Deterministic Atomic Buffering

Yuan Hsi Chou*¶, Christopher Ng*¶, Shaylin Cattell*, Jeremy Intan†, Matthew D. Sinclair†‖, Joseph Devietti‡, Timothy G. Rogers§ and Tor M. Aamodt*

*University of British Columbia, †University of Wisconsin, ‡University of Pennsylvania, §Purdue University ‖AMD Research
{yuanhsi, chris.ng}@ece.ubc.ca, shaylinc@student.ubc.ca, {intan, sinclair}@cs.wisc.edu, devietti@cis.upenn.edu, timrogers@purdue.edu, aamodt@ece.ubc.ca
¶equal contribution

***Abstract*—Deterministic execution for GPUs is a desirable property as it helps with debuggability and reproducibility. It is also important for safety regulations, as safety critical workloads are starting to be deployed onto GPUs. Prior deterministic architectures, such as GPUDet, attempt to provide strong determinism for all types of workloads, incurring significant performance overheads due to the many restrictions that are required to satisfy determinism. We observe that a class of *reduction workloads*, such as graph applications and neural architecture search for machine learning, do not require such severe restrictions to preserve determinism. This motivates the design of our system, Deterministic Atomic Buffering (DAB), which provides deterministic execution with low area and performance overheads by focusing solely on ordering atomic instructions instead of all memory instructions. By scheduling atomic instructions deterministically with atomic buffering, the results of atomic operations are isolated initially and made visible in the future in a deterministic order. This allows the GPU to execute deterministically in parallel without having to serialize its threads for atomic operations as opposed to GPUDet. Our simulation results show that, for atomic-intensive applications, DAB performs 4× better than GPUDet and incurs only a 23% slowdown on average compared to a non-deterministic GPU architecture. We also characterize the bottlenecks and provide insights for future optimizations.**

***Index Terms*—GPU architecture, determinism, performance, parallel programming**

I. Introduction

GPUs are extensively used to accelerate parallel workloads, such as machine learning [1], [2], [3] and graph workloads [4]. The utilization and adoption of machine learning and graph applications are growing rapidly, reaching a wide variety of areas such as autonomous agents [5], biomedical engineering, physics, commerce, and finance.

However, the non-deterministic nature of multi-threaded processors, such as GPUs, has become an issue in the field of machine learning. The network models trained by non-deterministic GPU architectures have non-trivial variance in achieved accuracy, even if all other aspects are held constant. Coupled with the long time periods required for training, GPU non-determinism presents a major challenge. This is especially important since the improvements in model accuracy often range within 1-3% of the baseline, in part due to the effects of non-deterministic GPUs. Reinforcement learning is also affected by non-determinism, with GPU variance around 12% [6]. Safety critical applications that adopt machine learning models, such as autonomous agents and medical diagnostics, require reproducibility to ensure that systems meet specifications or that experimental results can be replicated for verification [7], [8]. The variance caused by GPU non-determinism also affects graph applications, and can become problematic as graphs are starting to be used in graph neural networks as well [3]. In addition to providing clean and reproducible experiments, determinism can also improve debugging, e.g., if an algorithm converges only sometimes on non-deterministic hardware, determinism will allow us to accurately pinpoint the root cause of the divergence.

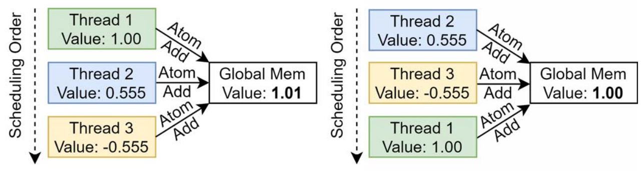


Fig. 1: Simplified non-deterministic reduction example: Base-10, 3-digit precision, rounding up (details in Section III-B).

Prior work attempted to provide deterministic execution for multi-threaded CPU code [9], [10], [11], GPUs [12], and more targeted solutions like reproducible floating-point arithmetic [13]. CPU-focused solutions such as Kendo [9] work well on a small number of threads but do not scale well as they incur non-trivial thread serialization. GPU solutions such as GPUDet [12] provide strong determinism for all types of workloads by handling all memory instructions. However, the generic deterministic architecture of GPUDet incurs high performance overheads since it places many restrictions on executions and threads are often required to stall or serialize. Domain-specific solutions such as the work by Collange et al. [13] focused on the reproducibility of floating point atomics by proposing to enforce floating point ordering and to use a wide accumulator to eliminate floating point rounding errors [13]. However, these methods incur high performance and area overheads, respectively.

In this work, we overcome these scalability and performance challenges by focusing on GPU reduction workloads and providing deterministic execution only for GPU atomic operations. Reduction workloads, although traditionally less common in general-purpose GPU (GPGPU) programs, have become increasingly popular in recent years as machine learning training and graph analytics workloads have targeted GPUs. For example, libraries such as Nvidia's cuDNN machine learning library [14], and graph applications such as Betweenness Centrality (BC) and PageRank, suffer from non-determinism issues in practice [6], [15], [16], [17], [18], [19]. For these workloads, there are several intertwined sources of non-determinism. The unpredictable states of the memory hierarchy and various heuristic-based GPU schedulers cause threads to be scheduled in a non-deterministic manner, which affects the order of operations. Coupled with non-associative floating point operations, this can lead to different results from the same program with the same inputs (Figure 1).

To overcome these issues we exploit the insight, much like modern memory consistency models [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], that we can provide provide low overhead, low performance penalty determinism for GPUs by focusing solely on atomic instructions. We demonstrate that determinism for atomics is sufficient to guarantee determinism under an assumption of data-race freedom (elaborated in Section IV-A), a property known as *weak determinism* [9]. Refining this notion of *weak determinism* further for GPUs, DAB exploits the relaxed atomics used in graph analytics and machine learning GPU workloads [29] to reduce the overheads of deterministic execution. The crux of our approach is to provide hardware buffers for atomic operations to keep them isolated from other threads. We evaluate the costs and benefits of providing this buffering at various levels of the GPU thread hierarchy, including the warp- and scheduler-levels. Scheduling of threads within the chosen level is done deterministically to avoid the deleterious affects of floating-point rounding, as is the process of flushing buffers periodically when they reach capacity. Broadly, when more threads share a buffer it lowers the hardware costs, but imposes more restrictions on GPU scheduling and can have mixed effects on buffer flushing, ultimately resulting in a complex set of trade-offs which we explore.

Overall, this work makes the following contributions:

1) We show that weak determinism can improve performance and provide correctness for reduction workloads that use atomic arithmetic instructions.
2) We propose DAB, an architecture extension that provides deterministic execution on GPUs with low overheads for reduction workloads.
3) We introduce different atomic buffering schemes and characterize them on atomic-intensive benchmarks.
4) We propose different determinism-aware schedulers to enable buffering at a coarser granularity, greatly reducing the area overhead required for atomic buffering.

II. Background and Motivation

This section gives an overview of the neural network and graph algorithms commonly deployed on GPUs, and their sources of non-determinism. It also describes the consequences of non-determinism for these workloads.

A. Neural Networks

Neural networks have emerged as a powerful tool to solve problems in domains such as natural language processing [30], [31], [32], image [1], [2], [33], [34], speech [35], [36], [37], [38], [39], and pattern recognition [40]. They repeat a computationally intensive training process numerous times to tune hyperparameters and search for optimal network architectures [41].

Due to its parallel nature training is performed on GPUs. One of the most common APIs used for neural network training is Nvidia's cuDNN library [14], which offers algorithms for training different types of networks. A subset of these algorithms are used to train convolutional neural networks (CNNs), where each evoked call trains either one layer of activations or weights. While computationally efficient algorithms such as Winograd [42] are often favored, they have high memory overheads and have restrictive dimensional constraints (3×3 or 5×5 filters). For layers where Winograd is not suitable, a non-deterministic algorithm is often used instead, since it has zero memory overhead, no dimensional restrictions, and is sometimes faster than deterministic algorithms. Increasingly 1×1 filters are employed to unlock deeper and wider networks [43].

Our analysis finds that non-determinism is caused by floating point atomics and the non-associativity of floating point values. [1] Fused multiply-add operations are executed on activation and gradients, and atomics update the weight gradients. Though the non-deterministic effects are small, they can propagate and amplify throughout the network [44], resulting in unacceptable variances in results. This is especially problematic during hyperparameter tuning or network architecture search where changes in accuracy may be due to changes to the model or non-deterministic execution.

B. Graph Algorithms

Graph algorithms are used in analyzing social [45] and biological networks [46], computer networking [47], [48], artificial intelligence [49]. For example, Betweenness-Centrality (*BC*) [50] is a well-known graph algorithm used to classify popular nodes within a network. Efficient GPU implementations of *BC* have been developed [4], [51]. Similar to cuDNN, the source of non-determinism in GPU implementations of *BC* is the non-associativity of floating point addition. *BC* performs a graph traversal and iteratively updates node data using atomic adds. *BC* is used in applications of machine learning to physical [52] and biological sciences [53], [54], [55], [56], and in reinforcement learning [57], where physical phenomena are represented as graphs.

[1] According to [14], a non-deterministic algorithm is available for calculating a particular loss function for recurrent neural networks (RNNs). However, we were unable to find the source of non-determinism in RNNs, regardless of the chosen algorithm. The disassembled PTX also suggests that the non-deterministic algorithm is not a reduction algorithm, so we leave RNNs as future work.

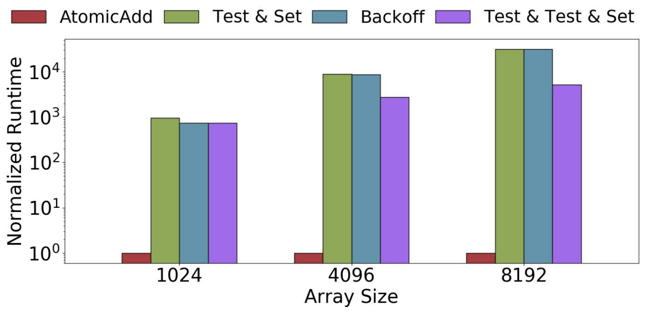


Fig. 2: AtomicAdd running on DAB vs locking algorithms on non-deterministic GPU, normalized to AtomicAdd, simulated on GPGPU-Sim.

