HiRace: Accurate and Fast Data Race Checking for GPU Programs

John Jacobson
Kahlert School of Computing
University of Utah
Salt Lake City, USA
john.jacobson@utah.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, USA
burtscher@txstate.edu

Ganesh Gopalakrishnan
Kahlert School of Computing
University of Utah
Salt Lake City, USA
ganesh@cs.utah.edu

***Abstract*—Data races are egregious concurrency bugs that are especially problematic in performance-oriented GPU codes where large thread counts and multiple shared memory regions tend to exacerbate them. In this work, we present a new dynamic data-race checker called HiRace, whose key novelty is an innovative state machine designed to capitalize on the bulk-synchronous hierarchical GPU programming model. This state machine condenses an arbitrarily long access history into a constant-size state. We evaluate HiRace on a large, calibrated data-race benchmark suite. In over 3,500 studied executions of 580 CUDA kernels, 346 of which contain data races, we found HiRace to detect races missed by other tools without raising false alarms and to be more than 10 times faster on average than the current state of the art with half the memory overhead.**

***Index Terms*—GPU Programming; Parallelism; Debugging; Data-Race Detection.**

I. Introduction

GPUs are central to supporting massive levels of parallelism and are today the mainstay of virtually all advances in HPC and AI. Performance-seeking GPU programmers aim to maximize concurrent computations and to increase data throughput through the use of shared memory. This can easily lead to situations where the resulting program allows one thread to write a value to a shared memory location while other threads concurrently access the same location, creating a *data race* that prevents the system from maintaining a consistent view of memory for all threads. Although GPU languages provide synchronization primitives to enforce orderings on memory actions, their use is not always intuitive and they have significant performance costs. Since GPU programs emphasize performance, developers often utilize complex code patterns to avoid synchronization overheads, making the code difficult to reason about and more likely to contain subtle data races.

Data races can cause non-deterministic behaviors and produce "out of thin air" values [1] (that is, a value may be read from memory that was never written). Racy code may also appear to work normally under low compiler optimization levels but misbehave when optimized [2]. The non-deterministic nature of data races, coupled with the complexity of shared memory semantics, makes both conventional testing and manual inspection unreliable for identifying them. The rapid pace of development in GPU architectures often results in races remaining dormant on current hardware, only to later manifest on newer architectures.

While numerous solutions have been proposed for automatically identifying GPU data races, most struggle to scale with the GPU programming model, leading to either prohibitive overhead or compromised completeness. For instance, NVIDIA's own Compute Sanitizer [3] Racecheck tool—a dynamic analysis tool—does not check for data races occurring within the GPU global memory space, presumably due to an inability to scale to full device memory. This is a serious practical limitation as most large-scale codes shared data through GPU global memory. Another promising dynamic analysis tool designed specifically for NVIDIA GPUs—iGUARD [4]—relies on traditional CPU race-checking methodology but sacrifices completeness to maintain tolerable memory overheads. Unfortunately, iGUARD is not actively maintained and, due to its dependence on the NVBit dynamic binary instrumentation tool [5], it does not run on NVIDIA architectures released since its publication. Many static GPU race-checking tools have been created (e.g., PUG and GPU-Verify [6], [7]), but these present false alarms to the user, which are costly to handle, and few of these tools are maintained.

The Faial [8] family of tools offer a static analysis-based approach to race checking, and have been shown to be highly effective in field studies. Their approach is compared with the dynamic approach of HiRace in §VI.

Evaluating Race-Checkers: To fairly evaluate the bug-finding ability of various race-detection tools, we employ a large *calibrated* benchmark suite, namely Indigo [9], that presents scenarios of different difficulty levels.

Emphasis on Widespread Usability: HiRace works at the source-level by automatically instrumenting code through Clang's source rewriting API [10], part of the widely adopted and actively maintained LLVM project. Functionally, the HiRace methodology is designed to support the GPU programming model, providing significant improvements in completeness, run time, and memory overhead for analyzing codes utilizing GPU synchronization primitives. As a dynamic checker, HiRace does not generate false alarms.

