GPUDet: A Deterministic GPU Architecture

Hadi Jooybar[1] Wilson W. L. Fung[1] Mike O'Connor[2] Joseph Devietti[3] Tor M. Aamodt[1]

University of British Columbia[1] Advanced Micro Devices, Inc. (AMD)[2] University of Washington[3]

jooybar@ece.ubc.ca, wwlfung@ece.ubc.ca, mike.oconnor@amd.com, devietti@cs.washington.edu, aamodt@ece.ubc.ca

Abstract

Nondeterminism is a key challenge in developing multithreaded applications. Even with the same input, each execution of a multithreaded program may produce a different output. This behavior complicates debugging and limits one's ability to test for correctness. This non-reproducibility situation is aggravated on massively parallel architectures like graphics processing units (GPUs) with thousands of concurrent threads. We believe providing a deterministic environment to ease debugging and testing of GPU applications is essential to enable a broader class of software to use GPUs.

Many hardware and software techniques have been proposed for providing determinism on general-purpose multi-core processors. However, these techniques are designed for small numbers of threads. Scaling them to thousands of threads on a GPU is a major challenge. This paper proposes a scalable hardware mechanism, GPUDet, to provide determinism in GPU architectures. In this paper we characterize the existing deterministic and nondeterministic aspects of current GPU execution models, and we use these observations to inform GPUDet's design. For example, GPUDet leverages the inherent determinism of the SIMD hardware in GPUs to provide determinism within a wavefront at no cost. GPUDet also exploits the Z-Buffer Unit, an existing GPU hardware unit for graphics rendering, to allow parallel out-of-order memory writes to produce a deterministic output. Other optimizations in GPUDet include deterministic parallel execution of atomic operations and a workgroup-aware algorithm that eliminates unnecessary global synchronizations.

Our simulation results indicate that GPUDet incurs only 2× slowdown on average over a baseline nondeterministic architecture, with runtime overheads as low as 4% for compute-bound applications, despite running GPU kernels with thousands of threads. We also characterize the sources of overhead for deterministic execution on GPUs to provide insights for further optimizations.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures—GPU Architecture; D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Design, Performance, Reliability

Keywords GPU, Deterministic Parallelism

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

1. Introduction

Massively parallel processors are designed to run thousands of threads. The energy efficiency of these processors has driven their widespread popularity for many inherently-parallel applications, such as computer graphics. We anticipate that increasing demands for energy efficiency will motivate a broader range of applications with more complex data structures and inter-task communication to use these processors.

Unfortunately, developing parallel applications that can utilize such architectures is hampered by the challenge of coordinating parallel tasks, especially in ensuring that a parallel program behaves correctly for every possible combination of interactions among these tasks. With the thousands of threads required to utilize current massively parallel architectures like GPUs, the number of interactions to consider is substantial. Nondeterminism makes this challenge even harder, because it thwarts attempts to reproduce anomalous behavior or to provide guarantees about future executions.

Providing deterministic execution for GPU architectures is a crucial aid for simplifying parallel programming at scale. Determinism simplifies the debugging process by making program behavior reproducible for a given input. Determinism is also useful once code is deployed, to reproduce bugs that manifest in production settings. Finally, determinism amplifies the power of testing, by providing a guarantee that once a particular input has passed its tests, those tests will continue to pass in the future. Current nondeterministic parallel architectures provide no such guarantee, which allows testing to provide only probabilistic guarantees.

The key challenge of determinism is providing these properties with acceptable performance overhead. To this end, different levels of determinism have been proposed in the literature [35]. We are proponents of *strong determinism*, which provides determinism even in the presence of data races. Some deterministic schemes require data-race freedom [35], offering determinism in a more limited context but with lower performance overheads. In contrast, a strongly deterministic system makes no assumptions about the program it is running, which is well-suited to dealing with the buggy programs that need determinism the most. The programmability of GPUs can benefit greatly from strong determinism.

1.1 Debugging with a Deterministic GPU

Many existing GPU workloads are embarrassingly parallel applications free of inherent data-races. It may be tempting to dismiss the benefit of deterministic execution on GPUs for these applications. However, these applications usually feature complex data tiling to improve data locality. Subtle mistakes during development can result in nondeterministic bugs that are hard to detect. A deterministic GPU allows programmers to analyze such bugs more effectively with a consistent output.

Indeed, race conditions have appeared in GPGPU algorithms from the published literature. To illustrate how a deterministic GPU can simplify debugging, we use a buggy version of

breadth-first-search (BFS) graph traversal distributed by Harish and Narayanan [21] (BFSr). Below is a simplified version of the kernel function in BFSr, which contains a race condition.

```
0: __global__ void BFS_step_kernel(...) {
1:   if( active[tid] ) {
2:     active[tid] = false;
3:     visited[tid] = true;
4:     foreach (int id = neighbour_nodes) {
5:       if( visited[id] == false ) {
6:         cost[id] = cost[tid] + 1;
7:         active[id] = true;
8:         *over=true;
9: } } } }
```

Presumably, this kernel is designed to traverse one level of the breadth-first-search nodes during each kernel launch. Each thread (`tid` = thread ID) is assigned a node in the graph. In each kernel launch (at each step), each thread updates the cost of its neighbouring nodes (`cost[id]`) that have not been visited, and sets their active flag (`active[id]`), which indicates these nodes should update their own neighbours during the next kernel launch. A race condition can occur since a thread can be reading its `active[tid]` and `cost[tid]` (at line 1 and 6) while its neighbour is updating these locations (at line 6 and 7). This causes `cost[id]` to be updated nondeterministically, which can lead to incorrect output. Harish and Narayanan have discovered the race conditions in this code and have posted a corrected version, which is now part of the Rodinia benchmark suite [14].

With a deterministic GPU, the programmer can observe the same incorrect output across different runs. This allows him to set a watch point in a GPU debugger at the location in `cost[]` that has an incorrect output value. The programmer can then pause the execution at every modification to this location and identify the threads responsible for the incorrect output value. Notice that this approach will not work with a nondeterministic GPU. The programmer cannot be sure that the race condition would occur at the same memory location in the second run via the debugger. This difficulty with nondeterministic parallel architecture has motivated the supercomputing community to create extensive logging tools that capture activities across the entire system, and analyze the (enormous) logs for errors [3]. We argue that providing a deterministic GPU to programmers is a more effective solution.

1.2 Our Contribution: GPUDet

This paper introduces GPUDet, which is the first deterministic massively parallel architecture. GPUDet provides strong determinism for current-generation GPU architectures. GPUDet leverages the approach suggested by Bergan et al. [6] for multi-core CPUs. In this approach, threads are executed in an isolated memory space, communicating with other threads only at deterministic, fixed intervals. Thread isolation is realized by appending all stores to a private buffer instead of directly updating global memory. Thread updates are made globally visible via a commit process which leverages existing GPU Z-Buffer hardware to execute in a deterministic but highly parallel fashion. Read-modify-write operations that need to be made globally visible have their execution deferred until this communication phase. GPUDet executes these operations in parallel and maintains a deterministic order among them by leveraging the ordering property in existing GPU memory systems.

Our key contributions in this paper are:

1. We propose GPUDet, the first hardware model for a fully deterministic GPU architecture.
2. We characterize the inherently deterministic and nondeterministic aspects of the GPU execution model and propose optimizations to GPUDet that leverage these inherent deterministic properties.

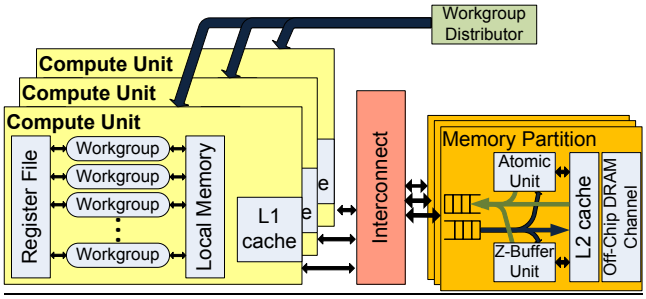

Figure 1. Baseline GPU Architecture

3. Our design exploits existing GPU hardware for Z-buffering to accelerate deterministic parallel committing of store buffers.
4. We introduce a workgroup-based quantum formation algorithm to enable larger quanta, which better amortized the overheads of quantum formation.
5. We exploit the point-to-point ordering in GPU memory subsystem to regain parallelism within each compute unit for read-modify-write operations that are executed in serial mode.

The rest of the paper is organized is follows. Section 2 summarizes our baseline GPU architecture and discusses sources of nondeterminism in GPUs. Section 3 summarizes existing techniques for providing deterministic execution and highlights the challenges in applying them on GPUs. Section 4 describes GPUDet. Section 5 presents a comprehensive evaluation of GPUDet. Section 6 discusses the related work. Section 7 concludes the paper.

