate ATGR such as CLR and MIS, higher utilization levels consistently increase performance slightly. This happens because the larger working sets in the higher utilization graphs of CLR and MIS dominate compared to the additional reuse they offer. Nevertheless, since these differences are small and the overall performance gains are similar for all four utilization levels, we focus on the full (100%) utilization graphs in the remaining analysis.

High Locality & ATGR: As shown in Figure 1, *PR* (0.72) and *CCA* (0.84) have high ATGRs ratio and many commutative atomics. For CCA, a small subset of vertices are very strongly connected. Thus, even a small LAB significantly improves performance: across all LAB sizes, PR improves performance up to 74% (42% on average) and CCA up to 95% (92% on average), with similar improvements in energy consumption, interconnection traffic, and miss rate. As LAB size increases, LAB buffers more atomics locally, but also reduces L1 cache space. However we observed that for both CCA and PR (especially CCA) the average reuse distance often decreases for increasingly strongly connected nodes: CCA and PR's average reuse distance decreases by 96% for the most strongly connected nodes). Moreover, larger LAB sizes are tolerant to higher reuse distances and

hence capture more reuse and further improve performance. Reducing the cache size has less effect because the load locality is relatively lower. The interconnect traffic reduction follows similar trends: a maximum reduction of 89% for LAB-Inf for PR and 88% for CCA. Finally, the overall energy trends are directly proportional to interconnect traffic since device-scoped atomics dominate: on average, energy is reduced by 16% (max 79%) for PR and by 48% for CCA.

LAB also improves CCA's and PR's performance by decreasing serialization cost. As shown in Figure 9, since small LABs (min results) have fewer coalescing opportunities, the serialization cost reduction provides 15% of PR's benefits. However, as LAB size increases (average and max results), coalescing opportunities increase and progressively make up a larger percentage of PR's and CCA's improvements.

Low ATGR & Locality: BC's (0.05) and SSSP's (0.19) ATGRs are significantly lower than CCA and PR. However, as also observed in prior work [26], these atomic accesses cause bottlenecks in BC and SSSP due to serialization. As a result, across all LAB sizes on average performance improves by 30% for BC and 21% for SSSP. Nevertheless, since there are few atomics, even with an infinite LAB, energy and network traffic gains are small (e.g., 3% less network traffic for BC with up to 1% less energy). Interestingly, although BC and SSSP have less locality than CCA and PR, their average reuse distance is 82% lower than PR and CCA for weakly connected nodes. Consequently, all LAB sizes capture similar amounts of reuse, resulting in smaller differences in miss rate until the data completely fits in the LAB (LAB-Inf). Since BC and SSSP have fewer coalescing opportunities, unsurprisingly the vast majority of LAB's benefits come from reducing serialization latency (Figure 9).

Moderate ATGR & Locality: CC, CLR, and MIS have fewer device-scope atomics than CCA and PR but more than BC and SSSP. Although CC, CLR, and MIS have similar ATGR and locality, they have different access patterns. Some of CC's kernels are streaming with little or no reuse; in these kernels most of LAB's benefits come from reducing serialization costs. Nevertheless, most of CC's kernels have moderate to good reuse; in these kernels LAB provides more benefits from coalescing. Like BC and SSSP, the kernels in CC that have at least moderate reuse have small average reuse distances, enabling even small LABs to provide good performance. CLR (ATGR: 0.24) initially performs device-scoped atomics on many cache lines, then reduces the working set as the application proceeds. Thus, all LAB sizes perform similarly once working set decreases. Furthermore, when the working set is large, larger LABs reduce more atomic traffic, but the reduced cache size also increases cache misses. However, again LAB's benefits outweigh the reduced cache locality, although the reduce cache hits reduce the performance difference between the LABs. Somewhat similar to CLR, MIS (ATGR: 0.43) has a large device-scoped atomic working set. Thus, a larger LAB is needed to capture the possible reuse. Accordingly, reducing serialization provides most (69%) of the benefit for small LABs, but a smaller portion (47%) for larger LABs. Moreover, the burstiness of the flushing the LAB is small (Figure 9): 0.5% (BC) - 5% (MIS) on average, and outweighed by LAB's overall gains. Overall, on average performance increases by 37% for CC, 14% for CLR, and 16% for MIS.