Low Book-keeping Overhead: A key novelty of HiRace is its low book-keeping overhead enabled by our observation

Supported in part by NSF Awards 1955367, 1956106, 2124100, 2217154, and 2426055.

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

that within a barrier-oriented hierarchical GPU programming model, a single accessor record suffices to identify races (elaborated in §III). In comparison, methodologies that are designed around the CPU parallel programming model maintain multiple accessor records per memory location to dynamically reconstruct the access history. This requires significantly more memory in the best case. In the worst case, the memory overhead can grow unsustainably while still missing races.

Explicit State-Machine: HiRace achieves a modest size (constant space per memory location) access history through the use of a distributed finite-state machine that encodes access histories compactly (5 bits of state per tracked memory location). This is achieved in part by equating threads to thread groups following the GPU concurrency hierarchy (§III).

```
1  __global__ void
2  test_kernel(int* nindex int* nlist,
3      data_t* data1, data_t* data2,
4      int numv)
5  {
6    int idx = threadIdx.x + blockIdx.x * blockDim.x;
7    for (int i = idx; i < numv; i += gridDim.x * blockDim.x) {
8      int beg = nindex[i];
9      int end = nindex[i + 1];
10     for (int j = beg; j < end; j++) {
11       int nei = nlist[j];
12       if (i < nei) {
13         data1[nei] = min(data1[nei], data2[i]);
14       }
15     }
16   }
17 }
```

Listing 1: A CUDA kernel containing a data race from the Indigo suite - Atomic bug (line 13)

Correctness: Representing our model as an explicit state-machine also allowed us to develop a Murphi [11] formal reference model. This model was model-checked and verified against HiRace's behavior. This extra step gives increased confidence in the correctness of our design and aids in extending our methodology to new and more complex memory models (a similar verification exercise was reported in [12]).

Our extensive benchmarking against the Indigo suite provides added confidence as these codes contain irregular graph algorithms that exhibit data-dependent memory accesses and control flow. Such behavior makes programs harder to debug because even buggy codes will execute correctly for inputs where (1) the control flow avoids the problematic code sections (e.g., Listing 1) or (2) not all memory index values are exercised. HiRace's reporting matches Indigo's calibration and, additionally, identified an unintended data race within some Indigo benchmarks. We found that **HiRace can be 30× to 50× faster than the state of the art—with median speedups of at least 7×** (Figure 5).

In summary, HiRace:

- employs an innovative state-machine-based design,
- detects more races than the state of the art in dynamic GPU data-race detection,
- is on average 10× faster and scales to much larger programs compared to the state of the art,
- has been tested against a calibrated set of benchmarks,
- uses source-level instrumentation based on Clang, which facilitates porting to newer GPU languages, and
- has been verified with the Murphi model checker.

Roadmap: After a background on shared-memory race checking and GPU data-race checking (§II), we present a systematic walk-through of HiRace's design (§III) followed by our experimentation methodology (§IV), analysis of our results (§V), and concluding remarks (§VII).

II. Background

Data-race checking is a topic with a long history [13]. In this section, we outline important background on data races and CUDA programming paradigms. We also present prior work and contrast it with HiRace.

A. Data Races

A parallel program has a data race if multiple threads access the same memory location, at least one of the accesses is a write, and the accesses are not ordered under a *happens-before* [1], [14] relation (thus are concurrent accesses). An example may be seen in Listing 1 on line 13, where multiple threads may write to the same index in the `data1` array.