2. Baseline Architecture

This section summarizes the details of our baseline GPU architecture relevant to deterministic execution on GPUs. Figure 1 shows an overview of our baseline GPU architecture modeled after the NVIDIA Fermi GPU architecture [32]. In this paper, we use OpenCL [24] terminology. However, to make the paper easier to read we use *thread* instead of *work item*. A GPU program (CUDA [33] or OpenCL [24]) starts running on the CPU and launches parallel compute kernels onto the GPU for processing. Each kernel launch consists of a hierarchy of threads organized in equal-sized *workgroups*. All threads in a kernel launch execute the same *kernel function*, while each thread may follow a unique execution path (with some performance penalty). Threads within a workgroup can synchronize quickly via *workgroup barriers* and communicate via an on-chip scratch-pad memory called the *local memory*. Communication among threads in different workgroups occurs through *global memory* located in off-chip DRAM. Each thread has exclusive access to its own *private memory* for non-shared data.

Our GPU microarchitecture consists of a set of compute units that access multiple partitions (i.e., channels) of off-chip DRAM via an interconnection network. Each compute unit can sustain up to 1536 threads in its register file. It has a scratch-pad memory that is partitioned among the different workgroups that run concurrently on the unit. The large register file allows a compute unit to freely context switch between different threads to tolerate long-latency accesses to global memory. The *workgroup distributor* on the GPU ensures that a compute unit has sufficient hardware resources to sustain all threads in a workgroup before dispatching the entire workgroup onto the unit. A GPU program may launch a kernel with more threads than the GPU hardware may execute concurrently. In this case, the GPU hardware starts executing as many workgroups as on-chip hardware resources permit, and dispatches the remaining ones as resources are released by the workgroups that have finished execution.

2.1 SIMT Execution Model

GPU architectures use Single-Instruction, Multiple-Data (SIMD) hardware for enhancing computational efficiency. Rather than exposing the SIMD hardware directly to the programmer, GPUs employ a Single-Instruction, Multiple-Threads (SIMT) execution model [29]. It groups scalar threads specified by the programmer into SIMD execution groups called *wavefronts* [2] (warps [33] in NVIDIA nomenclature). Threads in a wavefront execute in lockstep on the SIMD hardware. Since each thread can follow a unique execution path, a wavefront may *diverge* after a branch. The GPU hardware automatically serializes the execution of subsets of threads that have diverged to different control flow paths. Each wavefront has a SIMT stack [15, 20] maintaining the different control flow paths that still need to be executed. This allows for nested control flow divergence.

2.2 Memory Subsystem

Individual scalar accesses to global memory from threads in a wavefront are coalesced into wider accesses to 32, 64, or 128-byte chunks. The L1 data cache services one coalesced access per cycle. It caches data from both global memory and private memory, but with different policies. It acts as a writeback cache for accesses to private memory. Writes to global memory evict any line that hits in the L1 cache to make the memory updates visible to all compute units. However, the GPU hardware does not maintain coherence among the L1 data caches in different compute units. Each unit is responsible for purging stale data from its L1 cache.

Accesses that miss at the L1 cache are sent to the corresponding memory partition that contains the requested data. Each memory partition is responsible for its portion of the memory space. It contains a slice of the shared L2 cache. Accesses that miss at the L2 cache slices are in turn serviced by the off-chip DRAM controlled by the partition.

To support a rich set of synchronization primitives, the GPU programming model provides atomic operations, which are read-modify-write operations that update a single 32/64-bit word in the global memory. The wavefront sends each operation to its corresponding memory partition as with global memory accesses. Atomic operations are executed at the memory partitions with a set of specialized atomic operation units that operate directly on the L2 cache slices.

2.3 Sources of Nondeterminism in GPU Architectures

The standalone-accelerator nature of current GPU architectures helps isolate them from some sources of nondeterminism, like interrupt handlers and context switches, present in more general-purpose architectures. However, these sources will likely be increasingly relevant to future GPU architectures as GPUs become more general-purpose. To illustrate that nondeterminism exists on GPUs, we have developed GPU-Racey, a CUDA deterministic stress test, based on CPU deterministic stress test Racey [22]. Racey computes a signature that is extremely sensitive to the relative ordering of memory accesses among threads. Across multiple runs, it should generate different signatures on a nondeterministic system, and the same signature on a deterministic system. Running GPU-Racey on real GPU hardware (a Quadro FX 5800 and a Tesla C2050) with two or more concurrent wavefronts (warps) produces nondeterministic outputs on different runs. While the exact sources of nondeterminism in commercial GPUs are undocumented, we have postulated several potential sources.

First, each GPU usually consists of multiple clock domains [34], with each domain running at its optimal frequency. The synchronizer circuits interfacing between these domains can introduce nondeterministic delays to cross-domain messages due to phase drifting among the different clocks [38]. This source of nondeterminism may be exacerbated in the future as more aggressive power management features such as dynamic voltage and frequency scaling (DVFS) are introduced to GPU architectures.

Second, the access timing to off-chip memory on a GPU depends on the physical location of the data. Accesses to different memory partitions have an observable delay difference [45]. With the GPU shared by many different processes in the system, it is improbable for the GPU driver to starting in the same memory partition for every run of an application. The variance in DRAM cells may also encourages more adaptive refreshing techniques that change the refreshing interval according to the dynamic status of different cells [30]. This introduces nondeterministic delays for DRAM accesses.

Third, arbitration/scheduling units with uninitialized states can introduce nondeterminism by ordering thread execution or memory requests in an application differently between different runs. This includes the hardware wavefront schedulers in each compute unit, the workgroup distributor and the arbiters in the interconnection network. Although these units are reset to a known initial state at power up, the operating system is unlikely to reset them between kernel launches. This makes the states of these units dependent on the previously executed workload, which is usually not predictable.

Finally, as circuit process technology scales, transient faults in memory cells have become increasingly common. Transient failure in either on-chip or off-chip memory can trigger recover routines randomly [32], thus introducing nondeterministic latencies to memory accesses.

3. GPU Deterministic Execution: Background and Challenges

Many hardware and software techniques have been proposed for providing determinism on general-purpose multi-core processors [4, 6, 7, 16, 17, 23, 31, 35]. However, these techniques are designed for systems supporting tens, not thousands, of threads. Issues of serialization, global synchronization and per-thread fixed costs have much greater impact at GPU-scale than at CPU-scale.

In this section, we first summarize the deterministic multiprocessor schemes CoreDet and Calvin [6, 23], which serve as a basis for GPUDet. We then discuss the major challenges these prior approaches face in scaling up to work on GPU architectures. We have chosen to start with Calvin and CoreDet over more recent proposals because their simplicity – particularly Calvin's in-order pipeline design that avoids hardware speculation – is a good match for current GPU architectures. Furthermore, subsequent improvements in deterministic execution hardware provide lighter-weight synchronization operations [17], but do not address the scaling challenges inherent in making GPU architectures deterministic. Incorporating these subsequent improvements with GPUDet would improve performance for the applications that have frequent communications among threads.

3.1 Background: CoreDet and Calvin

CoreDet [6] is a compiler and accompanying runtime system that provides *strong determinism* for multithreaded programs. A multithreaded program compiled with CoreDet always produces the same result, even if the program contains data races. Calvin [23] is a processor design that provides deterministic execution, using an algorithm very similar to CoreDet's but incorporating the specifics of multi-core hardware. We provide a Calvin-centric overview as its hardware implementation is most similar to the GPUDet approach.

The basic mechanism both schemes use to enforce determinism is described in Figure 2. The execution of the program is divided into *quanta*, deterministically-sized sequences of instructions, e.g. every 1000th instruction executed by a wavefront marks the start

of a new quantum. Each quantum is in turn composed of three phases. The first phase is *parallel mode*, wherein each processor executes in isolation. Calvin's special coherence protocol ensures that a processor sees only its own writes but not those of remote processors. In effect, each processor executes a single-threaded program which is inherently deterministic. At the end of parallel mode, all processors enter a global barrier before transitioning into *commit mode*. In commit mode the writes from each processor are made globally visible, using a deterministic parallel commit algorithm that maps well to GPU hardware (Section 4.4.1). Another global barrier separates commit mode from *serial mode*, during which atomic operations (which cannot execute correctly under the relaxed-consistency coherence protocol of parallel mode) execute in a deterministic, serial order. A final global barrier ends the current quantum and allows a new quantum's parallel mode to begin.

3.2 Deterministic GPU Execution Challenges

The first challenge for deterministic execution on the GPU is the **lack of private caches and cache coherence**. A GPU's multiplexing of hundreds of threads onto a single first-level cache means that Calvin's mechanism of using private caches to provide isolation between threads is not readily employable. Physically or logically partitioning each cache for use amongst hundreds of threads would dramatically reduce the cache's effectiveness. Even if per-thread private caches were available, the lack of cache coherence in a GPU's per-core caches rules out Calvin's modified coherence protocol as a way of providing low-overhead thread isolation. Implementing isolation in software (as in CoreDet) allows physical resources to be shared but has high runtime overhead.

