iGUARD: In-GPU Advanced Race Detection

Aditya K Kamath
Indian Institute of Science
Bengaluru, India
adityakamath@iisc.ac.in

Arkaprava Basu
Indian Institute of Science
Bengaluru, India
arkapravab@iisc.ac.in

Abstract

Newer use cases of GPU (Graphics Processing Unit) computing, e.g., graph analytics, look less like traditional bulk-synchronous GPU programs. To cater to the needs of emerging applications with semantically richer and finer grain sharing patterns, GPU vendors have been introducing advanced programming features, e.g., scoped synchronization and independent thread scheduling. While these features can speed up many applications and enable newer use cases, they can also introduce subtle synchronization errors if used incorrectly.

We present iGUARD, a runtime software tool to detect races in GPU programs due to incorrect use of such advanced features. A key need for a race detector to be practical is to accurately detect races at reasonable overheads. We thus perform the race detection on the GPU itself without relying on the CPU. The GPU's parallelism helps speed up race detection by 15× over a closely related prior work. Importantly, iGUARD detects newer types of races that were hitherto not possible for any known tool. It detected previously unknown subtle bugs in popular GPU programs, including three in NVIDIA supported commercial libraries. In total, iGUARD detected 57 races in 21 GPU programs, without false positives.

CCS Concepts: • **Software and its engineering** → **Parallel programming languages**; **Correctness**.

Keywords: Data races; GPU program correctness; Debugging

1 Introduction

A large and growing swath of software today relies on GPUs for their computation. Thus, the correctness of GPU programs (kernels) is critical to the reliability of a significant portion of the software ecosystem. Decades of research in multi-threaded CPU software have shown that subtle data races due to improper synchronization can introduce unpredictable failures [5, 8, 14–16, 18, 20, 31, 32, 43, 44, 56]. GPU programs with hundreds of thousands of threads and advanced synchronization mechanisms are even more vulnerable to obscure races [17, 52]. In this work, we build a runtime tool to detect races in GPU programs due to improper use of *advanced* synchronization and programming features of modern GPUs.

Traditionally, GPU's massive data-level parallelism was leveraged by bulk-synchronous programs where interactions among threads were infrequent and happened at coarse grain. GPU programming languages require programmers to divide a GPU kernel's hundreds of thousands of threads into equal-sized threadblocks of up to 1024 threads to keep the parallelism tractable. Threads within a threadblock share a scratchpad memory and could synchronize via threadblock barrier (syncthreads in CUDA). This, along with the implicit barriers across all threads at the end of a kernel's execution, fulfilled most synchronization needs of bulk-synchronous GPU programs. However, they fall short for programs with semantically richer and fine-grain sharing patterns.

Advanced GPU features: To better support emerging use cases such as graph processing, GPU vendors have progressively introduced advanced synchronization operations and more expressive execution paradigms. To balance the need for advanced synchronization operations with the requirement to scale-up performance on ever-larger modern GPUs, vendors have introduced *scoped* atomic and fence operations (e.g., threadfence in CUDA). The *scope* qualifier guarantees the effect of an atomic or fence to be visible *only* within a specified subset of threads (i.e., the scope of the operation), eschewing global visibility. This makes operations with *narrower* scope faster. For example, on a recent NVIDIA Titan RTX GPU, the block-scope threadfence that guarantees its effects to be visible only within a threadblock is 21× faster than the device scope fence that ensures global visibility across all threads on a GPU. However, the use of *inadequate* scope in synchronization, which does not include both the producer and consumer of a data item, leads to a race.

GPUs schedule threads in a threadblock in groups of 32 to 64 threads called a warp. Traditionally, threads in a warp would execute in lockstep. However, the lockstep execution could deadlock if threads within a warp use distinct locks – a situation not possible in multi-threaded CPU programs. To broaden the set of applications that can leverage GPUs, NVIDIA introduced Independent Thread Scheduling (ITS) that allowed threads in a warp to make independent progress

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483545

(Volta architecture, circa 2017, onward). However, applications that implicitly relied on lockstep execution would now require adding warp-level barriers (syncwarp) for correctness.

Finally, NVIDIA introduced the software abstraction of Cooperative Groups (CG) that enables programmers to synchronize across an (almost) arbitrary set of threads (unlike scopes, that are fixed in the hardware). While semantically rich, improper use of CG leads to a race. We refer to scoped synchronization, ITS, and CG together as the advanced synchronization and programming features of modern GPUs.

Limitations of current GPU race detectors: We are not the first to notice that GPU programs have races. Several works focused on detecting races that occur among threads of a single threadblock via the scratchpad memory (in KBs) [9, 54, 55]. NVIDIA's Racecheck is one such commercial tool [37]. However, they ignore the harder class of races that can happen among any pair of hundreds of thousands of GPU threads through GPU's global memory (tens of GBs in size).

Even proposals that focus on global memory races fall short in the presence of races induced by the incorrect use of advanced GPU features. For example, Barracuda and CURD do not support scoped atomic operations [17, 39]. Barracuda also does not detect races due to missing syncwarp under ITS. Notably, Barracuda incurs performance overheads of over 50×, on average. Its extension, CURD, reduces overheads for applications that use *only* traditional threadblock barrier (syncthreads), but falls back to Barracuda for everything else (e.g., for fences and atomics). We noticed that a key reason behind the high overheads is their reliance on the CPU for race detection and the consequent serialization. Furthermore, these tools require recompilation of the code, which is a significant detriment given the wide use of closed-source GPU libraries.

To limit performance overheads, researchers have proposed hardware-based GPU race detectors, e.g., HaccRG [25] and our previous work ScoRD [28]. However, these need redesign of GPU architectures and are inapplicable to current GPUs. Further, HaccRG is oblivious to scopes/ITS. ScoRD detects races due to improper scopes but does not support ITS [28]. None detect races due to CG, since one needs to fully support atomics, fences, and ITS for it. § 4 details more shortcomings.

Our contributions: As a growing class of applications leverage GPUs, a tool to detect races due to improper use of advanced GPU features is desirable. To be useful in practice, it should limit performance overheads incurred during race detection. Toward this, we propose iGUARD – in-GPU Advanced Race Detector. It extends ScoRD's logic for detecting scoped races to also detect races due to ITS and CG. Importantly, it alleviates the need for modifying the GPU hardware.

iGUARD is a runtime tool that executes on NVIDIA GPUs. It uses NVIDIA's NVBit binary instrumentation tool [47] to instrument GPU memory and synchronization operations for race detection. iGUARD performs the entire race detection on the GPU, without CPU involvement. The race detection happens along with application execution and leverages GPU parallelism, unlike prior software-based detectors. This reduces iGUARD's overheads and helps it scale to larger GPUs.

iGUARD instruments synchronization operations to track the active synchronization status (including scopes) of each warp. Under ITS, if threads within a warp diverge, the detector automatically tracks the same for each thread. This information is later used for race detection to infer if adequate synchronization separates two conflicting accesses to a memory location. Since a GPU can run hundreds of thousands of threads concurrently, it is impractical to track pairwise interactions across so many threads to detect races. Therefore, iGUARD keeps metadata for each unit of global memory (e.g., 4 bytes) to identify of the last read or write accessor to that location, along with the synchronization status of the accessor/writer at the time of its accessing the given location.

iGUARD instruments every load, store, and atomic operation and looks up the corresponding metadata that contains a summary of previous accesses to an address and synchronization. It compares the information in the metadata with that of the load (store/atomic) and its active synchronization status for inferring happens-before ordering between the accesses to a given address. iGUARD reports a race when no happens-before order could be inferred. This is akin to ScoRD's detection logic for scoped races. While ScoRD implements these in new hardware, iGUARD relies on binary instrumentation.

While CUDA does not intrinsically support locks/unlocks, atomics and threadfences can be combined to create them [1, 41, 42]. Further, under ITS, applications can use either coarse-grain locking (e.g., one lock per warp) or fine-grain per-thread locks. iGUARD dynamically infers locks and whether per-thread locking is employed by tracking thread divergences. It then uses the well-known lockset algorithm [43, 56] to detect the possibility of a race due to improper locking.

A unique challenge for an in-GPU software-based race detector is the serialization of metadata accesses. In GPU programs, thousands of threads can concurrently access a shared variable. Accesses to the corresponding metadata need to be serialized for correct race detection. However, this serialization surrenders the benefits of in-GPU race detection for kernels with many shared variables. Toward this, we introduce two optimizations. ① We opportunistically coalesce metadata accesses by observing that loads and atomics to a shared variable by active threads of the same warp cannot race. ② We employ dynamically adjusted backoff based on the number of concurrent threads to limit contention for the metadata. These play key roles in reducing performance overheads for many kernels.

Impact: iGUARD detected several previously unreported races in popular GPU libraries and applications. It caught 12 races across a popular graph analytics library, Gunrock [50], and a GPU-based irregular application suite, LonestarGPU [11]. Developers of these software already acknowledged eight of them. It caught races in NVIDIA-supported commercial

libraries, e.g., CG, CUB, cuML. The developers acknowledged all of these. In total, iGUARD correctly reported 57 races across 21 applications without any false positives. It not only catches the most comprehensive list of global memory races but also keeps performance overheads limited to 5.1×, on average. Thanks to iGUARD's ability to leverage the GPU's parallelism in race detection, overheads reduced by 15×, on average, over a prior work called Barracuda [17].