C. ML Training Workloads

Overall, on average across all DNN workloads LAB improves performance by 18%, energy by 11%, and interconnect traffic by 19%. However, different training algorithms exhibit different trends in terms of benefits, especially as the number of layers, parameters, and batch sizes vary. For the ML benchmarks LAB's gains are larger for deeper networks and bigger batch sizes – trends that are expected to continue in next generation ML workloads. However, the overall gains are sometimes limited because CNNs are largely compute bound on modern GPUs. Nevertheless, given the significant efforts to optimize compute for CNNs, the memory system will become more of a bottleneck, increasing LAB's utility.

AlexNet, Tiny YOLO: *AlexNet* (AN) and *Tiny YOLO* (TY) have relatively low ATGRs because, like other CNNs, they are compute bound [33]. Thus, LAB improves interconnect traffic and performance for kernels with atomics, but the overall gains are smaller. AN is relatively unaffected by increasing LAB size (9% average performance improvement). Here, reducing cache size while incrementing LAB size is sometimes detrimental: performance declines a little when using a larger LAB due to reduced cache locality. Although AN and TY have a similar number of layers, TY spends more time in kernels with device-scoped atomics (85% compared to 74% for AN). Thus, LAB improves TY's performance more than AN's. On average LAB improves performance by 10%, decreases interconnect traffic by 18%, and decreases energy by 12% for AN and TY.

Moreover, AN's reuse pattern is different for bwd_filter and bwd_data. Bwd_filter is similar to RN, where addresses only exhibit temporal reuse after a certain number of accesses to unique addresses. Thus, the overall reuse depends on batch size and layer parameters. Bwd_data has a lower ATGR than bwd_filter, but more temporal reuse since a small subset of addresses are repeatedly reused. Thus, miss rate decreases as LAB size increases: LAB 64 captures an average of 71% of the reuse for the most heavily accessed addresses, while LAB 16 only provides 28% of the same reuse.

VGG19, SqueezeNet: *VGG19* (VGG) and *SqueezeNet* (SN) are deeper networks, and VGG has a larger batch size (64). Thus, they have more device-scoped atomics than smaller networks like AN, and accordingly larger improvements from LAB: for VGG performance improves up to 24% (16% average), energy decreases up to 22% (19% average), and interconnect traffic decreases up to 61% (29% average). Similarly, more of SN's bwd_filter and bwd_data kernels have

ATGR $> 30\%$. As a result, SN has better average performance improvements (13%) than AN, but smaller improvements than VGG, which has a larger batch size and thus more device-scoped atomics. Similar to the other DNNs, on average LAB reduces interconnect traffic by 21% and energy by 18%. Although VGG and SN see additional benefits from larger LABs, due to more temporal reuse in the bwd_data kernels and to a lesser extent in bwd_filter kernels, a size 16 LAB again provides the majority of the benefits, for the same reasons as previously described networks.

D. Comparison to hLRC

hLRC improves reuse and reduces the serialization penalty by obtaining ownership for atomics. However, hLRC struggles for large working sets and high contention because atomics and data accesses contend for L1 cache entries. Moreover, when multiple SMs perform atomics on the same cache line, hLRC must forward ownership to remote L1s, adding additional overhead. Although hLRC performed well for smaller graphs in prior work [4], our larger graphs have more frequent remote L1 ownership requests, larger working sets, and higher contention. Consequently, hLRC performs poorly for them, especially for CCA, MIS, PR, and Histogram which have high inter-SM contention for atomics. In Histogram and CCA this is amplified by heavy contention across a small subset of addresses, significantly increasing network traffic due to remote invalidations and further hurting performance. Conversely, most ML workloads have less contention or perform atomics in a small window, reducing remote invalidations and enabling more reuse. Consequently, hLRC provides similar performance to LAB for ML workloads, especially for small LAB sizes. However, as LAB size increases (e.g., 64+ entries for AN and VGG), LAB batches more updates without evictions than hLRC. Overall for the full sized benchmarks, compared to the baseline hLRC reduces performance by 76% (3% performance improvement without CCA and Histogram), energy by 72% (12% reduction without CCA and Histogram), and network traffic by 102% (20% reduction without CCA and Histogram). Thus, LAB outperforms hLRC by enabling multiple SMs to update local copies before sending out partial updates.

E. Comparison to PHI

Overall, PHI outperforms hLRC and the baseline. Although, like hLRC, PHI caches atomic updates locally, it does not suffer from remote invalidations [68]. This helps for Histogram, PR, and CCA, which have numerous commutative atomics that PHI buffers locally, significantly reducing network traffic and improving its performance and energy versus hLRC. Furthermore, avoiding expensive remote invalidations also helps PHI outperform hLRC for some applications with moderate or low ATGR and locality like SSSP, CLR, and MIS. However, hLRC outperforms PHI for CC and ML workloads. In the ML workloads hLRC has fewer remote invalidations since SMs usually update their own fixed set of weights. Unlike PHI, hLRC also must wait for ownership for atomics, which reduces these workload's contention and stalls compared to PHI.