Another major concern in building deterministic GPUs is dealing with **very large numbers of threads**. This in turn leads to a number of related problems. Large numbers of threads make the global barriers inherent in the CoreDet/Calvin deterministic execution algorithms much more expensive. Relatedly, atomic operations require serialization in the Calvin and CoreDet models, so their presence in a GPU kernel can quickly erode performance. By serializing thousands of threads, an atomic operation has effectively many orders of magnitude higher cost when run deterministically than when run nondeterministically.

The GPU hardware features various hardware mechanisms to efficiently manage thousands of threads. These mechanisms need to be extended accordingly to support deterministic execution. Specifically, with the **SIMT execution model**, arbitrarily pausing a scalar thread that has exhausted its instruction quota while permitting the others to proceed causes the wavefront to diverge. Handling this divergence requires substantial modification to the SIMT hardware and the extra determinism-induced divergence lowers the SIMD hardware utilization.

Finally, GPU kernels have different program properties than multithreaded CPU programs. Executing larger quanta is a natural solution to amortize expensive global barriers and atomic operations, and works well in the CPU space where threads are long-lived. Unfortunately, GPU kernels often contain a **large number of short-running threads**, making global barriers both expensive *and* frequent. GPU threads tends to synchronize frequently within a workgroup to communicate via the on-chip scratch-pad memory. This form of **hierarchical, localized communication** fits poorly with the infrequent global communication model in the CoreDet/Calvin deterministic execution algorithms. GPU threads also typically exhibit **less locality** than CPU threads, particularly in terms of memory accesses: frequently-used values are instead cached in a GPU's large register files or scratch-pad memory. This reduced locality makes Calvin's cache-based isolation mechanism a poor fit for GPU kernels.

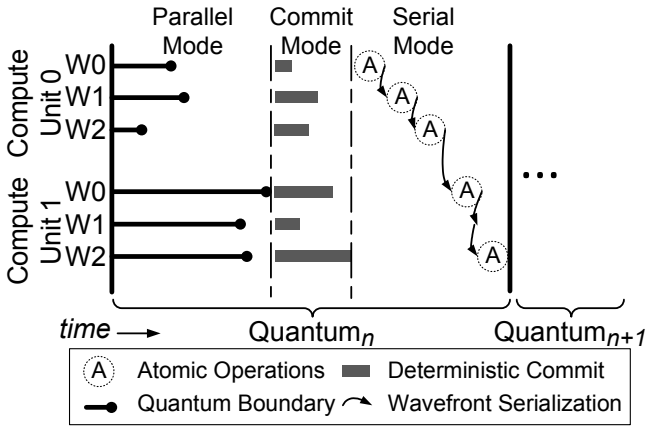

Figure 2. GPUDet-base architecture. Vertical lines show the global synchronization before the start of each mode. Wavefront 1 (W1) of compute unit 1 has been skipped in the serial mode since it has not reached an atomic operation in the $quantum_n$.

4. GPUDet

In this section, we present GPUDet, the first hardware proposal that provides strong determinism for deterministic massively parallel architecture with thousands of concurrent threads. We design GPUDet starting with a naive adaptation of the deterministic multiprocessor schemes from CoreDet/Calvin onto the GPU. This scheme divides GPU kernel execution into *quanta*. Each quantum has three phases: parallel mode, commit mode and serial mode as illustrated in Figure 2. GPUDet features optimizations for each mode, leveraging the inherent determinism in the GPU architecture as well as common-case GPU program behavior to recover more parallelism while reducing complexity required to support deterministic execution.

For parallel mode, GPUDet exploits the inherent determinism of current SIMT architectures to provide deterministic interaction among threads within a wavefront at no cost (Section 4.1). GPUDet also introduces a novel wavefront-aware quantum formation algorithm to eliminate unnecessary global synchronizations and replace them with local, deterministic synchronizations (Section 4.2). The store buffers that provide isolated execution among deterministic wavefronts are described in Section 4.3. For commit mode, GPUDet extends the Z-Buffer hardware designed for graphics rendering to accelerate its deterministic parallel commit algorithm (Section 4.4). For serial mode, it exploits the point-to-point ordering in the GPU's memory subsystem to reduce the amount of serialization required in this mode (Section 4.5). Section 4.6 discusses limitations in GPUDet's current design.

4.1 Deterministic Wavefront

The deterministic execution algorithm used in CoreDet and Calvin imposes determinism by pausing the execution of threads at deterministic intervals for communication [6, 23]. A naive GPU adaptation of this scheme would involve executing each scalar thread in isolation during the parallel mode and pausing the thread when it has exhausted its instruction count quota for the quantum. As mentioned in Section 3.2, this behavior interacts poorly with the implicit synchronizations of threads within a wavefront imposed by the SIMT execution model. Example 1 shows the execution of a wavefront with 4 threads (T0 to T3) through a branch hammock. The wavefront diverges at line `B`. In response, the SIMT stack deactivates T2 and T3 to execute line `C` with only T0 and T1 active. After executing line `C`, T0 and T1 have exhausted their instruction

Example 1 Thread-level isolation introduces divergence in SIMT architecture. In this example, each thread may execute up to 3 operations per quantum. The number of operations executed by each thread is shown beside each line of the code.

```
                                    T0  T1  T2  T3
A: v = 1;                        \\  1   1   1   1
B: if( threadIdx.x < 2 ){        \\  2   2   2   2
C:   v = input[threadIdx.x];     \\  3   3   2   2
D: }                             \\
E: output[threadIdx.x] += v;     \\  4   4   3   3
```

count quota for the quantum, and enter the global barrier for commit mode. A deadlock has occurred, because this global barrier is also waiting for T2 and T3, which are in turn paused by the SIMT stack to reconverge with T0 and T1 for full SIMD execution at line E. The SIMT stack can be modified to resolve this deadlock, but doing so introduces extra complexity and lowers the SIMD hardware utilization.

GPUDet eliminates these complexities and deadlock concerns altogether by exploiting the fact that the execution of the entire wavefront is inherently deterministic. Deterministic execution within a wavefront also eliminates the need to have thread-level isolation and allows threads in the same wavefront share a common store buffer. The inherent determinism of wavefront execution arises from two properties: 1) Pausing the execution of a wavefront causes each thread to execute a deterministic (but not necessarily equal) number of instructions, and 2) existing GPU architecture already handles data-races between threads within a wavefront.

As described in Section 2.1, the control flow of each wavefront, and the activity of its threads at every dynamic instruction, are controlled by its SIMT stack. The SIMT stack of every wavefront has deterministic initial state, since all wavefronts execute at the start of the kernel program. As the wavefront executes in parallel mode, its SIMT stack is updated to handle any branch divergence that occurs. This update is deterministic, because every thread in the wavefront is executing with input data from a deterministic global memory state produced by the previous quantum round. Since the SIMT stack has a deterministic initial state and it is updated deterministically, we can infer that the SIMT stack always maintains a deterministic state.

The deterministic SIMT stack provides deterministic control flow for the wavefront. More importantly, it ensures that the activity of each thread in the wavefront is deterministic for every dynamic instruction. This means that a wavefront can pause after any dynamic instruction to end its parallel mode for communication. Each thread in the wavefront may have executed a different amount of work in the parallel mode due to divergence, but the amount executed by each thread is always deterministic.

Data-races can occur between threads within a wavefront, but these data-races are reproduced deterministically on current SIMT hardware [25], a property we exploit in GPUDet. Data-races that occur between threads executing different dynamic instructions are always ordered deterministically, because instructions from the same wavefront are executed in-order and the control flow of each wavefront is deterministic as per the above discussion. Data-races that occur between threads at the same dynamic memory instruction are ordered by the coalescing unit, which combines the scalar memory accesses from threads within a wavefront into accesses to wider memory chunks. When multiple threads in a wavefront write to the same location the coalescing unit generates only one memory access for the location and chooses one of the threads' store values. To ensure determinism, GPUDet relies upon the fact that the coalescing unit selects the same thread's store value for a given access pattern. A NVIDIA patent [26] describes a way to deterministically handle the write collision based on thread IDs (e.g. sending data of the thread with largest thread ID). Samuli and Tero [25] have exploited this deterministic behavior to optimize their software rasterization engine for the Fermi GPU architecture [32]. We have verified this observation on current GPU hardware as well as our simulation infrastructure with the GPU-Racey stress test described in Section 2.3. The deterministic interaction of threads within a wavefront enables GPUDet to use a per-wavefront store buffer (described in Section 4.3) for execution isolation.