2 Background

GPUs organize their hardware resources into a hierarchy to scale to hundreds of thousands of concurrent threads. The basic execution block of a GPU is a Streaming Multiprocessor (SM). Modern GPUs contain up to around 108 SMs. These SMs contain multiple Single-Instruction-Multiple-Data (SIMD) units, which in turn contain multiple lanes of execution (16 - 32). All lanes of a SIMD unit execute the same instruction on different data items in parallel. The SIMD units of an SM share an L1 data cache and a scratchpad (shared memory). While the hardware manages the cache; the programmer decides the contents of a scratchpad. L1 caches and scratchpads are private to each SM, while all the SMs share a larger L2 cache. The GPU's global memory is accessible to all GPU threads and primarily includes the GPU's onboard HBM or GDDR memory. Modern GPUs can have up to 80 GB of global memory. The hardware caches contents of global memory in the L1 and L2 caches.

GPU programming languages, e.g., CUDA or OpenCL, require programmers to arrange threads in a hierarchy of execution groups that mimics the hardware. The smallest execution entity is a thread, which runs on a single SIMD lane. In CUDA, typically 32 threads make up a warp, the smallest hardware-scheduled unit of work. A threadblock is a collection of warps guaranteed to reside within the same SM, while the grid is the largest unit of execution, comprising of multiple threadblocks that together execute a common GPU kernel (CUDA function).

2.1 Advanced GPU features

Traditionally, GPU programs relied on bulk-synchronous parallelism. This model is ably supported by ① threadblock barriers (syncthreads in CUDA) that synchronize threads within a threadblock, ② lockstep execution of threads in a warp resulting in implicit barriers across a warp after every instruction, and ③ implicit barrier across all threads when a kernel finishes execution. However, they fall short for many emerging applications that require semantically richer and finer grain synchronization. Vendors such as NVIDIA, thus, enhanced GPU's synchronization and execution model, enabling a broader set of multi-threaded CPU programs to leverage GPU. We describe three such advanced features.

Scoped synchronization: A globally visible synchronization across thousands of GPU threads is slow. It is also often unnecessary in a GPU's hierarchical programming paradigm. Modern GPUs, thus, provide means to synchronize across only the threads of a particular level in the execution hierarchy. This is referred to as the *scope* of an operation. Currently, CUDA provides three scopes – block, device, and system. An operation's effect is guaranteed only for the set of threads in its scope. For example, an atomic operation with block scope guarantees only atomicity with other atomic operations from within the threadblock of the calling thread. For an operation to impact all the threads in a GPU, the device scope is needed (default). The system scope is useful if an operation should be visible across multiple GPUs and the CPU. OpenCL also provides similar scopes. In this work, we focus on a single GPU and thus, ignore the system scope.

Independent thread scheduling (ITS): Before NVIDIA Volta architecture (circa 2017), threads in a warp always executed in a lockstep. This created an effect of implicit warp-level barriers after every instruction. However, it caused deadlocks in some GPU programs [19]. Consider threads in two warps that compete for the *same set* of locks. A subset of threads of the first warp may acquire their locks, while the rest wait on locks acquired by threads from the second warp. Similarly, a subset of threads of the second warp acquire their locks, but the rest wait for the locks to be released by the first warp. Due to lockstep execution, none of the warps can progress.

Since Volta architecture, a hardware feature called Independent Thread Scheduling (ITS) [22] avoided such deadlocks by allowing threads in a warp to execute divergent paths concurrently, i.e., make independent progress. Consequently, implicit warp-level barriers are no more guaranteed. Instead, programmers should explicitly add warp-level barriers (syncwarp) if lockstep execution is needed for correctness.

Cooperative groups: NVIDIA introduced Cooperative Groups (CG) [24] to enable more flexible synchronization. For example, CG allows programmers to synchronize a chosen subset of warps in a threadblock instead of the entire threadblock. It allows multiple threadblocks of an entire grid to synchronize.

CG is a software abstraction that uses atomics, threadfences, and barriers. For example, to synchronize multiple threadblocks, it uses a counter updated by device-scoped atomics with fences to enforce ordering and synchronizes the threads within the participating threadblocks using syncthreads.

3 Races due to advanced GPU features

This section demonstrates how advanced features of modern GPUs can introduce subtle races if used improperly with the help of examples.

3.1 Races due to scopes

In CPU programs, data races arise when two or more threads access a shared memory location (at least one of the accesses is a write) without intervening synchronization. In GPUs, the mere presence of synchronization cannot prevent a race if it is

```
1  __device__ int getWork(...)
2  {
3    if(tid != 0) return -1; // Only leader thread
4    // Get work from own local partition
5    currHead[blockId] =
6      atomicAdd_block(&nextHead[blockId],
7      NTHREADS); // block scope
8    // Work left in own partition?
9    if(currHead[blockId] < partitionEnd[blockId])
10     return currHead[blockId];
11   // Otherwise steal work
12   int victimBlock = getPartitionToStealFrom();
13   if(victimBlock == -1) return -1; //No work
14   currHead[blockId] =
15     atomicAdd(&nextHead[victimBlock],
16     NTHREADS);
17   if(currHead[blockId] <
18     partitionEnd[victimBlock])
19     return currHead[blockId];
20   return -1; //No work left
21 }
```

Figure 1. Insufficient scope in the atomic operation causing a race in the example CUDA snippet (adopted from ScoRD [28]).

of *insufficient* scope: it fails to encompass both the producer and consumer of data [26, 28].

In current GPUs, both atomic and fence operations can be qualified with a scope. Further, these two operations can be combined to create (scoped) lock/unlock operations [1, 41, 42]. Insufficient scopes in any of these operations lead to a scoped race.[1] For brevity, we only show an example of a scoped race involving atomics.

Let us consider the graph coloring problem [28]. Each thread assigns a color one vertex at a time. The number of vertices typically exceeds the number of GPU threads. The vertices are partitioned among the threadblocks, which then perform multiple iterations to color all vertices in their respective partitions. The amount of work needed to color depends on the number of edges incident on a vertex. Therefore, threadblocks could take different amounts of time to color vertices in their partitions. To improve performance, threadblocks that finish early steal work from others' partitions.

The function getWork() in Figure 1 shows how the leader thread of each threadblock obtains the next set of vertices to be colored after each iteration. The partitionEnd[] holds the tail index of each block's partition in the global array of vertices. Arrays currHead[] and nextHead[] hold the starting indices of vertices to be colored in the current and next iteration, respectively. In lines 5-7, the leader thread updates currHead[] with the present value of nextHead[], while atomically updating nextHead[] using a block-scope atomic. The leader then checks if the threadblock's original partition is empty (lines 9-10). If so, the leader steals from other partitions. It chooses the threadblock (victimBlock) to steal from (line 12). The leader then steals by incrementing the victim's nextHead[] using a device-scope atomic (lines 14-16).

[1] We discuss scoped races to make this paper self-contained. Prior works already described them, and our hardware-based tool ScoRD detects them [28].

```
1  __global__ void reductionKernel(...)
2  {
3    ...
4    if (blockSize >=  4 && tid < 2)
5      sdata[tid] = mySum = mySum + sdata[tid +  2];
6    //__syncwarp(); <-- Needed to avoid race
7    if (blockSize >=  2 && tid == 0)
8      sdata[tid] = mySum = mySum + sdata[tid +  1];
9    ...
10 }
```

Figure 2. Missing warp barrier causing ITS race.

```
1  __global__ void reduce(float *in, float *out, int N)
2  {
3    cg::thread_block block = cg::this_thread_block();
4    cg::grid_group grid = cg::this_grid();
5    reduceBlock(in, out, N, block);
6    cg::sync(block); // Racey, should be grid
7    if (grid.thread_rank() == 0)
8      for (int blk = 1; blk < gridDim.x; blk++)
9        out[0] += out[blk];
10 }
```

Figure 3. Insufficient granularity of sync causing CG race.

One may think that the block-scope atomic used to update nextHead[] in lines 5-7 is sufficient. It is, in the common case, when *no* stealing occurs since the leader updates a variable read only by threads within its threadblock. However, if a threadblock tries to steal from victimBlock's partition when the leader of victimBlock was also assigning itself the next vertex set from its own partition, a subtle scoped race arises. The thread stealing work may not see the update as it falls outside the block scope of the victimBlock.

Barriers and scoped races: Note that threadblock barriers include the effect of a block-scope fence. A barrier additionally waits for all threads in the block to reach the barrier and thus, could be slower. However, functionally, a race preventable by a block-scope fence is also prevented by a barrier.

3.2 Races due to ITS

Any program that implicitly relied on lockstep execution but failed to use syncwarp where needed will have races on modern GPUs. Figure 2 shows a code snippet from a reduction kernel, where threads within a warp sum four elements of an array. In line 5, the first two threads add the last two elements to their current sum and store it. In line 8, the first thread of the warp adds the second element to its current sum and stores it. Under ITS, a syncwarp is needed between these (line 6). Otherwise, the first thread may proceed ahead and execute line 8 while the second thread executes line 5, creating a race.

3.3 Races due to Cooperative Groups (CG)

Races happen if a programmer fails to choose the *right* group of threads in CG that covers the producer and consumer of a data item. Figure 3 shows a simplified code snippet for performing reduction on an array using CG. The input array in is divided among threadblocks, who reduce their subarray and store the result in the array out. The reduction happens in the function reduceBlock (line 5). It takes the thread group performing the subarray reduction as a parameter. Finally, the

Table 1. Comparison of features and requirements of GPU race detectors. *CURD's perf. is Med *only* for syncthreads.