C. Software Based Determinism

While deterministic execution can be achieved through software solutions for both neural network training and graph algorithms, there are some drawbacks. Many software-based determinism schemes may require non-trivial effort in order to have comparable performance to the non-deterministic alternative. For example, while cuDNN's deterministic convolution algorithm has good performance, it is the result of hand-tuned SASS code optimized for a given GPU architecture [14]. The required engineering work is a significant overhead for enabling deterministic execution on a general set of workloads. Without heavy optimization, software solutions that render generic reduction workloads deterministic can incur significant performance overheads [13].

To understand the performance impact of using deterministic versus non-deterministic GPU algorithms, we designed a simple microbenchmark that summed up elements of an array into a single output variable. In the non-deterministic algorithm, each thread atomically adds an array element to the output. Since the ordering of atomic operations is non-deterministic, the output is non-deterministic as well. To achieve deterministic results, reduction trees or ticket lock-style GPU algorithms [58], [59] could be used to sum up the elements. While reduction trees are fast, they assume the summed values are available a priori, whereas reduction workloads require calculating the summed values on the fly. This means that applying reduction trees to reduction workloads would require expensive barriers or a separate kernel to ensure all dynamically generated values are available. Although locks are sometimes not preferred for GPUs due to poor performance and potential SIMT deadlock issues [60], [61], we utilize the deterministic behavior provided by them to compare to our non-deterministic approach. We implemented three locking algorithms: a basic *Test&Set*-style centralized ticket lock where each thread has the same ticket number for every run (and thus the order threads perform their atomics in is deterministic), a variant that reduces overhead by performing exponential *backoff* in software when the Test&Set's lock acquisition fails, and a *Test&Test&Set* algorithm that reduces Test&Set's overhead by only attempting to acquire the lock when it is likely to succeed. Figure 2 compares the execution time of atomicAdd running on DAB to the three different locking algorithms on a non-deterministic GPU in GPGPU-Sim, normalized to atomicAdd. Although the optimized deterministic approaches reduce the overhead over the base *Test&Set* algorithm, especially as array size (and thus contention) increase, all three locking algorithms take substantially longer than the non-deterministic atomicAdd version.

Our microbenchmark demonstrates that non-deterministic algorithms can significantly outperform deterministic ones. However, for realistic workloads the performance gap between deterministic and non-deterministic algorithms is more blurred. Similar to our microbenchmark, cuDNN's non-deterministic algorithms often utilize atomics to improve performance. However, as customers preferred deterministic algorithms, NVIDIA focused their efforts on optimizing deterministic algorithms, which closed the performance gap between deterministic and non-deterministic algorithms [62]. Thus, for CNN training, we found that the gap between the deterministic and non-deterministic convolution algorithms heavily depends on the dimensions of the convolution, and neither algorithm consistently performs better than the other across all dimensions. Prior work has also demonstrated similar conclusions between deterministic (pull-based) and non-deterministic (push-based) graph algorithms, where the relative performance between the push- and pull-based algorithms is dependent on the input graph [16], [19].

D. Lack of Reproducibility

Reproducibility and verifiability are essential aspects in software research and development. It allows us to verify the correctness of experimental results and easily build off prior work. However, both graph analytics and machine learning research is faced with the problem of a lack of reproducibility. Recent work surveyed recent publications in major machine learning conferences and concluded that the high variance between trials makes it difficult to isolate the impact of the novel contributions introduced in each work [15]. This is particularly an issue since the improvements in accuracy often range within 1-3% compared to the baseline, making it difficult to differentiate between the effects of random initialization, seeding, non-deterministic results, and legitimate improvements. Similarly, additional work investigated the sources of non-determinism in vision-based reinforcement learning and identified several key sources: non-deterministic GPU operations, network initialization, learning environment, batching, and exploration [6]. Crucially, despite holding every other aspect constant and only introducing non-determinism due to non-deterministic GPU operations, the results had a variance of 12%. This is especially crucial for safety critical applications such as autonomous vehicles and medical diagnostics, where rigorous safety regulations demand reproducibility and verifiability.

III. Challenges of Determinism

This section outlines the challenges of achieving deterministic execution for GPU reduction workloads and the limitations of prior work.

A. Reduction Workloads

As discussed in Section II, workloads that use atomics for reductions are commonly deployed on GPUs for their inherent parallel nature. The structure of reduction workloads splits computation into two phases. The first phase partitions computation among threads, where each thread stores their partial computations in memory. Then, a reduction kernel is invoked which reduces these partial computations into the final result before rewriting back to memory. However, the expensive read-modify-write cycles to accumulate the partial results and the memory overhead incurred to store the partial results can be eliminated by using atomic addition operations instead [63].

B. Non-Determinism in GPUs

When executing reduction workloads on GPUs, there are many intertwined sources of non-determinism. First, unpredictable states of the memory hierarchy and various heuristic-based schedulers cause threads to be scheduled in a non-deterministic manner, affecting order of operations. For example, whether a warp is scheduled can be dependent on a cache hit or miss, which cannot be statically determined since GPU state is unknown from previously executed kernels.

The second reason is the non-associative nature of floating point arithmetic. Floating point numbers represent real numbers in hardware using fractions of base 2. Due to this limitation, real numbers cannot be expressed exactly, which leads to representation error. Additionally, the bit width of a floating point ALU is limited, so it is unable to calculate the exact result, leading to floating point rounding errors when calculated in hardware, regardless of the rounding mode. This causes different results to be produced when performing arithmetic in a different order. To demonstrate rounding errors caused by ordering of reduction operations between threads, Figure 1 illustrates a simplified example, adapted from Goldberg [64]. For ease of understanding, in the example we assume a base-10 floating-point representation, three digits of precision and that non-significant digits are rounded up after performing addition. Assume Thread 1, 2 and 3 increment the reduction variable with values $a = 1.00$, $b = 0.555$, and $c = -0.555$. Under the ordering on the left, the reductions compute $(a+b)+c = 1.56+(-0.555) = 1.01$. With the ordering on the right, the reductions compute $(b+c)+a = 0+1.00 = 1.00$, which differs. Similar differences can occur with higher-precision base-2 floating-point and other rounding modes. While the differences introduced by each individual change in reduction ordering may be small, during lengthy computations rounding errors can compound and become significant [65]. Such non-determinism can cause issues for debugging and validation, including deadlocks [66].

C. Problems with Prior Deterministic GPUs

While there have been many prior works on deterministic execution for massively parallel systems [67], [68], [69], [70], [71], most of these solutions focus on software. GPUDet focuses on providing deterministic execution on GPUs through its hardware architecture [12]. GPUDet provides strong determinism by handling all global memory instructions. It appends all global stores to a per-thread store buffer instead of directly writing to global memory. Execution of programs is divided into phases called quanta, where a thread executes up to a fixed number of instructions in parallel mode and then waits for all other threads to end their quanta as well. If a thread encounters an atomic instruction, it will prematurely mark the end of its quantum and end parallel mode. Once all threads have reached the end of their quanta, the threads enter commit mode and global stores in the store buffers are made visible in a deterministic manner, accelerated by Z-buffer hardware in the GPU. Atomics are handled in serial mode, by issuing warps serially in a set order, essentially serializing the GPU. The imposition of frequent quantum barriers and serialization of atomics causes GPUDet to have slowdowns of up to 10× in some applications.

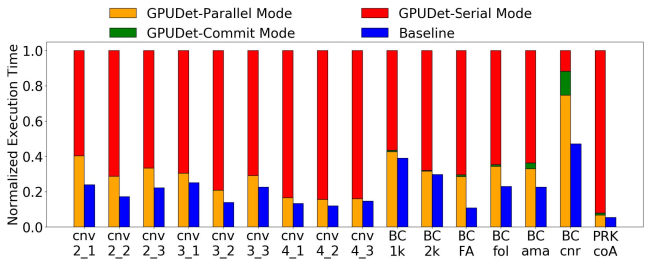


Fig. 3: GPUDet Execution Mode Breakdown.

Figure 3 breaks down the execution modes of GPUDet and compares the execution time to a non-deterministic GPU baseline for convolution and graph applications (workloads and methodology described in Section V). The high execution times previously reported for BFS on GPUDet [12] are observed in Figure 3 for BC, which also has BFS kernels. The serial mode execution times are relatively high since these are with atomic intensive workloads as opposed to the benchmarks in the original work. For these benchmarks, GPUDet spends the majority of the execution time in serial mode dealing with atomic operations, which is the root cause of performance slowdown. Thus, new approaches are needed.

IV. Deterministic Atomic Buffering

In this section, we first describe DAB's memory consistency model, which states the assumptions DAB makes about programs and what guarantees it provides in return. Then we describe how DAB provides determinism for reduction workloads via atomic buffering by locally reducing atomic operations within a core before serializing between cores. This hierarchical approach exploits GPU atomics that become reduction operations [29] to significantly improve performance over serializing all atomics directly (Section III-C).

A. Memory Consistency Model

DAB uses the sequentially consistent for heterogeneous-race-free (SC-for-HRF, or HRF) memory consistency model [24],

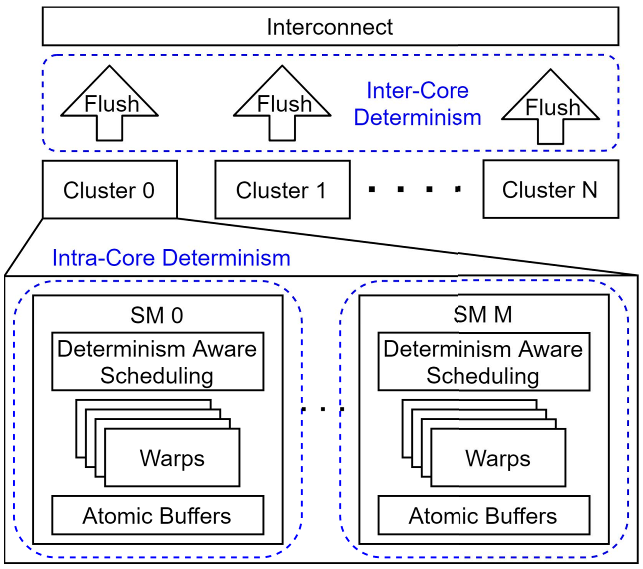


Fig. 4: Block diagram of DAB hardware. Intra-core determinism is enforced by atomic buffers and determinism-aware schedulers (Sections IV-B-IV-C), while inter-core determinism is enforced by a deterministic buffer flushing order (Section IV-D).

