SuperCollider: Scalable and Effective Data Race Detection for CUDA

MARK STEPHENSON, NVIDIA, USA
SANA DAMANI, NVIDIA, USA
MOHAMED TAREK IBN ZIAD, NVIDIA, USA
ANIS LADRAM, NVIDIA, USA
MICHAEL GARLAND, NVIDIA, USA

Data races, which occur when two or more threads incorrectly access the same memory location without appropriate synchronization, cause CUDA programmers considerable pain. Even experts struggle to reason about extreme parallelism, multiple memory spaces and scopes, and diverse synchronization mechanisms. Races can manifest as subtle data corruption, deadlock, or livelock, which are difficult to reproduce in a debugging environment. To date, there is no generally reliable tool for identifying races in GPU programs at scale. State-of-the-art dynamic GPU race detectors attempt to detect data races by faithfully tracking the memory operations, barriers, and memory fences executed by tens of thousands of threads. While this complexity enables the detection of subtle races, it comes with a cost: existing tools incur substantial memory-footprint overheads (*e.g.*, greater than 9×) and/or runtime overheads (*e.g.*, 1000×), and tend to report false positives. This paper introduces SuperCollider, a simple race detector for CUDA that tracks only thread-level state, is agnostic to synchronization protocols, introduces no memory-footprint overhead, seamlessly handles shared, global, and tensor memory spaces, does not produce false positives, and is relatively fast. Although SuperCollider cannot detect all races, we demonstrate that it effectively catches many data races in professionally authored CUDA programs, including races that prior art cannot. Our approach exposed over 90 data races across libraries and CUDA benchmarks, and even found a race in the CUDA Programming Guide. Unlike prior art, SuperCollider can operate at the extreme scales required by modern GPU workloads.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → *Software security engineering*; • **Computer systems organization** → Heterogeneous (hybrid) systems.

Additional Key Words and Phrases: CUDA, GPU, thread safety, data races

ACM Reference Format:
Mark Stephenson, Sana Damani, Mohamed Tarek Ibn Ziad, Anis Ladram, and Michael Garland. 2026. SuperCollider: Scalable and Effective Data Race Detection for CUDA. *Proc. ACM Program. Lang.* 10, PLDI, Article 261 (June 2026), 25 pages. https://doi.org/10.1145/3808339

1 Introduction

From self-driving cars and robotics to confidential computing, cancer treatment planning, and chatbots, modern life depends on GPU compute. However, even in robustness-critical domains, data races—types of concurrency errors in parallel programming—remain a significant liability in GPU programs. A data race arises when two or more threads can concurrently access the same memory

Authors' Contact Information: Mark Stephenson, NVIDIA, Austin, USA, mstephenson@nvidia.com; Sana Damani, NVIDIA, Santa Clara, USA, sdamani@nvidia.com; Mohamed Tarek Ibn Ziad, NVIDIA, Westford, USA, mtarek@nvidia.com; Anis Ladram, NVIDIA, New York, USA, aladram@nvidia.com; Michael Garland, NVIDIA, Santa Clara, USA, mgarland@nvidia.com.

ACM 2475-1421/2026/6-ART261
https://doi.org/10.1145/3808339

```
mul vr0 = vr1, 4
add vr2 = %addr, vr0
read.global vr3 = [vr2]  // Weak load
mul vr4 = vr3, 3.0
```

(a) Original code.

```
mul vr0 = vr1, 4
add vr2 = %addr, vr0
read.global vr3 = [vr2]                // Weak load
nanosleep.rand #rdelay                 // Random delay
read.global.sys.strong vr5 = [vr2]     // Strong load
if vr3 != vr5 {                        // Comparison
   // Race detected! Report error.
   call sc_err(pRuntimeStr, PC, %addr, CLOBBERED_READ)
}
mul vr4 = vr3, 3.0
```

(b) Instrumented code.

Fig. 1. SuperCollider instrumentation for weak loads. (a) Original code. (b) Instrumented code with delay and strong load for race detection. We extend this approach to similarly handle writes and asynchronous memory copies.

location, at least one access is a write, and there is no synchronization mechanism to control the order of accesses. Because the value at the shared location[1] depends on which thread wins the race, program behavior becomes nondeterministic, making these bugs difficult to reproduce and debug.

Although races are not unique to GPUs, the scale of parallelism enabled by the CUDA programming model exacerbates the programming challenge. CUDA organizes kernels into a grid of thread blocks, which are further composed of warps that contain 32 threads each. Kernels can execute concurrently with other kernels and CUDA provides primitives for synchronizing threads at each level. The thread hierarchy is mirrored by a multilevel memory hierarchy, comprising *global*, *shared*, and *local* memory, each with distinct access speeds and visibility scopes. The multilevel structure of threads and memory spaces means races can occur anywhere in the system: between threads within a single warp, warps in the same block, threads in different blocks, threads in different kernels, threads in different GPUs, or between CPU and GPU threads.

State-of-the-art dynamic GPU race detectors operate by tracking every memory and synchronization operation performed by every thread in the kernel. Such tracking provides excellent race coverage and is the gold standard for GPU race detection. Tools such as iGUARD [14], HiRace [10], and NVIDIA's Compute Sanitizer [23] are indispensable and can expose subtle races. However, anyone who has extensively used these tools quickly recognizes their limitations, primarily because tracking all of that state is a tricky proposition. Many CUDA kernels launch tens of thousands of threads that collectively access multiple gigabytes of data.

Unsurprisingly, iGUARD [14] and HiRace [10], for instance, require memory footprint overheads of $17x$ and $9x$, respectively, to store metadata to track memory accesses at byte granularity. Compute Sanitizer [23], on the other hand, trades memory overhead for run time overhead, regularly incurring over $1,000\times$ the runtime of a native execution.

In this paper, we introduce SuperCollider, a tool that takes a stateless and synchronization-agnostic approach to detecting races in CUDA applications. Inspired by DataCollider [5], a successful race detector for operating system kernel development, SuperCollider applies a straightforward transformation to all of the so-called *weak* loads and stores in device-side CUDA code. Figure 1 illustrates the core idea: a compiler pass expands every weak memory operation in the program (excluding *strong* memory operations and atomics) with two accesses to the same address separated by a random delay, followed by an equality check of the values fetched from memory. The key insight is simple: if back-to-back loads separated by a random delay—which unpredictably perturbs

[1]We use the term *shared location* to denote a memory location accessible by two or more threads, differentiating it from the GPU-specific scratchpad called *shared memory*.

thread scheduling—return different values, we can confidently declare a data race. The values from weak memory operations should not change under our feet!

Because weak memory operations constitute the majority of GPU memory operations, SuperCollider is effective at discovering races. While we do not view SuperCollider as a substitute for prior art, its simplicity allows developers to check any GPU application for races, including applications that operate at massive scale. Additionally, because SuperCollider is agnostic to synchronization and memory fence primitives, it naturally supports numerous features out of the box. For instance, to the best of our knowledge, no other tool can detect races across multiple kernels, across OS processes, between the CPU and GPU, or across multiple GPUs.

In this paper, we make the following contributions:

- We demonstrate the effectiveness of SuperCollider using real CUDA applications and libraries, exposing dozens of race conditions in the process.
- We extend DataCollider's core algorithm to detect "lost update" hazards, including intra-warp hazards.
- We add support for tensor memory accelerator (TMA) operations, including bulk copies.
- We expose races using layered scheduling randomization, including block, warp, and subwarp shuffling, promoting opportunities for distant races to occur.

2 Background and Motivation

Our work adds to the growing body of evidence that data races are widespread in CUDA applications [10, 14]. This section aims to give the reader a sense of why that is and why CUDA programmers should care. We start with the definition of a data race.

2.1 Data Races

We adopt the definition of a data race presented by Erickson and Musuvathi [5]. Two memory accesses are *conflicting* if:

(1) The accesses overlap (*i.e.*, touch some of the same bytes).
(2) At least one of them is a write.
(3) At least one of them is not a synchronization access.

A *data race* occurs when a multiprocessor performs two conflicting memory accesses simultaneously. Although the definition is unchanged on GPUs, GPU programs often contain tens of thousands of concurrently executing threads, in stark contrast to the scale of multithreaded CPU applications. Additionally, CUDA's memory model programmatically distinguishes regular accesses from synchronization accesses, which SuperCollider leverages to guarantee the absence of false positives.

2.2 Why Bother with Race Detection?

Data races are dangerous because they introduce nondeterminism and unpredictable behavior into concurrent programs. Races are difficult to debug and verify, and can lead to subtle, intermittent, and severe issues, such as data corruption, logical errors, security vulnerabilities, and even death [17].

In our experiments, the randomness introduced by SuperCollider caused several racy programs to fail, for example. However, in an unsafe language like CUDA, we must also worry about the platform's memory model. SuperCollider identified several race conditions in programs that are "benign" in the sense that the compiler and architectures *currently* handle the programs according to the programmer's expectations. However, a new just-in-time compiler in a future driver update could result in trouble for applications with such races. Data races are considered undefined behavior (UB) in CUDA. As prior research demonstrates, compilers leverage UB to perform specific

```
1  __global__ void racyKernel(int *ret) {
2      // Initialize shared with a weak store.
3      __shared__ int sum = 0;
4
5      __syncthreads();
6
7      for (int i = 0; i < 4; i++) {
8          atomicAdd(&sum, 1);
9      }
10
11     __syncthreads();
12
13     if (threadIdx.x == 0) {
14         // We expect sum == (4 * blockDim.x)
15         ret[blockIdx.x] = sum;
16     }
17
18 }
```

(a) Racy code.

```
__global__ void optimizedRacyKernel(int *ret) {
    // Convert memory to thread local storage
    int sum = 0;

    for (int i = 0; i < 4; i++)
        sum += 1;

    if (threadIdx.x == 0)
        ret[blockIdx.x] = sum;
}
```

(b) Optimized code.

```
__global__ void optimizedRacyKernel(int *ret) {
    if (threadIdx.x == 0)
        ret[blockIdx.x] = 4;
}
```

(c) Constant folded code.

Fig. 2. A hypothetical optimization that converts a data race into an optimization "opportunity".

optimizations. John Regehr's tutorial on undefined behavior quotes the alarming consequences of UB in the C FAQ [38]:

> *Anything at all can happen*; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.