Features / requirements	Barracuda [17]	CURD [39]	Simulee [52]	HaccRG [25]	ScoRD [28]	iGUARD
Sc. fence	Yes	Yes	No	No	Yes	Yes
Sc. atomic	No	No	No	No	Yes	Yes
ITS	No	Lim	No	No	No	Yes
CG	No	No	No	No	No	Yes
Perf. overhead	High	Med*	Med	Low	Low	Med
Needs recompile	Yes	Yes	Yes	No	No	No
Extra H/W	No	No	No	Yes	Yes	No

first thread in the grid reduces the array out to a single value. But before that, all threads in the grid, across all threadblocks, must finish reducing their respective sub-arrays. Therefore, all blocks of the grid should be part of the cooperative group to synchronize. However, in line 6 of Figure 3, only individual threadblocks are synchronized. This leads to a race.

4 Prior race detectors and iGUARD's goals

We are not the first to propose a race detector for GPUs. Thus, we set our goals in the context of existing works.

There exist commercial tools and prior works to efficiently detect races that occur amongst the threads of a threadblock via the scratchpad in an SM [9, 37, 54, 55]. However, these ignore the races that can happen among any pair of hundreds of thousands of GPU threads through the global memory (tens of GBs vs. KBs of scratchpad). We focus on detectors that can find the challenging global memory races.

Table 1 summarizes a qualitative comparison between existing proposals and ours. Closest to our work is Barracuda, a software tool to detect GPU races, including scoped races due to threadfences [17]. It, however, does not detect races due to wrongly scoped atomics. It cannot catch missing warp-level synchronization (syncwarp) under ITS, even if it detects races due to if-else divergence. Barracuda instruments GPU kernels to collect metadata for race detection but does not perform the detection on the GPU. Instead, it serializes and ships the metadata to the CPU for race detection. This simplifies the design; detecting GPU races effectively reduces to that on the CPU. However, it incurs large performance overheads (10-1000×) as the race detection fails to leverage GPU's parallelism.

CURD [39] extends Barracuda by speeding up race detection for traditional kernels that use *only* threadblock barriers through compiler-directed source-code instrumentation. It falls back on Barracuda in the presence of atomics or fences. While CURD reduces overheads for traditional bulk-synchronous programs to 3×, that for the rest remains. It could, in theory, detect races due to ITS but does not support warp-level barriers. Simulee [52] goes beyond race detection to also detect bugs caused when threads in a threadblock do not reach a barrier. However, it focuses only on barriers and is incapable of detecting races caused by atomics or fences.

Barracuda requires recompilation for its shared runtime, while CURD and Simulee are compiler-directed techniques. Consequently, their applicability is limited in the presence of closed-source low-level libraries (e.g., cuDNN, cuBLAS).

HaccRG [25] and ScoRD [28] can check races at low overheads ($< 1\times$) but require significant new hardware that does not exist in today's GPUs. HaccRG ignores scopes and ITS, but ScoRD detects all scoped races. In fact, iGUARD borrows its race detection logic to detect improper use of scopes. However, ScoRD does not detect missing syncwarp under ITS. Note that none of the detectors detect races due to improper use of CG on modern GPUs since none fully support all scoped operations and detect missing syncwarp under ITS.

Goals: We aim to build a GPU race detector with the following goals. ① Comprehensively detect global memory races, including those due to scopes, ITS, and CG *without* new hardware. ② Limit performance overheads of software-based race detection (e.g., <10×, instead of 100×). ③ Avoid reserving GPU's capacity-constrained memory for the race detection in a way that limits its usability to small kernels only. ④ Avoid requiring recompilation or re-linking of applications. ⑤ The detector should work out-of-the-box for a wide range of kernels without developer intervention.

5 Design overview of iGUARD

We first present a high-level overview of iGUARD and how its design philosophy caters to our goals. We will detail its implementation in the next section.

In-GPU race detection: The *entire* process of race detection in iGUARD happens on the GPU without requiring any hardware modifications. As thousands of GPU threads execute in parallel, the corresponding race detection also happens in parallel on the GPU. This is key to iGUARD's performance since there is no serialization due to the CPU.

iGUARD is implemented on top of NVIDIA's NVBit tool [46, 47]. NVBit is a dynamic binary instrumentation tool that enables inspection and modification of CUDA kernel assembly code (SASS) on the GPU without recompilation. iGUARD performs two primary tasks via the instrumentation. ① It collects and updates metadata for active synchronization operations (synchronization metadata) and accesses to each global memory location (memory metadata). The metadata is later used for race detection. It instruments synchronization operations – fences and barriers, and keeps track of their issuing thread, warp, and the scope qualifiers. All loads, stores, and atomics are instrumented to update the memory metadata. ② It implements the race detection using the information of the current instruction and the metadata for the address of the access. Race detection happens on loads, stores, and atomics.

An in-GPU race detector needs to consider thousands of threads. Tracking pairwise interactions for that many threads (e.g., via vector clocks) is infeasible, unlike for CPUs. Therefore, the memory metadata tracks read and write accessors, along with relevant synchronization information, including scopes, for each unit of global memory (here, 4 bytes by default). Similar to ScoRD, we then use classic happens-before relations [29] to check if there have been conflicting accesses to a memory location (i.e., at least one of them is a write) that are not separated by adequate synchronization.

Unfortunately, keeping track of all accessors to a memory location is unrealistic since any of the thousands of threads can be an accessor. Instead, we keep the identity (thread and warp ID) of the last writer and last accessor (reader/writer). This may lead to false negatives if a writer correctly synchronized with the last reader of the same location but not with previous readers. However, based on our experiments, we find this unlikely in practice. If a writer synchronized with the latest reader, it is likely to have synchronized with other readers, directly or transitively. Further, iGUARD tracks if threads accessing a location fall within a single threadblock or span across threadblocks to determine inappropriate use of scopes. In § 6.7, we discuss the trade-offs between keeping detailed accessor information and the practicality of the detector.

To catch races due to ITS, iGUARD tracks thread IDs, beside warp IDs in the metadata and tracks warp-level barriers (syncwarp). iGUARD reports conflicting accesses to a memory location by threads within the same warp if the accesses are not separated by a syncwarp or syncthreads. Note that syncthreads synchronizes threads within a warp too.

Inferring locking protocols: While there is no instruction or intrinsic in CUDA for lock/unlock operations, the CUDA guidebook suggests that atomics and threadfences can be paired to create one [42]. iGUARD infers these instruction pairs as lock/unlock without needing programmer annotation, similar to prior works [17, 25, 28].

However, iGUARD faces a new challenge since it should support ITS. Under ITS, some kernels employ per-thread lock within a warp (e.g., matrix multiplication), while others take one lock per warp where a leader thread performs lock/unlock on behalf of the warp. To fulfill our goal of not requiring developer intervention, iGUARD infers the locking protocol used by an application at runtime. It assumes a warp-level lock by default. However, it monitors if two or more threads from the same warp attempt to acquire locks simultaneously. If so, iGUARD switches to race detection for per-thread locking.

iGUARD detects inappropriate lock/unlock (acquire/release) using the lockset technique [38], instead of happens-before as in ScoRD. Lockset has the advantage that it can detect races that do not manifest during an execution. However, the lockset's useability is limited to lock/unlock only. We augment the synchronization metadata with lock tables used for inferring locks and locking protocol. The memory metadata is augmented with a summary of locks held while accessing a location. The metadata is later used to find if a given location is accessed without holding a common lock(s).

Metadata management: iGUARD requires 16 bytes of memory metadata per 4 bytes of data (4× memory overhead), and a total ∼2MB of synchronization metadata. If the entire metadata needs to reside on GPU's limited device memory, it will inevitably constrain the kernels that the iGUARD can run. It, thus, allocates the metadata using CUDA's Unified Virtual Memory (UVM) feature [23]. With UVM, memory is not pinned (reserved) on the GPU. It is moved between the CPU and GPU on demand by the driver. Since only a small fraction of metadata is updated and used for race detection at a given time, UVM helps keep the "right" amount of metadata in the device memory. In short, iGUARD avoids pinning of GPU's device memory for metadata by leveraging UVM.

Optimizing metadata access: A unique challenge for in-GPU software race detection is the serialization of metadata accesses. Thousands of GPU threads can concurrently access a shared variable. Accesses to the corresponding metadata need to be serialized for race detection correctness. However, serialization can hamstring GPU software. iGUARD, thus, opportunistically coalesces metadata accesses due to race detection to reduce serialization. Further, it employs dynamically adjusted exponential backoff. It adjusts the backoff length based on the number of concurrent threads at runtime to limit metadata contention while also avoiding unnecessary wait. These are key in ensuring low performance overheads for kernels with frequent accesses to shared variables.

Race reporting: iGUARD reports identities of instructions, the address of the data participating in a race, and the cause. iGUARD allocates a 1MB buffer to accumulate information of races without stopping execution on detecting a race. This buffer is sent to the CPU and reported to the programmer when full or when the program ends. It may sometimes happen that a kernel livelocks due to a race. Thus, iGUARD provides a parameterized timeout. On a timeout, iGUARD sends the details of detected races to the CPU before terminating.

6 Implementation of iGUARD

A familiarity with iGUARD's metadata layout is a pre-requisite to appreciate iGUARD's implementation details.

6.1 Metadata layout and allocation

The metadata layout is akin to ScoRD but is extended for ITS, and for better race detection accuracy and performance.