Although PHI provides some of LAB's benefits, and offers the second best performance of all configurations, overall LAB outperforms PHI (on average 20.94% better performance, 0.2% better on network traffic and 3% better on energy, and , respectively, for LAB-64; 20% performance, 1.5% energy, and 0.3% network traffic for LAB-Inf). For the histograms, AN, CCA, and SSSP, PHI provides most or all of LAB's benefits because their working set fits in the cache. Moreover, PHI outperforms LAB for BC by better leveraging BC's limited locality across atomics, since PHI can evict either a regular data read or an atomic when an atomic misses, unlike LAB. However, PHI significantly increases L1 cache stalls due to increased L1 contention, which usually occurs when PHI utilizes all the MSHRs for pending data read misses that are evicted by interspersed atomics. Thus, while PHI reduces network traffic for some benchmarks, like LAB, in others it increases stalls and contention between reads and writes. Accordingly, PHI cannot provide all of LAB's benefits in applications with large working sets that cause frequent L1 cache evictions (e.g., CC, CLR, MIS, SN, and VGG). Although these workloads sometimes evict data loads or stores with limited locality, in other situations PHI's co-mingling of data and atomic accesses increases stalls and limits atomic reuse. As a result, LAB provides more consistent improvements than either hLRC or PHI, by both explicitly exploiting commutativity and decoupling where data and atomic accesses are stored.

F. Traditional GPU Workloads

In Figure 10 we evaluate the baseline's, LAB's, PHI's, and hLRC's performance for more traditional GPGPU workloads. We allocate LAB space only for applications that have atomics (Huffman and HybridSort). Thus, we use LAB-0 for all other applications in Figure 10 (adjusted for LAB-0's overheads, see Section V-G), and LAB-64, which was big enough for other applications, for Huffman and HybridSort. Since Huffman and HybridSort perform histograms, unsurprisingly both PHI and LAB improve performance by 24% and 26% respectively, while hLRC again suffers from inter-SM contention. Since the other applications do not use atomics, both LAB and hLRC perform similar to the baseline: all were within 1.2% of the baseline. However, PHI again increases contention between reads and writes, hurting performance for Backprop, BFS, DWT2D, MummerGPU, and SRAD. PHI also performs worse for NW, NN, pathfinder and particlefilter, though the performance degradation is smaller, either because these applications have fewer writes or because their read locality is low enough that writes taking up cache space does not significantly impact performance. Conversely, PHI does better for LUD, gaussian and b+tree, which have write locality. Overall, on average

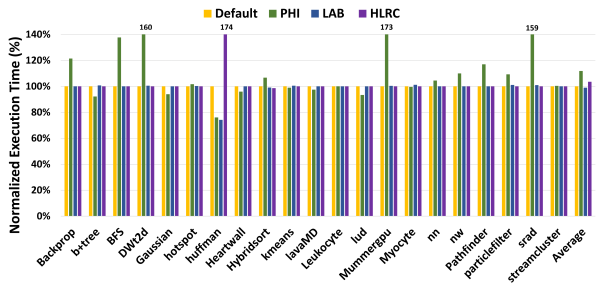

Figure 10. GPGPU results for PHI, hLRC and LAB, normalized to baseline.

PHI is 14% worse and LAB is 0.9% better than the baseline. This further highlights the importance of decoupling data and atomics in GPUs, like LAB, whose reconfigurability allows it to work well across both synchronization-heavy and traditional streaming GPGPU workloads, unlike PHI.

G. Area

Although LAB dynamically partitions the local SRAM, LAB does have some small overheads within the reconfigured SRAM block. Each LAB line requires 4 additional bits (e.g., 32 bytes in total for an LAB of size 64) to identify the atomic operation. Additionally, LAB also adds a 2-1 mux (Section III-A) and a 4 bit comparator to ensure the atomic operation matches. In total, this requires 4 caches lines of overhead in a 128 KB L1 GPU cache, and our results in Section V-F show this has minimal impact on performance. Alternatively, to avoid extra storage, we can exploit the fact that the atomic values did not require all of the data bits, and instead use 4 data bits per line for this.