Figure 2a illustrates a typical pattern (with a race condition) to initialize a shared-memory variable. Leveraging the CUDA memory model, a hypothetical optimizing compiler can assume that only one thread performs the weak memory store on line 3. Otherwise, there would be a race condition, which is UB. Because CUDA expressly forbids UB, a compiler could justify converting the `atomicAdd` to thread-local arithmetic, and constant-folding the computation, as the panels on the right show. A kernel that stores "4" is efficient, but not what the programmer anticipated.

The example above is hypothetical, but it is not far-fetched. The interested reader can find many such examples in the literature of compilers that optimize around UB [16, 47]. Contentious compiler bug reports clearly show a tension between compiler writers and developers who use seemingly-benign UB [7]. While the notion of protecting one's code from the compiler can be unsettling, the safest approach to future-proofing an application is to remove UB, even "benign" data races.

2.3 The CUDA Programming Model

Programming in CUDA is challenging. CUDA programs run a kernel function in parallel— in single-instruction, multiple-threads (SIMT) fashion— across a massive number of threads organized into *grids* of *thread blocks*, with each block containing *warps* of 32 threads. Threads within a warp execute in lockstep, but this synchronized execution splits when a divergent branch causes different threads to follow different execution paths. When this happens, the warp divides into two or more *subwarps* of threads, executing their code paths serially. Recent versions of CUDA offer a formal abstraction, called cooperative groups, for grouping threads to augment traditional thread-block synchronization with more granular control. Furthermore, multiple kernels may run on the same GPU or across multiple GPUs in parallel streams, which may require periodic synchronization to enforce data dependencies. While the GPU scheduler dispatches thread blocks to physical Streaming

Multiprocessors (SMs) in *waves* on the GPU, a warp scheduler selects the next ready warp to execute to hide pipeline stalls.

Each GPU thread can access different memory spaces that mirror the underlying architecture.

- Shared Memory: This is a small, low-latency memory on the SM that is visible to all threads within the same thread block. With the introduction of distributed shared memory, a cluster of thread blocks can pool their shared memory in a coherent address space [26]. Intra-block data races on shared memory are common.
- Global Memory: This is the large, high-latency device memory visible to all threads and blocks on the GPU. Accesses to global memory can lead to inter-block data races, which typically require global synchronization primitives or atomic operations.
- Unified Memory (UM) and Heterogeneous Memory Management (HMM) [9] Support: These features allow the CPU and GPU to share a single virtual address space. This sharing enables races not only among GPU threads, but also between CPU and GPU threads accessing the same managed data, further complicating the happens-before relationship across the host and device.
- Constant and local memory: These memories are for holding arrays of constants, and for thread-local storage respectively. Data races are extremely unlikely to occur in these memory spaces, so most tools do not check them.

CUDA provides explicit mechanisms for ordering and visibility, which are critical for establishing happens-before relationships. There are mechanisms for synchronizing threads within a warp (`__warpsync`), within a thread block (`__syncthreads`) or thread block cluster (`cluster.sync`), and between kernels (`cudaStreamSynchronize` and `cudaDeviceSynchronize`). There are also memory fences (`__threadfence`, `__threadfence_block`, `__threadfence_system`) designed to enforce memory ordering and visibility across the different memory spaces, and at multiple scopes.

NVIDIA's Ampere architecture added support for asynchronous data copies from global memory to shared memory. Further, the Hopper architecture introduced the Tensor Memory Accelerator (TMA) unit to enable efficient asynchronous transfer of bulk data between global memory and shared memory in a bidirectional manner [35]. These instructions require additional barriers to synchronize with the calling thread. Finally, `cudaMemCpyAsync` is a non-blocking CUDA function that initiates parallel memory transfers (host ↔ device or device ↔ device). These transfers execute concurrently with kernel execution and require explicit synchronization.

The point of this section is to convey the complexities CUDA programmers grapple with: coordinating tens of thousands of threads that must synchronize in different memories and at different scopes. We have glossed over the nuances of the CUDA programming model, but as our results later show, professional CUDA developers struggle with the details. In fact, we found a race condition in the CUDA Programming Guide.

2.4 Weak vs. Strong Memory in the CUDA Programming Model

Fortunately, SuperCollider is agnostic to the peculiarities of synchronization and scope. Instead, it focuses on weak memory operations. The PTX ISA documentation defines a data race as follows: "Two conflicting memory operations are said to be in a data-race if they are not related in causality order and they are not *morally strong*. [31]" (See Lustig et al. for a formal definition of the memory model [22].) As a necessary (but insufficient) condition for two operations to be morally strong with respect to each other, both operations must be *strong*. The PTX documentation lists strong memory operations as those with one of the following opcode qualifiers: "`.relaxed`", "`.acquire`", "`.release`", "`.acq_rel`", "`.volatile`", or "`.mmio`". Strong memory operations inform the compiler and architecture that the specified address may hold a value that changes sporadically. Programmers

use them for tasks such as synchronization and reading Memory-Mapped Input/Output (MMIO) control registers.

By contrast, *weak* memory operations use the `.weak` qualifier, indicating "a memory instruction with no synchronization. The effects of [weak instructions] become visible to other threads only when synchronization is established by other means." [31] The implication is that, in the absence of synchronization, locations touched by weak memory operations cannot change sporadically. This nicely mirrors point (3) in Erickson and Musuvathi's definition of a data race and establishes SuperCollider's correctness. If the value at an address touched by a weak memory operation changes before the next synchronization operation, there is a *definite* data race.

Weak operations are prevalent in most CUDA programs and provide the compiler with additional scheduling opportunities.

2.5 DataCollider as Inspiration

Erickson and Musuvathi introduced a clever approach to race detection, called DataCollider [5]. Unlike prior race detectors, DataCollider does not rely on tracking state across multiple threads. Instead, DataCollider randomly samples memory references and leverages ubiquitous CPU hardware support for code and data breakpoints to detect, at runtime, whether two concurrent memory instructions conflict.

While DataCollider's preferred solution uses data breakpoints to discover races, at the OS kernel level, breakpoints alone are insufficient. Breakpoints operate on virtual addresses, and the authors observed that they miss cases where hardware devices directly access memory and where different virtual addresses map to the same physical address. Therefore, DataCollider also issues redundant memory operations as a backstop for the cases in which breakpoints do not work. For every memory operation, the algorithm issues a redundant read to the same address, separated by a random delay, and checks that the value has not changed. The authors note that the redundant-read approach was their first strategy, and that it was highly effective. Our solution, which we introduce next, uses the redundant-read approach as its foundation.

3 Our Approach: SuperCollider

Our goal is to build a tool for detecting *device-side* data races, leaving CPU-side race detection to other tools. While DataCollider's relative statelessness makes it a good candidate for GPU race detection, it does not apply wholesale to CUDA. For one thing, GPUs do not provide hardware data watchpoints. One could conceivably emulate data watchpoints in software, but the overhead of searching through a mutable data structure for every memory operation would be prohibitive. More importantly, a sample-based approach would not work well for GPU kernels. DataCollider randomly selects memory operations to sample and sets a control breakpoint. If the breakpoint triggers, the DataCollider algorithm sets a data watchpoint for the address of the memory operation. Even if GPUs provided hardware data-watchpoint support, sampling remains questionable. In the *typical case*, tens of thousands of threads will trigger the control breakpoint, and the SIMT memory operation will reference thousands of *unique* addresses. It is infeasible to watch all of them efficiently, and picking one at random would be unlikely to catch a race.

For these reasons, we choose the redundant-read approach as the basis for SuperCollider and extend it to handle lost updates, leveraging the peculiarities of SIMT and asynchronous memory operations. Unlike DataCollider, SuperCollider is not sample-based: by default, SuperCollider checks *every weak* memory operation (Section 2.4 defines weak). Furthermore, we apply GPU-specific techniques to alter thread schedules to expose data races.

```
write.global [%addr] = vr0            // Original store
nanosleep.rand #wdelay                // Shuffle warps
read.global.sys.strong vr1 = [%addr]  // Redundant read
if vr0 != vr1 {
   // Race detected! Report error.
   call sc_err(pRuntimeStr, PC, %addr, LOST_UPDATE)
}
```

(a) Universal lost update instrumentation.

```
match vr0 = %addr   // Mask of threads with same %addr
popcount vr1 = vr0  // How many match?
if vr1 > 1 {
   // More than one thread writing to same address!
   call sc_err(pRuntimeStr, PC, %addr, LOST_UPDATE)
}
write.global [%addr] = vr2   // Original store
```

(b) Intra-warp lost update instrumentation.

Fig. 3. Lost update instrumentation.

3.1 Read Instrumentation: Clobbered Read

Figure 1 shows the compiler-level instrumentation for all *weak* read operations to the *global*, *shared*, or *generic* memory spaces. The instrumentation preserves the original operation and introduces a *strong*, system-scoped duplicate load, which not only prevents the compiler from reordering the duplicate but also bypasses the cache hierarchy. The instrumentation inserts a `nanosleep.rand` instruction between the original load and the duplicate load, which sleeps for a random amount of time, capped by a customizable *maximum* time, and alters both thread and warp schedules [32]. Intuitively, the longer the sleep window, the more likely the technique is to expose latent races. Section 5 experiments with different maximum sleep times. Finally, the instrumentation compares the values returned by the two loads and calls an error-reporting routine on a mismatch.

The compiler uses *generic* addresses when it cannot disambiguate the memory space to which a pointer maps. Therefore, reads with a generic address can target *local*, *shared*, or *global* memory [30]. Because *local* memory is thread-private, data races in the *local* space are not possible, and our instrumentation is unnecessary, but harmless.

3.2 Write Instrumentation: Lost Updates

Figure 3a shows the instrumentation that the compiler inserts for all *weak* write operations to the *global*, *shared*, or *generic* memory spaces. It closely mirrors the instrumentation for reads, with the exception that the approach now checks whether another thread overwrites this thread's update. We considered inserting an additional write with a garbage value before the update to uncover cases like those in Figure 2, where multiple racy threads write the same value to a shared location. That typical use case motivates the specialized check we describe next.