Synchronization metadata: The active synchronization status of threads, warps, and threadblocks are maintained in the synchronization metadata. iGUARD keeps counters for different synchronization operations to identify the latest synchronization operation performed by individual threads, warps, and threadblocks. Specifically, a threadblock barrier counter is kept for *each* threadblock and is incremented upon encountering a syncthreads. Similarly, a warp barrier counter is

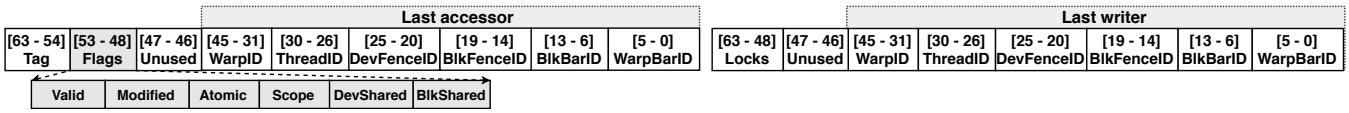

Figure 4. Layout of a single memory metadata entry (16 bytes).

kept for each warp and is incremented when the given warp executes a syncwarp. iGUARD keeps two threadfence counters – one for block-scope and another for device-scope fence for each thread. We keep threadfence counters per thread since CUDA defines the semantics of threadfences for each thread [34]. Under ITS, each thread may have executed different threadfences at a given point in the execution. Finally, we keep a lock table to infer locks and locking protocol at runtime. We will detail the lock table later while explaining how we infer the locking protocol. The synchronization metadata takes $\sim$ 2MB of space in total.

Memory metadata: iGUARD keeps metadata for each unit of global memory (4 bytes, by default). The memory metadata keeps the identity of accessors and access type to a given location, along with the synchronization information. The synchronization information is copied from the synchronization metadata of the accessor at the time of the access. Instead of keeping the identity of all accessors to a location, iGUARD tracks the last writer and last accessor (reader/writer). This keeps the metadata overhead low. In § 6.7, we discuss why limited metadata may not lead to false negatives in practice.

Figure 4 shows the layout of a unit of metadata (16 bytes). The Tag keeps the address tag to uniquely identify 4-byte global memory addresses corresponding to metadata. Flags contains bits to track whether the metadata entry is valid (initialized) and types of accesses to the location. For example, if it has been written to (Modified), accessed via atomics (Atomic) and if so, whether block or device scope was used (Scope). The atomics are treated as stores, and thus, memory metadata tracks atomic operations. Further, DevShared and BlkShared note whether the accessors to a location are spread across multiple threadblocks or part of the same threadblock, respectively. These help to detect incorrect use of scopes.

The metadata keeps the identity (WarpID and ThreadID) of the last writer and last accessor. The ThreadID is necessary to detect races under ITS. The metadata contains the latest synchronization status of the last writer and accessor when those accesses occurred. For example, a 6-bit DevFenceID identifies the latest threadfence executed by the writer/accessor with device scope. The BlkFenceID identifies the same with threadblock scope. The 8-bit BlkBarID tracks the latest syncthreads executed by the accessor, while WarpBarID tracks the latest syncwarp. The WarpBarID is unique to iGUARD. It is needed for tracking missing syncwarp under ITS. Later in this section, we discuss the impact of limited counter sizes on race detection accuracy.

Allocating metadata: While the synchronization metadata needs just 2MB, memory metadata incurs 4$\times$ overhead. If the entire metadata is pinned on the capacity-constrained GPU memory, only a 5th of its capacity would be available to execute the GPU kernel itself. For example, prior works, e.g., Barracuda reserves 50% of the memory capacity for buffers.

Instead, we use the UVM feature of NVIDIA GPUs. We allocate the entire metadata ($\sim 4\times$ of GPU memory capacity) using cudaMallocManaged while initializing the detector. This does not reserve any physical memory – it only allocates virtual addresses. When the detector accesses the metadata, page faults are triggered. The UVM driver then allocates the physical memory for the page containing the requested metadata. The driver also migrates pages between the CPU and the GPU memory based on access patterns. This way, only the needed portion of the metadata resides on the GPU memory.

A drawback of UVM is the overhead of page faults incurred on the first access to the metadata and the cost of migrating pages between the CPU and the GPU. Toward this, iGUARD ensures that the overheads of UVM are incurred only if unavoidable. It keeps an account of free memory available on the GPU by tracking the amount of memory reserved by the application kernel. It instruments CUDA memory allocation API (e.g., cudaMalloc()) invoked by the application kernel for the purpose. If free GPU memory is available after satisfying the application's memory needs, iGUARD pre-faults (all or part of) the metadata onto the GPU by initializing the memory through cudaMemset. Later accesses to the pre-faulted metadata during race detection does not trigger page faults. This way, iGUARD pays the cost of faults and triggers ferrying of data between GPU and CPU only when it is necessary – i.e. when the aggregate of the application kernel's memory needs and metadata exceeds the GPU's memory capacity.

6.2 Metadata updation

iGUARD instruments synchronization operations using the NVBit tool [47]. Figure 5 (a) shows instrumentation code snippet in CUDA for updating synchronization metadata on encountering syncthreads. One of the active threads (line 6) in the threadblock increments the corresponding block barrier counter (BlkBarID). All threads in threadblock are synchronized thereafter. On encountering syncwarp, the warp barrier counter is similarly incremented (code not shown).

Figure 5 (b) shows instrumentation code on encountering a fence operation. iGUARD simply increments the corresponding threadblock or device scope fence counters for the corresponding thread. Threadfences also participate in constituting lock/unlock operations. We will later see how the routine activateLocks helps infer the use of locks on fence operation. These actions are also depicted in Figure 6.

```
__device__ void instr_barrier(uint8_t **syncMD) {
  if(threadId < WARP_SIZE) {
    unsigned mask = __activemask();
    unsigned chosen = ((mask - 1) & mask) ^ mask;
    if((1 << threadId) & chosen) // Leader thread
      ++syncMD[BlkBarId][blockId];
  }
  __syncthreads();
}
```

(a) Instrumentation code for threadblock-level barriers.

```
__device__ void instr_fence(scope_t scope,
  uint8_t **syncMD, uint64_t *locks) {
  switch(scope) {
    case DEV: ++syncMD[DevFenceId][threadId]; break;
    case BLK: ++syncMD[BlkFenceId][threadId]; break;
  }
  activateLocks(scope, locks); // Set lock active bit
}
```

(b) Instrumentation code for scoped fences.

Figure 5. Instrumentation (simplified) for synchronizations.

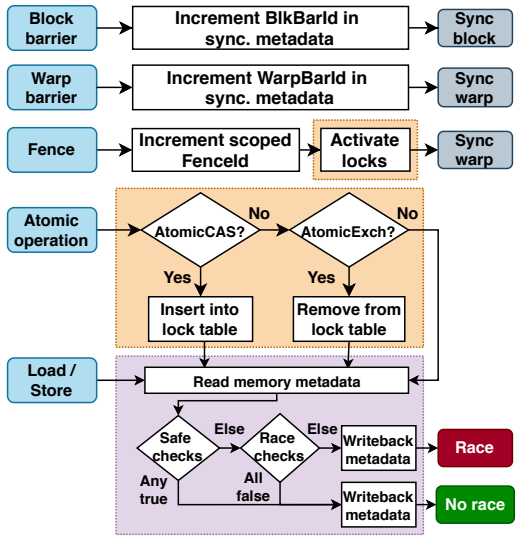

Figure 6. Overview of iGUARD's operation. Purple box shows race detection. Orange boxes shows lock inference.

All memory instructions – loads, stores, atomics – are instrumented to update memory metadata and for race detection itself. On a load, iGUARD first reads the memory metadata corresponding to the load address. It then calculates the threadblock ID of the last accessor by dividing the WarpID in the metadata by the number of warps per threadblock in the executing kernel. The number of warps per threadblock remains constant for a kernel but can vary across kernels. If this calculated threadblock ID differs from the current instruction's theadblock ID, then the DevShared flag in the metadata is set. If the threadblock IDs match but the WarpID in the metadata (last accessor) and that of the load differs, then the BlkShared flag in the metadata is set.

iGUARD updates the WarpID and ThreadID of the last accessor in the metadata with those of issuing load instruction. iGUARD then looks up the synchronization metadata corresponding to the thread/warp issuing the load instruction

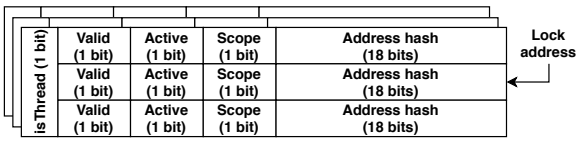

Figure 7. Structure of the lock table.

and copies the latest synchronization information (e.g., barrier counters, fence counters, locks held) into the memory metadata for last accessor (e.g., BlkBarID, WarpBarID, DevFenceID, BlkFenceID, Locks). The Locks field keeps a 16-bit summary (2-way bloom filter) of lock addresses (available in the lock table) currently held by the accessor. In the case of a store, a similar process is followed except for two differences. First, the identity and synchronization information is updated for both the last accessor and the last writer in the memory metadata. Second, the Modified bit in the flag is set. The atomic operations update the metadata in the same way as a store operation, except for one difference. An atomic operation also sets the Atomic bit in the flag and updates Scope bit based on its scope – 0 if device scope, 1 if threadblock scope.

6.3 Inferring the locking protocol

While there are no explicit lock/unlock instructions, the CUDA guidebook [41, 42] specifies that an atomicCAS on a lock variable followed by a threadfence can be used as a lock. A threadfence followed by atomicExch could be used for unlocking. Therefore, we infer these instruction sequences as lock and unlock, like previous works [17, 25, 28].

Similar to ScoRD, iGUARD uses a lock table to infer locks. Each lock table is a 64-bit structure depicted in Figure 7 and a part of the synchronization metadata. On an atomicCAS, iGUARD adds an entry to the lock table of the corresponding warp with an 18-bit hash of the variable's address. The Valid bit is set but the Active bit is not. The Scope bit stores whether the atomic is device or threadblock scope.