Tools such as FastTrack [15] and Google's Thread Sanitizer [16] are built around mechanisms to track happens-before, albeit for (POSIX-like) threads. Consequently, detecting data races requires first identifying all memory accesses, particularly those that are *conflicting* (accesses to the same location from different threads), and second determining whether any conflicting accesses thus identified are happens-before ordered. While doing so, data-race-detection tools aim for a balance between two properties: reporting all races ("no omissions", or *completeness*) and NOT reporting any non-races ("no false alarms", or *soundness*). There are two primary approaches to making these determinations: *static* and *dynamic* analysis. Although static analysis methods can be more general (verify for unknown values of the number of threads and input data), they suffer from a key weakness: they easily produce false alarms that are costly to validate manually.

Dynamic analysis is performed by monitoring the underlying program at run-time. Doing so avoids the burden of determining reachability of instructions since all executed instructions can be directly witnessed. However, dynamic methods are restricted to analyzing program instructions that are observed in a given test execution. Any conditional code paths that are not explored by a given configuration cannot be tested, potentially leaving regions of code unanalyzed. This problem can be alleviated by testing the code with multiple inputs that elicit the various possible program behaviors.

Many modern tools employ a combination of these methods (*hybrid* analysis), such as Archer [17] (that employs very limited static analysis to black-list specific parts of the code) and GKLEE [18] (that uses dynamic-symbolic execution [19]).

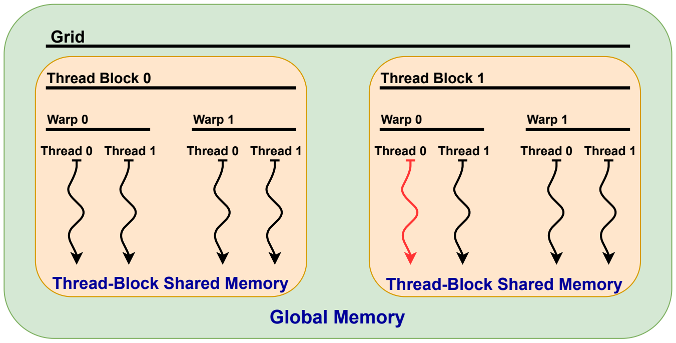

Fig. 1: Simplified grid showing the hierarchical structure of CUDA threads and shared memory.

B. *CUDA Shared-Memory Concurrency*

CUDA devices present large numbers of threads to the user, which are organized into hierarchical structures to be dynamically scheduled and executed on a number of streaming multiprocessors (SMs). Each SM has several SIMT vector units. Modern GPUs have over 100 SMs on a single device, each capable of hosting over 1,000 resident threads.

Logically, individual threads are arranged into *thread blocks*, which are executed on a single SM. The threads of a thread block are arranged into *warps*, which are executed on a single SIMT unit. Device memory is composed of several spaces (or *scopes*) with differing accessibility. *Global memory* is accessible by all threads on the device, whereas *shared memory* is allocated to individual thread blocks and is accessible only to threads within the same thread block. A simplified view of this structure is shown in Figure 1.

For managing this thread hierarchy, CUDA provides a set of block-scope synchronization primitives (*barriers*) as well as a set of warp-scope synchronization primitives.

The most basic block-scoped barrier is `__syncthreads`. Informally, any thread that executes a `__syncthreads` instruction will wait for all threads within the same thread block to execute the same `__syncthreads` before proceeding to its next instruction. There are variants of this primitive that provide the same guarantee but also perform a thread-block-wide reduction operation and broadcast the result to all threads in the block. The most basic warp-scoped barrier is `__syncwarp`. It provides the same functionality as `__syncthreads` but is limited to the threads within the same warp. Additionally, `__syncwarp` takes a *mask* argument that restricts the participating threads to only those specified in the mask (which must all be members of the same warp). Like for the block-level barriers, there are additional warp synchronization primitives that allow thread communication alongside the synchronization effects.

These hierarchical structures and synchronization primitives are the core of the CUDA programming model. Effective use of GPU resources entails maximizing parallel computation (or *occupancy*) and reducing data movement. Unfortunately, many traditional parallel programming patterns, such as mutex-based programming and release-acquire communication patterns, are designed for fine-grained synchronization of resources with low contention. These patterns are less practical on GPUs since they create bottlenecks for large numbers of threads. As evidence of this, CUDA does not provide an explicit API for these patterns and encourages the use of scoped synchronization primitives (barriers and atomic operations) to create more efficient device programs.