3.3 Write Instrumentation: Intra-warp Lost Updates

GPUs rely on SIMT execution to efficiently amortize the work of an instruction over a *warp* of threads. Figure 3b illustrates how SuperColider leverages SIMT behavior to ensure that multiple threads in a warp do not write to the same address. The check uses the `match` warp-collective operation to compare addresses across active threads [33], and records an error if more than one thread writes to the same address. The check catches the race in the initialization pattern in Figure 2. While we urge programmers to fix such races, we provide an optional flag that allows developers to suppress race reports unless a warp writes *distinct* values to the same address.

Note that we must filter out generic local addresses because they cannot race, as the following program snippet demonstrates:

```
int a;                              // Stack allocated variable.
int *p = __cvta_local_to_generic(&a); // Generic pointer to local.
*p = threadIdx.x;                   // Not racy.
```

Here, all threads write to the *same* generic local address, p, which the hardware ensures maps to a unique, thread-private location on each thread's stack. We filter out local addresses to avoid reporting false positives.

3.4 Handling Asynchronous Copies

NVIDIA Hopper GPU Architecture introduced the Tensor Memory Accelerator, an asynchronous copy engine that allows for high-speed, bulk data movement between the GPU's off-chip global memory and on-chip shared memory without using any registers [35]. This feature is exposed in CUDA with `memcpy_async` and in PTX with the `cp.async` and `cp.async.bulk` TMA instructions. Because these instructions are asynchronous with respect to the issuing thread, the executing thread cannot access the shared memory destination without intervening synchronization. Hence, unlike regular weak memory accesses that maintain per-thread sequential consistency, asynchronous copies can create intra-thread races in addition to inter-thread races. Furthermore, because these instructions touch both global and shared memory spaces, races may exist at either the source or target address and our instrumentation must insert redundant reads for both the global memory and shared memory addresses.

Figure 4a shows the original uninstrumented code with an intra-thread data race. The `cp.async` PTX instruction asynchronously copies 4, 8, or 16 bytes of data from a source address in global memory to a destination address in shared memory [31]. The following shared memory read executes without intervening synchronization and may read stale data.

If our instrumentation simply inserted memory reads to both the source and destination addresses after a delay, they could both read stale data and result in false positives. On the other hand, if we first inserted barrier synchronization to ensure the asynchronous copy was completed before inserting the memory reads, we would identify races across threads but we might mask intra-thread missing synchronization races.

Instead, we handle the inter-thread and intra-thread races separately in our instrumentation logic. Figure 4b shows the SuperCollider instrumentation for `cp.async`. We first emulate the `cp.async` operation by issuing a regular (synchronous) weak load from the global memory source, followed by a weak store to the shared memory destination. Next, we insert a delay to allow other warps to issue potentially conflicting memory accesses, followed by strong loads from both the source and destination addresses. Because the PTX memory model guarantees per-location sequential consistency for strong operations at a given scope, these strong loads are guaranteed to read the most recently stored value at each address. We then compare the retrieved data to the initial weak load to ensure that no other thread clobbered either the global address or the shared address during the delay. This helps us identify the inter-thread data races.

To identify the intra-thread missing synchronization case, we dirty the destination shared-memory location and then issue the original `cp.async`. Our read instrumentation (Section 3.1) identifies intra-thread races caused by missing synchronization. When the thread loads the shared memory written by `cp.async`, our instrumentation checks for erratic values. Note that if we do not dirty the shared-memory location, there will be no value mismatch, and the race will go undetected. Hence, we can identify both inter-thread and intra-thread races for cp.async instructions. Next, we describe our instrumentation for the bulk asynchronous TMA instruction, cp.async.bulk.

Unlike `cp.async`, `cp.async.bulk` can bidirectionally transfer large blocks of data between global and shared memory. Figure 4c shows an uninstrumented kernel with a bulk TMA copy instruction. Bulk copies face the same risks of potential data races at both global and shared memory addresses, and the same intra-thread races due to missing synchronization. Hence, as before, we employ emulation-based instrumentation for bulk copies. Additionally, because data movement is bulk, the original and delayed copies cannot both reside in registers or shared memory for

```
/* Each thread uses cp.async to copy 4 bytes from
     global to shared memory */
cp.async [%shared_addr] = [%global_addr], 4

/* Missing synchronization: cp.async may not have
     completed before the read */
read.shared vr3 = [%shared_addr]; // Race!
```

(a) Intra-thread race with async. copies.

```
+ // Emulate cp.async
+ read.global vr0 = [%global_addr]
+ write.shared [%shared_addr] = vr0

+ // Enable warp shuffling
+ nanosleep.rand #rdelay

+ // Redundant read and compare
+ read.global.sys.strong vr1 = [%global_addr]
+ read.shared vr2 = [%shared_addr]
+ if vr1 != vr0 || vr2 != vr0 {
+   // Error reporting
+   call sc_err(pRuntimeStr, PC, ...)
+ }
+ // Clobber destination
+ write.shared [%shared_addr] = 0xbadbeef

  // Original cp.async instruction
  cp.async [%shared_addr] = [%global_addr], 4

  // Original shared memory read instruction
  read.shared vr3 = [%shared_addr];

+ // Enable warp shuffling
+ nanosleep.rand #rdelay

+ // Redundant read and compare
+ read.shared vr4 = [%shared_addr]
+ if vr3 != vr4 {
+   // Error reporting
+   call sc_err(pRuntimeStr, PC, ...)
+ }
```

(b) Instrumenting cp.async.

```
if tid == 0 {
  // cp.async.bulk issued by one thread per block
  cp.async.bulk [%global_addr] = [%shared_addr], 256
}
```

(c) cp.async.bulk.

```
if tid == 0 {
+  // Duplicate bulk copy
+  cp.async.bulk [%global_addr] = [%shared_addr], 256
+  // Compute hash
+  hash0 = FNV_offset_basis
+  for (i = 0 to 255) {
+    read.global.sys.strong vr0 = [%global_addr + i]
+    mul hash0 = hash0, FNV_prime
+    xor hash0 = hash0, vr0
+  }
+  // Shuffle warps
+  nanosleep.rand #rdelay
+  // Duplicate bulk copy
+  cp.async.bulk [%global_addr] = [%shared_addr], 256
+  // Compute hash again
+  hash1 = FNV_offset_basis
+  for (i = 0 to 255) {
+    read.global.sys.strong vr1 = [%global_addr + i]
+    mul hash1 = hash1, FNV_prime
+    xor hash1 = hash1, vr1
+    // Clobber destination
+    write.global.sys.strong [%global_addr + i] = ...
+  }

+  // Compare hashes, report error on mismatch
+  if hash0 != hash1 {
+    call sc_err(pRuntimeStr, PC, ...)
+  }

  // Original cp.async.bulk
  cp.async.bulk [%global_addr] = [%shared_addr], 256
}
```

(d) Instrumenting cp.async.bulk.

Fig. 4. Handling asynchronous copies. Lines marked with + denote SuperCollider instrumentation.

direct value-by-value comparison. Instead, we use a checksum-based solution that walks over all destination data values to compute a hash, then compares the hash to detect potential data races. Figure 4d shows the SuperCollider instrumentation for bulk copies.

We use the FNV-1a algorithm to compute the hash because it is simple, inexpensive, and has a good distribution, which helps reduce collisions and false negatives. The reason we only hash the destination values is partially performance-driven. However, we surmise that if there is a race on the source values, the hash of the destination values will change as well.

Our implementation handles bulk copies only from shared to global memory. The global-to-shared copy case is more challenging because it requires a shared-memory barrier. Reusing an existing barrier could mask races, and allocating a new barrier requires additional shared memory. A potential future solution is to either emulate the bulk copy instruction (which might mask other intra-thread races) or to reserve a small amount of shared memory for barrier allocation.

3.5 Rejiggering Thread Scheduling

Our approach is blind to synchronization protocols. Unlike state-tracking race detectors, if a race condition does not physically manifest at runtime, SuperCollider will not catch it. Therefore, SuperCollider uses techniques to randomly perturb subwarp, warp, and block schedules to expose latent races at runtime.

3.5.1 Warp and Subwarp Shuffling. The `nanosleep` in Figures 1 and 3 performs two tasks: (1) putting the active threads to sleep for a random amount of time, capped by a user-defined maximum, and (2) switching the GPU's scheduler to another ready subwarp or warp. The Volta architecture introduced the threads-are-threads programming model, used by subsequent NVIDIA architectures [34]. In this model, when a convergent group of threads in a warp sleeps, the scheduler can consider scheduling other sets of "ready" threads, either from the same or another warp, which can interleave the execution of potentially conflicting subwarps or warps, thereby exposing races that might not otherwise manifest. Section 5 studies how the sleep duration affects race detection.

3.5.2 Block Shuffling. Consider the following pedagogical example with a race between the first and last element of the `C` array:

```
__global__ void racy_add(const float *A, const float *B, float *C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    // Set a race between the first and last element of the array.
    int ix = i == (N-1) ? 0 : i;
    if (i < N) {
        C[ix] = A[i] + B[i] + 0.0f;
    }
}
```

At one end of the spectrum, consider a GPU with a single SM that must execute thread blocks serially. SuperCollider would never expose the race for sufficiently large N, because the race occurs between threads of two different blocks. Even on NVIDIA's most powerful GPUs, SuperCollider will not catch the race if the first and last blocks are in different *waves*. Although in practice, the thread block scheduler appears to operate predictably [8], the CUDA C++ Programming Guide states: "thread blocks are required to execute independently. It must be possible to execute blocks in any order, in parallel, or in series" [24]. Put another way, a well-formed CUDA program should produce the same results under any permutation of block schedules. If there are data races that depend on block ordering, deadlock is possible.

SuperCollider injects a kernel header that allows users to perturb block schedules, which we call *block shuffling*. The approach pseudo-randomly alters the architecture's thread block schedule by rewriting `blockIdx.x` with an *affine cipher* of the block's *original* `blockIdx.x`:

$$\texttt{blockIdx.x}_{new} = (a \cdot \texttt{blockIdx.x} + b) \bmod N_x, \tag{1}$$

where N_x is the grid size in the x dimension, a and N_x are coprime, and all blocks in the grid agree on a and b. The coprimality of a and N_x ensures that all block IDs are permuted to different values covering the same set, creating a bijection that pseudo-randomly reorders block scheduling. Block shuffling effectively changes the composition of waves, and enables SuperCollider to expose races between distant thread blocks.