On a threadfence, iGUARD sets the Active bit in the corresponding lock table for all entries with matching or narrower scope. An activated entry signifies a lock that is currently held by the given warp. On an atomicExch, the warp's lock table is looked up, and the Valid bit of any entry matching the address hash and scope is unset. Note that even if a programmer misses a threadfence, we will infer the atomicExch as unlock. We will shortly see how a race caused by missing threadfence is caught separately. Further, we keep track of up to 3 separate locks held by each warp at any given time. We found that this is sufficient for practical purposes, but this can easily be extended by provisioning more entries in the table.

To infer the locking protocol under ITS, iGUARD faces a unique challenge. Typically, one thread from each warp acquires/releases locks. However, in some kernels, individual threads within a warp use distinct locks under ITS. iGUARD dynamically infers a kernel's locking protocol, without developer intervention, by ① provisioning both per-warp and per-thread lock tables, and ② switching to the per-thread table on

Table 2. Conditions used during race detection. Memory and synchronization metadata are represented by mm and sm, respectively. Current accessor details is curr.

Definitions	

if(curr.Type == STORE) md = mm.LastAccessor;
if(curr.Type == ATOMIC) md = mm.LastAccessor;
if(curr.Type == LOAD) md = mm.LastWriter;	

Preliminary checks for race-free access	
Type	**Condition**
P1: First access	mm.Valid == false
P2: No write access	mm.Modified == false AND curr.Type == LOAD
P3: Program order access	!md.DevShared AND !md.BlkShared AND curr.ThreadID == md.ThreadID
P4: Warp-synced access	!md.DevShared AND !md.BlkShared AND curr.WarpID == md.WarpID AND (md.WarpBarID != sm.WarpBarID OR curr.ActiveMask contains md.ThreadID)
P5: Barrier access	!md.DevShared AND md.BlockID == curr.BlockID AND md.BlkBarID != sm.BlkBarID
P6: Safe atomic access	mm.Atomic AND curr.Atomic AND (md.BlockID == curr.BlockID OR mm.Scope == DEVICE)
Conditions for racey access	
Type	**Condition**
R1: Scoped-atomic race	mm.Atomic AND mm.Scope == BLOCK AND mm.LastWriter.BlockID != curr.BlockID
R2: Intra-warp race	md.WarpID == curr.WarpID AND md.DevFenceID == sm.DevFenceID AND md.BlkFenceID == sm.BlkFenceID AND !mm.DevShared AND !mm.BlkShared
R3: Intra-block race	md.BlockID == curr.BlockID AND md.DevFenceID == sm.DevFenceID AND md.BlkFenceID == sm.BlkFenceID AND !mm.DevShared
R4: Inter-block race	md.BlockID != curr.BlockID AND md.DevFenceID == sm.DevFenceID
R5: Missing lock race	(mm.Locks != EMPTY OR sm.Locks != EMPTY) AND (md.Locks BITWISE_AND sm.Locks) == 0

detecting simultaneous use of locks by threads within a warp at runtime. iGUARD keeps an isThread bit in the per-warp table (unset by default). On an atomicCAS, iGUARD checks the active mask of the executing warp. The active mask identifies the threads within a warp that are executing an instruction in lockstep. If there is more than one thread performing atomicCAS, iGUARD infers the use of thread-level locking. It sets the isThread bit and proceeds to use the per-thread lock table. The isThread bit is never unset, and the detector does not revert to per-warp locks after it detects the use of per-thread locking. iGUARD accesses the warp-level lock table first but switches to the per-thread table if the isThread bit is set. The locking protocol typically does not change within a single kernel. However, a program can have many kernels, each having its own locking protocol. iGUARD would infer the locking protocol of each kernel independently.

6.4 Detecting and reporting races

Races occur when conflicting memory instructions access a location without intervening synchronization. Thus, iGUARD looks for possible races upon loads, stores, and atomics. Atomics are treated similar to stores for race detection.

Preliminary race checks: Most instructions do not participate in a race [28]. Thus, similar to ScoRD, iGUARD implements a two-tier race detection logic where preliminary checks ascertain if an access is trivially race-free. A detailed race check is employed only if all preliminary checks fail. iGUARD inherits race check conditions from ScoRD but extends them to handle execution under ITS.

The top half of Table 2 lists six possible conditions (P1-P6) for an access to be trivially race-free. The condition P1 states that the first access to a memory location cannot be a race. The second condition says that if a location is unmodified and the current access is not a store, it is race-free.

In the rest of the conditions (P3 - P6), for loads, iGUARD uses the information (e.g., WarpID, ThreadID, synchronization) of the last writer to determine if that load is race-free. For stores, that of the last accessor is used. The condition P3 states that two accesses from the same thread in program order cannot race. The condition ensures that a given location has never been accessed across threadblocks or warps, and the same thread accessed the location the last time.

The condition P4 is unique to iGUARD. It checks if the current and previous accesses are from the same warp (possibly from different threads) but are separated by a warp barrier (syncwarp). A mismatch between the warp barrier ID in the memory metadata and the current value of the warp barrier ID in the synchronization metadata indicates intervening syncwarp(s). If the threads have not diverged (i.e., accessors are within the warp's active mask), their accesses are synchronized at every instruction due to lockstep execution. The fifth condition checks if there has been an intervening threadblock barrier (syncthreads) and if the accessors to the location fall within the same threadblock. Finally, if a memory location is accessed using atomics of sufficient scope, a race cannot occur due to the access.

Detailed race checks: If *all* six conditions (P1-P6) fail, conditions in the bottom half of the table (R1-R5) are checked to confirm the existence of a race and its type. These conditions are checked in the order given in the table. If a condition is satisfied, a race is declared, and later conditions are skipped for that instruction. Race detection continues for the following instructions until the execution finishes.

The condition R1 detects the use of insufficient scope in atomic operations. If the last writer and the current accessor belong to different threadblocks, but the memory location was previously accessed with a block-scoped atomic, a race is declared. As before, if the instruction is a load, then the memory metadata of the last writer is used, while for stores, the last accessor is used. The condition R2 in iGUARD is

specific to its ability to catch races that occur under ITS. It checks if the last and current access to a memory location by threads within a warp are separated by a fence. If not, an intra-warp race due to ITS is declared. Note that it does not check if threads are executing in lockstep. If so, the fourth condition of preliminary checks would have been satisfied.

The condition R3 checks if accesses from threads within a single threadblock are not ordered by a threadfence. If so, an intra-threadblock race is detected. Similarly, R4 declares an inter-threadblock race if it finds accesses from two different threadblocks are not separated by threadfence(s) of device scope. The condition R5 relates to the detection of races due to improper use of locks. If no locks were held during previous accesses to a given memory location or during the current access, locks were not used to order accesses. If locks were used, but the intersection of locks held during the previous access and the current one is empty, it still causes a race.

Note that a race is declared only if all conditions P1-P6 fail and one of R1-R5 is satisfied. It is possible that none of the P or R conditions satisfies. For example, this will happen for accesses correctly protected by locks. No race is declared in such cases. Also, there are no specific checks for cooperative groups. As discussed in § 2.1, NVIDIA created cooperative group primitives using synchronization and ITS support. Since iGUARD supports constituent features, it automatically detects races due to improper use of cooperative groups.

When a race is detected, iGUARD collects the instruction pointer, instruction type, address of the racey memory location, thread and warp ID of the accessor, the type of race, and metadata about the previous accesses to that location. If the binary was compiled with debug information, iGUARD also reports the source line number of the instruction. These are accumulated in a small buffer (1MB) to be passed to the CPU.

Summary: Figure 6 summarizes how iGUARD works. On threadfences and barriers, iGUARD updates the synchronization metadata. On a load/store, iGUARD looks up memory metadata corresponding to the load (store) address. It then performs two-tier checks based on happens-before relations using the information of previous accesses to the location (from metadata) and the current access. The atomics are treated as (special) stores. Thus, on atomics, iGUARD updates memory metadata and performs race checks. iGUARD dynamically infers locks and locking protocol. The race detection for improper locking uses the lockset technique. The entire process happens on the GPU alongside kernel execution.

6.5 Reducing serialization in metadata access

GPU applications thrive in parallelism. We observed that thousands of threads could concurrently access a shared variable in many applications. Even when they do not cause races (e.g., read-only accesses), all accesses to the variable's metadata need to be serialized for correct race detection since the non-existence of a race cannot be affirmed unless checks are complete [51]. Therefore, accesses to a metadata entry needs to be serialized. iGUARD keeps a fine-grain lock per metadata entry to ensure that race detection for accesses to different locations can leverage the GPU's parallelism.

However, applications with many accesses to shared variables still suffer from lock contention for metadata. Fine-grain lock contention is a bigger nemesis to GPU's performance than to CPU's since thousands of threads can concurrently contend for a lock [48, 53]. This challenge is unique to iGUARD's in-GPU software race detection. For example, a prior software-based race detector Barracuda does not update metadata or perform race detection on the GPU. It only logs GPU accesses for the CPU to process later. On the other hand, hardware-based race detectors such as ScoRD propose dedicated new hardware to order metadata accesses and perform race detection – a luxury iGUARD does not have.

iGUARD introduces two novel optimizations to reduce performance overheads due to serialization of metadata accesses.

Coalesced metadata access: We note that not all concurrent accesses to shared variables can race. Consider active threads in a warp that are all loading or performing an atomic operation on a single variable. The scope of the atomic operation is not a concern since all threads within a warp are part of even the smallest possible scope, i.e., threadblock. Also, loads alone cannot race with each other.

Driven by this observation, on a memory access, iGUARD checks if the active threads within a warp are accessing the same location using load or atomic instructions. Specifically, threads in a warp use warp instrincs (e.g., *activemask*, *shfl_sync*) that allow them to quickly communicate with each other to check whether the active threads are accessing a shared location. If so, only a single thread checks for races with threads outside its warp on behalf of all active threads. This allows for opportunistic coalescing of up to 32 serial metadata accesses and race checks into one without the possibility of missing a race.