[25], [27], [29], [72], [73], which is widely used in modern GPUs. HRF adds scoped synchronization to the popular sequentially consistent for data-race-free (SC-for-DRF, or DRF) memory consistency model that is widely used in multicore CPUs [20], [21], [22], [23], [28].

Like other work [74], [75], DAB assumes CUDA programs are DRF. Moreover, we also assume that programs respect strong atomicity – i.e., that within a given kernel, if an address is ever accessed atomically, all accesses must be atomic [76]. Given a compatible program, DAB guarantees more than just SC behavior, but additionally a deterministic outcome by imposing deterministic semantics on atomic operations. DAB is similar to and inspired by the Kendo [9] scheme for deterministic CPU multithreading, which also makes a DRF assumption. For compatible kernels, DAB provides the *SyncOrder* determinism of Lu et al. [77], which states that each load returns the same value on each execution. DAB-incompatible kernels, e.g., those with data races, are not guaranteed to execute deterministically.

While Kendo focuses on lock acquires and releases, DAB leverages the fact that GPU reduction workloads can benefit from relaxed memory orderings [29] to reduce the overheads of determinism. For example, in CUDA, all atomic operations in the programs we study are compiled into atomics with no implicit ordering [27], and do not implicitly include a memory fence, making them equivalent to *relaxed* atomics in the C, C++, HSA, and OpenCL memory consistency models [23], [24], [26] (separate CUDA fence instructions exist when memory ordering is desired). This lack of memory ordering allows these atomic operations to be aggressively buffered within each streaming multiprocessor (SM), reducing the rate of inter-core communication which is a key overhead in DAB.

Additionally, CUDA atomic operations can be compiled into one of two PTX instructions: atom that returns a value in a register or red (for "reduction") with no output. red instructions avoid dependencies that cross thread or warp boundaries. While red instructions are not emitted by Nvidia's nvcc compiler for our workloads, we confirm through manual inspection that the return values of atoms in our workloads are never used, and we believe this no-return optimization is leveraged in the SASS machine code. The lack of a return value again enables aggressive buffering of atomics.

DAB supports all of PTX's red instructions, including non-associative ones like floating-point addition. DAB can deterministically execute atom instructions, atomic loads/stores, volatile accesses, and memory fences (none of which are found in our workloads) by incurring a buffer flush (Section IV-D) to provide global ordering. For simplicity, in the rest of this paper, *atomic operations* refers to red instructions. DAB also supports CUDA's syncthreads local barrier (found in our cuDNN convolution workload) which includes a CTA-level memory fence, again via a buffer flush. Though complex, the rich interface exposed by CUDA atomics allows DAB to provide determinism at low cost in the common case.

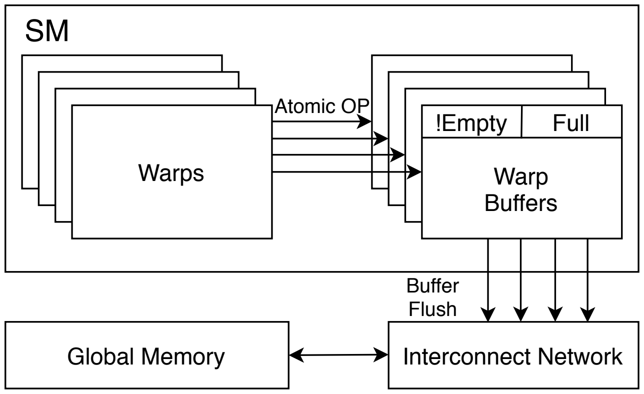
B. Warp-Level Atomic Buffers

In DAB, atomic instructions operate on storage in dedicated buffers, instead of writing directly to global memory. Each atomic buffer contains multiple entries, where each entry holds a *memory address*, an *argument*, an *opcode* and a *valid* bit, e.g., an atomic operation incrementing address 0xB0BA by 1 would be represented as the tuple (0xB0BA, 1, add.f32, valid). Atomic buffers support associative search by memory address. Each buffer has full and non-empty bits to facilitate the buffer flushing process, which makes the partial results stored in each atomic buffer globally visible (Section IV-D).

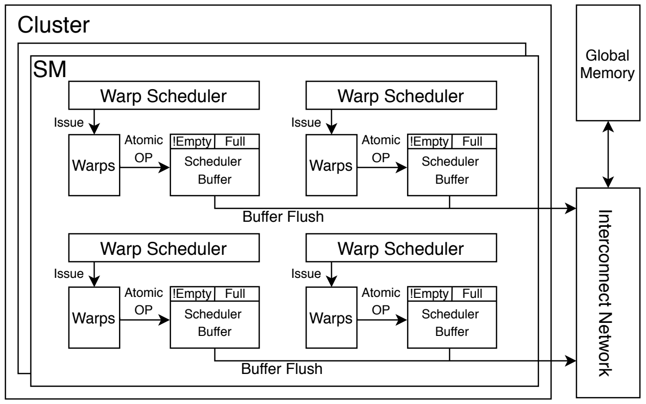
We begin with a simple, though impractical, scheme where each warp has its own atomic buffer with at least 32 entries to support all 32 threads in the warp performing an atomic operation (Figure 5a). An atomic is executed provided sufficient space exists in the per-warp buffer. If there are insufficient entries the warp is blocked from issuing and the full bit is set. Warps are kept active while the buffer is non-empty and wait for the flushing process before they can be reclaimed.

Figure 6 shows a simplified example of warp-level buffering illustrating operation for a single warp. For simplicity, the example warp has one thread. We assume a 2-entry atomic buffer so at most two atomic instructions can be executed before stalling. Initially the buffer is empty, then an atomic add is performed which fills the first buffer entry (❶). Later, another atomic add is performed, filling the second buffer entry (❷). Since there are no more available entries in the buffer, the `full` bit is set. Further atomic instructions from this warp are blocked from issuing until the buffer is flushed.

The contents of a warp-level atomic buffer are deterministic since they are filled based on program order. If threads in the same warp write to the same address, we fill atomic entries

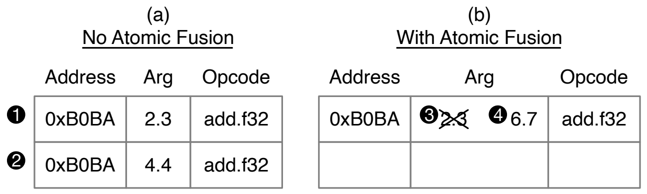


(a) Architecture of Warp Level Buffering. Each warp performs atomic operations to its own warp buffer in isolation. The buffer is flushed at a later time to global memory.



(b) Architecture of Scheduler Level Buffering. Each scheduler has its own buffer. Warps issued by a scheduler write to the corresponding scheduler buffer.

Fig. 5: Atomic Buffering Architectures


(a)
No Atomic Fusion

	Address	Arg	Opcode
❶	0xB0BA	2.3	add.f32
❷	0xB0BA	4.4	add.f32

(b)
With Atomic Fusion

Address	Arg	Opcode
0xB0BA	❸~~2.3~~ ❹6.7	add.f32

Fig. 6: Atomic Buffers, without (left) and with Atomic Fusion (right).

by increasing thread ID. This ensures a warp-level buffer has deterministic contents under any scheduling scheme.

While simple, warp-level buffers require substantial area overheads – each buffer entry is 9 bytes (5B address, 4B argument, 1B for opcode + valid bit). With 32 entries per buffer, and a maximum of 64 warps per SM, the area overhead is quite significant at about 20 KB per SM. This motivates our next scheme which shares atomic buffers across warps.

C. *Scheduler-Level Buffering*

Modern GPUs such as Nvidia's TITAN V have 4 warp schedulers per SM, with each scheduler responsible for issuing a subset of warps in the SM. Scheduler-level buffering (Figure 5b) allocates one atomic buffer per warp scheduler, reducing the area of atomic buffers by 16× compared to warp-level buffering (as we move from 64 warps to 4 schedulers).

With warps sharing a single atomic buffer, program order and thread ID no longer suffice to deterministically order atomic operations, as two warps may "race", e.g., based on the non-deterministic latency of cache accesses, to fill a particular buffer entry. This requires us to adopt determinism-aware warp scheduling, described next. We start with a simple round-robin scheme, then successively relax this to improve performance while preserving determinism. We defer discussion of deterministically assigning warps to schedulers until Section IV-C5.

1) Strict Round Robin: The most straightforward determinism-aware warp scheduler is a strict round robin policy, where warps belonging to the same scheduler are issued in a fixed order (skipping threads blocked on syncthreads). This scheme has non-trivial overheads compared to our non-deterministic baseline, however, as it does not allow warps that could issue to start early.

Figure 7a shows how Strict Round Robin (SRR) orders the execution of two warps *A* and *B* on a given scheduler, with the height of each rectangle illustrating variable latency for non-atomic (light gray) and atomic (dark gray) instructions. SRR always issues from warp *A* first. Only when warp *A* is issued, is warp *B* considered for issuing. If warp *A* is blocked from issuing, either from hazards or unfetched instructions, no instruction is issued, even if warp *B* is unblocked (❶).

2) Greedy Then Round Robin: To improve the performance of SRR, we observe that only atomics need to be ordered to preserve determinism. Thus, we can use a more relaxed scheduling policy that runs any conventional scheduling policy up until atomic instructions are reached. Once all warps reach their first atomic instruction (or have exhausted all instructions), the scheduler switches to the SRR scheduling policy until the kernel ends. Prior work has demonstrated that the Greedy-Then-Oldest scheduling policy (GTO) performs well across most workloads, and is often used as a starting point for more elaborate scheduling policies [78]. So we use GTO scheduling to run prior to any atomic instructions, and we call this scheduling policy Greedy-Then-Round-Robin (or GTRR).