VI. Discussion

Software vs. Hardware: Histogram obtains significant benefits from using shared memory to exploit commutativity. However, as discussed in Sections II-B and IV-C, the larger graph analytics and ML training workloads cannot use shared memory for their vertices (graph analytics) and weights (ML training) because GPUs must statically allocate the entire array in shared memory – even if a given TB only accesses a small subset of the locations. Moreover, these large arrays exceed the maximum shared memory size per TB. To demonstrate this effect, we increased the number of histogram bins from 256 to 8192. Since Histogram Shared creates partial histograms for each TB, using more bins reduces how many TBs can run simultaneously from 8 to 3 TBs per SM, hurting performance.

In comparison, Histogram has no such limitation. Thus, it outperforms Histogram Shared by at least 3X for 8192 bins with LAB (not shown due to space constraints). Moreover, as histogram bins increase, Histogram Shared eventually cannot run even 1 TB per SM. Thus, using GPU software optimizations like shared memory can improve performance, but only when the working set is sufficiently small. In comparison, LAB dynamically retains the most highly used locations, improving reuse even for applications with large working sets. For example, on average AlexNet's weight array size is 466540 bytes, which requires ∼1900 KB of storage – whereas modern GPUs only allow up to 192 KB of shared memory per SM. Finally, for graph analytics algorithms, the vertex updates are not predictable at compile time and hence it is difficult to use shared memory to improve performance.

It is also possible to virtualize and manage shared memory allocations manually in software [6], [18], [56]. This enables programs with large shared memory requirements to run. However, this requires programmers to handle issues such as evictions, significantly increasing overhead (especially from thread divergence), and prior work has shown that such approaches provide mixed results for CPUs [54].

Applicability to Other ML Training Algorithms: Our results focused on CNN training algorithms (Section IV-C). However, LAB is also applicable to other ML training algorithms: any ML training algorithm that atomically updates shared weights at the end of a training iteration, which is common in data parallel training, could utilize a similar approach. Similar to DAB [16], we attempted to examine recurrent neural network (RNN) training. Like DAB, we found that current versions of cuDNN do not use atomics for weight updates in RNN training. Nevertheless, we expect that other ML training algorithms such as Reinforcement Learning and GANs would obtain similar benefits to CNNs.

Simplicity: Although our proposed additions are relatively simple, LAB still provides significant benefits by intelligently exploiting algorithmic properties. Moreover, LAB seamlessly fits in the existing, per-SM reconfigurable SRAM, which allows programmers to utilize the LAB only when it is useful (unlike prior approaches). Prior approaches (Section VII) provide some of the same benefits as LAB, but often require more invasive coherence protocol or consistency model changes [4], [9], [93], [107] or suffer from cache contention [2], [68]. Thus, LAB's simplicity is a strength and demonstrates how the additional complexity of prior approaches is unnecessary, while also improving efficiency over the state-of-the-art (Section V).

VII. Related Work

Table IV compares LAB to prior work.

Remote Memory Operations [34], [55], [89], [104]: RMOs send and perform update operations to a fixed memory location or memory controller, and have been used in Cray T3E, NYU Ultra, SGI Origin, and NVIDIA's Fermi GPUs. RMOs avoid contention for cache lines since updates are sent to a fixed, shared memory location. However, this approach increases memory traffic, which hurts performance, and sometimes require programmers to explicitly allocate shared memory locations. In comparison, LAB buffers commutative atomics locally and does not increase network traffic.

Coherence Protocol or Consistency Model Changes [4], [5], [9], [30], [93], [107]: Other work exploited commutativity by extending or modifying the coherence protocol or memory

Feature	hLRC [4]	DeNovo [93]	COUP [107]	PHI [68]	RMOs [34], [89]	TS [85], [98]	LAB
No coherence protocol change	X	X	X	✓	✓	X	✓
No memory consistency model change	X	✓	✓	✓	✓	✓	✓
Low degree of atomic L2 traffic	✓	✓	✓	✓	X	X	✓
Reduces atomic serialization penalty	✓	✓	✓	✓	X	X	✓
No overhead for remote invalidations or ownership requests	X	X	✓	✓	✓	✓	✓
Decouple data & atomic accesses	X	X	X	X	X	X	✓
Applied to GPUs	✓	✓	X	X	✓	✓	✓

Table IV
COMPARING LAB TO PRIOR WORK.