4.2 Quanta Formation

At certain points of execution GPUDet pauses thread execution to commit store buffers. In order to have deterministic results, these points should be selected deterministically. In other words, execution of threads should be paused at the exact same instructions in different runs of a program. These termination points are called *quantum boundaries* in this paper. This section describes the quantum formation algorithm and how GPUDet determines quanta boundaries for each wavefront.

GPUDet divides the program execution into quanta similar to previous work [6, 23, 35]. Each wavefront is allowed to execute a deterministic number of instructions during the parallel mode in each quantum. To avoid deadlocks and to handle atomic operations, a wavefront may end its parallel mode before reaching its instruction limit. The following are the events that can cause a wavefront to end its parallel mode:

Instruction Count. A wavefront ends its parallel mode once it finishes executing a fixed number of instructions. This number is the *quantum size*.

Atomic Operations. Atomic operations are handled in GPUDet like they are in CoreDet. A wavefront ends its parallel mode whenever it reaches an atomic operation to execute the operation in serial mode.

Memory Fences. GPUs provide memory fence instructions for programmers to impose ordering on memory operations. Similar to CoreDet/Calvin, a fence instruction causes a wavefront in GPUDet to end its parallel mode to commit its store buffer in commit mode.

Workgroup Barriers. The GPU architecture provides workgroup-level synchronization operations called workgroup barriers. As discussed in Section 2, threads inside a workgroup cannot exit a workgroup barrier before all other threads in the same workgroup have reached the barrier. Some of these other wavefronts may have exhausted their instruction count limit before reaching the workgroup barrier and are waiting at the global barrier for transition into commit mode. To prevent a deadlock in this case, the wavefront at the workgroup barrier should end its parallel mode to unblock the other wavefronts.

Execution Complete. When a wavefront finishes kernel execution before exhausting its instruction count limit, it ends its parallel mode.

Figure 3 shows the breakdown of these events for our GPU applications (more details in Section 5) running on GPUDet with this baseline quantum formation logic. In this figure, GPUDet is configured to have quantum size = 200 instructions. In ATM, CL and HT, wavefronts usually end their quanta by reaching an atomic operation. In AES, HOTSP, LPS and SRAD, most wavefronts end their quanta at workgroup barriers. In BFSr, BFSf and CFD, wavefronts end more then 50% of their quanta at the end of a kernel program, illustrating the short-running thread challenge discussed in Section 3.2. Collectively, these events constrain the number of instructions a wavefront may execute in each quantum, and thus limit the effectiveness of increasing quantum size to amortize the synchronization overhead at each quantum.

With the above observations, we have designed workgroup-aware logic to determine the end of parallel mode of each quantum.

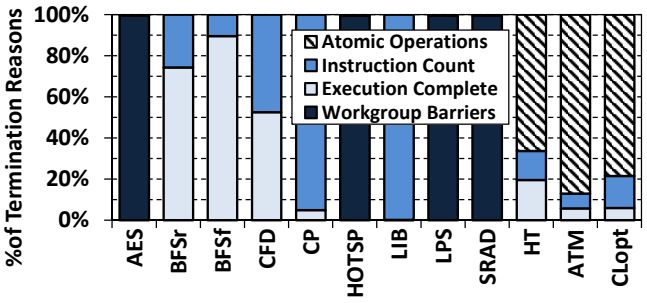

Figure 3. Breakdown of events that causes a wavefront to end its quantum in GPUDet with baseline quantum formation logic.

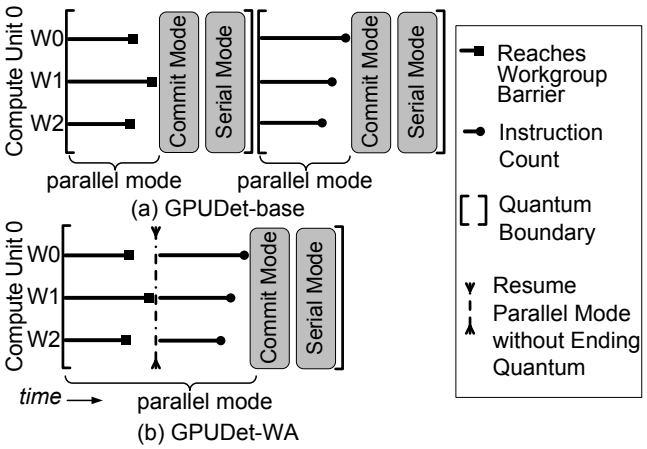

Figure 4. GPUDet quantum formation. (a) GPUDet-base behavior toward quantum termination events. (b) Workgroup-aware quantum formation of GPUDet (GPUDet-WA) allows wavefronts of one workgroup to continue the execution in parallel mode after all reaching workgroup barrier.

In Section 4.2.1 and Section 4.2.2, we describe this workgroup-aware quantum formation logic with two optimizations. One optimization allows a wavefront to execute beyond a workgroup barrier without ending its parallel mode; the other permits the GPU to issue a new workgroup within a quantum.

4.2.1 Workgroup-Aware Quantum Formation

Our workgroup-aware quantum formation logic (GPUDet-WA) extends the parallel mode illustrated in Figure 2 with an intermediate *wait-for-workgroup* mode. A wavefront advances to this wait-for-workgroup mode after encountering one of the termination events listed in Section 4.2 and waits for other wavefronts in the same workgroup to arrive. In this intermediate mode, each wavefront can deterministically observe the states of other wavefronts in the same workgroup. This allows all wavefronts in a workgroup to collectively decide the next mode.

Figure 4 illustrates how this mechanism reduces synchronization overhead by eliminating an unnecessary quantum boundary introduced by a workgroup barrier. This mechanism, called *Barrier Termination Avoidance* (BTA), allows wavefronts to continue execution past certain quantum boundary formation conditions improving load balance when the quantum size is increased. In this figure W0, W1 and W2 belong to one workgroup. In Figure 4 all wavefronts of the workgroup have been terminated by reaching a workgroup barrier. In the baseline GPUDet (Figure 4(a)), wavefronts will end their quantum and wait for the global synchronization before start of commit mode. However, GPUDet wavefront aware quantum formation (GPUDet-WA) allows the wavefronts to exit the barrier and resume the parallel mode without ending the quantum (BTA). Since reaching all the wavefronts of workgroup is compulsory to exit the barrier, the wavefronts would not be able to exit the barrier if a wavefront in that workgroup stopped execution before the barrier. To avoid the resulting deadlock condition, all of the wavefronts of the workgroup finish the parallel phase if any of them have stopped their parallel mode before the barrier (e.g., reaching instruction limit).

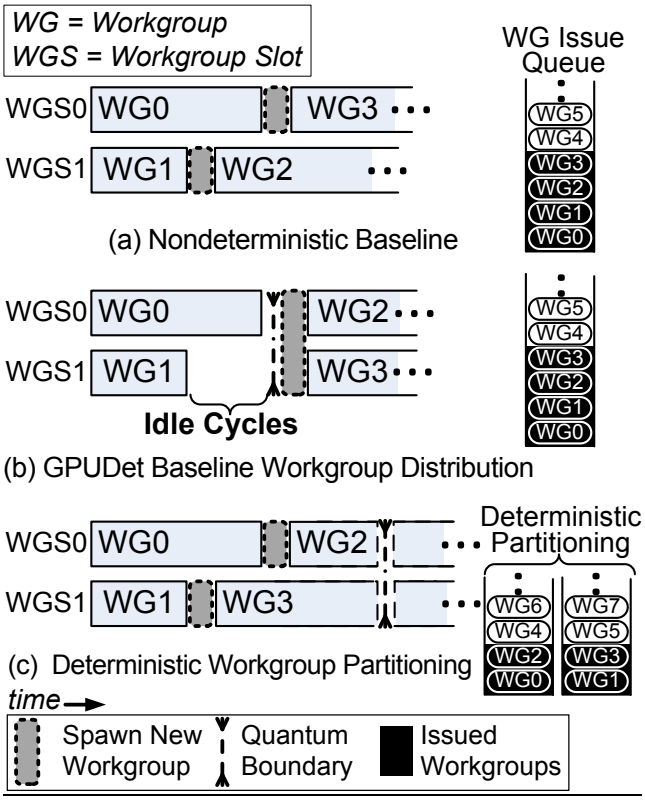

Figure 5. Workgroup distribution

A similar scenario arises when all wavefronts of the workgroup are terminated by reaching the end of their kernels. In this case, GPUDet launches a new workgroup and the execution continues in the parallel phase helping to ameliorate the impact of load imbalance between workgroups and gaining larger quanta.

4.2.2 Deterministic Workgroup Distribution

In Section 2, we described how a hardware workgroup distributor assigns a new workgroup to a compute unit whenever it has a free "slot" (subset of wavefronts with size of workgroup). This distribution of workgroups to compute units occurs both at the start of a kernel launch and when a running workgroup has finished execution.