The operation of GTRR is shown in Figure 7b. The initial use of GTO scheduling (❷) avoids stalling for non-atomic instructions. Once all warps either encounter an atomic reduction instruction (❸), or have exited, scheduling switches to SRR. This inflection point is deterministically reached because our memory consistency model assumptions (Section IV-A) ensure all communication between threads uses atomics or appropriate fencing. With warp divergence, this still holds true when divergence is handled by SIMT stacks, where both sides of a branch do not execute concurrently and which side executes first is deterministic [12]. As atomics only occur after the switch to SRR, the ordering of atomic operations remains deterministic. The switch to SRR until the kernel ends can however lead to additional stalling (❹).

3) Greedy Then Atomic Round Robin: We can further relax GTRR scheduling by observing that the execution between atomic instructions does not need to be deterministic. After the warps issue their atomic instruction, conventional scheduling can be employed again up until the next "round" of atomic instructions is reached, so long as the relative issue order of the atomics is deterministic. After the "round" of atomics is complete, non-deterministic scheduling (like GTO) of non-atomic instructions can resume once more. This treats each atomic instruction as a scheduler-level barrier. For warps that do not have any atomic instructions, this "barrier" is reached once it executes all its instructions (similar to the condition to switch to SRR in GTRR).

While the scheduler is running atomics in round robin order, a warp that has already executed its atomic instruction can start executing its subsequent non-atomic instructions without violating determinism. We call this scheduling policy Greedy Then Atomic Round Robin (GTAR).

GTAR is demonstrated in Figure 7c. GTO scheduling is used initially (❺) until all warps reach an atomic instruction (❻), at which point atomic instructions issue and execute in round robin order. As soon as warp A is done with its atomic instruction, it can proceed with a non-atomic instruction (❼) without needing to wait for warp B's atomic to complete. Once all warps finish its atomic, scheduling reverts to GTO.

4) Greedy With Atomic Token: Our final determinism-aware scheduling algorithm exploits the observation that atomic instructions do not need to be executed strictly one after the other, so long as deterministic ordering is preserved. Similar to GPUDet's serial mode [12], we pass a single "token" among warps, and only the warp possessing the token can issue an atomic instruction. At any kernel launch, the warp with the smallest warp ID is initially granted the token. It passes the token to the next warp if it either exhausts all of its instructions, or if an atomic instruction is issued. If a warp wants to issue an atomic instruction, but does not hold the token, it stalls and other warps will have priority to be issued over it. This creates a deterministic ordering of atomic instructions across warps, while still permitting heuristic-based scheduling for non-atomic instructions. We call this scheduling policy Greedy-With-Atomic-Token (GWAT). Similar to GTAR, GWAT enforces the implicit barriers inserted between warps executing in the same hardware slot, and warps from different kernels.

Figure 7d shows GWAT in action. Initially, GTO scheduling occurs while warp A holds the token. At ❽, warp A encounters an atomic and is allowed to issue the atomic since it holds the token. After warp A issues its atomic, the token is passed to warp B, which issues an atomic at ❾ and then passes the token back to A. When warp A completes at ❿, the token is passed back to warp B.

5) Deterministic CTA Distribution: Even with determinism-aware warp scheduling, determinism additionally requires the set of warps assigned to each scheduler is also deterministic, which we refer to as deterministic CTA distribution. Otherwise, order of atomics in each buffer will be affected by differing CTA distributions. We statically partition CTAs among each scheduler in each SM.

Warp and kernel exits must also be handled specially. Within a given scheduler, CTAs are dispatched in *batches*. All atomics from batch b_i must complete before any atomics from b_{i+1}. Note that non-atomic instructions from b_{i+1} can run earlier (except with SRR scheduling); only atomics are confined to run within their batch. Batching CTAs ensures a fixed set of warps share one atomic buffer, and determinism-aware scheduling orders the atomics from those warps.

D. Buffer Flushing

Buffer flushing is the process in which all values stored in all buffers are made globally visible by writing them to memory in a deterministic order. To ensure determinism, DAB flushes buffers only when either 1) all buffers are full, 2) the kernel exits, or 3) a memory fence is reached.

Additionally, DAB addresses the non-deterministic ordering from the interconnect network with a protocol to buffer and reorder memory packets as shown in Figure 8. Once all the buffers are ready to be flushed, we first send pre-flush messages to each memory sub-partition, indicating how many buffer entry flushes from each cluster to expect (Figure 8a). Each memory sub-partition should expect pre-flush messages equal to the number of GPU compute clusters. Next, each buffer pushes its contents to the interconnect (Figure 8b). Each memory sub-partition first waits until all pre-flush messages have been received. Each sub-partition will then use the expected number of messages from each SM to re-order the atomics in round robin order. If the number of expected messages from each SM is not equal, SMs with less messages are eventually skipped once every message from that SM has arrived. As buffer contents from different SMs arrive, they are sent to the ROP to perform the actual atomic operation based on this ordering, and any atomic arriving out of order is buffered in a write queue called a flush buffer, waiting for its turn. If an incoming atomic is buffered while the atomic next in order is waiting in the flush buffer, that atomic is popped from the flush buffer and is sent to the ROP. In order to keep the number of buffered transactions manageable, buffer flushes may not overlap, and buffer flushing can only occur once every transaction write-back from the previous buffer flush has been received. One way to enable this large buffer for reordering the potentially many atomics sent to the memory partitions is to employ virtual write queues, where the a portion of the L2 cache is repurposed as buffering space [79]. This protocol prevents compute clusters from serializing buffer flushes. The buffers are then cleared and execution resumes across all cores.

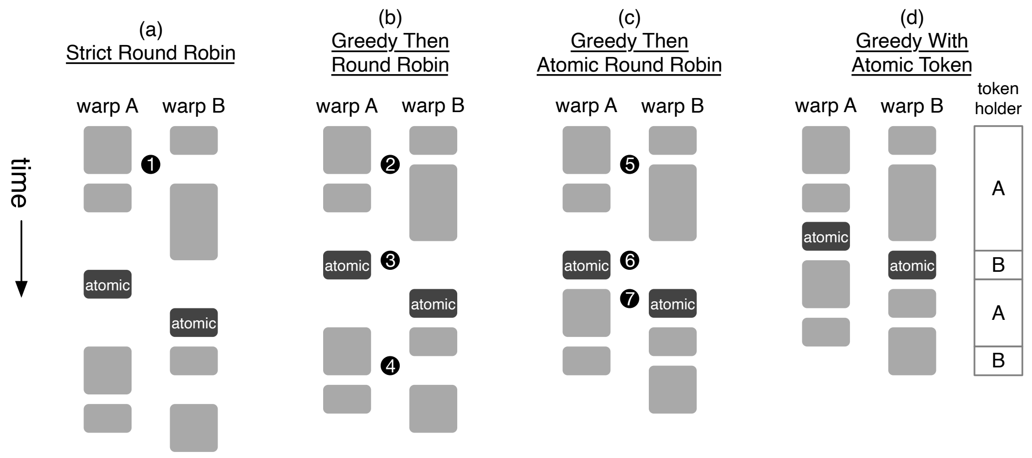


Fig. 7: An example schedule of two warps under each of DAB's four determinism-aware schedulers. Light gray boxes indicate non-atomic instructions and dark gray boxes atomic instructions; box height indicates latency.

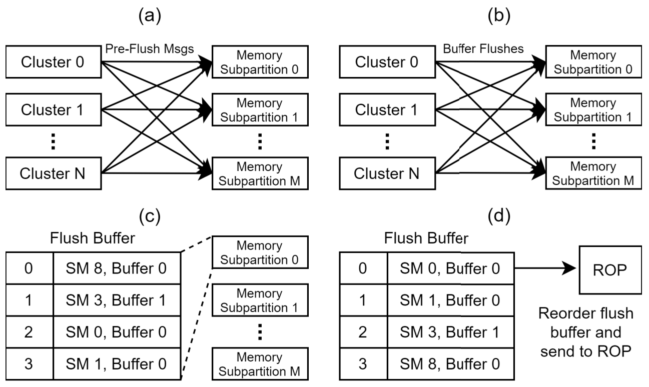


Fig. 8: Mechanism of inter-core determinism. (a) Each cluster sends a pre-flush message to each memory sub-partition, indicating expected number of messages. (b) Each cluster pushes its buffer flushes to interconnect. (c) Each memory sub-partition waits and buffers arriving flushes that are out of order. (d) Stalled buffer flushes are reordered and sent to ROP.

E. Atomic Fusion

To further optimize our design, we implement the fusing of identical atomic operations to the same address within atomic buffers, effectively performing a local reduction. We call this *atomic fusion*. Atomic fusion saves buffer space and delays a costly buffer flush. Atomics to the same location can be fused, even if they are not from the same warp. Since the atomic buffer is a fully-associative structure, atomic fusion has low latency impact when searching for matching addresses.

Figure 6b illustrates the operation of a scheduler-level buffer with atomic fusion enabled. In contrast with Figure 6a, which does not employ atomic fusion, the two adds to address 0xB0BA can use the same buffer entry, summing the argument from the first atomic (❸) with that from the second atomic (❹) to save space.

Both applications with irregular access patterns (graph algorithms), and ordered atomic access patterns (convolutions) benefit from atomic fusion. For example, cuDNN's convolution algorithm partitions the filter into n even regions, and $m \cdot n$ CTAs are launched, with m CTAs atomically adding to each region. The CTAs that atomically access the same region also have the same memory access pattern, meaning additional fusion opportunities if these CTAs are distributed to the same scheduler where they share an atomic buffer.

F. Atomic Coalescing

The baseline GPU coalesces atomics into a single transaction per cache sector. This can be done for the buffer entries as well in DAB. While the entries remain separate within the buffer, entries that write to the same cache sector can be marked, and can be coalesced together into a single transaction while flushing, effectively lowering memory traffic.

G. Limitations

Enabling/Disabling DAB A potential solution for disabling DAB is to extend existing API calls to toggle DAB's hardware structures that enforce determinism. Also, API calls should toggle schedulers to be determinism-aware (e.g. for GWAT, stall a reduction instruction if the warp does not have the token). For CUDA workloads without reductions, the schedulers require no toggling since GTRR, GTAR and GWAT operate like GTO in the absence of reductions.