consistency model. Sinclair, et al. extended DeNovo to CPU-GPU systems and showed how obtaining ownership for written data reduced global memory traffic for atomics [5], [93], [94]. Subsequently, hLRC extended DeNovo to only obtain ownership for atomics [4]. Coup [107] and CCache [9] apply similar concepts to CPU coherence protocols and software. Although these approaches provide some of LAB's features, LAB outperforms hLRC (Section V-D), and they significantly change the coherence protocol or consistency model. Moreover, GPU coherence protocols differ significantly from CPU coherence protocols [40], [41], [95], which makes adopting CPU coherence-based techniques like Coup or CCache on GPUs difficult. Similarly, AtomicCoherence makes caching GPU atomics in the L1 easier, but requires coherence and interconnect changes [30]. Finally, GPU timestamp (TS)-based protocols [85], [95], [98] improve efficiency, especially for streaming GPGPU workloads. However, these protocols are write-through or write-no-allocate for stores (and atomics) and block further accesses to the addresses that are written through to the L2, limiting intra- and inter-TB reuse opportunities.

Add Buffers to Caches: AIM uses special instructions to perform aggregation for commutative updates throughout the memory hierarchy [2]. However, similar to hLRC, AIM uses coherence to transfer aggregation updates between remote caches. For workloads with large working sets and high contention, this will hurt performance as discussed in Section V-D. PHI [68] improves commutative access performance without changing the coherence or consistency by buffering and coalescing updates in the cache. As shown in Section V-E, LAB outperforms PHI, since the buffered and data lines are not partitioned, contention from other GPU memory accesses can evict buffered lines prematurely and increase stalls, reducing coalescing and increasing in global memory traffic. In comparison, since LAB utilizes a separate space it is unaffected by data accesses, and thus increases reuse. Additionally, PHI requires both a buffered update bit and bits to identify the atomic operation for the entire cache, whereas the LAB only needs atomic operation bits (Section V-G).

Avoiding Collisions [26]: Egielski, et al. reduced GPU atomic collisions with software optimizations to coalesce atomics [26]. Although this reduces serialization, LAB targets an orthogonal problem: performing atomics locally to reduce serialization. Moreover, their approach could further improve LAB's performance by increasing hits.

ML Training: Prior work also optimized ML training. However, unlike our work, most of this work optimizes the width or number of memory accesses [44], [101], [113], utilizing compression [86], optimizing synchronization in distributed training [108], or rewriting code to keep memory accesses in the register file or shared memory [24], [51], [71], [111]. In comparison, we focus on a different bottleneck – fine-grained synchronization. Moreover, these works are complementary because applying them removes other sources of inefficiency and makes fine-grained synchronization even more important. Prior work also converted fine-grained synchronization into data accesses [21], [70]. This removes atomics, but potentially increases convergence time. In comparison, we make device-scoped atomic accesses cheaper without increasing iterations.

VIII. CONCLUSION

As GPGPU applications increasingly use fine-grained synchronization, improving device-scoped atomics support is imperative. We exploit the insight that atomic accesses in graph analytics and ML training are commutative, and utilize recent work to identify commutative atomics. Next, we introduce a small, per-SM buffer (LAB) that combines commutative atomics and utilizes reconfigurability to avoid hurting applications that do not use commutative atomics. LAB improves locality for commutative atomics, reduces serialization costs, and reduces execution time, energy consumption, and on-chip memory traffic compared to state-of-the-art solutions.

REFERENCES

[1] M. Ahmad and O. Khan, "GPU Concurrency Choices in Graph Analytics," in *IISWC*, 2016, pp. 1–10.

[2] J. Ahn, S. Yoo, and K. Choi, "AIM: Energy-Efficient Aggregation Inside the Memory Hierarchy," *ACM TACO*, vol. 13, no. 4, Oct. 2016.

[3] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing," in *ISCA*, 2016, pp. 1–13.

[4] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood, "Lazy Release Consistency for GPUs," in *MICRO*, 2016, pp. 26:1–26:13.

[5] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: A Flexible Interface for Efficient Heterogeneous Coherence," in *ISCA*, 2018, pp. 261–274.

[6] L. Alvarez, L. Vilanova, M. Moreto, M. Casas, M. Gonzàlez, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero, "Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures," in *ISCA*, 2015, pp. 720–732.

[7] D. A. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner, *Benchmarking for Graph Clustering and Partitioning*, 2018, pp. 161–171.

[8] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, April 2009, pp. 163–174.

[9] V. Balaji, D. Tirumala, and B. Lucia, "Flexible Support for Fast Parallel Commutative Updates," 2017.

[10] H.-J. Boehm and S. V. Adve, "Foundations of the C++ Concurrency Memory Model," in *PLDI*, 2008, p. 68–78.