III. Design of HiRace

The design of HiRace is based on the enumeration of access patterns to a given memory address. While this space is unbounded in the general case of fine-grained release-acquire thread communication, we observe that it is bounded when restricted to the bulk-synchronous GPU programming model. By choosing a hierarchy-aware representation that opportunistically equates threads to the largest possible hierarchical thread group, the number of distinct and relevant access patterns is actually quite small. We represent all such access patterns within 25 distinct states, which encompasses all interleavings of reads, writes, and atomic read-modify-write instructions to a single memory location when constrained to thread synchronization exclusively through scoped thread barriers (such as CUDA's *__syncthreads*).

This model does not improve the current state of the art for monitoring synchronization using release-acquire communication or lock-based synchronization. HiRace's methodology falls back to existing methods for tracking these forms of synchronization (such as maintaining lock tables for analyzing lock-based programs). In practice, these traditional CPU parallel programming patterns are not common in high-performance GPU programs as they are not core components of the bulk-synchronous programming model. This is evidenced by GPU programming languages such as CUDA not providing explicit lock API's, instead relying on barriers.

HiRace is based on a finite state machine (FSM) that encodes the happens-before logic introduced earlier. Whereas the FSM is tailored to the current CUDA programming model, it can easily be extended to other models and deeper hierarchies. The HiRace FSM has 25 states and 1200 transitions between these states. They are sufficient to describe the interactions of all atomic and non-atomic reads and writes to a single memory location as well as all needed synchronization history from warp-scoped and block-scoped barrier primitives.

Rather than exhaustively detailing each state of the FSM (whose complete specification is available at https://github.com/JohnJacobsonIII/HiRace-Artifact-SC24), this section incrementally develops similar state machines for simpler concurrency models. To this end, we present two scenarios that introduce the key ideas behind our approach: how the FSM performs atomic updates, and how we succinctly express the FSM transitions.

A. *Simple Scenarios*

First, we consider the set of CUDA programs containing only non-atomic reads and writes to a single memory space

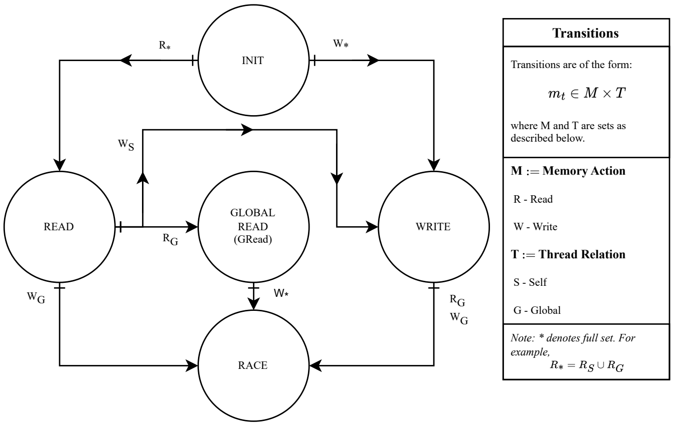

Fig. 2: State machine for code without barriers; any unmentioned outgoing transition from a state means we stay in the same state—a "self loop"

(global memory) that use no form of synchronization. Listing 2 provides an example of such a program.

```
1 __global__ void race_noSync(int* data) {
2   int i=threadIdx.x + blockIdx.x * threadsPerBlock;
3   int val = data[0];
4   data[0] = i + val;  // race!
5 }
```

Listing 2: Read-write data race: A simple CUDA kernel exhibiting a data race on line 4. All device threads read the same global memory address (data[0]), and then subsequently all write the same address causing a data race between all threads on the device.

We claim that the FSM in Figure 2, along with a single prior accessor ID, is sufficient to track all data races that may occur in such programs along a single control-flow path.