Context Switching Prior work has proposed to use context switching for preemptive multitasking on GPUs [80], [81], [82], [83]. In addition to saving architectural and scheduler states, DAB requires additional support for context switching. For interrupt-driven context switches, DAB would have to either statically partition the buffer between the switched kernels, or save the buffer contents. For context switches triggered by some deterministic virtual clock (like logical clocks in [9]), buffers are flushed during each context switch to preserve determinism. However, note that training DNNs typically involves running

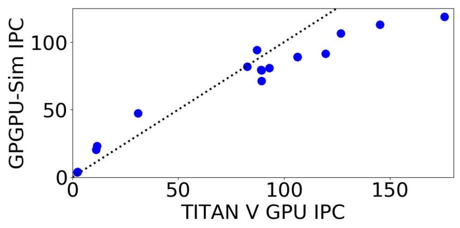


Fig. 9: IPC Correlation of GPGPU-Sim with TITAN V. Dotted line represents a perfect correlation between simulation and hardware.

GPUs for long periods with time-sharing accomplished using check-pointing between GPU kernel launches for individual minibatch training iterations using frameworks such as PyTorch and TensorFlow).

Independent Thread Scheduling Under Volta's independent thread scheduling, each sub-warp can be treated as a separate warp until convergence. Once the sub-warps of a given warp diverge, their execution can be interleaved in strict round robin fashion (while determinism-aware scheduling is still employed at the warp-level), in order to maintain a deterministic ordering of atomics between subwarps.

V. Methodology

To evaluate DAB, we extended GPGPU-Sim 4.0.0 [84], [85], [86] to model the atomic buffers, determinism-aware schedulers, and the buffer flushing mechanism as described in Section IV using the configuration in Table I. For buffer flushing, we simulate an unbounded message buffer for reordering buffer flushes in the memory partition after arriving from compute clusters. Large buffers could be implemented with low area overhead similar to virtual write queues [79], as noted in Section IV-D. To demonstrate the feasibility of this scheme, we repeated our simulations with each out-of-order atomic triggering L2 cache evictions to mimic the virtual write queue. On average, these extra evictions increased the total L2 cache miss rate by less than 1% compared to our original simulations.

The non-deterministic baseline is an unmodified GPGPU-Sim using GTO scheduler, where branch divergence is handled by SIMT stacks. Figure 9 shows IPC correlation of 96.8% and error rate of 32.5% on our evaluation benchmarks comparing GPGPU-Sim and TITAN V GPU. We also compare against an updated version of GPUDet [12] as the deterministic baseline. Our GPUDet updates enabled it to run on newer GPUs, which required adding support for sector caches, increasing the Z-cache size in GPUDet to 260k sets, and disabling deterministic CTA distribution to fully simulate some benchmarks. These changes should only inflate GPUDet's performance for a better performing deterministic baseline. All benchmarks are evaluated with CUDA 8.0, and the convolution benchmarks are evaluated with cuDNN 7.1. We use this version of CUDA because it is the latest version of cuDNN that embeds the PTX, which our version of GPGPU-Sim requires [86].

To validate that DAB produces deterministic results, we extended the baseline GPGPU-Sim and DAB to model non-determinism in GPUs as shown in [12]. In addition, we created a benchmark whose output is sensitive to the order of atomics, and validated that the injected non-determinism leads to different bitwise results on the baseline simulator, while DAB obtained the same results across different runs.

TABLE I: GPGPU-Sim Configuration

# Compute Clusters	40
# SM / Compute Cluster	2
# Streaming Multiprocessors (SM)	80
Max Warps / SM	64
Warp Size	32
Number of Threads / SM	2048
Baseline Scheduler	GTO
Number of Warp Schedulers / SM	4
Number of Registers / SM	65536
Constant Cache Size / SM	64KB
Instruction Cache	128KB, 128B line, 24-way assoc.
L1 Data Cache + Shared Memory	128KB, 128B line, 64-way assoc. LRU
L2 Unified Cache	4.5MB, 128B line, 24-way assoc. LRU
Compute Core Clock	1200 MHz
Interconnect Clock	1200 MHz
L2 Clock	1200 MHz
Memory Clock	850 MHz
DRAM request queue capacity	32
Interconnect Flit Size	40
Interconnect Input Buffer Size	256
Cluster Ejection Buffer Size	32

TABLE II: Graph Configurations for BC and PageRank.

Benchmark	Graph	Nodes	Edges	Atomics PKI
BC	1k	1,024	131,072	6.92
BC	2k	2,048	1,048,576	12.4
BC	FA	10,617	72,176	4.12
BC	foldoc (fol)	13,356	120,238	4.14
BC	amazon0302 (ama)	262,111	1,234,877	0.70
BC	CNR	325,557	3,216,152	0.004
PageRank (PRK)	coAuthor (coA)	299,067	1,955,352	47.2

A. Atomic Workloads

The benchmarks focus on workloads that stress atomic arithmetic operations. Since DAB only affects atomic operations, benchmarks without atomic instructions see no change in performance. Thus, we evaluate the performance impact of DAB on a set of benchmarks from Pannotia [4], and convolution layers using the cuDNN library [14].

BC and PageRank: Pannotia [4] provides push based algorithms for graph applications that use atomic instructions such as Betweenness Centrality (BC), introduced Section II, and PageRank, which extensively uses atomics. The graphs used for evaluation are shown in Table II.

cuDNN Convolutions: We tested backward filter convolutions using cuDNN's Algorithm 0. The evaluated layers are a subset of the ResNet building blocks described in [2] on the ImageNet dataset [87], with batch size 16. Each building block contains three layers, and are repeatedly stacked in order to increase the depth of the network. These layer configurations are shown in Table III. Each layer will be referred to as {Block}_{Lay} (e.g cnv2_1 for layer 1 of block conv2).

VI. Evaluation

Figure 10 presents the overall performance of DAB compared to the non-deterministic baseline and GPUDet, with results

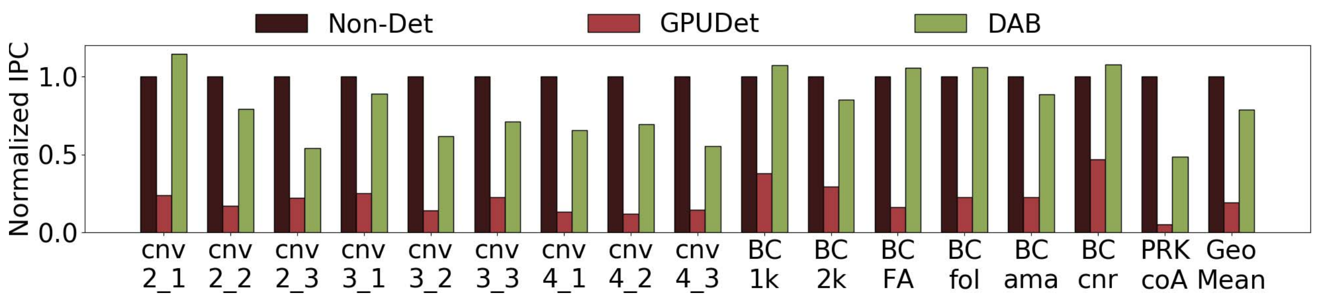
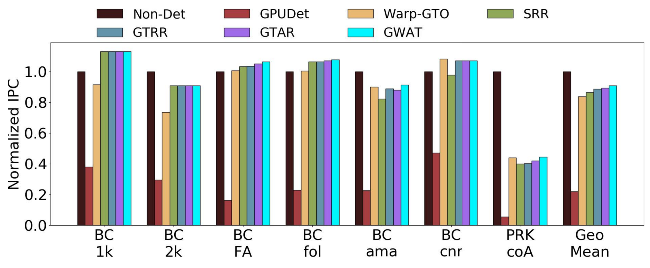


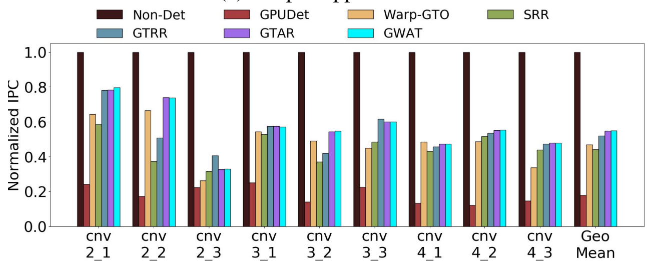
Fig. 10: Performance of DAB (GWAT-64-AF-Coalescing) compared to prior deterministic work.

TABLE III: ResNet Layer Configurations for Convolution

Block	Lay	Input Size ($C \times H \times W$)	Output Size ($C \times H \times W$)	Filter Size ($K \times C \times H \times W$)	Atomics PKI
cnv2_x	1	$256 \times 56 \times 56$	$64 \times 56 \times 56$	$64 \times 256 \times 1 \times 1$	1.08
	2	$64 \times 56 \times 56$	$64 \times 56 \times 56$	$64 \times 64 \times 3 \times 3$	1.09
	3	$64 \times 56 \times 56$	$256 \times 56 \times 56$	$256 \times 64 \times 1 \times 1$	1.72
cnv3_x	1	$512 \times 28 \times 28$	$128 \times 28 \times 28$	$128 \times 512 \times 1 \times 1$	1.70
	2	$128 \times 28 \times 28$	$128 \times 28 \times 28$	$128 \times 128 \times 3 \times 3$	1.70
	3	$128 \times 28 \times 28$	$512 \times 28 \times 28$	$512 \times 128 \times 1 \times 1$	1.96
cnv4_x	1	$1024 \times 14 \times 14$	$256 \times 14 \times 14$	$256 \times 1024 \times 1 \times 1$	3.74
	2	$256 \times 14 \times 14$	$256 \times 14 \times 14$	$256 \times 256 \times 3 \times 3$	3.75
	3	$256 \times 14 \times 14$	$1024 \times 14 \times 14$	$1024 \times 256 \times 1 \times 1$	3.74



(a) Graph Applications



(b) Convolutions

Fig. 11: Performance impact of scheduling.

normalized to stock GPGPU-Sim with GTO scheduling. DAB provides deterministic execution and incurs only a 23% geomean performance slowdown with low area overhead, while GPUDet is 2-4× slower. With 4 schedulers per SM, 64 entries per buffer and 9B per entry, total area overhead of DAB after using L2 cache as a virtual write queue is 2.3 KB per SM (and negligible logic area). Below we analyze DAB's performance in greater depth.