[11] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *IEEE International Symposium on Workload Characterization*, ser. IISWC, 2012, pp. 141–151.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, Oct 2009, pp. 44–54.

[13] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *IISWC*, 2010, pp. 1–11.

[14] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *IISWC*, Sept 2013, pp. 185–195.

[15] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016, pp. 367–379.

[16] Y. H. Chou, C. Ng, S. Cattell, J. Intan, M. D. Sinclair, J. Devietti, T. G. Rogers, and T. M. Aamodt, "Deterministic Atomic Buffering," in *MICRO*, Oct 2020, pp. 981–995.

[17] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk, "Full-Speed Deterministic Bit-Accurate Parallel Floating-Point Summation on Multi- and Many-Core Architectures," INRIA - Centre de recherche Rennes - Bretagne Atlantique, Tech. Rep., 2014.

[18] H. Cook, K. Asanovic, and D. A. Patterson, "Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments," Electrical Engineering and Computer Sciences University of California at Berkeley, Tech. Rep., 2009.

[19] W. J. Dally, "Hardware for Deep Learning," SysML Keynote, Feb 2018.

[20] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable Heterogeneous Computing (SHOC) Benchmark Suite," in *GPGPU-3*, 2010, p. 63–74.

[21] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, "Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent," in *ISCA*, 2017, pp. 561–574.

[22] D. Defour and S. Collange, "Reproducible floating-point atomic addition in data-parallel environment," in *FedCSIS*, 2015, pp. 721–728.

[23] C. Demetrescu, "9th DIMACS Implementation Challenge," http://users.diag.uniroma1.it/challenge9/download.shtml, 2006.

[24] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Y. Hannun, and S. Satheesh, "Persistent RNNs: Stashing Recurrent Weights On-Chip," in *ICML*, 2016, pp. 2024–2033.

[25] S. Dong and D. Kaeli, "DNNMark: A Deep Neural Network Benchmark Suite for GPUs," in *GPGPU*, 2017, pp. 63–72.

[26] I. J. Egielski, J. Huang, and E. Z. Zhang, "Massive Atomics for Massive Parallelism on GPUs," in *ISMM*, 2014, p. 93–103.

[27] I. El Hajj, J. Gomez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu, "KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism," in *MICRO*, 2016, pp. 1–12.

[28] B. Feinberg, B. C. Heyman, D. Mikhailenko, R. Wong, A. C. Ho, and E. Ipek, "Commutative Data Reordering: A New Technique to Reduce Data Movement Energy on Sparse Inference Workloads," in *ISCA*, 2020, pp. 1076–1088.

[29] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A Configurable Cloud-scale DNN Processor for Real-time AI," in *ISCA*, 2018, pp. 1–14.

[30] S. Franey and M. Lipasti, "Accelerating atomic operations on gpgpus," in *NoCS*, 2013, pp. 1–8.

[31] B. R. Gaster, D. Hower, and L. Howes, "HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models," *ACM TACO*, vol. 12, no. 1, pp. 7:1–7:26, Apr. 2015.

[32] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor," in *MICRO*, 2012, pp. 96–106.

[33] Google, "Hot Chips 2017: A Closer Look At Google's TPU v2," September 2017.

[34] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer," *IEEE TOCS*, vol. C-32, no. 2, pp. 175–189, Feb 1983.

[35] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-Tuning a High-Level Language Targeted to GPU Codes," in *InPar*, 2012, pp. 1–10.

[36] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *MICRO*, 2016, pp. 1–13.

[37] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *ISCA*, 2016, pp. 243–254.

[38] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *HPCA*, Feb 2018, pp. 620–629.

[39] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[40] B. Hechtman, S. Che, D. Hower, Y. Tian, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood, "QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs," in *HPCA*, Feb 2014, pp. 189–200.

[41] B. A. Hechtman and D. J. Sorin, "Evaluating Cache Coherent Shared Virtual Memory for Heterogeneous Multicore Chips," in *ISPASS*, 2013, pp. 118–119.

[42] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free Memory Models," in *ASPLOS*, 2014, pp. 427–440.

[43] HSA Foundation, "HSA Platform System Architecture Specification," http://www.hsafoundation.com/?ddownload=4944, 2015.

[44] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient Data Encoding for Deep Neural Network Training," in *ISCA*, 2018, pp. 776–789.

[45] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, "A Distributed Multi-GPU System for Fast Graph Processing," *Proc. VLDB Endow.*, vol. 11, no. 3, p. 297–310, Nov. 2017.

[46] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ISCA*, 2017, pp. 1–12.