Dynamic exponential backoff: Coalescing alone is not sufficient. We further use traditional binary exponential backoff to spread out contending accesses to locks [45]. However, we noticed that the upper limit for backoff latency affects its efficacy. If the limit is too low, contention remains, while if it is too high, threads unnecessarily wait.

In GPUs, the number of concurrent threads can vary widely from hundreds to hundreds of thousands. We empirically found that a single upper limit on backoff does not fit all cases. Instead, it should be set dynamically based on the number of concurrent threads in a kernel at runtime. A higher limit is preferable for kernels launched with a larger number of threads than those with fewer threads.

We combine the opportunistic coalescing of metadata accesses with dynamic exponential backoff to reduce the contention for metadata. As a result, iGUARD's race check sped up by 7× for eight applications that originally experienced severe metadata contention, on average (more in § 7).

```
1  __global__ void reductionKernel(...)
2  {
3    ...
4    if (blockSize >=   4 && tid < 2)
5      sdata[tid] = mySum = mySum + sdata[tid +  2];
6    //__syncwarp(); <-- Needed to avoid race
7    if (blockSize >=   2 && tid == 0)
8      sdata[tid] = mySum = mySum + sdata[tid +  1];
9    ...
10 }
```

Figure 8. Missing warp barrier causing ITS race.

```
1  __global__ void lockingKernel(...)
2  {
3    ...
4    // Spin until lock acquired
5    while(atomicCAS(&lock[lockId], 0, 1) != 0);
6    __threadfence();
7    ... // Critical section
8    data[warpId] += value[threadId];
9    __threadfence();
10   atomicExch(&lock[lockId], 0);
11   ...
12 }
```

Figure 9. Race with per-thread locking.

6.6 Tying it all together with examples

We illustrate iGUARD's overall race detection with examples for two of its unique features – detecting races due to ITS and inference of an application's locking protocol at runtime.

Catching ITS races: Figure 8 depicts a CUDA snippet that contains a race due to ITS (repeated from § 3.2). iGUARD instruments all loads/stores and synchronization operations to perform race checks and to update metadata. After thread 1 (tid = 1) executes the load on line 5 (sdata[tid + 2]), the instrumentation code of iGUARD performs checks for potential races (Table 2). It reads the memory metadata corresponding to the address of the variable. Since this is the first access to the address, the Valid bit of the metadata was not set. The access would thus proceed without flagging a race by satisfying preliminary check P1. The instrumentation code then updates the metadata with its thread and warp ID and sets the Valid flag. Similarly, the store operation on line 5 (sdata[tid]) would not trigger a race, but the corresponding metadata would have its Modified flag set.

Thread 0 (tid = 0) later accesses the stored data through a load on line 8 (sdata[tid + 1]). When iGUARD's instrumentation code performs race detection upon this instruction, the preliminary checks P1 - P3 fail as the metadata was marked Valid and Modified by a different thread. P4 - P6 fail too since there were no intervening barriers or atomics.

Next, full race check conditions are evaluated since all preliminary checks failed. Here, condition R2 will evaluate to true since the WarpID of thread 1 matches the WarpID of thread 0 and no intermediate fences have been executed. Upon detecting the race, iGUARD does not stop execution. Instead, it inserts the details of the race (e.g., address of the variable, line number) into a buffer along with its type (here, ITS). Later, a CPU thread asynchronously reads the details of detected races from the buffer and informs the programmer that a race is detected on line 8 due to a load operation by thread 0 to a location previously modified by thread 1.

If there was a syncwarp in line 6, iGUARD would have instrumented it. Upon encountering the syncwarp, the instrumentation code would increment the WarpBarId field of the warp in the synchronization metadata. In this case, when iGUARD performs race detection upon the load on line 8, the WarpBarId of synchronization metadata would not match that in the memory metadata. This would cause preliminary check P4 to evaluate true, and thus, no race would be declared.

Lock inference: Figure 9 shows a code snippet where threads in a warp take individual locks for their critical section and update a variable. Note that without ITS, for example, on GPUs before the Volta architecture, such programs would deadlock. We will next show how iGUARD infers the use of per-thread locking in this code snippet at runtime.

To accommodate varied locking protocols in applications under ITS, iGUARD provisions both per-warp and per-thread lock tables (§ 6.3). On line 5, threads within a warp contend for different locks, causing them to execute the atomicCAS simultaneously. iGUARD detects this by monitoring the warp's active mask – multiple threads within the warp are active while executing the atomicCAS. It thus infers that per-thread locking is used by the kernel and sets the isThread bit of the warp's lock table (Figure 7) to indicate this. It then switches to the individual thread's lock table. It places a hash of the lock's address in the lock table and sets the entry's Valid bit.

Upon encountering the threadfence on line 6, iGUARD would first look up the warp's lock table. However, since the isThread bit is set, the per-thread lock table is consulted instead. The Active bit is set for all entries having their Valid bit set, as a lock acquire has been completed.

On line 8, multiple threads from the same warp concurrently update data[] causing a race. When the first thread accesses data[], iGUARD inserts the details of its active lock table entries into the lock bloom filter in the corresponding memory metadata. When the next thread accesses data[], iGUARD finds that the intersection of locks held by the thread with those stored in the bloom filter is EMPTY (null). This causes condition R5 to fail, leading iGUARD to declare a race due to missing locks. If iGUARD had used only a per-warp lock table, all threads in the warp would have identical lock entries, and the race would not be detected.

6.7 Potential false positives and negatives

We aim to create a practical detector of races due to advanced GPU features. iGUARD excels in catching races across a variety of applications (§ 7). However, it does not guarantee the detection of all races. For example, the size of the barrier and threadfence counters are limited to 6-8 bits. Although very unlikely, counters can wrap around, leading to false positives and negatives. For example, a threadblock should issue exactly 256 syncthreads to cause an error in detection.

Besides counters, iGUARD tracks only the last writer and accessor to a memory location. It is possible that a race constitutes across old accesses but not among the recent ones. However, in practice, it is unusual to have recent accesses properly synchronized but not the distant ones (in time). We empirically confirmed this by tracking the last 2, 4, and 8 accessors to a memory location in the metadata instead of only the last accessor (default in iGUARD). Tracking longer access history did not find any new races for any of the programs we evaluated in this work. A previous work on detecting data races in CPU multi-threaded applications, called FastTrack [20], also made similar observations for CPU applications – reads to a given location are typically totally ordered in practice.

On the other hand, the overhead of the metadata grows linearly with the number of previous accessors that it keeps. Therefore, iGUARD keeps the identity of only the last accessor as a pragmatic choice. Alternatively, it is also possible to collect metadata and ship it over to the CPU. However, the CPU would then have to perform race detection, as in previous works (e.g., [17]). In summary, for in-GPU race detection with low-performance overhead, iGUARD judiciously chose the metadata layout and its size without impacting the usefulness of the race detection in practice.

Table 3. System configuration

CPU	Intel Xeon Gold 6242 (16 cores) @ 2.80GHz
GPU	NVIDIA Titan RTX (72 SMs, 24 GB GDDR6)
DRAM	768 GB DDR4 @ 2933 MHz
Software	CUDA 11.0, Ubuntu 20.04

7 Evaluation

Table 3 details of the evaluation platform. We use workloads from previous works, e.g., Barracuda, CURD [17, 39] (CUB, Rodinia, SHoC), and ScoRD [28] (ScoR) for a comprehensive comparison. Besides, we used two popular graph processing frameworks, Gunrock [50] and LonestarGPU [11], a GPU hash table called SlabHash [2], and example applications from NVIDIA's CG [24] and cuML [40] libraries for a wide coverage. In total, we evaluated iGUARD on 42 workloads from 10 workload suites [2, 10, 12, 13, 21, 28, 33, 35, 40, 49]. Half of these workloads had races while the rest did not.

Table 4 and Table 5 contain the names of the workload suites and the applications used from that suite. Table 4 contains applications with global memory races and those in Table 5 are race-free. The applications with races help us determine the accuracy of race detection. Those without races aid in evaluating if iGUARD reports false positives. Both help in measuring the overheads of race detection.

To put iGUARD's accuracy and performance in perspective, we compare it with Barracuda. For a fair comparison, we disable shared memory race detection in Barracuda since iGUARD focuses only on global memory races. Further, iGUARD runs on the latest CUDA 11.0 on recent ISA (SM70). However, Barracuda cannot run on SM70 and is compiled to SM35 for backward compatibility. To ensure fair performance comparison, we report Barracuda's overheads by normalizing its runtime over that for applications (without Barracuda) also compiled to SM35. Similarly, the overhead of iGUARD is normalized over the application runtimes compiled to SM70. This way, any potential differences due to ISA cancel out.

Table 4. Races detected by Barracuda and iGUARD.
IL: Improper locking, AS: Insufficient atomic scope,
ITS: ITS induced race, BR: Intra-block race,
DR: Intra-device race *Did not terminate.

Suite	Application	Barracuda	iGUARD	Types
ScoR	matrix-mult	-	4	IL, AS, BR
	1dconv	-	1	AS
	graph-con	-	5	AS, BR, DR
	reduction	-	7	ITS, BR, DR
	rule-110	-	2	AS, DR
	uts	-	6	IL, AS
	graph-color	-	6	AS, BR, DR
CG	conjugGMB	-	1	CG (DR)
	reduceMB	-	1	CG (DR)
NVlib_CG	grid_sync	-	1	DR
Gunrock	louvain	-	3	ITS
	pr_nibble	-	1	BR
	sm	-	1	BR
	color	-	2	BR
Lonestar (LS)	mis	-	2	BR, DR
	cc	-	3	BR, DR
SlabHash	slabhash_test	-	1	DR
cuML	cuML_gsync	-	1	DR
Kilo-TM	interac	3*	4	BR, DR
	hashtable	2	2	DR
SHoC	shocbfs	2	2	BR
CUB	cub_gridbar	1	1	DR

Table 5. Applications without any reported races.