A. *Warp & Scheduler-Level Buffering*

Scheduler-level buffering performs similarly to warp-level buffering but could reduce area overhead up to 16× (In the Figure 11 comparison, area overhead is halved). Thus, the remainder of evaluation focuses on scheduler-level buffering.

1) Scheduling Policies: Figure 11 presents the performance of different scheduling policies (SRR, GTRR, GTAR, GWAT) on graph applications and convolution. We increase buffer capacity to 256 to reduce bottlenecks due to frequent stalls from reaching buffer capacity. In general, more restrictive schedulers such as SRR with scheduler-level buffering incur only a geometric mean of 4% slowdown over WarpGTO. However, scheduler-level buffering matches or exceeds the performance of WarpGTO by up to 7% with more relaxed schedulers such as GTRR, GTAR, and GWAT.

Scheduling policies have a significant effect on the performance of convolutions when using scheduler-level buffering. The varying gap between SRR and the rest of the schedulers can be explained by the number of active warps during runtime. The large gap in cnv2_1, cnv2_2, and cnv3_2 can be attributed to schedulers having 6 warps active, allowing heuristic-based schedulers more options to select a better warp to schedule. For other layers (except cnv3_1), only 4 warps are active, giving heuristic-based schedulers fewer options to schedule a better warp. The relative performance of GTRR can be explained by the number of warps executed per hardware slot. GTRR performs close to heuristic-based schedulers when SMs are not saturated (cnv2_1, cnv3_3), allowing GTRR to run mostly under GTO mode up until atomics, or when only 2 warps are distributed to each hardware slot (cnv3_1, cnv4_1, cnv4_3). For the latter case, only 2 hardware slots are active when the 2nd set of warps are distributed, meaning even if SRR is run, the performance should not deviate much from heuristic-based schedulers. In contrast, for layer 2 of all blocks, GTRR performs closer to SRR than GWAT and GTAR, since each hardware slot now runs more total warps, and only the first executed warp is under GTO mode, while the rest are under SRR mode, causing GTRR's performance to be closer to SRR.

For graph workloads, the relative performance of each scheduler varies between the different types of graphs. For the small, dense graphs (1k, 2k), all schedulers perform the same since there is only one warp to schedule in each scheduler. For the smaller, sparse graphs (FA, foldoc (fol)), the SMs cannot be fully saturated either, leaving each scheduler with at most 2 warps to schedule, so SRR performs very similarly to the other schedulers. For CNR and amazon0302 (ama), SMs are fully saturated, so the gap between SRR and other schedulers starts to grow. However, for BC, only a subset of threads are active at each kernel call, since each kernel operates on one layer of nodes in the breadth-first search tree (refer to Section II-B), meaning many threads and warps may exit without executing any atomics. In addition, even with SMs fully saturated, each hardware slot executes at most 2 warps. These two points allow GTRR to execute under GTO more often, thus performing closer to the heuristic-based schedulers. However, for PageRank (PRK), every thread performs atomic updates at every iteration, and the number of atomics executed per thread varies greatly. With atomics forming an implicit barrier, the irregular atomic pattern causes all schedulers to have non-trivial overheads. This is expected, with prior work showing that even dedicated graph algorithm accelerators struggle to accelerate PageRank [88].

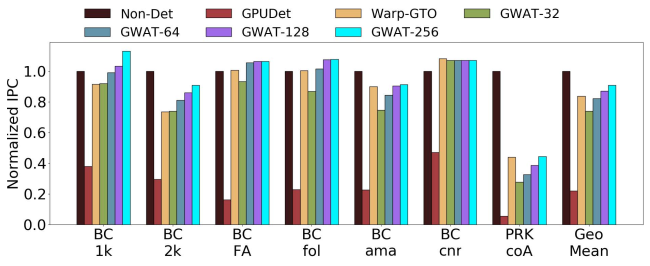
DAB's speedup over the non-deterministic baseline can be attributed to exploiting relaxed atomics. Atomic arithmetic operations are treated like regular arithmetic operations during execute and are allowed to write an entry to the atomic buffers without blocking execution of future atomics. Since GWAT, the most relaxed scheduling scheme, performs the best across the evaluated benchmarks, further evaluation will be shown with only GWAT.

2) Buffer Capacity: Figure 12 shows the effect of buffer capacity on graph applications and convolutions. The evaluated configurations use GWAT scheduler with 32, 64, 128, and 256 buffer entries, labeled as GWAT-32, 64, 128, and 256. Increasing buffer capacity generally improves IPC as it decreases stalls on a full buffer and reduces time spent on flushing. For dense graphs, performance increases with buffer capacity since there are enough non-zero edges to be atomically added. Sparse graphs see a huge gain when increasing buffer capacity from 32 to 64 but lower gains thereafter, since there are not enough atomics to fill up the buffers, meaning the extra capacity is not utilized.

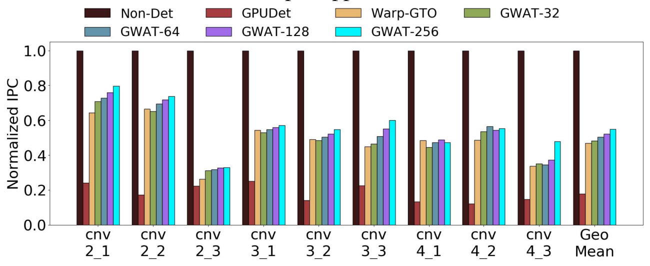
Increasing buffer size for scheduler-level buffering on convolution however, only results in small improvements and even performance loss in some cases. This can be attributed to how convolution workloads are structured. The algorithm performs calculations first, before performing a long sequence of atomic adds. Since the total number of atomic adds do not change with buffer size, having a larger buffer capacity only reduces the frequency of flushing. Additionally, large buffers can cause more atomics to be densely bunched together and pushed to the interconnect at the same time, causing an increase in interconnect stalls. With smaller buffers, flushing is more spread out. Convolution performance is addressed by atomic fusion which is discussed in Section VI-B1.

B. Buffer Optimizations

This section explores the effects of applying optimizations on DAB to further reduce performance overheads.



(a) Graph Applications



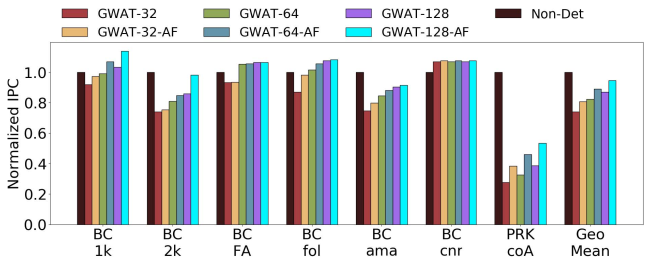
(b) Convolutions

Fig. 12: Performance impact of buffer size.

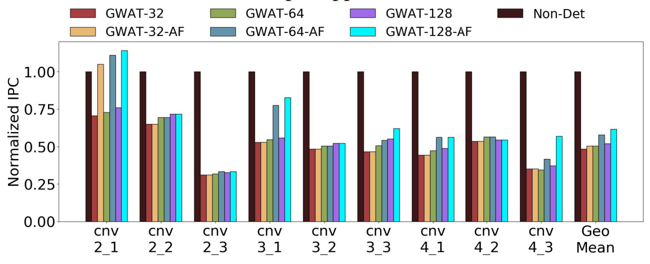
1) Atomic Fusion: Figure 13 shows the effect of enabling atomic fusion on graph applications and convolutions. Atomic fusion increases performance for graph applications for all evaluated graphs, since it reduces the number of atomic operations performed at the ROP. Atomic fusion also increases the effective capacity of the atomic buffers as multiple atomic operations to the same address now only occupy 1 entry instead of multiple entries in the buffer. This results in fewer stalls due to the atomic buffer being full. Additionally, the GPU will have fewer interconnect stalls it flushes fewer buffer entries. Atomic fusion has a lesser effect with larger buffer sizes since most atomic operations are already able to fit within the large buffer without atomic fusion. Thus the extra effective capacity from atomic fusion does not result in better performance.

Atomic fusion also increases performance for most convolution layers similar to graph applications. Most layers do not see any improvement for the 32-entry case, since 2 warps of a CTA are mapped to a scheduler, meaning 64 unique addresses are written to before reuse occurs. However, for Layer 2 of each block, atomic fusion does not improve performance.This is due to misalignment of CTAs that leads to no buffer entry reuse. For these layers, the filter is evenly partitioned into 18 sections (according to Section IV-E), and CTAs whose ids are congruent *modulo 18* write to the same partition. However, with 80 SMs, the CTAs that write to the same partition are never assigned to the same scheduler under the static partition scheme described in Section IV-C5. In order to force atomic fusion, we evaluated these layers by assigning CTAs to 72 cores instead, and obtained a speedup over using 80 cores, despite using 8 fewer cores, as demonstrated in Figure 14.