[47] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "AccelWattch: A Power Modeling Framework for Modern GPUs," in *MICRO*, 2021, p. 738–753.

[48] A. Kerr, D. Merrill, J. Demouth, and J. Tran, "CUTLASS: Fast Linear Algebra in CUDA C++," https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/, 2017.

[49] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *ISCA*, 2020, pp. 473–486.

[50] M. Khairy, A. Jain, T. M. Aamodt, and T. G. Rogers, "Exploring Modern GPU Memory System Design Challenges through Accurate Modeling," *CoRR*, vol. abs/1810.07269, 2018.

[51] F. Khorasani, H. A. Esfeden, N. Abu-Ghazaleh, and V. Sarkar, "In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization," in *MICRO*, 2018.

[52] Khronos OpenCL Working Group, "The OpenCL Specification," https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf, 2021.

[53] J. Kim and C. Batten, "Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists," in *MICRO*, 2014, p. 75–87.

[54] W. Kim, S. Tavarageri, P. Sadayappan, and J. Torrellas, "Architecting and Programming a Hardware-Incoherent Multiprocessor Cache Hierarchy," in *IPDPS*, 2016, pp. 555–565.

[55] B. Klenk, N. Jiang, G. Thorson, and L. Dennison, "An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives," in *ISCA*, 2020, pp. 996–1009.

[56] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, P. Srivastava, M. Kotsifakou, S. V. Adve, and V. S. Adve, "Stash: Have Your Scratchpad and Cache it Too," in *ISCA*, 2015, pp. 707–719.

[57] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras, "Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead," *ACM TACO*, vol. 13, no. 1, pp. 1:1–1:22, Mar. 2016.

[58] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," in *OSDI*, Oct. 2012, pp. 31–46.

[59] J. H. Lee, J. Sim, and H. Kim, "BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models," in *PACT*, 2015, pp. 241–252.

[60] Lee Howes and Aaftab Munshi, "The OpenCL Specification, Version 2.0," Khronos Group, 2015.

[61] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013, p. 487–498.

[62] J. Lew, D. Shah, S. Pati, S. Cattell, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, and T. M. Aamodt, "Analyzing Machine Learning Workloads Using a Detailed GPU Simulator," *CoRR*, vol. abs/1811.08933, 2018.

[63] J. Lew, D. Shah, S. Pati, S. Cattell, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, and T. M. Aamodt, "Analyzing Machine Learning Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2019.

[64] B. Li, J. Wei, J. Sun, M. Annavaram, and N. S. Kim, "An Efficient GPU Cache Architecture for Applications with Irregular Memory Access Patterns," *ACM TACO*, vol. 16, no. 3, Jun. 2019.

[65] D. Lustig, S. Sahasrabuddhe, and O. Giroux, "A Formal Analysis of the NVIDIA PTX Memory Consistency Model," in *ASPLOS*, 2019, p. 257–270.

[66] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication," in *USENIX ATC*, Jul. 2017, pp. 195–207.

[67] D. Merrill, "NVIDIA CUB Library," https://nvlabs.github.io/cub/, 2020.

[68] A. Mukkara, N. Beckmann, and D. Sanchez, "PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates," in *MICRO*, 2019, pp. 1009–1022.

[69] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-Free Irregular Computations on GPUs," in *GPGPU-6*, 2013, p. 96–107.

[70] F. Niu, B. Recht, C. Re, and S. J. Wright, "HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent," in *NeurIPS*, 2011, pp. 693–701.

[71] NVIDIA, "NVIDIA cuDNN: GPU Accelerated Deep Learning," https://developer.nvidia.com/cudnn, 2018.

[72] NVIDIA, "CUDA C++ Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2020.

[73] NVIDIA, "libcu++: The C++ Standard Library for Your Entire System," https://nvidia.github.io/libcudacxx/, 2020.

[74] NVIDIA Corp., "Inside Volta: The World's Most Advanced Data Center GPU," https://devblogs.nvidia.com/parallelforall/inside-volta/, May 2017.

[75] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood, "Synchronization Using Remote-Scope Promotion," in *ASPLOS*, 2015, p. 73–86.

[76] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems," in *MICRO*, 2017, p. 41–54.

[77] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *ISCA*, 2017, pp. 27–40.

[78] V. Podlozhnyuk, "Histogram calculation in CUDA," https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf, 2007.

[79] B. Pourghassemi, C. Zhang, J. H. Lee, and A. Chandramowlishwaran, "On the Limits of Parallelizing Convolutional Neural Networks on GPUs," in *SPAA*, 2020, p. 567–569.