Suite	Applications			
CUB	b_radix_sort	b_reduce	b_scan	d_part_flag
	d_part_if	d_radix_sort	d_reduce	d_scan
	d_sel_flag	d_sel_if	d_sel_uniq	d_sort_find
Rodinia	dwt2d	hotspot	hybridsort	kmeans
	needle	nn	pathfinder	srad
CG	warpAA			

```
1  void sync_grid(int gridSize, volatile int *arrived)
2  {
3    //__threadfence();
4    __syncthreads();
5    if(threadId == 0) {
6      __threadfence();
7      atomicAdd(arrived, 1);
8      while(*arrived != gridSize);
9    }
10   __syncthreads();
11 }
```

Figure 10. Simplified implementation of grid-level sync.

7.1 Detecting races

Table 4 shows the races caught by iGUARD and their types. For comparison, we report races caught by Barracuda. Table 5 lists applications that Barracuda and iGUARD did not report

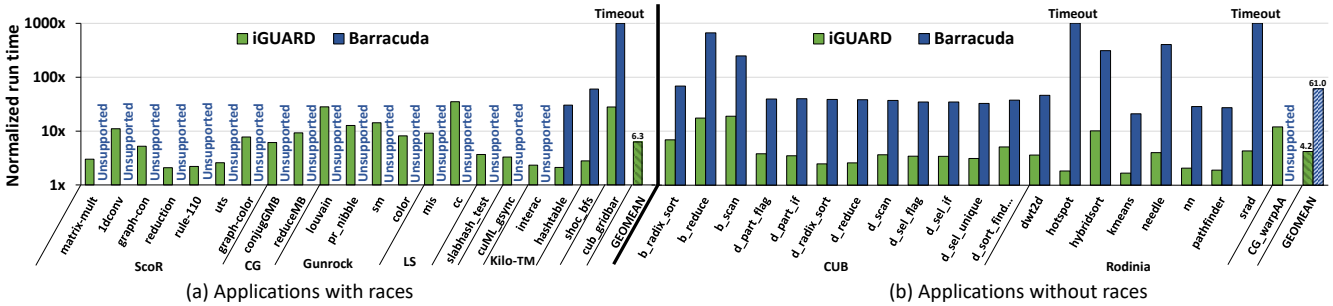

Figure 11. Performance overhead of iGUARD and Barracuda normalized with no race detection. Note the log scale.

any global memory races for. In total, iGUARD detected 57 races in 21 applications without any false positives. The other 21 workloads did not have races.

Across the workloads in Kilo-TM, SHoC, CUB and Rodinia used by Barracuda, iGUARD detected all seven races that were reported. Further, Barracuda did not terminate for interac kernel of Kilo-TM and misses a true race. iGUARD correctly caught that race and successfully executed the application. Barracuda could not run workloads from the ScoR suite [27] and CG [35] since they have operations like scoped atomics which it does not support. Barracuda also failed to run Gunrock, LonestarGPU, SlabHash, and cuML. Since Barracuda executes on PTX, the intermediate representation for NVIDIA GPUs [36], and not the binary, it requires a single PTX file to be embedded in a binary. It cannot handle large, multi-file real-world GPU libraries like Gunrock or LonestarGPU.

iGUARD detected all 26 races reported in ScoR [27]. In addition, iGUARD caught 5 more previously unreported true races in ScoR due to ITS. ScoRD did not report them since it does not support ITS. To test if iGUARD detects races due to improper use of cooperative groups, we modified example applications in NVIDIA's CG library to introduce improper use of cooperative groups. iGUARD detected all those races.

Reporting true previously unreported races: iGUARD detected several previously unreported races. It detected a race in NVIDIA's CG library implementing grid-level synchronization (NVlib_CG in Table 4). Figure 10 shows a simplified code snippet implementing grid-level synchronization. As per CUDA, barrier synchronization operations ensure that threads proceed after barrier only after all threads in its group reach the barrier (execution barrier). It also guarantees that writes performed by threads before the barrier are visible to all threads in that group after the barrier (memory barrier). However, the grid sync implementation fails to guarantee the memory barrier property, causing races in applications. Observe that threadfence in line 6 of Figure 10 is executed only by the leader thread of each threadblock. However, the effect of a threadfence is limited to writes of the *calling thread* only. Therefore, after the grid sync, threads are not guaranteed to observe writes performed by all threads in the grid before the sync, except for the leader threads. NVIDIA communicated that they filed an internal bug report based on this. iGUARD caught similar races in cuML's and CUB's grid sync implementation, which developers have acknowledged.

iGUARD detected 7 races in Gunrock [49, 50] ($>$ 7700 LOC), and 5 races in LonestarGPU [10, 11] ($>$ 6400 LOC) – two popular GPU graph frameworks. We detected intra-thread block races and ITS races. Gunrock developers acknowledged 3 races, while LonestarGPU developers acknowledged all.

7.2 Comparing performance overhead

Figure 11 (a) and (b) shows (log scale) performance overhead introduced by iGUARD for applications in Table 4 and Table 5, respectively. We also compare overheads with Barracuda for whichever applications it could run.

We could report numbers for Barracuda only for three applications in Figure 11 (a). This is because it failed to run for other applications either for lack of support for scoped atomics (ScoR), lack of ITS support (interac, CG) or because of application complexity where compilers fail to embed PTX for large libraries (Gunrock, Lonestar). This also demonstrates the wider applicability of iGUARD. Figure 11 (a) shows for several compute-heavy applications, e.g., rule-110, reduction, the overheads are limited to 2-3×. To put it in perspective, we compared against Barracuda, wherever it ran. For example, Barracuda slowed down hashtable and shocbfs by 30× and 60×, respectively. That for iGUARD were 2.1× and 2.8×.

Figure 11 (b) provides a picture of relative performance overheads as many applications with traditional synchronization ran on Barracuda. On average, Barracuda had an overhead of 61× for the 18 workloads in this graph that did not time out. iGUARD had a 4.2× overhead for the 21 workloads in this graph. Across all 20 workloads (2 in Figure 11 (a) and 18 in Figure 11 (b)) that Barracuda ran without timing out, it incurred a 58.9× overhead, on average. For the same set, iGUARD's overhead is 3.9×. Thus, iGUARD not only has broader applicability but also significantly lowers performance overhead. Across all 42 workloads, iGUARD incurred 5.1× slowdown over no race detection, on average.

7.3 Benefits of reduced serialization in metadata access

We now analyze the performance improvement due to reduced serialization in metadata access thanks to iGUARD's coalesced metadata access and dynamic backoff (§ 6.5).

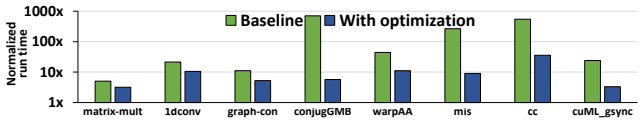

Figure 12. Performance overhead of iGUARD with and without contention optimisations normalized against no detection.

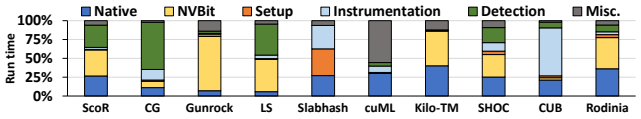

Figure 13. Breakdown of application runtime with detection.

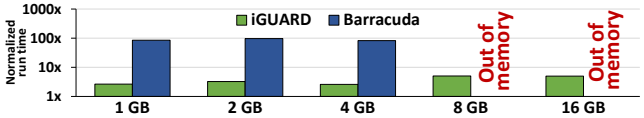

Figure 14. Overheads with memory footprint scaling.

Figure 12 shows the performance overhead of race detection, with and without the aforementioned optimizations for a subset of workloads that originally experienced high overhead due to serialization in metadata access. The overheads of these applications were reduced by 7×, on average. Notably, conjugGMB's overhead dropped from 706× to 6×. It launches many threads (73728) that synchronize by spinning on a shared variable. Other applications outside this subset did not experience much serialization and are agnostic to these optimizations. We also confirmed that these optimizations did not affect the accuracy of race detection in any way.

7.4 Understanding sources of overheads

Figure 13 presents the fraction of runtime contributed by different components of iGUARD for each benchmark suite (averaged). Native refers to the application runtime without race detection. NVBit refers to time spent by the NVBit tool analyzing the binary and inserting instrumentation calls. Setup is the time spent on allocating and initializing metadata. Instrumentation is the delay to the kernel execution due to instrumentation without race detection. Detection captures the time taken to perform the race detection, while Misc. captures all other overheads like loading kernel etc.

While the primary source of overheads varies based on applications, NVBit itself is often a key contributor. If NVBit is optimized in later versions, or a faster instrumentation tool emerges, iGUARD will quicken up. Applications in the CG suite are designed to demonstrate different synchronization operations and have limited computation. Therefore, race detection overheads dominate in them. Applications in CUB are short running and thus, the Instrumentation time dominates.

7.5 Scaling with memory footprint

iGUARD does not reserve GPU memory for metadata. To demonstrate the value of this approach, compared to the memory reservation approach adopted in previous works, we ran both detectors for an application with varying memory footprint (d_reduce from CUB). Figure 14 shows the performance overheads with iGUARD and Barracuda. Barracuda's overheads are significantly more. Importantly, beyond 8 GB of application memory footprint, Barracuda failed due to lack of memory. However, iGUARD continued to detect races, albeit with larger overheads due to additional page faults introduced by the UVM driver to accommodate larger metadata sizes. This demonstrates iGUARD's ability to work with practical data sizes with graceful degradation for increased memory usage instead of failing.