Since the commit order of wavefronts is defined by how they are assigned to hardware wavefront slots, any change in this assignment will affect the commit order and hence the execution results. To achieve deterministic results, workgroups should be distributed among compute units deterministically. GPUDet has two different schemes to achieve deterministic workgroup distribution.

In the default scheme illustrated in Figure 5(b), GPUDet spawns workgroups only at the start of each quantum. The state of the whole system is deterministic when a quantum starts, so when

employing this policy the workgroup distributor has deterministic information about free workgroup slots. To ensure determinism, workgroups are assigned to free slots deterministically based on hardware slot ID. To achieve this, GPUDet enters a special *work distribution mode* right before entering the parallel mode. Workgroups are spawned only in this work distribution mode. All wavefronts stay in this mode until the workgroup distributor finishes assigning all free workgroup slots on all compute units with new workgroups. A deterministic set of free slots and of remaining workgroups results in spawning a deterministic set of workgroups.

Although implementation of the default workgroup distribution scheme in GPUDet is simple, it performs poorly when the kernel launch consists of short-running threads. If new workgroups are only issued in work distribution mode, then the number of instructions executed by a wavefront in one quantum will be limited by the amount of work per workgroup. This leads to load imbalance when some workgroups have more work than others.

To address this problem, we have proposed a *Deterministic Workgroup Partitioning* technique (DWP) that allows workgroups to be issued deterministically in the middle of parallel mode. To prevent nondeterministic workgroup distribution inside parallel mode, GPUDet partitions the issue-pending workgroups among hardware workgroup slots before starting parallel mode. Each workgroup slot can only be replenished with the workgroups in its partition. As workgroup partitioning is done in a deterministic state (at the start of each quantum round), the final workgroup partitioning is deterministic.

4.3 Per-Wavefront Store Buffer

In GPUDet, each wavefront has a private store buffer that contains all of its global memory writes from the parallel mode in the current quantum. The store buffer is located in the private memory of the wavefront, cached by the L1 data cache and written back to off-chip DRAM. It has been observed that in GPU kernel programs, data written to global memory is rarely accessed by the writer thread/wavefront again [19]. This insight suggests organizing the store buffer as a linear write log. Each entry in this write log represents a coalesced global memory write to a coalesced memory access size -128-Byte- chunk. It has an address field indicating the destination location of the chunk, a data field, and a 128-bit byte mask to indicate the valid portion in the data field. The store buffer has a Bloom filter that summarizes the addresses updated in the write log. Each coalesced global memory read in parallel mode first queries the Bloom filter with its chunk address. A hit in the Bloom filter triggers a linear search through the write log; a miss redirects the global memory read to the normal access sequence.

The cost of using a large Bloom filter to reduce false positives is amortized with a wavefront-shared store buffer. A 1024-bit Bloom filter only takes 8kB of storage per compute unit (assuming 48 wavefronts per unit). The Bloom filter of each wavefront can be stored in the register file space allocated to the wavefront. This eliminates any need for permanent storage at the expense of extra register file bandwidth and capacity.

4.4 Parallel Commit of Store Buffers

GPUDet commits the store buffers from all wavefronts into the global memory in the commit mode to allow wavefronts to communicate deterministically. We use the deterministic parallel commit algorithm used in CoreDet [6]. This algorithm tags the entries from the store buffer from a wavefront with a deterministic ID. This ID defines the commit order of this wavefront with respect to the other wavefronts. The wavefronts can attempt to commit the entries in their store buffers in parallel. The algorithm uses the deterministic ID to determine the final writer, in commit order, to each memory location, and guarantees that the location contains the value written by this wavefront after the commit mode. Currents GPUs have non-coherent private caches. To avoid reading stale data GPUDet flushes the L1 data caches after the commit mode. Supporting cache coherency on GPUs [39] would eliminate this overhead.

While CoreDet implemented this algorithm in software using fine-grained locks, we recognized that this algorithm is analogous to the Z-Buffer algorithm that controls the visibility of overlapped polygons in graphics rendering. GPUDet adopts the Z-Buffer Unit for graphics rendering to implement a hardware accelerated version of the deterministic parallel commit algorithm.

Currently, no GPU vendors have exposed any instruction for using the Z-Buffer Unit directly in general purpose programming models like CUDA and OpenCL. We believe exposing this unit in general purpose programming model will not be expensive in terms of area or complexity.

4.4.1 Z-Buffer Unit

Z-Buffer algorithm is designed to control the visibility of overlapped 3D objects displayed on screen. In graphics rendering, 3D objects are represented by triangles that are transformed according to a given camera view to be displayed. Each pixel rendered from a triangle is assigned a depth value representing its logical order with respect to pixels from other triangles. These depth values are stored in a set of memory locations called the *Z-Buffer*. A specialized hardware unit in the GPU, which we call the *Z-Buffer Unit*, manages the Z-Buffer. Using the depth values in the Z-Buffer, the Z-Buffer Unit prevents overlapped triangles from updating the colors of the pixels that have been updated by a foreground triangle. The Z-Buffer Unit allows out-of-order writes to produce a deterministic result. Namely, each pixel on the screen displays the color of the foremost triangle covering that pixel regardless of the order of the triangle updates.

There is little publicly-available information regarding the internal architecture of Z-Buffer Unit in current GPUs. The Z-Buffer Unit evaluated in this paper is inspired by the details disclosed in a patent by ATI Technologies [43]. As shown in Figure 1, there is a Z-Buffer Unit in each memory partition, each responsible for servicing the color update requests to the memory locations managed by the partition. Each Z-Buffer Unit contains a *request buffer* for keeping the status of the incoming requests. Each request contains a 128-byte aligned address to the chunk of pixels it attempts to update, together with 32 color values and 32 depth values. It also has a 128-bit byte mask indicating the valid values in the color update request. A color update request first tries to retrieve the latest depth values for its pixels from a cache for the depth values, called the *Z-cache*. The depth values from the request are then compared against the retrieved depth values. The Z-Buffer Unit then updates the color of the pixels that passes the comparison by sending write requests to the L2 cache, and updating the depth values in the Z-cache. A request that misses at the Z-cache allocates a block in the cache, locks the block and defers its comparison until the depth values are fetched. Requests hitting at a locked block are deferred similarly. The Z-Buffer Unit overlaps multiple color update requests to tolerate the Z-cache miss latency.

4.4.2 Deterministic Parallel Commit using Z-Buffer

To adopt the Z-Buffer Units for deterministic parallel commit of store buffers, GPUDet allocates a corresponding Z-Buffer for each writable global memory buffer. The allocation routine co-locates both depth and data of a memory location at the same memory partition. In commit mode, each wavefront publishes the entries in its store buffer by traversing through the linear write log. From each entry, it generates a color update request containing the buffered data, with the depth equal to the logical quantum ID concatenated with its hardware wavefront ID:

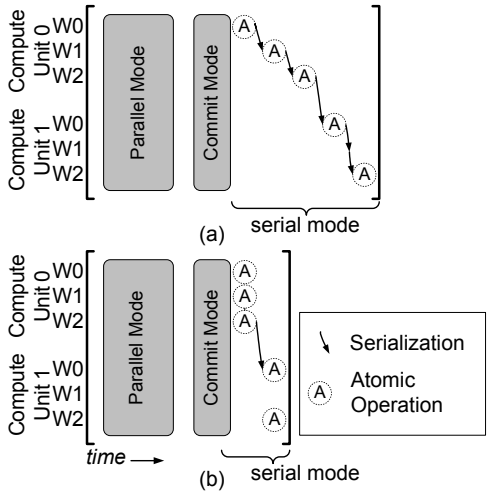

Figure 6. Serial mode in GPUDet. (a) Serializing execution of all atomic operations in GPUDet-base. (b) Overlapping execution time of atomic operations from each compute unit by GPUDet.

$$Depth\_value = -(quantum\_id \times 2^m) + wavefront\_id$$

Here, m is number of bits needed to represent the maximum number of wavefronts that can run concurrently on the hardware. We include the logical quantum ID in the depth to ensure that data from an earlier quantum is always overwritten by memory updates from a later quantum. Without the logical quantum ID, the depth values for all writeable locations in global memory must be reset to the largest positive integer at every quantum boundaries. With the quantum ID, GPUDet amortizes this overhead across a large number of quanta by resetting the Z-buffer only when the ID overflows.

4.5 Compute Unit Level Serialization

In serial mode, the wavefronts execute atomic operation in a deterministic serial order, with the operations updating the global memory directly. Each wavefront executes only one atomic operation in serial mode. Each atomic operation can take 100s of cycles to execute in the GPU's throughput-optimized memory subsystem. Our evaluation in Section 5.2.3 shows that this naive, wavefront-level serialization (W-Ser) significantly slows down applications that use atomic operations.