[80] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *MICRO*, 2013, pp. 457–467.

[81] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs," *CoRR*, vol. abs/1811.08309, 2018.

[82] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling Low-power, Highly-accurate Deep Neural Network Accelerators," in *ISCA*, 2016, pp. 267–278.

[83] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-time Object Detection," in *CVPR*, 2016, pp. 779–788.

[84] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems," in *HPCA*, 2020, pp. 582–595.

[85] X. Ren and M. Lis, "Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence," in *HPCA*, 2017, pp. 625–636.

[86] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks," in *HPCA*, 2018, pp. 78–91.

[87] D. E. Rumelhart, R. Durbin, R. Golden, and Y. Chauvin, *Backpropagation: The Basic Theory*, 1995, p. 1–34.

[88] G. Salvador, W. H. Darvin, M. Huzaifa, J. Alsop, M. D. Sinclair, and S. V. Adve, "Specializing Coherence, Consistency, and Push/Pull for GPU Graph Analytics," in *ISPASS*, 2020.

[89] S. L. Scott, "Synchronization and Communication in the T3E Multiprocessor," in *ASPLOS*, 1996, p. 26–36.

[90] A. Segura, J. Arnau, and A. González, "SCU: A GPU Stream Compaction Unit for Graph Processing," in *ISCA*, 2019, pp. 424–435.

[91] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars," in *ISCA*, 2016, pp. 14–26.

[92] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN Accelerator Efficiency Through Resource Partitioning," in *ISCA*, 2017, pp. 535–547.

[93] M. D. Sinclair, J. Alsop, and S. V. Adve, "Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models," in *MICRO*, December 2015, pp. 647–659.

[94] M. D. Sinclair, J. Alsop, and S. V. Adve, "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems," in *ISCA*, 2017, pp. 161–174.

[95] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *HPCA*, 2013, pp. 578–590.

[96] T. Sorensen, S. Pai, and A. F. Donaldson, "One Size Doesn't Fit All: Quantifying Performance Portability of Graph Applications on GPUs," in *IISWC*, November 2019.

[97] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, *IMPACT Technical Report, IMPACT-12-01*, University of Illinois, at Urbana-Champaign, 2012.

[98] A. Tabbakh, X. Qian, and M. Annavaram, "G-TSC: Timestamp Based Coherence for GPUs," in *HPCA*, 2018, pp. 403–415.

[99] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *MICRO*, 2019, p. 372–383.

[100] V. Vineet and P. J. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," in *CVPR Workshops*, 2008, pp. 1–8.

[101] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks," in *PPoPP*, 2018, pp. 41–53.

[102] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-Performance Graph Processing Library on the GPU," in *PPoPP*, 2016.

[103] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, "Remote-Scope Promotion: Clarified, Rectified, and Verified," in *OOPSLA*, 2015, p. 731–747.

[104] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, March 2011.

[105] Y. Yang, Z. Li, Y. Deng, Z. Liu, S. Yin, S. Wei, and L. Liu, "GraphABCD: Scaling out Graph Analytics with Asynchronous Block Coordinate Descent," in *ISCA*, 2020, p. 419–432.

[106] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism," in *ISCA*, 2017, pp. 548–560.

[107] G. Zhang, W. Horn, and D. Sanchez, "Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-Coherent Systems," in *MICRO*, Dec 2015, pp. 13–25.

[108] H. Zhang, Y. Li, Z. Deng, and X. Liang, "AutoSync: Learning to Synchronize for Data-Parallel Distributed Deep Learning," in *NeurIPS*, 2020.

[109] B. Zheng, A. Tiwari, N. Vijaykumar, and G. Pekhimenko, "Ecornn: Efficient computing of lstm rnn training on gpus," *arXiv preprint arXiv:1805.08899*, 2018.

[110] Z. Zheng, W. Jiang, and G. Wu, "SpeeDO: Parallelizing Stochastic Gradient Descent for Deep Convolutional Neural Network," in *LearningSys*, 2015.

[111] F. Zhu, J. Pool, M. Andersch, J. Appleyard, and F. Xie, "Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-Chip," in *ICLR*, 2018.

[112] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "TBD: Benchmarking and Analyzing Deep Neural Network Training," in *IISWC*, October 2018.

[113] M. Zhu, M. Rhu, J. Clemons, S. W. Keckler, and Y. Xie, "Training long short-term memory with sparsified stochastic gradient descent," https://openreview.net/forum?id=HJWzXsKxx, 2016.