We implemented block shuffling as a compiler transformation, and therefore applying the technique does not require source code modifications. However, it requires all code referencing `blockIdx.x` to be recompiled with SuperCollider for correctness. Depending on the race's nature, exposing a cross-CTA race may require many runs.

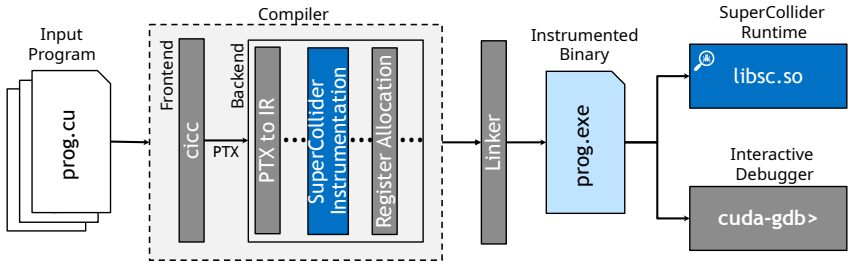


Fig. 5. SuperCollider workflow. Compiler-instrumentation yields a binary that a user can either run under the SuperCollider runtime or CUDA-GDB.

3.6 Summary

SuperCollider is massively scalable. In the absence of races, SuperCollider is totally stateless, except for the short-lived, thread-local variables used to store duplicate-operation values. SuperCollider can seamlessly detect races within a thread, between threads within a warp (including across divergent threads), within a block, and within a kernel. In principle, because SuperCollider detects overwrites of memory locations, it can detect races on weak memory between two or more threads *anywhere* in the system,[2] and in any shared address space. Our solution can detect races between kernels, between CUDA API calls (*e.g.*, between `cudaMemCpyAsync` calls and kernels), between GPUs, and between the CPU and the GPU. Appendix A in the supplemental material formalizes the execution and memory models and proves soundness and semantics preservation for our detectors.

4 Implementation and Usage

Figure 5 shows SuperCollider's tooling flow. There are two primary flows: one uses a custom runtime, and the other uses CUDA-GDB for interactive debugging. Both flows rely on a modified compiler. This section describes typical usage, the compiler, and the runtime.

4.1 Usage

The first step in both flows is to compile all sources with the SuperCollider instrumentation pass, which we discuss later in this section. To enable the instrumentation pass, a user passes a new command-line option to our compiler, `-sanitize=supercollider`.

A developer must then run the instrumented application through the SuperCollider runtime to check for data races. Our prototype uses the `CUDA_INJECTION64_PATH` environment variable to force the CUDA runtime to load the SuperCollider runtime, which is a shared object library [28].

4.2 Error Reporting

Figure 6 shows an abridged error report from our prototype on the `simpleAttributes` example from the CUDA Samples. This report contains clobbered reads, lost updates, and intra-warp lost updates. Our prototype aggregates races by program counter (PC), and uses DWARF debugging information to display the source-code locations of the PCs at which races occurred. One shortcoming of SuperCollider is that, except for intra-warp lost updates, a developer only gets information about a single thread involved in a race. For example, in the first race shown in the figure, the report identifies the reading thread, but provides no information about the writing thread that clobbered the data. We later demonstrate that, while imperfect, the error reports are beneficial.

[2]Since our compiler only instruments device code, race detection involving CPU or co-processor threads is limited to cases where those threads perform writes. DataCollider can help discover races where device threads write and CPU threads read.

```
./simpleAttributes Starting...
    +======================== SUPERCOLLIDER ENGAGED! ========================+
    Note         : This tool disables lazy loading
    MAX_ERRORS   : 64
    BLOCK_SHUFFLE: 1
    +============================== WARNING ===============================+
                              9 RACE CONDITIONS DETECTED!
    +======================================================================+

! In function kernCacheSegmentTest at simpleAttributes.cu:74
Type         : This thread READ from an address that another thread clobbered
PC offset    : 0x1110
PC           : 0x7bd5d1285110
First noticed: Thread (1,24,0), Block (82,0,0), Address 0x0
Occurrences  : 3411421133 times (possibly by other threads at other addresses)

! In function kernCacheSegmentTest at simpleAttributes.cu:85
Type         : This thread WROTE a value that another thread clobbered
PC offset    : 0xff0
PC           : 0x7bd5d1284ff0
First noticed: Thread (16,5,0), Block (74,0,0), Address 0x7bd5949ea0f0
Occurrences  : 14673949 times (possibly by other threads at other addresses)

! In function kernCacheSegmentTest at simpleAttributes.cu:85
Type         : This warp WROTE more than one value to the same address
Lanes in warp: { 12 122 }
PC offset    : 0xee0
PC           : 0x7bd5d1284ee0
First noticed: Thread (2,27,0), Block (55,0,0), Address 0x7bd5957e29bc
Occurrences  : 9700 times (possibly by other threads at other addresses)
```

Fig. 6. An abridged error report that shows three of the nine detected race conditions in `simpleAttributes`. Note that the shared memory address, `0x0`, is a valid address.

```
// Error reporting routine.
__device__ void sc_err(
   volatile void *pRuntimeStr,  // Runtime structure
   uint64_t pc,                 // PC
   uint64_t address,            // Racy data address
   RaceType rtype               // Race type
)
{
    if (ptrRuntimeStr == nullptr) {
        // If our runtime isn't running, just issue a
        // breakpoint, which will trap here when
        // run natively or in the debugger.
        __brkpt();
        return;
    }
    // Record error with key=pc in dictionary...
    ...
}
```

Fig. 7. CUDA error reporting callback. Our instrumentation adheres to the CUDA ABI, allowing tools writers to customize reporting behavior using a high-level language.

4.3 Compiler

We implemented SuperCollider as a pass within NVIDIA's production `ptxas` back-end compiler that consumes PTX [31] and emits NVIDIA assembly (SASS). The compiler is based on CUDA 13, and allows us to sanitize massive codebases. As Figure 5 shows, our compiler pass runs early in the back-end compiler's flow and performs the code expansions described in Section 3. Because our pass runs before register allocation, the instrumentation code efficiently uses registers to store the values fetched from duplicate loads.

Inserting the duplicate load, the sleep value, and the comparison is straightforward and involves adding appropriate intermediate representation (IR) instructions. However, the error-recording code that runs when the comparison fails is relatively complicated. For the SuperCollider prototype, we implement the error-recording code as a CUDA callback. For example, Figure 3 shows calls to `supercollider_report`, which target a callback like that in Figure 7. While this costs the tool some performance, especially in the uncommon case where a race is detected, the approach affords the tool's maintainer the flexibility to write the error-reporting logic in a high-level language rather than tediously crafting instrumentation in compiler IR instructions. Figure 7 shows the arguments that our prototype passes to the callback: a pointer to the runtime's management data structure, the program counter of the racy memory operation (for source line correlation), the data address at which the race occurred, and the type of race (*i.e.*, clobbered read, lost update, TMA).

Our error-reporting callback handler uses an unsorted list to maintain the set of PCs where at least one race occurred. It uses this list to aggregate other races at the same PC. For each PC, SuperCollider records the thread and block IDs, and the address of the initial detected race.

Table 1. Races detected for two delay configurations.

	Fastest #rdelay=1ns, #wdelay=1ns									Balanced #rdelay=5000ns, #wdelay=1ns								
Application	1/N	2/N	3/N	4/N	5/N	Total	CR	LU	ILU	1/N	2/N	3/N	4/N	5/N	Total	CR	LU	ILU
Gunrock	0	0	1	0	15	16	93%	0%	7%	0	0	0	5	11	16	93%	0%	7%
CUTLASS	0	0	0	0	1	1	100%	0%	0%	0	0	0	0	1	1	100%	0%	0%
CUDA-samples	0	0	0	0	14	14	52%	10%	38%	0	4	0	0	10	14	46%	11%	43%
HeCBench	0	0	0	0	35	35	51%	8%	41%	3	1	1	0	43	48	58%	6%	36%

4.4 Runtime

The runtime's functionality is straightforward and relies on a library such as CUPTI [28] to interpose between the application and the CUDA driver and runtime APIs. Its primary responsibility is to report any errors to users coherently. Our prototype's runtime performs the following tasks:

- At CUDA context creation time, it allocates a small, fixed-length, context-specific buffer that the instrumentation passes as a parameter to the error-reporting callback, *i.e.*, the `pRuntimeStr` parameter in Figure 7.
- At CUDA module load time, it extracts per-PC source-code correlation information from the module's DWARF.
- When the application exits, it reports any errors in the contexts' buffers, using source-code correlation to report line information when available.
- It initializes a and b in Equation 1 to perturb block schedules.

4.5 CUDA-GDB Mode

Instead of using the SuperCollider runtime, one can run an instrumented application directly in CUDA-GDB. The SuperCollider runtime allocates a management structure and, indirectly, passes a pointer to it to the callback in Figure 7. However, if the user runs the application without the runtime, the pointer will be NULL, and the callback code will trap. Within CUDA-GDB, the trap halts execution and allows a developer to debug interactively.

5 Evaluation

This section evaluates SuperCollider's ability to uncover races, its overhead, and compares it with state-of-the-art GPU race-detection tools.

5.1 Experimental Setup

5.1.1 System. We evaluate SuperCollider using a CUDA 13 toolkit, replacing the `ptxas` compiler with our SuperCollider-enabled PTX compiler. All SuperCollider experiments run with the SuperCollider runtime as an injected shared library. Our evaluation uses an NVIDIA RTX Ada 6000 GPU (48 GB memory) combined with a 32-core Intel i9-13900K (128 GB memory).

5.1.2 Applications. We use applications from several suites as part of our evaluation: HeCBench [12], CUDA-Samples [29], Gunrock [48], Indigo [21], and CUTLASS 4.3.0 [27].

5.1.3 SuperCollider Configuration. Throughout this section we adjust two basic knobs: #*rdelay*, and #*wdelay*. The #*rdelay* knob controls the `nanosleep` duration for *clobbered read* instrumentation (Section 3.1), and `#wdelay` controls the sleep delay for *lost update* instrumentation (Section 3.2). Intuitively, these knobs affect both the performance overhead of SuperCollider, as well as the window of time in which a shared location can be overwritten before the duplicate read executes.