To recapture parallelism, GPUDet exploits the point-to-point ordering guaranteed by our GPU memory model and common in network-on-chip designs [44] to overlap the execution of atomic operations within a compute unit (CU-Ser). As long as the interconnection network guarantees point-to-point ordering, atomic operations from the same compute unit to the same memory partition will arrive in the original order. With little if any potential performance gain at stake, the memory partitions do not reorder accesses to the same memory location. By resetting the wavefront scheduler in each compute unit at the start of serial mode, GPUDet forces the unit to issue its atomic operations in a deterministic order. This order is preserved by the point-to-point ordering in the GPU memory subsystem. On the other hand, since there is no guaranteed ordering between requests from different cores, GPUDet has to serialize memory operations from different compute units.

Figure 6 demonstrates how this optimization reduces serialization overhead in GPUDet by eliminating serialization within each compute unit. In Figure 6(a), the atomic operations from all workgroup are executed serially. In Figure 6(b), each compute unit overlaps execution of their wavefronts in serial phase, so the serialization overhead is reduced by a factor of number of wavefronts in the compute unit.

4.6 Limitations

The current design of GPUDet does not enforce determinism in the use local memory (or shared memory in NVIDIA terminology). Local memory accesses update the on-chip scratch-pad memory directly and rely on the application to remove data-races via workgroup barriers. The wavefront scheduler at each compute unit can be modified to issue wavefronts in deterministic order. This modified scheduler will provide deterministic execution for local memory accesses even in the presence of data-race. We leave proper support for deterministic local memory accesses as future work.

Since the current Z-buffer design binds each 32-bit color pixel in memory with a depth value, the Z-buffer-based parallel commit algorithm is not directly applicable to applications with byte-granularity writes. An intermediate solution, without modifying the Z-Buffer Unit, is to commit (only) the store buffer entries with byte-granularity writes in the serial mode. We leave evaluation of this as future work. Note that the per-wavefront store buffer is already designed to support byte-granularity accesses. Each store buffer entry contains a 128-bit mask indicating the modified bytes, and the Bloom filter is only responsible for identifying accesses to the same 128-byte chunk.

5. Evaluation

We extended GPGPU-Sim 3.0.2 [5] to model a nondeterministic GPU architecture by randomizing the interconnect injection order of requests from different compute units. We evaluate the performance impact of GPUDet on a set of CUDA/OpenCL benchmarks (listed in Table 1) from Rodinia[14], Bakhoda et al.[5] and Fung et al.[19]. We ran each benchmark to completion. We do not exclude any benchmarks because of poor performance. We do exclude one benchmark that assumes workgroups are spawned in ascending ID order, and five benchmarks that contain hard-to-eliminate byte-granularity writes. We include a version of cloth simulation (CLopt) with a GPU-optimized work distribution scheme that transforms the non-coalesced memory accesses into coalesced accesses. This optimized version performs 30% faster than the original one on the nondeterministic baseline architecture. On GPUDet, this optimization significantly reduces the number of entries in the per-wavefront store buffers, lowering the overhead of each write-log search. It also generates fewer write-log searches by eliminating the aliased read-write accesses within a wavefront accessing different bytes in the same 128-bytes. These two effects cause CLopt to perform significantly better than CL on GPUDet (Section 5.3.1). We also include the version of BFS graph traversal with data-races (BFSr) from Section 1.1 as well as the corrected version (BFSf) from Rodinia [14]. Both BFSr and BFSf are modified to use 32-bit boolean flags to eliminate byte-granularity writes.

We used GPU-Racey (described in Section 2.3) to verify our nondeterminism extension to GPGPU-Sim, and to verify that our model of GPUDet can provide deterministic execution under this nondeterministic simulation framework.

Our modified GPGPU-Sim is configured to model a Geforce GTX 480 (Fermi) GPU [32] with the configuration parameters distributed with GPGPU-Sim 3.0.2. In our default GPUDet configuration, each Z-Buffer Unit runs at 650MHz, has a 16kB Z-cache, and a 16 entry request buffer. The per-wavefront store buffer uses a 1024-bit bloom filter, implemented with a Parallel Bloom Filter [37] with 4 sub-arrays each indexed with a different hash function. This default configuration assumes that each global barrier between parallel, commit, and serial modes in GPUDet takes zero cycles. Section 5.3.4 investigates the sensitivity of GPUDet's performance to various Z-cache sizes, smaller store buffer bloom fil-

Table 1. Benchmarks

Name	Abbr.
Without Atomic Operations	
AES Cryptography [5]	AES
BFS Graph Traversal (with Data-Race) [5]	BFSr
BFS Graph Traversal (Race-Free) [14]	BFSf
Computational Fluid Dynamics Solver [14]	CFD
Coulumb Potential [5]	CP
HotSpot [14]	HOTSP
LIBOR [5]	LIB
3D Laplace Solver [5]	LPS
Speckle Reducing Anisotropic Diffusion [14]	SRAD
With Atomic Operations	
Cloth Simulation [19]	CL
Cloth Simulation (Optimized)	CLopt
Hash Table [19]	HT
Bank Account [19]	ATM

ters, and higher global barrier latencies. Our modified version of GPGPU-Sim and the benchmarks are available online [1].

5.1 Overall Performance

To evaluate our system, we compare the execution time of the benchmarks on a baseline nondeterministic architecture (NONDET) and the optimized version of GPUDet. We configure the quantum size to 200 instructions. We increment the instruction count whenever a wavefront executes an instruction, regardless of how many threads in the wavefront execute that instruction. Figure 7 shows the total execution time of each application with GPUDet, normalized to execution time on a nondeterministic architecture. Our deterministic model causes about 105% performance penalty on average. The execution time of each application is broken down into time spent in parallel, commit and serial modes. Wavefronts spend the most time in parallel mode. We discuss the sources of performance overhead in parallel mode below. As discussed in Section 4.5, GPUDet only serializes the execution of atomic operations: applications without atomic operations (all except CLopt, HT, ATM) skip serial mode entirely.

We have found that some applications (CP, AES) perform slightly better with deterministic execution. A deeper inspection reveals that our workgroup distribution algorithm (Section 4.2.2) results in a more even distribution of workgroups compared to the baseline architecture. The baseline architecture tries to find the first free hardware slot to spawn a workgroup. This mechanism can cause uneven distribution at the very end of a kernel launch when all workgroups of one compute unit finish their executions within a short period. In this case, all remaining workgroups are assigned to the sole available compute unit. When other compute units become available, no workgroups are left to be assigned, and the compute units are underutilized (a similar observation was noted by Bakhoda et al [5]). For these applications, GPUDet's deterministic workgroup distribution will distribute workgroups more evenly, resulting in a small speedup.

5.2 Impact of GPUDet Optimizations

This section provides data to evaluate our optimization techniques discussed in Section 4.

5.2.1 Quantum Formation

To assess our proposed optimization techniques for quantum formation (Section 4.2) we implemented three versions of GPUDet shown in Figure 8. In the GPUDet-base configuration all optimizations are disabled. In the GPUDet-WA(BTA) the workgroup barrier optimization technique (Section 4.2.1) is enabled, allowing workgroups to synchronize without a quantum boundary. Finally, the GPUDet-WA(BTA+DWP) version additionally allows new workgroups to be spawned in parallel mode by deterministically partitioning of the workgroups (DWP) among workgroup slots (Section 4.2.2).

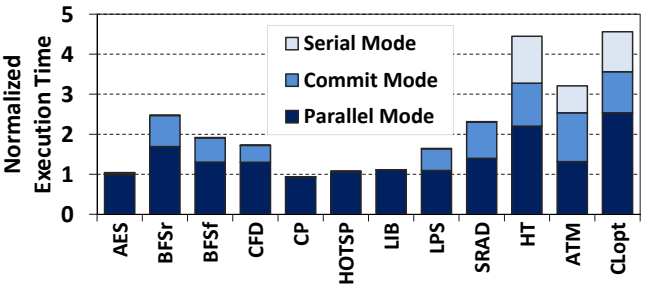

Figure 7. Breakdown of execution cycles. Normalized to NONDET execution time.

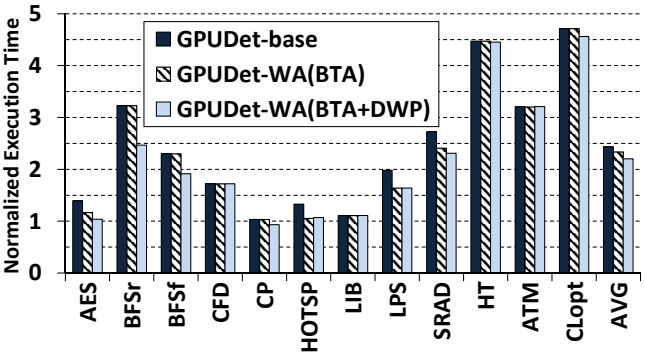

Figure 8. Performance impact of Barrier Termination Avoidance (BTA) and Deterministic Workgroup Partitioning (DWP) techniques. Bars Normalized to NONDET execution time.