2) Offset Flushing: From Section VI-A1, we see DAB does poorly on cnv2_3. This is because every CTA atomically writes to the same memory addresses. While this is exploited at the



(a) Graph Applications



(b) Convolutions

Fig. 13: Atomic Fusion on scheduler-level Buffering

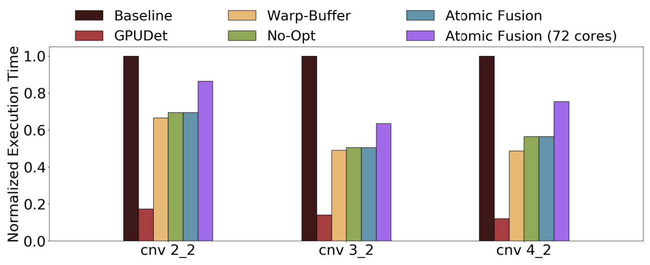


Fig. 14: Effects of "gating" SMs on GWAT-64-AF

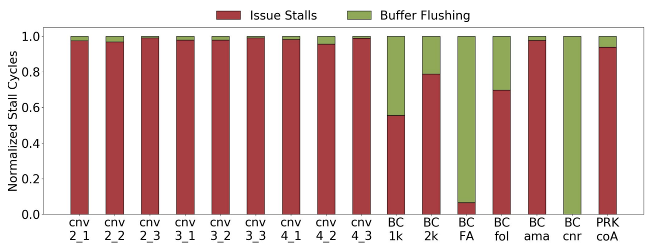


Fig. 15: Performance Overhead of DAB

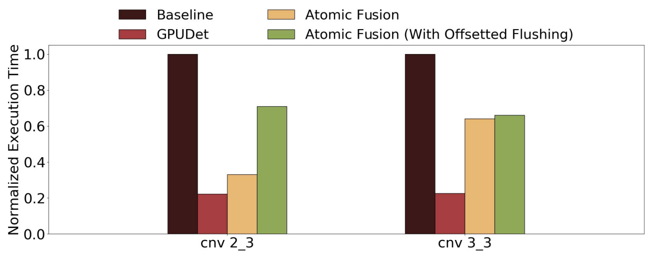


Fig. 16: Effect of offset flushing on GWAT-64-AF

intra-core level by atomic fusion, at the inter-core level, if every buffer flushes to the same set of memory addresses in the same order (hence, the same memory partitions), the interconnect becomes congested, which would lead to longer buffer flushes and more issue stalls due to full buffers. We remedy this with *offset flushing*, where each SM starts flushing at a different index to spread out the writes to each memory partition at a given time. Since both SM id and the buffer contents are deterministic, having some SMs start flushing at a different index preserves determinism. Figure 16 demonstrates speedup of offset flushing. Every SM with an even SM id starts flushing at the 32^{nd} index. Applying *offset flushing* to cnv3_3, where every 4 CTAs write to the same set of memory addresses yields minimal performance gain, hinting a lack of congestion.

3) Flush Coalescing: Coalescing buffer flushes on convolutions for GWAT-64-AF improves performance by a geomean of 13%, shown in Figure 17. Since atomic instructions in convolution access strided memory locations, buffer flushes that access the same cache sector can be coalesced to a single transaction, reducing memory traffic (Section IV-F). However, graph workloads did not improve much due to irregular data access patterns. Figure 15 breaks down the different overheads of DAB for different benchmarks.

4) Limitation Study: In this section, we relax various constraints of DAB and observe its impact on performance. While these relaxed versions are no longer deterministic, they help identify bottlenecks in DAB. In Figure 18, we first relax the constraint of reordering atomics at the memory partition and instead, send the atomics to the ROP unit in the order they arrive at the memory partition (DAB-NR). Next, we relax the constraint of not allowing flushes to overlap (described in Section IV-D, DAB-NR-OF). Finally, we relax DAB by lowering the granularity of buffer flushing from GPU to clusters, meaning each individual cluster flushes their buffers independently as they become full, relaxing the implicit barrier imposed across SMs (DAB-NR-CIF). From Figure 18, we observe that relaxing the last constraint usually provides the most speedup, implying that the implicit barriers across SMs hamper performance, especially for graph benchmarks that have irregular atomic accesses. Note that naively implementing the same reordering scheme for this relaxed version of buffer flushing may lead to intractable and an unbounded number of buffered transactions at the memory partition, and more sophisticated ordering methods are required at the memory partition in order to enforce determinism.

VII. Related Work

Previous sections discussed the design and performance of GPUDet [12], a proposal for deterministic GPU hardware and the most relevant point of comparison. Here we survey additional related work in determinism for CPU programs.

Software Determinism Schemes Kendo [9] introduced *weak determinism*, which tackles non-determinism caused by races on synchronization objects (like lock acquires) by relying on deterministic logical clocks. While Kendo is feasible on CPUs, it is unlikely to scale up to GPU thread counts, and does not

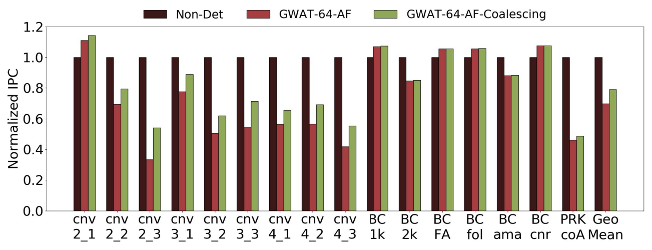


Fig. 17: Coalescing Buffer Flushes on Convolutions

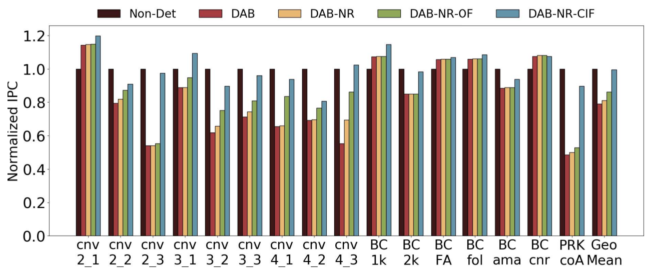


Fig. 18: DAB with different constraints relaxed

take into account the specific semantics of GPU atomics as DAB does. The CoreDet [10] compiler and runtime system uses the same algorithm as that in GPUDet [12], leading to similar scalability bottlenecks. More recent software schemes [67], [89], [90] have coupled the Kendo scheduling algorithm with sophisticated OS virtual memory support to reduce overhead of per-thread store buffers. There are a wide range of deterministic parallel programming languages which leverage type systems [91], [92], [93] or functional programming [94], [95] to enable high-performance determinism within a restricted programming model.

Hardware Determinism Schemes A number of systems [11], [96], [97] have explored deterministic CPU hardware support. These schemes adopt the same deterministic scheduling technique with global barriers as used in CoreDet and GPUDet, which imposes a scalability bottleneck, especially in the presence of frequent atomic operations. DAB takes inspiration from schemes like Calvin [11] and RCDC [97] which used relaxed memory consistency models to improve performance, similar to how we exploit semantics of GPU atomics.

Deterministic Floating-Point Collange et al. [13], [98] proposed software techniques to address floating point rounding errors. They use a wide super-accumulator to cover the whole range of 32 bit floating point numbers. However, it incurs up to 10× performance overhead compared to unordered floating point operations, while also imposing high area overhead. Thus, DAB tackles the problem of reproducibility by ensuring a deterministic order of floating point atomic operations.

VIII. Conclusion

In this paper, we presented DAB, a GPU architecture that provides deterministic execution with low overheads for reduction workloads like graph algorithms and machine learning. DAB exploits the GPU's relaxed atomic semantics and an assumption of data-race freedom to enable the use of isolated atomic buffers for atomic operations, allowing atomics to be performed deterministically in parallel. Coupled with determinism-aware warp scheduling inside each core, these buffers can be area-efficient while eliminating the non-determinism caused by floating point rounding. Simulation results show DAB outperforms GPUDet [12], a state-of-the-art deterministic GPU baseline, by a significant margin of 2-4×.

Acknowledgment

We would like to thank Johnathan Lew for enabling cuDNN support in GPGPU-Sim. This research is funded in part by the Computing Hardware for Emerging Intelligent Sensory Applications (COHESA) project. COHESA is financed under the National Sciences and Engineering Research Council of Canada (NSERC) Strategic Networks grant number NETGP485577-15. This research was also supported by US NSF grant #1703541 and the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA. Professor Aamodt serves as a consultant for Huawei Technologies Canada and Professor Sinclair serves as a Visiting Research Scholar for AMD Research.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proc. Conf. and Workshop on Neural Information Processing Systems (NIPS)*, 2012, pp. 1097–1105.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[3] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," *CoRR*, vol. abs/1901.00596, 2019.

[4] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *Proc. IEEE Symp. on Workload Characterization (IISWC)*, 2013, pp. 185–195.

[5] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to End Learning for Self-Driving Cars," *CoRR*, vol. abs/1604.07316, 2016.

[6] P. Nagarajan, G. Warnell, and P. Stone, "Deterministic Implementations for Reproducibility in Deep Reinforcement Learning," *CoRR*, vol. abs/1809.05676, 2018.

[7] H. Kuwajima, H. Yasuoka, and T. Nakae, "Open Problems in Engineering and Quality Assurance of Safety Critical Machine Learning Systems," *CoRR*, vol. abs/1812.03057, 2018.

[8] M. B. A. McDermott, S. Wang, N. Marinsek, R. Ranganath, M. Ghassemi, and L. Foschini, "Reproducibility in Machine Learning for Health," *CoRR*, vol. abs/1907.01463, 2019.

[9] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient Deterministic Multithreading in Software," in *Proc. ACM Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*, 2009, pp. 97–108.

[10] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution," in *Proc. ACM Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*, 2010, pp. 53–64.

[11] D. Hower, P. Dudnik, M. D. Hill, and D. A. Wood, "Calvin: Deterministic or not? Free will to choose," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2011, pp. 333–334.

[12] H. Jooybar, W. W. Fung, M. O'Connor, J. Devietti, and T. M. Aamodt, "GPUDet: A Deterministic GPU Architecture," in *Proc. ACM Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*, 2013.

[13] D. Defour and S. Collange, "Reproducible floating-point atomic addition in data-parallel environment," in *Proc. Federated Conf. on Computer Science and Information Systems (FedCSIS)*, 2015, pp. 721–728.

[14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *CoRR*, vol. abs/1410.0759, 2014.

[15] L. Li and A. Talwalkar, "Random Search and Reproducibility for Neural Architecture Search," in *Proc. Conf. on Uncertainty in Artificial Intelligence (UAI)*, 2019, p. 129.

[16] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 93–104.

[17] A. L. Beam, A. K. Manrai, and M. Ghassemi, "Challenges to the Reproducibility of Machine Learning Models in Health Care," *JAMA*, vol. 323, no. 4, pp. 305–306, 2020.

[18] O. E. Gundersen and S. Kjensmo, "State of the Art: Reproducibility in Artificial Intelligence," in *Proc. Conf. on Artificial Intelligence (AAAI)*, 2018, pp. 1644–1651.

[19] 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 *IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS, 2020.

[20] S. V. Adve and M. D. Hill, "Weak Ordering - A New Definition," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 1990.