Table 2. Quantitative comparison of race detection techniques using the Indigo micro-benchmark. ✓ means that the race is correctly found whereas × means the tool missed that error.

	Ground Truth			HiRace		iGUARD		Compute Sanitizer		SuperCollider						
Input Graph	Total Tests	Expected Races	Race-Free Tests	✓	×	✓	×	✓	×	1/N ✓	2/N ✓	3/N ✓	4/N ✓	5/N ✓	Missed ×	0/N Correct
DAG_100n_200e	446	266	180	266	0	366	8	40	226	4	2	1	8	220	31	180
DAG_200n_400e	446	266	180	266	0	366	8	40	226	0	7	5	8	227	19	180
DAG_5n_5e	446	266	180	154	112	158	154	40	226	4	3	7	3	43	206	180
counterDAG_200n_1000e	446	266	180	266	0	366	8	40	226	3	5	2	11	226	19	180
counterDAG_5n_5e	446	266	180	234	32	192	126	40	226	6	5	6	11	141	97	180
power_law_200n_1000e	446	266	180	266	0	366	8	40	226	4	2	2	7	232	19	180

5.2 Detecting Races

Table 1 summarizes the races SuperCollider discovered across the applications we surveyed. We present results for two configurations. On the left, we set #*rdelay* and #*wdelay* to their minimal settings, yielding the generally lowest-overhead configuration. On the right, we show a balanced setting that, in our experiments, was the fastest configuration capable of uncovering nearly all the races we found at other settings. For each configuration, under the *Total* column, we show the total number of bugs SuperCollider finds for each benchmark suite. The table provides a histogram that shows the likelihood of finding each bug. For example, for Gunrock using the fastest configuration, we find 16 total bugs; we discovered 15 of the bugs all five times we ran the suite, but one bug only turned up three of the times. This statistic highlights the probabilistic nature of SuperCollider. However, it also shows that most bugs do not require much luck to find. We set #*wdelay* to the minimum setting of 1ns because our experiments showed that increasing this delay did not discover any additional races. The interested reviewer can consult the supplemental material for a study on SuperCollider's sensitivity to #*rdelay* and #*wdelay* on the HeCBench benchmarks.

We aggregate reported bugs by source code line number. For instance, CUTLASS tests reported several races that all mapped back to the same line number. For each benchmark suite and both configurations, we list the percentage of races reported in the tool's reports for the three race categories: *clobbered reads* (CR), *lost updates* (LU), and *intra-warp lost updates* (ILU). The data show that the clobbered read instrumentation is the most effective overall, but we were pleasantly surprised by the effectiveness of our intra-warp lost update instrumentation.

Crucially, our approach *cannot* report false positives. We have observed several instances of "benign" races, such as a confirmed bug in CUTLASS that closely resembles a CUDA Programming Guide example [25]. Even if the compiler currently produces expected code, the PTX memory model classifies every race our tool reports as undefined behavior. In the case of CUTLASS and other similar examples, the fix is straightforward and causes no performance degradation. The supplemental material lists the complete set of applications for which SuperCollider reports races.

5.2.1 Indigo. We further evaluate SuperCollider using microbenchmarks from the Indigo suite [21], comparing its error-detection capabilities against state-of-the-art tools. Although Indigo provides scripts to generate diverse bug patterns and configurations, we use the same scripts and microbenchmarks from the HiRace artifact repository [11] to enable a direct comparison with HiRace. Table 2 summarizes results on 6 small input graphs.

Across 446 CUDA kernels—266 of which contain races—SuperCollider detects up to 92% of races, depending on the input graph. Most races are found deterministically in all five runs (the 5/N column), with only a few detected in a subset of runs. Tiny graphs (with fewer nodes and edges), such as DAG_5n_5e and counterDAG_5n_5e, do not trigger all races and thus yield lower detection coverage with SuperCollider. Excluding these tiny graphs, HiRace achieves the highest detection rate (100%), followed by iGUARD (97%). However, iGUARD raises many false alarms, whereas

Table 3. Comparison of race detection tools on cuHadron microbenchmarks. Green ✓ indicates a true positive while a red ✓ indicates a false positive; green × indicates true negative and red × indicates a false negative.

Test Name	Description	iGUARD	Compute Sanitizer	SuperCollider
Inter-subwarp SMEM RW	Divergent threads in warp read/write the same address	×	✓	✓
Inter-subwarp SMEM WW	Divergent threads in warp write to the same address	×	✓	✓
cp.async GMEM RW	One thread reads via cp.async while another writes	×	×	✓
cp.async SMEM RW	Missing synchronization before read	×	✓	✓
cp.async SMEM WW	Multiple threads write to same address	×	✓	✓
cp.async.bulk (shared->global) SMEM RW	Bulk copy reads from smem while other threads write	×	×	✓
cp.async.bulk (shared->global) GMEM RW	Threads read from gmem before cp.async.bulk completes	×	×	✓
cp.async.bulk (shared->global) GMEM WW	Multiple threads write to same address	×	×	✓
cp.async.bulk (global->shared) SMEM RW	Bulk copy writes to smem while threads read from smem	×	✓	✓
cp.async.bulk (global->shared) SMEM WW	Bulk copy writes to smem while threads write to smem	×	✓	✓
cp.async.bulk (global->shared) GMEM RW	Bulk copy reads from gmem while threads write to gmem	×	✓	×
Distributed SMEM RW	CTAs access DSMEM without sync	×	✓	✓
Distributed SMEM WW	CTAs write DSMEM without sync	×	✓	✓
Inter-wave GMEM WW	Blocks in different waves write to same address	✓	×	✓*
Inter-kernel GMEM RW	Consumer kernel reads before producer kernel completes	×	×	✓
Inter-kernel GMEM WW	Kernels on different streams write same addresses	×	×	✓
cudaMemcpyAsync (HtoD) RW	Kernel reads before cudaMemcpyAsync completes	×	×	✓
cudaMemcpyAsync (DtoH) RW	cudaMemcpyAsync reads before kernel completes	×	×	×
Inter-GPU UVM RW	GPU reads before other GPU completes	×	×	✓
Inter-GPU UVM WW	Multiple GPUs write same addresses	×	×	✓
CPU-GPU UVM RW	CPU reads before GPU kernel completes	×	×	×
CPU-GPU UVM RW	GPU reads while CPU updates	×	×	✓
CPU-GPU UVM WW	CPU and GPU write same addresses	×	×	✓
Byte granularity SMEM RW	Different bytes, same word (false positive check)	×	×	×
Byte granularity SMEM WW	Different bytes, same word (false positive check)	×	×	×
Byte granularity GMEM RW	Different bytes, same word (false positive check)	✓*	×	×
Byte granularity GMEM WW	Different bytes, same word (false positive check)	✓*	×	×

HiRace and SuperCollider report none. Compute-Sanitizer has the lowest detection rate (15%) because it only detects shared-memory races.

5.2.2 cuHadron. To highlight the capabilities of SuperCollider, we created cuHadron, a microbenchmark suite for data races in CUDA programs that leverages modern CUDA features and explores lesser-known race scenarios. cuHadron includes 27 carefully crafted test cases covering asynchronous copy operations (`cp.async`, `cp.async.bulk`), inter-kernel races across streams, CPU-GPU and inter-GPU races using Unified Virtual Memory, distributed shared memory in thread block clusters, and inter-subwarp divergent execution. The benchmark spans multiple execution scopes and memory spaces, and includes granularity tests to check for false positives.

Table 3 shows a comparison of SuperCollider, iGUARD and Compute Sanitizer's racecheck on cuHadron. We were unable to run these tests with HiRace because it requires manual annotation. SuperCollider correctly identifies races in all but three tests and has no false positives. For the inter-wave GMEM race, block shuffling forces conflicting thread blocks to execute in parallel, enabling SuperCollider to detect the race. Note that this is a probabilistic approach due to the randomness of block shuffling. The two CPU-GPU tests where SuperCollider fails involve clobbered read races where the CPU performs a read operation. In this case, GPU side instrumentation is insufficient for race detection. Additionally, we do not currently support `cp.async.bulk` from global to shared. We discuss these limitations and how they can be addressed in section 7. While Compute Sanitizer successfully detected races in shared memory and avoided false positives, it was unable to detect races in global memory and UVM, a known limitation of the tool. iGUARD failed to detect races in any of our tests except the inter-warp gmem race. This is expected because our test suite intentionally tests for instructions and execution- and memory-scopes that are not commonly supported by GPU race detectors. Additionally, iGUARD reported two false positive tests where threads write to different bytes of the same word. If we decrease iGUARD's metadata granularity to 1-byte, it does not report these false positives.

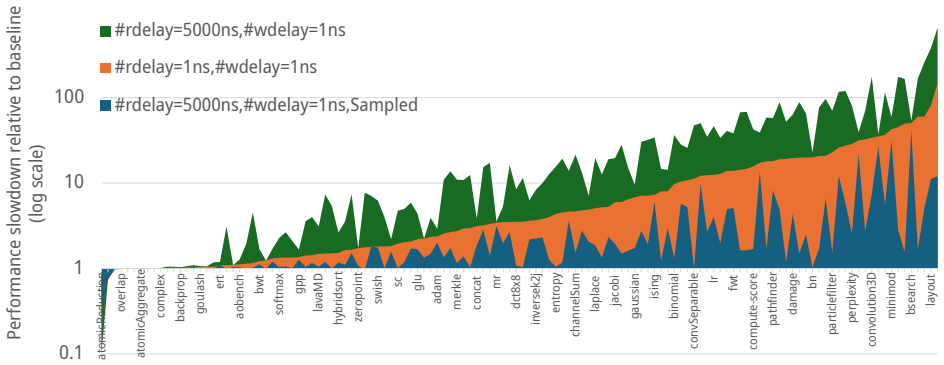


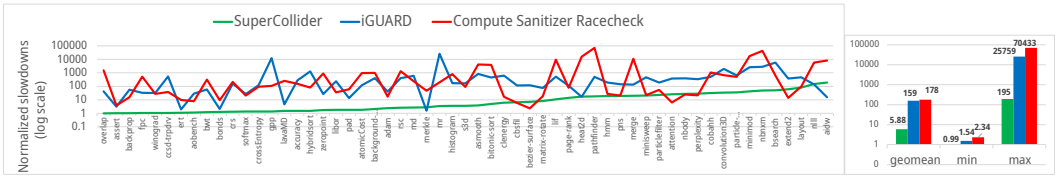
Fig. 8. SuperCollider's performance relative to native execution on 130 HeCBench applications.