Figure 8 shows that the barrier termination avoidance technique decreases the execution time by 4% on average over GPUDet-base. As expected, this improvement lies mostly with benchmarks that have frequent synchronization barriers (AES, HOTSP, LPS, SRAD). Figure 3 shows that encountering workgroup barriers is the dominant cause of quantum termination for these applications. Our experiments confirm that activating the barrier termination avoidance technique (GPUDet-WA(BTA)) forms 3.8× fewer quanta in these four benchmarks, improving their performance by 20% on average.

Figure 8 shows that applications with small kernel functions (BFSr, BFSf, CFD) benefit from spawning workgroups in parallel mode. Figure 3 confirms that most of the quanta in BFSr, BFSf and CFD are terminated by reaching the end of the kernel. The ability to start new workgroups deterministically in parallel mode (GPUDet-WA(BTA+DWP)) speeds up these applications by 19% on average.

5.2.2 Parallel Commit using Z-Buffer Unit

To evaluate the Z-Buffer Unit parallel commit algorithm, we implemented a lock-based version of committing the store buffer. The lock-based version simulates the software-based deterministic parallel commit algorithm proposed in CoreDet [6]. The algorithm locks the chunk of memory that corresponds to the store buffer entry address using atomic operations and performs the logical order priority comparisons and global memory updates in a mutually exclusive section. Although it has been reported that atomic operations contending for the same memory location can be up to 8.4× slower than non-atomic store operations in the Fermi architecture [40], in our evaluation we do *not* model this extra slowdown

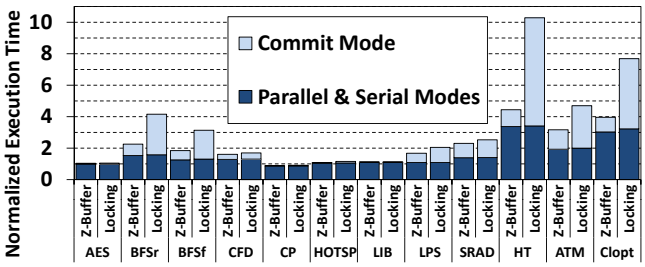

Figure 9. Execution time comparison of committing the store buffer between the Z-Buffer Unit parallel commit and the lock-based algorithms

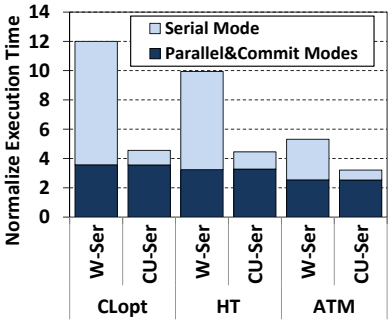

Figure 10. Execution time comparison between wavefront level (W-Ser) and compute unit level (CU-Ser) serialization of atomic operations (Section 4.5). Normalized to NONDET execution time

for the additional atomic operations required by the lock-based version.

Figure 9 shows the normalized execution time for both of the store buffer committing algorithms. The Z-Buffer Unit commit algorithm improves performance by 60% on average. Since spinning for a lock in compute units generates significantly more global memory accesses, it is expected that the applications with more accesses (e.g. HT) achieve more performance improvement by exploiting the Z-Buffer Unit commit algorithm. Figure 9 shows that the execution time of commit mode is decreased by 2.3× using the Z-Buffer Unit for the HT benchmark.

5.2.3 Serial Mode Optimization

Figure 10 evaluates the performance overhead of serial mode in GPUDet. Since only atomic operations are serialized, we omit benchmarks without atomic operations. W-Ser (wavefront level serialization) is the GPUDet-base configuration which serializes execution of all atomic operations. In CU-Ser (compute unit serialization), serialization is only performed among compute units by executing the atomic operations of a single compute unit in parallel. CU-Ser decreases overhead of serial mode by 6.1× for these applications.

5.3 Sensitivity Analysis

This section explores the performance impact of varying different GPUDet design parameters.

5.3.1 Quantum Size and Store Buffer Overhead

In this section we evaluate the effect of quantum size on performance. For better insight, we separate out the time spent in store buffer operations. Store buffer operations entail appending entries to store buffers for stores, and traversing store buffers for replying to loads. Figure 11 shows the execution time of GPUDet with quantum sizes of 100, 200, 600 and 1000 instructions.

Increasing quantum size has three kinds of effects on applications. For applications that have few stores to global memory (AES, CP, HOTSP and LIB) larger quanta improves performance. By increasing quantum size, we reduce the number of quantum boundaries, each of which involves expensive global synchronization.

For applications with frequent accesses to global memory (BFSr and BFSf), performance is degraded with increasing quantum size, because larger quanta lead to larger store buffers. Because GPUDet uses a linear write log traversal, load instructions hitting in store buffer must perform longer searches. Increasing the quantum size also has a side effect on the BFS benchmark. Since this benchmark is not work optimal [27] (it may do each task several times), the number of executed instructions varies in different configurations. In this application the number of redundant tasks depends on how frequently the execution results of each thread become visible to the other threads. Enlarging quantum size causes results to be visible less frequently, BFS runs more redundant tasks which decreases performance. Our experiments reveal that the number of executed instructions increases by 26% when the quantum size increases from 200 to 1000.

For the CL benchmark, the performance overhead of the linear write log is highly affected by the size of the quantum. We realized that in the Integrator, WriteBack and Driver Solver kernels [12] of this benchmark work is distributed block-wise among the threads, so wavefronts generate uncoalesced memory accesses. As described in Section 4.3, uncoalesced accesses generate many extra entries in the store buffer. Furthermore, due to Bloom filter aliasing, uncoalesced accesses result in unnecessary log traversals for load operations (Section 5.3.2). By using an interleaved distribution of work among the threads, we eliminated uncoalesced memory accesses in the CLopt application. Since eliminating uncoalesced accesses to the global memory is profitable for general GPU applications and it does not need substantial changes to CL source code, we have only included the optimized version of CL in our overall average.

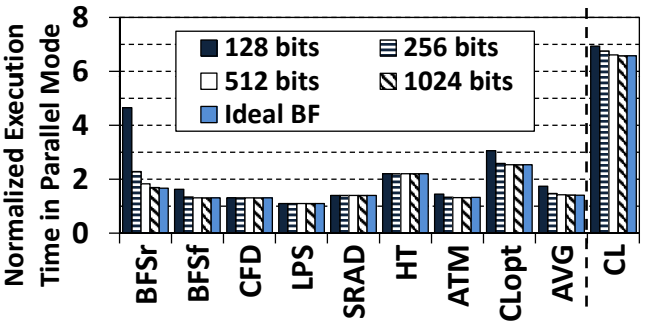

Figure 12. Execution cycles in parallel mode for various sizes of Bloom filter, normalized to NONDET execution. The AVG bar shows the average among all our benchmarks.

5.3.2 Bloom Filter

Figure 12 evaluates the effect of Bloom filter size on performance. Since the Bloom filter configuration does not affect commit and serial modes, we present only the execution time of parallel mode. Figure 12 shows the execution time of the GPUDet architecture with 128, 256, 512 and 1024-bit Bloom filters, and an ideal Bloom filter which has no false positives. We only present the execution time of individual applications that use the store buffer frequently, but the average is for all our workloads. Figure 12 shows that for most of the applications (except BFSr and CL) a 256-bit Bloom fil-

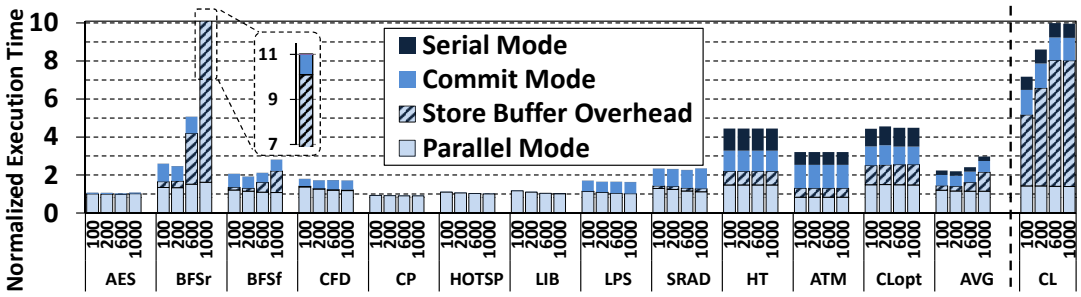

Figure 11. Sensitivity of GPUDet to quantum size. Execution time is normalized to NONDET. The unoptimized version of CL is not included in the average.

ter performs as well as ideal. Increasing Bloom filter size for BFSr reduces execution time because of a high number of store buffer entries in each quantum. For CL, because of the large number of uncoalesced accesses, the store buffer contains many 128B entries with sparsely-populated data portions. The large number of entries causes many loads to perform unnecessary log traversals.