[21] ——, "A Unified Formalization of Four Shared-Memory Models," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, pp. 613–624, 1993.

[22] S. V. Adve, "Designing Memory Consistency Models for Shared-Memory Multiprocessors," Ph.D. dissertation, University of Wisconsin, Madison, 1993.

[23] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*, 2008, p. 68.

[24] L. Howes and A. Munshi, "The OpenCL Specification, Version 2.0," Khronos Group, 2015.

[25] 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 *Proc. ACM Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*, 2014, pp. 427–440.

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

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

[28] J. Manson, W. Pugh, and S. V. Adve, "The Java Memory Model," in *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, 2005, pp. 378–391.

[29] M. D. Sinclair, J. Alsop, and S. V. Adve, "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2017, pp. 161–174.

[30] K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," in *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1724–1734.

[31] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *CoRR*, vol. abs/1810.04805, 2018.

[32] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation," *CoRR*, vol. abs/1609.08144, 2016.

[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proc. IEEE Int'l Conf. on Computer Vision (ICCV)*, 2015, pp. 1026–1034.

[35] C. Olah, "Understanding LSTM Networks," http://colah.github.io/posts/2015-08-Understanding-LSTMs/, August 2015.

[36] B. Bakker, "Reinforcement Learning with Long Short-Term Memory," in *Proc. Conf. and Workshop on Neural Information Processing Systems (NIPS)*. MIT Press, 2001, pp. 1475–1482.

[37] W. Chan and I. Lane, "Deep Recurrent Neural Networks for Acoustic Modelling," *CoRR*, vol. abs/1504.01482, 2015.

[38] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[39] A. Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks," http://karpathy.github.io/2015/05/21/rnn-effectiveness/, May 2015.

[40] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 35, no. 8, pp. 1798–1828, 2013.

[41] J. Frankle and M. Carbin, "The Lottery Ticket Hypothesis: Training Pruned Neural Networks," *CoRR*, vol. abs/1803.03635, 2018.

[42] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4013–4021.

[43] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[44] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[45] J. A. Danowski and N. T. Cepela, "Automatic Mapping of Social Networks of Actors from Text Corpora: Time Series Analysis," in *Proc. IEEE/ACM Int'l Conf. on Advances in Social Networks Analysis and Mining (ASONAM)*, 2009, pp. 137–142.

[46] W. Xiong, L. Xie, S. Zhou, and J. Guan, "Active learning for protein function prediction in protein–protein interaction networks," *Neurocomputing*, vol. 145, pp. 44–52, 2014.

[47] L. Maccari and R. L. Cigno, "Pop-routing: Centrality-based tuning of control messages for faster route convergence," in *Proc. IEEE Int'l Conf. on Computer Communications (INFOCOM)*, 2016, pp. 1–9.

[48] J. Magnusson and T. Kvernvik, "Subscriber Classification within Telecom Networks Utilizing Big Data Technologies and Machine Learning," in *Proc. Int'l Workshop on Big Data, IoT Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications (BigMine)*, 2012, pp. 77–84.

[49] P. Moradi, M. E. Shiri, and N. Entezari, "Automatic Skill Acquisition in Reinforcement Learning Agents Using Connection Bridge Centrality," in *Int'l Conf. on Future Generation Communication and Networking (FGCN)*, 2010.

[50] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[51] Z. Shi and B. Zhang, "Fast network centrality analysis using GPUs," *BMC Bioinformatics*, vol. 12, no. 1, p. 149, 2011.

[52] J. H. van der Linden, G. A. Narsilio, and A. Tordesillas, "Machine learning framework for analysis of transport through complex networks in porous, granular media: A focus on permeability," *Phys. Rev. E*, vol. 94, p. 022904, Aug 2016.

[53] K. Plaimas, R. Eils, and R. König, "Identifying essential genes in bacterial metabolic networks with machine learning methods," *BMC systems biology*, vol. 4, no. 1, p. 56, 2010.

[54] J. M. Fletcher and T. Wennekers, "From structure to activity: using centrality measures to predict neuronal activity," *International journal of neural systems*, vol. 28, no. 02, p. 1750013, 2018.

[55] H. Cheng, S. Newman, J. Goñi, J. S. Kent, J. Howell, A. Bolbecker, A. Puce, B. F. O'Donnell, and W. P. Hetrick, "Nodal centrality of functional network in the differentiation of schizophrenia," *Schizophrenia research*, vol. 168, no. 1-2, pp. 345–352, 2015.

[56] D. Koschützki and F. Schreiber, "Centrality Analysis Methods for Biological Networks and Their Application to Gene Regulatory Networks," *Gene Regulation and Systems Biology*, vol. 2, pp. GRSB–S702, 2008.

[57] O. Simsek and A. Barto, "Betweenness centrality as a basis for forming skills," *Technical report, University of Massachusetts, Department of Computer Science*, 2007.

[58] M. D. Sinclair, J. Alsop, and S. V. Adve, "HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs," in *IEEE International Symposium on Workload Characterization*, ser. IISWC, October 2017.

[59] J. A. Stuart and J. D. Owens, "Efficient Synchronization Primitives for GPUs," *CoRR*, vol. abs/1110.4623, 2011. [Online]. Available: http://arxiv.org/abs/1110.4623

[60] A. ElTantawy and T. M. Aamodt, "Warp scheduling for fine-grained synchronization," in *2018 IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA, 2018, pp. 375–388.

[61] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-Grained Synchronizations and Dataflow Programming on GPUs," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 109–118. [Online]. Available: https://doi.org/10.1145/2751205.2751232

[62] G. Lacey, private communication, Aug. 2018.

[63] I. J. Egielski, J. Huang, and E. Z. Zhang, "Massive Atomics for Massive Parallelism on GPUs," in *Proc. ACM SIGPLAN Int'l Symp. on Memory Management (ISMM)*, 2014, pp. 93–103.

[64] D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, Mar. 1991.

[65] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd ed. Cambridge University Press, 1992.

[66] K. Doertel, "Best Known Method: Avoid Heterogeneous Precision in Control Flow Calculations," 2013. [Online]. Available: https://software.intel.com/en-us/articles/best-known-method-avoid-heterogeneous-precision-in-control-flow-calculations

[67] T. Merrifield, S. Roghanchi, J. Devietti, and J. Eriksson, "Lazy Determinism for Faster Deterministic Multithreading," in *Proc. ACM Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*, 2019, pp. 879–891.

[68] V. Miller, "Determinism in GPU Programs-Real Time Applications on the NVIDIA Jetson TK1," Master's thesis, University of North Carolina, 2016.

[69] E. Bardsley and A. F. Donaldson, "Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels," in *NASA Formal Methods*, 2014, pp. 230–245.

[70] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, and G. L. Lee, "Determinism and reproducibility in large-scale HPC systems," in *Workshop on Determinism and Correctness in Parallel Programming (WODET)*, 2013.

[71] J. Yang, H. Cui, and J. Wu, "Methods, systems, and media for providing determinism in multithreaded programs," 2016, US Patent 9,454,460.

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

[73] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, "Remote-Scope Promotion: Clarified, Rectified, and Verified," in *Proc. ACM SIGPLAN Int'l Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2015, pp. 731–747.

[74] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras, "Building Heterogeneous Unified Virtual Memories (UVMs) without the Overhead," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, Mar. 2016.

[75] M. D. Sinclair, J. Alsop, and S. V. Adve, "Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2015, pp. 647–659.

[76] M. Martin, C. Blundell, and E. Lewis, "Subtleties of Transactional Memory Atomicity Semantics," *IEEE Computer Architecture Letters*, vol. 5, no. 2, p. 17, July 2006.

[77] L. Lu and M. L. Scott, "Toward a Formal Semantic Framework for Deterministic Parallel Programming," in *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS)*, 2011, pp. 460–474.

[78] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2012, p. 72–83.

[79] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John, "The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2010, pp. 72–82.

[80] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 3–16. [Online]. Available: https://doi.org/10.1145/3018743.3018748

[81] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *Proc. ACM Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*, 2015, p. 593–606.

[82] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling Preemptive Multiprogramming on GPUs," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. IEEE Press, 2014, p. 193–204.

[83] B. Wu, X. Liu, X. Zhou, and C. Jiang, "FLEP: Enabling Flexible and Efficient Preemption on GPUs," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 483–496. [Online]. Available: https://doi.org/10.1145/3037697.3037742

[84] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, 2009, pp. 163–174.

[85] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA, 2020, pp. 473–486.

[86] 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. [Online]. Available: http://arxiv.org/abs/1811.08933

[87] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, , and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *Int'l Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[88] A. Segura, J. Arnau, and A. González, "SCU: a GPU stream compaction unit for graph processing," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2019, pp. 424–435.

[89] T. Merrifield and J. Eriksson, "Conversion: multi-version concurrency control for main memory segments," in *Proc. European Conf. on Computer Systems (EuroSys)*, 2013, pp. 127–139.

[90] T. Merrifield, J. Devietti, and J. Eriksson, "High-performance Determinism with Total Store Order Consistency," in *Proc. European Conf. on Computer Systems (EuroSys)*, 2015.

[91] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for deterministic parallel Java," in *Proc. ACM SIGPLAN Int'l Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2009, pp. 97–116.

[92] R. L. Bocchino and V. S. Adve, "Types, regions, and effects for safe programming with object-oriented parallel frameworks," in *Proc. European Conference on Object-Oriented Programming (ECOOP)*, 2011, pp. 306–332.

[93] M. C. Rinard and M. S. Lam, "The design, implementation, and evaluation of Jade," *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 3, pp. 483–545, May 1998.

[94] M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow, "Data Parallel Haskell: a status report," in *ACM Workshop on Declarative Aspects of Multicore Programming (DAMP)*, 2007, pp. 10–18.

[95] L. Kuper, A. Turon, N. R. Krishnaswami, and R. R. Newton, "Freeze after writing: Quasi-deterministic parallel programming with LVars," in *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, 2014, pp. 257–270.

[96] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic Shared Memory Multiprocessing," in *Proc. ACM Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*, 2009, p. 85.

[97] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman, "RCDC: a relaxed consistency deterministic computer," in *Proc. ACM Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*, 2011, pp. 67–78.

[98] 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.