5.2.3 Impact of Block Shuffling. Block shuffling did not ultimately increase the number of data races exposed in our application suites. However, it has exposed latent programming errors related to thread block dependencies. For example, `segmentationTreeThrust` uses a `cub` feature that deadlocks unless the zeroth thread block executes in the first wave of thread blocks. Similarly, block shuffling induces hangs in some CUTLASS unit tests. By pseudo-randomly altering the thread block schedule, SuperCollider acts as a stress-tester for CUDA's independent thread block requirement.

5.3 Performance

5.3.1 Performance Sensitivity. We experimented with several combinations of load and store sleep times. Figure 8 shows the performance of three configurations over 130 HeCBench applications that do not trigger races. The slowest configuration, shown in green, uses $\#rdelay = 5000$ ns and $\#wdelay = 1$ ns, like the *balanced* configuration in Table 1. It achieves a median/geomean/max slowdown of 11.94×/11.21×/1066×. Twelve of the apps experience overheads of 100× or greater, but 29 of the applications perform well, with under 2× overheads. The *fast* configuration from the table, shown in orange in the figure, sets $\#rdelay = \#wdelay = 1$ ns. This configuration achieves a median/geomean/max slowdown of 3.57×/4.86×/229×. Only three applications are worse than 100× and approximately 50 applications incur less than 2× overheads. This chart, combined with Table 1 highlight a performance versus coverage tradeoff that SuperCollider users must balance. The final configuration in Figure 8 shows a *sampled* configuration with $\#rdelay = 5000$ ns and $\#wdelay = 1$ ns. Each point in the figure is the geometric mean of 13 runs, where on each run, our compiler only instruments a single memory operation. This configuration achieves a median/geomean/max slowdown of 1.54×/1.98×/41.3×, and explores SuperCollider's potential as an always-on tool. Interestingly, `atomicReduction` exhibits a speedup relative to the baseline. We suspect that the injected `nanosleep` acts as an unintentional back-off mechanism. By pseudo-randomly staggering thread execution, the delays likely reduce structural contention at the GPU's L2 atomic units, allowing the hardware to process heavily serialized updates more efficiently.

5.3.2 HeCBench. Figure 9 compares the performance slowdowns of SuperCollider, iGUARD, and Compute Sanitizer's racecheck on 59 HeCBench applications. The remaining HeCBench workloads timed out with racecheck (67 workloads) and/or iGUARD (39 workloads). We exclude HiRace from this comparison because its publicly-available toolchain requires manual source-code annotations. SuperCollider achieves the best performance, with a geometric mean slowdown of 5.88×, whereas



(a) (b)

Fig. 9. Performance slowdown for different tools relative to native execution on 59 HeCBench applications.

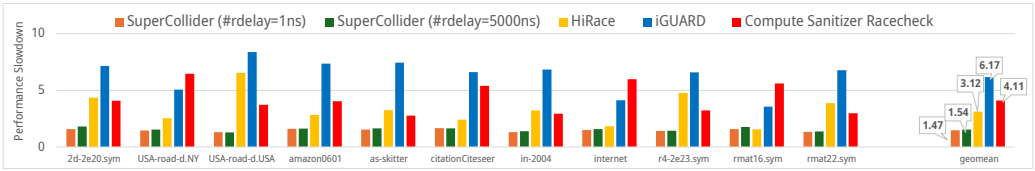


Fig. 10. Performance slowdown for different tools relative to native execution on Indigo kernels using 11 large input graphs with vertices ranging from 100K to 8 million.

iGUARD and Compute Sanitizer's racecheck incur geometric mean slowdowns of 159× and 178×, respectively. SuperCollider's worst-case slowdown (195×) is more than two orders of magnitude lower than iGUARD (25, 759×) and racecheck (70, 433×).

5.3.3 Indigo. Using six small input graphs, SuperCollider and HiRace have geometric mean slowdowns of 1.03× and 1.01×, respectively, whereas iGUARD and Compute Sanitizer incur 1.92× and 2.07× slowdowns, respectively. The performance gap widens on larger inputs. Figure 10 shows the results on 11 large graphs with vertices and edges ranging from 100K to 8 million [45]. SuperCollider incurs a geometric mean slowdown of 1.47× (worst case 409×) compared to HiRace's 3.12× (worst case 1,842×), iGUARD's 6.17× (worst case 7,021×), and Compute Sanitizer's 4.11× (worst case 1,922×). iGUARD timed out on 3% of runs despite a 4-hour timeout limit.

5.3.4 Memory Footprint. A primary factor limiting the scalability of state-of-the-art dynamic race detectors is memory overhead. State-tracking tools maintain metadata. For example, iGUARD defaults to 32-bit, word-level tracking, resulting in a 4:1 metadata-to-application-memory bloat. Although iGUARD lazily allocates metadata only for the memory locations the application actively touches, footprint expansion quickly becomes a bottleneck for data-intensive workloads. Figure 11 illustrates how prior tools force a severe compromise between absolute memory overhead and runtime performance across seven Gunrock workloads using large input graphs. Running the `bc`, `color`, `mst`, `pr`, `spgemm`, and `sssp` workloads under iGUARD inflated their native memory requirements (449 MiB to 3.2 GiB) by 1.83× to 4.63×. Crucially, on the `ppr` workload, which natively requires 15.3 GiB, iGUARD exhausted available device memory and crashed.

In contrast, SuperCollider is stateless. Its memory footprint is bounded by the fixed size of its error-reporting buffers and the increased size of the instrumented binary. Across these same workloads, SuperCollider's memory overhead is negligible. In absolute terms, the total additional footprint averaged only 109 MiB and never exceeded 282 MiB. As visualized in the ideal low-overhead zone of Figure 11, SuperCollider's memory efficiency enables it to scale to massive, memory-bound applications that can cause state-tracking tools to fail.

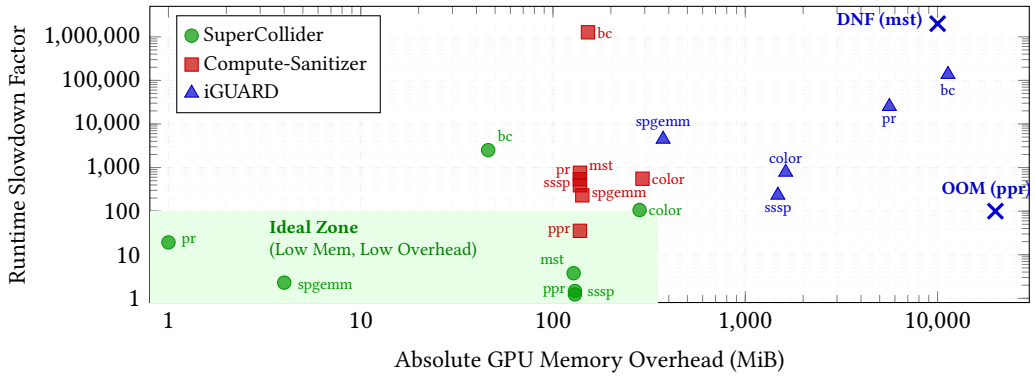


Fig. 11. Performance vs. Memory Overhead on Gunrock. The y-axis shows the runtime slowdown, where native execution is 1. The x-axis shows the additional GPU memory required (MiB). SuperCollider clusters in the low-overhead ideal zone, whereas Compute Sanitizer incurs large runtime penalties, and iGUARD experiences significant memory bloat (leading to OOM/DNF failures on large workloads).

6 Related Work

In this section, we summarize and position our work against the two main families of race detection: static analysis tools and dynamic race detectors.

6.1 Static Race Detectors

Static analysis tools attempt to identify data races by analyzing a program's source code or intermediate representation without executing it. Prominent examples of this approach for GPU programs include GPUVerify [2], GKLEE [19], and FAIAL [3]. The primary promise of static analysis is its theoretical completeness: a perfect analysis could cover all possible execution paths and identify all potential races, including those that might only manifest under rare, hard-to-replicate conditions.

Unfortunately, static analysis for GPU programs remains a profound challenge. The sheer scale of parallelism, combined with the complex interplay among multiple memory spaces (shared, global) and varied synchronization scopes (from warps to blocks and grids), creates a state space that is exceptionally difficult to model accurately. This complexity often leads to a high volume of false positives. Tools may flag code as racy even when the "racy" path is unreachable, or protected by subtle synchronization. False positives can overwhelm developers, rendering the tool impractical for effective debugging. On the contrary, SuperCollider is designed to be practical and generates no false positives.

6.2 Dynamic Race Detectors

Dynamic detectors, including SuperCollider, analyze a program during its execution to find races that actually occur. This category is diverse, ranging from heavyweight state-tracking methods (*e.g.*, happens-before, lockset, and hybrid detectors) to lightweight probabilistic methods (*e.g.*, sampling-based detectors).

6.2.1 Happens-Before Detectors. The happens-before relationship establishes the gold standard for dynamic race detection. Happens-before detectors instrument all memory and synchronization operations to build a graph of causal relationships, allowing them to detect races that did not deterministically manifest in a single run. For example, Google's Thread Sanitizer [41] is a highly effective happens-before detector for CPU code, but its state-tracking design is not tractable at the immense scale of tens of thousands of GPU threads.

Academic tools like CURD [37] and Barracuda [4] apply this model to GPUs, providing excellent race coverage by faithfully tracking memory and synchronization operations. HiRace [10] is a recent happens-before GPU detector that introduces a finite-state machine (FSM) to encode arbitrary-length access histories in a constant-size state. The FSM avoids false negatives that can affect state-tracking tools with finite history lengths. While the FSM compresses state, the approach still introduces significant memory overheads, proportional to the application's data footprint. All tools in this category, including HiRace, incur performance penalties due to their detailed instrumentation, which is a primary bottleneck that SuperCollider eliminates.