8 Related work

We discussed GPU race detectors closest to ours in § 4. Here, we discuss a few other GPU race detectors. Early works on GPU race detection ignored the harder-to-detect races on global memory and focused solely on races in the scratchpad among threads within a threadblock. Boyer et al. [9] utilize instrumentation and emulation to find such bugs at a heavy performance overhead. GRace [54], and GMRace [55] use static analysis and dynamic checking to detect races. LDetector [30] detects races in scratchpad and global memory by taking snapshots and comparing changes in values but ignores fences and atomics. SMT solving [3, 4, 6, 7] has been proposed to find data races at the risk of detecting false positives. However, no prior work covers all types of races due to modern GPU features, including scopes, ITS and CG.

Concurrency bug detection has been extensively explored for multi-threaded CPU software [5, 8, 14–16, 18, 20, 31, 32, 43, 44, 56]. While these provide inspiration for GPU race detection, they cannot be directly applied due to challenges that arise from the GPU's massive parallelism. CPUs also lack the advanced synchronization operations present in GPUs.

9 Conclusion

To enable a broader set of applications to leverage GPUs, vendors are progressively introducing semantically richer synchronization and expressive execution paradigms. However, improper use of these can lead to subtle races that could threaten GPU software reliability. We thus propose iGUARD, a detector of global memory races, including those induced by advanced features. The detector reported several previously unreported races, including some in NVIDIA's own libraries. The in-GPU race detection without CPU assistance allows iGUARD to detect races with significantly less overhead.

Acknowledgments

We thank the anonymous reviewers and our shepherd Junfeng Yang for their constructive feedback. We thank Ajay Nayak for helping us with workloads. This work is supported by research grants from VMware Inc. Arkaprava is supported by a Young Investigator Fellowship by Pratiksha Trust, Bangalore.

References

[1] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. ACM, New York, NY, USA, 577–591. https://doi.org/10.1145/2694344.2694391

[2] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. 2018. A Dynamic Hash Table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, USA, 419–429. https://doi.org/10.1109/IPDPS.2018.00052

[3] Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. 2014. Engineering a Static Verification Tool for GPU Kernels. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 226–242. https://doi.org/10.1007/978-3-319-08867-9_15

[4] Ethel Bardsley and Alastair F. Donaldson. 2014. Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels. In *Proceedings of the 6th International Symposium on NASA Formal Methods - Volume 8430*. Springer-Verlag New York, Inc., New York, NY, USA, 230–245. https://doi.org/10.1007/978-3-319-06200-6_18

[5] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-fly Maintenance of Series-parallel Relationships in Fork-join Multithreaded Programs. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Barcelona, Spain) *(SPAA '04)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/1007912.1007933

[6] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: A Verifier for GPU Kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. ACM, New York, NY, USA, 113–132. https://doi.org/10.1145/2384616.2384625

[7] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. 2015. The Design and Implementation of a Verification Technique for GPU Kernels. *ACM Trans. Program. Lang. Syst.* 37, 3, Article 10 (May 2015), 49 pages. https://doi.org/10.1145/2743017

[8] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

[9] Michael Boyer, Kevin Skadron, and Westley Weimer. 2008. Automated Dynamic Analysis of CUDA Programs. In *2008 Workshop on Software Tools for MultiCore Systems*.

[10] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A Quantitative Study of Irregular Programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '12)*. IEEE Computer Society, Washington, DC, USA, 141–151. https://doi.org/10.1109/IISWC.2012.6402918

[11] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2021. LonestarGPU. https://iss.oden.utexas.edu/?p=projects/galois/lonestargpu.

[12] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09)*. IEEE Computer Society, Washington, DC, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[13] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1735688.1735702

[14] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. 2015. Race Detection in Two Dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA '15)*. ACM, New York, NY, USA, 101–110. https://doi.org/10.1145/2755573.2755601

[15] Anne Dinning and Edith Schonberg. 1991. Detecting Access Anomalies in Programs with Critical Sections. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging* (Santa Cruz, California, USA) *(PADD '91)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/122759.122767

[16] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. 2012. IFRit: Interference-free Regions for Dynamic Data-race Detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. ACM, New York, NY, USA, 467–484. https://doi.org/10.1145/2384616.2384650

[17] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. BARRACUDA: Binary-level Analysis of Runtime RAces in CUDA Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 126–140. https://doi.org/10.1145/3062341.3062342

[18] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 245–255. https://doi.org/10.1145/1250734.1250762

[19] Ahmed ElTantawy and Tor M. Aamodt. 2016. MIMD Synchronization on SIMT Architectures. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 11, 14 pages.

[20] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[21] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. 2011. Hardware Transactional Memory for GPU Architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 296–307. https://doi.org/10.1145/2155620.2155655

[22] Olivier Giroux, Luke Durant, Mark Harris, and Nick Stam. 2017. Inside Volta: The World's Most Advanced Data Center GPU. https://devblogs.nvidia.com/inside-volta/. Accessed: 2019-11-20.

[23] Mark Harris. 2017. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[24] Mark Harris and Kyrylo Perelygin. 2017. Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups/. Accessed: 2020-11-19.

[25] Anup Holey, Vineeth Mekkat, and Antonia Zhai. 2013. HAccRG: Hardware-Accelerated Data Race Detection in GPUs. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing (ICPP '13)*. IEEE Computer Society, Washington, DC, USA, 60–69. https://doi.org/10.1109/ICPP.2013.15

[26] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free Memory Models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City,

Utah, USA) *(ASPLOS '14)*. ACM, New York, NY, USA, 427–440. https://doi.org/10.1145/2541940.2541981

[27] Aditya K Kamath, Alvin A George, and Arkaprava Basu. 2019. Scoped Racey Benchmark Suite. https://github.com/csl-iisc/ScoR/. Accessed: 2020-11-15.

[28] Aditya K. Kamath, Alvin A. George, and Arkaprava Basu. 2020. ScoRD: A Scoped Race Detector for GPUs. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 1036–1049. https://doi.org/10.1109/ISCA45697.2020.00088

[29] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[30] Pengcheng Li, Chen Ding, Xiaoyu Hu, and Tolga Soyata. 2014. LDetector: A low overhead race detector for GPU programs. In *5th Workshop on Determinism and Correctness in Parallel Programming (WODET2014)*.

[31] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic Race Detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 443–457. https://doi.org/10.1145/3009837.3009857

[32] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1168857.1168864

[33] Duane Merrill. 2015. Cub: Cuda unbound. *http://nvlabs.github.io/cub* (2015).

[34] NVIDIA. 2021. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2021-05-07.

[35] NVIDIA. 2021. CUDA Samples. https://docs.nvidia.com/cuda/cuda-samples/index.html. Accessed: 2021-05-07.

[36] NVIDIA. 2021. Parallel Thread Execution ISA Version 7.3. https://docs.nvidia.com/cuda/parallel-thread-execution/. Accessed: 2021-05-07.

[37] NVIDIA. 2021. Racecheck Tool. https://docs.nvidia.com/cuda/cuda-memcheck/index.html. Accessed: 2021-05-07.

[38] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/781498.781528

[39] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: A Dynamic CUDA Race Detector. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 390–403. https://doi.org/10.1145/3192366.3192368

[40] Sebastian Raschka, Joshua Patterson, and Corey Nolet. 2020. Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *arXiv preprint arXiv:2002.04803* (2020).

[41] Jason Sanders and Edward Kandrot. 2010. *CUDA by Example: An Introduction to General-Purpose GPU Programming* (1st ed.). Addison-Wesley Professional, Boston, MA, USA.

[42] Jason Sanders and Edward Kandrot. 2021. CUDA By Example - Errata Page. https://developer.nvidia.com/cuda-example-errata-page. Accessed: 2020-05-01.

[43] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[44] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[45] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs. In *IEEE International Symposium on Workload Characterization (IISWC)*.

[46] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 372–383. https://doi.org/10.1145/3352460.3358307

[47] Oreste Villa, Zi Yan, and David Nellans. 2019. NVBit Source Code. https://github.com/NVlabs/NVBit/. Accessed: 2020-11-15.

[48] Kai Wang, Don Fussell, and Calvin Lin. 2019. Fast Fine-Grained Global Synchronization on GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 793–806. https://doi.org/10.1145/3297858.3304055

[49] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

[50] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2021. Gunrock. https://github.com/gunrock/gunrock.

[51] Benjamin Wester, David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2013. Parallelizing Data Race Detection. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2451116.2451120

[52] Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2020. Simulee: Detecting CUDA Synchronization Bugs via Memory-Access Modeling. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 937–948. https://doi.org/10.1145/3377811.3380358

[53] Ayse Yilmazer and David Kaeli. 2013. HQL: A Scalable Synchronization Mechanism for GPUs. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. IEEE Computer Society, USA, 475–486. https://doi.org/10.1109/IPDPS.2013.82

[54] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: A Low-overhead Mechanism for Detecting Data Races in GPU Programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1941553.1941574

[55] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2014. GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. *IEEE Trans. Parallel Distrib. Syst.* 25, 1 (Jan. 2014), 104–115. https://doi.org/10.1109/TPDS.2013.44

[56] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. 2007. HARD: Hardware-Assisted Lockset-based Race Detection. In *2007 IEEE 13th*

International Symposium on High Performance Computer Architecture (HPCA '07). IEEE, Piscataway, NJ, USA, 121–132. https://doi.org/10.1109/HPCA.2007.346191