5.3.3 Z-Buffer Unit

We assess the sensitivity of GPUDet to the size of the Z-cache (Section 4.4.1) by adjusting the size of cache for each memory partition from 8KB to 128KB. As expected, execution time decreases with increasing Z-cache size. However, since wavefronts spend most of their execution time in parallel and serial modes, the execution time decreases by less than 3% when the Z-cache is enlarged from 8KB to 128KB.

5.3.4 Global Synchronization Overhead

GPUDet needs global synchronization between each execution mode. We analysed the sensitivity of GPUDet to the cost of global synchronization. Our data illustrates that the overall performance degradation with a 100 cycles synchronization overhead is negligible. However increasing the overhead to 1000 cycles reduces overall performance by 22% over the instantaneous synchronization model.

6. Related Work

The most highly-related work to GPUDet are proposals for providing deterministic execution of general CPU programs [4, 6, 7, 16, 23, 31, 35]. These schemes provide determinism for general multithreaded code on commodity [4, 6, 7, 31, 35] or modified general-purpose multicore CPUs [16, 23]. A naive translation of these schemes to a GPU architecture would be highly inefficient due to the massive numbers of threads involved. For example, the global barriers inherent in many deterministic execution schemes ([4, 7, 16, 23, 31]) would not scale well with 1000s of threads. Kendo's [35] more scalable approach provides determinism for data-race-free programs only, which would scale better in a GPU environment but also reneges on many of the debugging benefits of other determinism techniques that can handle data races.

There has also been extensive work on deterministic programming languages ([9, 10, 13, 18, 36, 41]). Programs written in these languages are deterministic by construction, and incur little runtime overhead during execution at the expense of a restricted programming model. SHIM [18] is a deterministic version of MPI that provides determinism for message-passing programs. NESL [9] and Data Parallel Haskell [13] are pure functional languages that support data-parallel operations which are deterministic and implicitly parallel. Jade [36] is an imperative language that relies on programmer-supplied annotations to extract parallelism automatically while preserving sequential (and deterministic) semantics. StreamIt [41] is a language for streaming computations that enforces determinism by restricting communication between parallel stream kernels to occur only via FIFOs, though a later version of StreamIt allowed for out-of-band control messages that could flow both upstream and downstream without breaking determinism [42]. Deterministic Parallel Java [10] is a version of Java augmented with a type-and-effect system that proves non-interference between parallel fork-join tasks. None of these languages are directly applicable to current GPU programming models, which support a rich set of synchronization primitives via atomic operations on global memory.

A number of projects have looked at improving GPU programmability through hardware or software mechanisms. Kilo TM [19] showed that transactional memory can be incorporated into the GPU programming model at low cost. GRace [46] proposed a race detector for GPU programs, achieving low overhead via a combination of static analysis and runtime instrumentation. Boyer and Skadron [11] describe a GPU emulator that detects race conditions and bank conflicts for CUDA programs, though the emulation overheads are quite high compared to native execution. The PUG system [28] and GPUVerify [8] leverage the compact nature of GPU kernels to perform static race detection, occasionally requiring annotations from programmers to avoid false positives.

7. Conclusion

Nondeterminism in parallel architectures hampers programmers' productivity significantly as bugs can no longer be reproduced easily. We believe this non-reproducibility problem presents a key challenge to GPU software development, discouraging use of GPUs in broader range of applications.

In this paper we presented GPUDet, a proposal for supporting deterministic execution on GPUs. GPUDet exploits deterministic aspects of the GPU architecture to regain performance. Specifically, it uses the inherent determinism of the SIMD hardware in GPUs to provide deterministic interaction among threads within a wavefront. This amortizes the complexity of store buffers required to isolate the execution of each wavefront and works seamlessly with the existing SIMT execution model in GPUs. GPUDet uses a workgroup-aware quantum formation scheme that allows wavefronts in parallel mode to coordinate via workgroup barriers and to accept work from a deterministic workgroup distributor. GPUDet also extends the Z-Buffer Unit, an existing GPU hardware unit for graphics rendering, to deterministically commit store buffers in parallel. Finally, GPUDet eliminates the serialization required among atomic operations from the same compute unit by exploiting the implicit point-to-point ordering in the GPU memory subsystem.

Our simulation results indicate that these optimizations allow GPUDet to perform comparably against a nondeterministic base-

line, despite running GPU kernels with thousands of threads. Our characterization of sources of overhead for deterministic execution on GPUs provides insights for further optimizations.

Acknowledgments

The authors would like to thank Inderpreet Singh, Tim Rogers, Henry Wong and the anonymous reviewers for their insightful feedback. We also thank Ayub Gubran for his work on developing the functional model for the Z-Buffer Unit in our simulator. This research was funded in part by grants from Advanced Micro Devices Inc., Qualcomm Inc., and the Natural Sciences and Engineering Research Council of Canada. Wilson W. L. Fung is supported by NVIDIA Graduate Fellowship Program.

References

[1] http://www.ece.ubc.ca/˜aamodt/GPUDet.

[2] *White Paper — AMD Graphics Cores Next (GCN) Architecture*. AMD, June 2012.

[3] D. Arnold et al. Stack Trace Analysis for Large Scale Debugging. In *IPDPS*, 2007.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[5] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[6] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, 2010.

[7] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic Process Groups in dOS. In *OSDI*, 2010.

[8] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPU kernels. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, 2012.

[9] G. Blelloch. NESL: A Nested Data-Parallel Language (Version 3.1). Technical report, Carnegie Mellon University, Pittsburgh, PA, 2007.

[10] 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 *OOPSLA*, 2009.

[11] M. Boyer, K. Skadron, and W. Weimer. Automated Dynamic Analysis of CUDA Programs. In *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[12] A. Brownsword. Cloth in OpenCL, 2009.

[13] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A Status Report. In *DAMP*, 2007.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.

[15] B. W. Coon et al. United States Patent #7,353,369: System and Method for Managing Divergent Threads in a SIMD Architecture (Assignee NVIDIA Corp.), April 2008.

[16] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

[17] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *ASPLOS*, 2011.

[18] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. In *EMSOFT*, 2005.

[19] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware Transactional Memory for GPU Architectures. In *MICRO-44*, 2011.

[20] W. Fung et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO*, 2007.

[21] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *HiPC*, 2007.

[22] M. Hill and M. Xu. http://www.cs.wisc.edu/ markhill/racey.html, 2009.

[23] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or Not? Free Will to Choose. In *HPCA*, 2011.

[24] Khronos Group. OpenCL. http://www.khronos.org/opencl/.

[25] S. Laine and T. Karras. High-Performance Software Rasterization on GPUs. In *HPG*, 2011.

[26] G. H. Lars Nyland, John R. Nickolls and T. Mandal. United States Patent #8,086,806: Systems and methods for coalescing memory accesses of parallel threads (Assignee NVIDIA Corp.), April 2011.

[27] C. E. Leiserson and T. B. Schardl. A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers). In *SPAA*, 2010.

[28] G. Li and G. Gopalakrishnan. Scalable SMT-Based Verification of GPU Kernel Functions. In *FSE*, 2010.

[29] E. Lindholm et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 2008.

[30] J. Liu, B. Jaiyen, R. Veras, and O. Multu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*, 2012.

[31] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: Efficient Deterministic Multithreading. In *SOSP*, 2011.

[32] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. NVIDIA, October 2009.

[33] *NVIDIA CUDA Programming Guide v3.1*. NVIDIA Corp., 2010.

[34] *NVML API Reference Manual v3.295.45*. NVIDIA Corp., 2012.

[35] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[36] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 20(3), May 1998.

[37] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *MICRO*, 2007.

[38] S. R. Sarangi, B. Greskamp, and J. Torrellas. CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging. In *DSN*, 2006.

[39] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache Coherence for GPU Architectures. In *HPCA*, 2013.

[40] J. A. Stuart and J. D. Owens. Efficient Synchronization Primitives for GPUs. *CoRR*, abs/1110.4623, 2011.

[41] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC '02*, 2002.

[42] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. P. Amarasinghe. Teleport Messaging for Distributed Stream Programs. In *PPoPP*, 2005.

[43] T. J. Van Hook. United States Patent #6,630,933: Method and Apparatus for Compression and Decompression of Z Data (Assignee ATI Technologies Inc.), October 2003.

[44] S. Vangal et al. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, Jan. 2008.

[45] H. Wong et al. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, 2010.

[46] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. GRace: A Low-Overhead Mechanism for Detecting Data Races in GPU Programs. In *PPoPP*, 2011.