6.2.2 Lockset Detectors. Lockset algorithms represent a different approach to dynamic detection. Pioneered by tools like Eraser [39], lockset detectors monitor the set of locks held by each thread as it accesses memory. A potential race is reported if two threads access the same variable without holding a common lock. While effective at finding bugs related to improper locking, traditional lockset tools can be prone to false positives. Moreover, there is no standardized approach in CUDA for acquiring and releasing locks, requiring lockset-based tools to infer these operations based on code patterns, which is an undecidable problem [13].

6.2.3 Hybrid Detectors. To get the benefits of both worlds, some tools combine a happens-before analysis with lockset algorithms. For example, iGUARD [14] uses a lockset algorithm for inferred lock-protected accesses and a happens-before mechanism to detect improperly synchronized accesses to a shared location. While thorough, iGUARD relies on state-tracking and instrumentation that increase the memory footprint by $5 - 17\times$, making it unusable for some memory-bound applications.

A key limitation of all state-tracking tools, including hybrids like iGUARD, is their complexity. Supporting the full, evolving CUDA feature set (*e.g.*, asynchronous memory operations, multi-GPU, CPU/GPU interaction) is challenging. While SuperCollider is not as comprehensive as iGUARD (and thus can miss some of the bugs it finds), SuperCollider's stateless, synchronization-agnostic nature provides major benefits. The stateless nature introduces no memory footprint overhead, allowing SuperCollider to scale to any application, whereas SuperCollider's simplicity seamlessly handles races across multiple kernels, processes, and between the CPU and GPU.

6.2.4 Thread Scheduling for Exposing Races. Randomness is a core tenet of some race detectors. Data Collider randomly samples memory operations to profile, and like SuperCollider, it forces threads to sleep for random durations [5]. Racemob is a clever, crowdsourced race detector that dynamically monitors whether statically-identified races ever occur. To this end, Racemob describes "schedule steering", an approach that tries to enforce specific thread orders at runtime [15]. Similar to block shuffling, the Schedule Amplifier fuzzer tests OpenCL applications on a fixed set of randomized block orders [36]. Additionally, prior work employed thread randomization to reveal weak memory bugs in GPU applications [1] [43] [6] or errors in the memory model [18]. SuperCollider randomizes schedules at multiple levels to jointly schedule racy threads.

7 Discussion

Table 4 compares SuperCollider to the current state of the art in GPU race checking. Overall, SuperCollider is a strong contender for the best CUDA race detector. It is not a substitute for prior art, however, because it cannot detect races in atomic and strong operations, which are essential for writing race-free programs. However, it shines in areas where other detectors fail. It can check *any* CUDA program for races, regardless of the program's scale, memory usage, or feature set. It is relatively fast, does not report false positives, supports a wide variety of CUDA features, can detect

Table 4. Qualitative comparison of race detection techniques. Circles illustrate feature support: Filled (●) for full support, partially-filled (◒) for partial support, and white (○) for no support. Gray fill indicates that the feature is probabilistic.

Execution Scope	**SuperCollider**	**HiRace**	**iGUARD**	**Compute Sanitizer**
Between instructions within a single thread	●	○	○	●
Between convergent threads within a warp	●	●	●	●
Between divergent threads within a warp	●	●	●	●
Between different warps within a block	●	●	●	●
Between different blocks within a kernel	●	●	●	●
Between kernels or API calls on different streams	●	○	○	○
Between threads on different GPUs	●	○	○	○
Between CPU/co-processor and GPU threads	◒	○	○	○
Memory Space				
Shared Memory	●	●	○	●
Distributed Shared Memory (DSMEM, SM 9.0+)	●	●	○	●
Global Memory	●	●	●	○
Unified/HMM/Managed Memory	●	●	●	○
Operations				
32-bit or greater loads and stores	●	●	●	●
8- and 16-bit loads and stores	●	◒	◒	●
Atomic and strong operations	○	●	●	●
`cp.async[.bulk]`	◒	○	○	◒
`cudaMemcpyAsync`, `cudaMemsetAsync`	◒	○	○	○
Usage				
Attribution	One-Sided	One-Sided+	Two-Sided	Multi-Sided
Instrumentation (compiler vs. binary instrumentation)	Compiler	Compiler	Binary	Binary

races at any scope, and can check applications interactively within CUDA-GDB. The rest of this section discusses limitations and potential extensions to SuperCollider.

7.1 The Element of Chance

While detecting races in happens-before detectors relies somewhat on fortuitous thread interleaving [15], Table 4 shows, with all the gray icons, that SuperCollider relies exclusively on forcing races to happen at runtime. Experimentally, our approach is extremely repeatable across the set of races SuperCollider *has detected*. Yet there could be races lurking in the applications we have tested that require a great deal of luck, time, and patience to uncover. Our prototype has plucked the low-hanging fruit of the approach with well-supported randomization. Still, we have considered extending it by shuffling the execution order of kernel calls across independent streams and by randomly selecting any legal (*i.e.*, not heuristically chosen) instruction schedule during compilation.

7.2 Value-Based False Negatives

SuperCollider uses a value-based strategy, and the lost-update instrumentation will miss data races in which multiple threads simultaneously write the same value. As Section 3 discussed, to reduce false negatives, we considered instrumentation that writes a garbage value to a location before performing the actual write. SuperCollider may also miss race conditions where threads write *different* values. Consider a memory location that experiences the following sequence of writes: a, b, a, a, a. The redundant read approach could miss the interloping b in the series of writes of a. DataCollider solves this problem with hardware watchpoints, which are unsupported on GPUs. Even with hardware watchpoint support, sampling addresses is untenable for GPU applications, because each instruction is likely to encounter many thousands of unique addresses. We cannot efficiently watch all of them, and sampling at that scale is not appropriate for interactive debugging.

7.3 Crowd Sourcing and Nightly Testing

However, recent research has demonstrated the effectiveness of hunting for bugs in the wild [15, 20, 42]. In contrast to the other approaches in Table 4, which painstakingly track global state, SuperCollider lends itself well to sampling. We extended SuperCollider with a naive sampling approach in which a JIT compiler selects a single instruction to instrument on each run. The prototype detected several races, though the overheads (2× average) may deter performance-conscious developers from adopting the approach for in-the-wild, always-on race detection. A sample-based approach might be helpful for "nightly" functional testing of software, which is less performance-critical, and runs frequently anyway. In a crowdsourced or nightly testing regime, it might make sense to apply a sampled (software-instrumentation-based) watchpoint approach.

7.4 Compiler versus Binary Instrumentation

We implemented SuperCollider as a compiler pass. An attractive alternative would be to implement the tool atop a binary instrumentation framework like NVBit [46], which like Compute Sanitizer and iGUARD, would allow the tool to handle raw binaries. The downside of a compiler-based debugging tool is that debugging requires compiling the code base, which of course, requires source code access. However, as Jacobson et al. note (and we reinforce in Section 2.2) data races are *undefined behavior*, which means that in the presence of races, the compiler may generate *anything*, including an unrelated race-free program [10]. The earlier in the compilation flow we insert the SuperCollider instrumentation, the better, ideally before any optimizations that take advantage of undefined behavior. Future work may consider running earlier, in the front-end compiler, but a tremendous advantage of running in the back-end compiler is that we can theoretically sanitize any source language that targets PTX, such as Triton, OpenCL, and even graphics shader languages. Once a developer has paid the price of recompiling their codebase, compiler-inserted instrumentation typically runs considerably faster than a binary-instrumentation approach [10, 40].

7.5 Attribution and the Performance-Completeness Tradeoff

You cannot have your cake and eat it too. The "*Attribution*" line in Table 4 compares SuperCollider's error reporting quality against three state-of-the-art techniques. The attribution improves from left to right in the table. For the races they detect, SuperCollider provides the least informative error reports, while Compute Sanitizer provides the best reports. As Figure 6 shows, for each detected race, SuperCollider reports the thread ID and source line information of a single thread involved in the race. We call this one-sided attribution, because we lack information about the other thread(s) involved in the race.[3] On the other hand, our approach is scalable because it minimally tracks execution. HiRace tracks significantly more state than SuperCollider. HiRace cleverly compresses context history with a finite state machine in order to reduce the high cost of metadata tracking, but in so doing, it loses source line information for all but the thread that eventually triggered the race. On the other hand, iGUARD uses twice as much storage as HiRace, and fully records information about the last writer and last accessor of each word of addressable memory. While the extra footprint and machinery to record the history hinders performance, attribution improves as iGUARD reports helpful additional details. Compute Sanitizer is comparatively slow because it analyzes complete traces of memory requests, but as a result, it can report multiple threads involved in a race on a memory location, including source code locations.

Compute Sanitizer's reports are immediately actionable, but it comes with a cost. HiRace and iGUARD are faster and more scalable and can run applications Compute Sanitizer cannot, albeit with degraded attribution. Finally, SuperCollider is the fastest and most scalable approach, but it

[3]The intrawarp lost-update check is complete and shows all lanes involved in a race.

provides minimal information. Developers have already demonstrated the effectiveness of one-sided attribution by verifying several races that SuperCollider reported; and a one-sided error report is better than nothing, which is what developers are currently stuck with for at-scale applications.

We propose a potential solution for two-sided attribution that activates only after a race is detected. The method replays the application with specialized instrumentation that first saves the racing thread's address at the offending PC. Subsequent checks then identify other threads that modified the same memory location between the original access and the redundant read.

7.6 Detecting CPU-GPU Races

SuperCollider is unable to detect Read-After-Write (RAW) races in applications with CPU-GPU races where the CPU only reads because we do not instrument host-side code, which is compiled using a separate host compiler. Hence, in the case of a RAW race, while the GPU-side thread will not detect any modification to the location, the uninstrumented CPU thread will remain ignorant of the overwrite and SuperCollider will yield a false negative. We envision a combined race detection solution that uses both SuperCollider and DataCollider to instrument device-side and host-side races respectively. This hybrid approach, which we leave to future work, would enable the detection of races between GPU threads, CPU threads, or across the GPU and CPU.

7.7 Maintenance

As CUDA evolves, so too must the tool. While SuperCollider's core logic operates independently of synchronization primitives, tool maintainers must update the compiler pass to support any newly introduced PTX memory access instructions. However, a key advantage of our design is its native support for partial instrumentation, *i.e.*, executing programs that contain uninstrumented operations. As a result, SuperCollider seamlessly handles unsupported PTX instructions, applications linked with pre-compiled binaries, and user-directed random sampling for low-overhead execution. Although incomplete instrumentation naturally introduces the risk of false negatives, the tool's foundational guarantee of zero false positives remains.

7.8 Real World Impact

SuperCollider uncovered several data races in production code, including the CUTLASS library, the official CUDA Programming Guide, the CUDA Samples, HeCBench, and Gunrock. We disclosed these findings to NVIDIA and the respective open-source maintainers. NVIDIA engineers acknowledged the reports as genuine data races. Because these accesses violate the PTX memory model, they constitute undefined behavior, even if they do not cause issues on current hardware. NVIDIA prioritized critical software, correcting the Programming Guide and resolving CUTLASS bugs alongside several races in undisclosed libraries that caused runtime failures.

8 Conclusion

SuperCollider's strength is its simplicity. It leverages the clever observation that, in the absence of synchronization, it is strictly illegal for the target of a weak memory operation to change. SuperCollider takes a different approach than other GPU race detectors; it cannot detect every race they find, but in return, it is uniquely capable of uncovering races that others miss, such as those between kernels, and between a GPU and any memory-writing processor, including other GPUs. More importantly, it scales. Unlike prior art, the approach can handle any CUDA application. It can handle any memory space. It can handle practically any memory footprint size. Ultimately, this unparalleled scope and scalability have enabled SuperCollider to detect several previously-undiscovered data races, including in applications that existing detectors fail to execute to completion.

Data-Availability Statement

The SuperCollider artifact is available on Zenodo [44]. The cuHadron benchmark suite is available at https://github.com/NVlabs/cuHadron.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback, which significantly improved this paper. We are grateful to Jaewook Shin, Lloyd Cunningham, Cody Addison, Ameya Ekbote, and Girish Bharambe for their invaluable assistance in implementing the infrastructure that made this work possible. We extend our gratitude to Dan Lustig, Gonzalo Brito, and David Goldblatt for helping us navigate the PTX memory model and shaping our approach. Finally, we thank Yuzhong Wen for directing us to the DataCollider paper and suggesting the approach for GPU race checking.

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)*. Association for Computing Machinery, New York, NY, USA, 577–591. doi:10.1145/2694344.2694391

[2] 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*. 113–132. doi:10.1145/2384616.2384625

[3] Tiago Cogumbreiro, Julien Lange, Dennis Liew, and Hannah Zicarelli. 2024. Memory Access Protocols: Certified Data-Race Freedom for GPU Kernels. *Formal Methods in System Design* 63, 1 (2024), 134–171. doi:10.1007/s10703-023-00415-0

[4] 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*. 126–140. doi:10.1145/3062341.3062342

[5] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 151–162. https://www.usenix.org/conference/osdi10/effective-data-race-detection-kernel

[6] Cormac Flanagan and Stephen N. Freund. 2010. Adversarial memory for detecting destructive races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 244–254. doi:10.1145/1806596.1806625

[7] GCC. 2007. Bug 30475 - assert(int+100 > int) optimized away. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475. Accessed: 10-30-2025.

[8] Guin Gilman, Samuel S. Ogden, Tian Guo, and Robert J. Walls. 2021. Demystifying the Placement Policies of the NVIDIA GPU Thread Block Scheduler for Concurrent Kernels. *SIGMETRICS Perform. Eval. Rev.* 48, 3 (March 2021), 81–88. doi:10.1145/3453953.3453972

[9] John Hubbard, Gonzalo Brito, Chirayu Garg, Nikolay Sakharnykh, and Fred Oh. 2023. Simplifying GPU Application Development with Heterogeneous Memory Management. https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/.

[10] John Jacobson, Martin Burtscher, and Ganesh Gopalakrishnan. 2024. HiRace: Accurate and Fast Data Race Checking for GPU Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 36, 14 pages. doi:10.1109/SC41406.2024.00042

[11] John Jacobson, Martin Burtscher, and Ganes Gopalakrishnan. 2024. HiRace-Artifact-SC24. https://github.com/JohnJacobsonIII/HiRace-Artifact-SC24/.

[12] Zheming Jin and Jeffrey S. Vetter. 2023. A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 325–327. doi:10.1109/ISPASS57527.2023.00041

[13] Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. 2005. Reasoning about Threads Communicating via Locks. In *Proceedings of the 17th International Conference on Computer Aided Verification* (Edinburgh, Scotland, UK) *(CAV'05)*. Springer-Verlag, Berlin, Heidelberg, 505–518. doi:10.1007/11513988_49

[14] Aditya K. Kamath and Arkaprava Basu. 2021. iGUARD: In-GPU Advanced Race Detection. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for

Computing Machinery, New York, NY, USA, 49–65. doi:10.1145/3477132.3483545

[15] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced Data Race Detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 406–422. doi:10.1145/2517349.2522736

[16] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 633–647. doi:10.1145/3062341.3062343

[17] N.G. Leveson and C.S. Turner. 1993. An Investigation of the Therac-25 Accidents. *Computer* 26, 7 (1993), 18–41. doi:10.1109/MC.1993.274940

[18] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. 2023. MC Mutants: Evaluating and Improving Testing for Memory Consistency Specifications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 473–488. doi:10.1145/3575693.3575750

[19] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P Rajan. 2012. GKLEE. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM. doi:10.1145/2145816.2145844

[20] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug Isolation via Remote Program Sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 141–154. doi:10.1145/781131.781148

[21] Yiqian Liu, Noushin Azami, Avery Vanausdal, and Martin Burtscher. 2024. Indigo3: A Parallel Graph Analytics Benchmark Suite for Exploring Implementation Styles and Common Bugs. *ACM Trans. Parallel Comput.* 11, 3, Article 13 (Aug. 2024), 29 pages. doi:10.1145/3665251

[22] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. 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, 257–270. doi:10.1145/3297858.3304043

[23] NVIDIA. 2025. Compute Sanitizer v2025.3.1. https://docs.nvidia.com/cuda/sanitizer-docs/pdf/ComputeSanitizer.pdf

[24] NVIDIA. 2025. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 9-5-2025.

[25] NVIDIA. 2025. CUDA C++ Programming Guide : Atomic Functions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomic-functions. Accessed: 11-13-2025.

[26] NVIDIA. 2025. CUDA C++ Programming Guide : Distributed Shared Memory. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#distributed-shared-memory. Accessed: 11-12-2025.

[27] NVIDIA. 2025. CUTLASS 4.3.0. https://github.com/NVIDIA/cutlass

[28] NVIDIA. 2025. NVIDIA CUDA Profiling Tools Interface (CUPTI) - CUDA Toolkit. https://developer.nvidia.com/cupti

[29] NVIDIA. 2025. NVIDIA/cuda-samples. https://github.com/NVIDIA/cuda-samples

[30] NVIDIA. 2025. PTX ISA 9.0 Documentation - Instruction Operands : Generic Addressing. https://docs.nvidia.com/cuda/parallel-thread-execution/#generic-addressing

[31] NVIDIA. 2025. PTX ISA 9.0 Documentation - Memory Consistency Model. https://docs.nvidia.com/cuda/parallel-thread-execution/#morally-strong-operations

[32] NVIDIA. 2025. PTX ISA 9.0 Documentation - Miscellaneous Instructions : nanosleep. https://docs.nvidia.com/cuda/parallel-thread-execution/#miscellaneous-instructions-nanosleep

[33] NVIDIA. 2025. PTX ISA 9.0 Documentation - Parallel Synchronization and Communication Instructions. https://docs.nvidia.com/cuda/parallel-thread-execution/#parallel-synchronization-and-communication-instructions-match-sync

[34] NVIDIA Corporation. 2017. NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Whitepaper.

[35] NVIDIA Corporation. 2022. NVIDIA H100 Tensor Core GPU Architecture Whitepaper. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c. Whitepaper.

[36] Chao Peng and Ajitha Rajan. 2020. Automated Test Generation for OpenCL Kernels Using Fuzzing and Constraint Solving. In *Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit* (San Diego, California) *(GPGPU '20)*. Association for Computing Machinery, New York, NY, USA, 61–70. doi:10.1145/3366428.3380768

[37] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: A Dynamic CUDA Race Detector. *ACM SIGPLAN Notices* 53, 4 (2018), 390–403. doi:10.1145/3296979.3192368

[38] John Regehr. 2010. A Guide to Undefined Behavior in C and C++. https://blog.regehr.org/archives/213. Accessed: 10-30-2025.

[39] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411. doi:10.1145/265924.265927

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28. https://www.usenix.org/conference/atc12/addresssanitizer-fast-address-sanity-checker

[41] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. 62–71. doi:10.1145/1791194.1791203

[42] Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyrklevich, Christian Holler, Julian Lettner, David Kilzer, and Lander Brandt. 2024. GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice* (Lisbon, Portugal) *(ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 168–177. doi:10.1145/3639477.3640328

[43] Tyler Sorensen and Alastair F. Donaldson. 2016. Exposing errors related to weak memory in GPU applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 100–113. doi:10.1145/2908080.2908114

[44] Mark Stephenson, Sana Damani, Mohamed Tarek Ibn Ziad, Anis Ladram, and Michael Garland. 2026. Reproduction Package for Article 'SuperCollider: Scalable and Effective Data Race Detection for CUDA'. Zenodo. doi:10.5281/zenodo.19058944

[45] Texas State University. 2019. ECL graphs. https://userweb.cs.txstate.edu/~burtscher/research/ECLgraph/index.html.

[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*. ACM. doi:10.1145/3352460.3358307

[47] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined Behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems* (Seoul, Republic of Korea) *(APSYS '12)*. Association for Computing Machinery, New York, NY, USA, Article 9, 7 pages. doi:10.1145/2349896.2349905

[48] 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. *SIGPLAN Not.* 51, 8, Article 11 (Feb. 2016), 12 pages. doi:10.1145/3016078.2851145

Received 2025-11-13; accepted 2026-04-03