To monitor the $data$ array for races, we create a $shadow[k]$ entry for each array element $data[k]$, as shown in Figure 3. Each $shadow[k]$ entry maintains the following information: the block-scalar clock (BC), the warp-scalar clock (WC), the ID of the thread performing the *most recent access* (TID), and the state of the FSM from Figure 2 ($State$).

Initially, all $State$ values are set to INIT, indicating that no memory accesses have occurred yet. Let us assume the thread organization shown in Figure 1. We label the eight threads T_{000} to T_{111} to indicate their block, warp, and TID. For example, the "red" thread is T_{100}.

In the scenarios we describe, all CUDA threads accessing $data[k]$ atomically copy, due to our source instrumentation, $shadow[k]$ into $oShadow$, which is the tuple $\langle oBC, oWC, oTID, oState \rangle$[1]. Then, they determine the new tuple $\langle nBC, nWC, nTID, nState \rangle$ to write back to $shadow[k]$ in a lock-free manner using an ATOMICCAS instruction. Only one thread will succeed; the "losers" re-obtain a copy of $shadow[k]$ and try again. To illustrate this process, let us walk through a few specific scenarios.

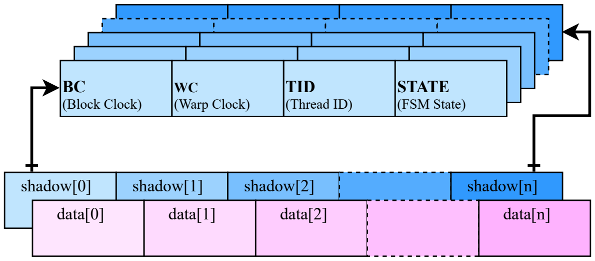

Fig. 3: Shadow information used by HiRace

Scenario 1: Suppose T_{011} and T_{100} concurrently execute Line 3 in Listing 2. Suppose further that T_{011} is the winner. At this point, the instrumented code transitions the $State$ from INIT to READ (the transition is labeled R_*, meaning "any read by any thread"). Since T_{100}'s update attempt fails, it re-reads $State$ and finds it to be READ with a different TID. Thus, it advances the state to GREAD—meaning **G**lobal read—since this memory location was previously read by a different thread. If thread T_{011} executes Line 4 next, the state machine follows the W_* arc (meaning "any write at all"), transitioning the state to RACE.

Note that there is no obligation for the instrumented code handling T_{100} to update $shadow[k]$ before T_{011} executes Line 4. In other words, *there is no requirement that a location access and its instrumented code execute atomically.*[2]

Of course, it is quite possible that only T_{011} has updated $shadow[k]$ when T_{011} reaches Line 4, in which case it updates the state from READ to WRITE following the W_S transition, which stands for **S**ame-thread write. If the instrumented code for T_{100} now updates $shadow[k]$, it enters the RACE state since any read or write access by a different thread is a race.

```
__global__ void race_blockSync(int* data)
{
  int tid = threadIdx.x;
  int val = data[0];
  __syncthreads();
  if (tid > 0) {
    data[tid − 1] = tid;  // race!
  }
}
```

Listing 3: Read-write data race with syncthreads (block-scope barrier)

Scenario 2: For this scenario, we consider the program in Listing 3, which contains a block-scoped syncthreads barrier. To explain the state-machine activities in the presence of barriers, we extend the notation on how reads and writes are subscripted. The subscripts are of the form $\{S, T\}$, where S is the synchronization status, which is either Us ("unsynchronized") or Bs ("block-synchronized"), and T is the thread relation, which now has a B ("block") option in addition to the former S ("self") and G ("global"). The new thread relation B indicates that a barrier has "block-synchronized" the access

[1] Prefix o indicates "old" values, i.e., the values stored by a prior accessor, whereas prefix n denotes the "new" values determined by the current accessor.

[2] This is also important for performance.

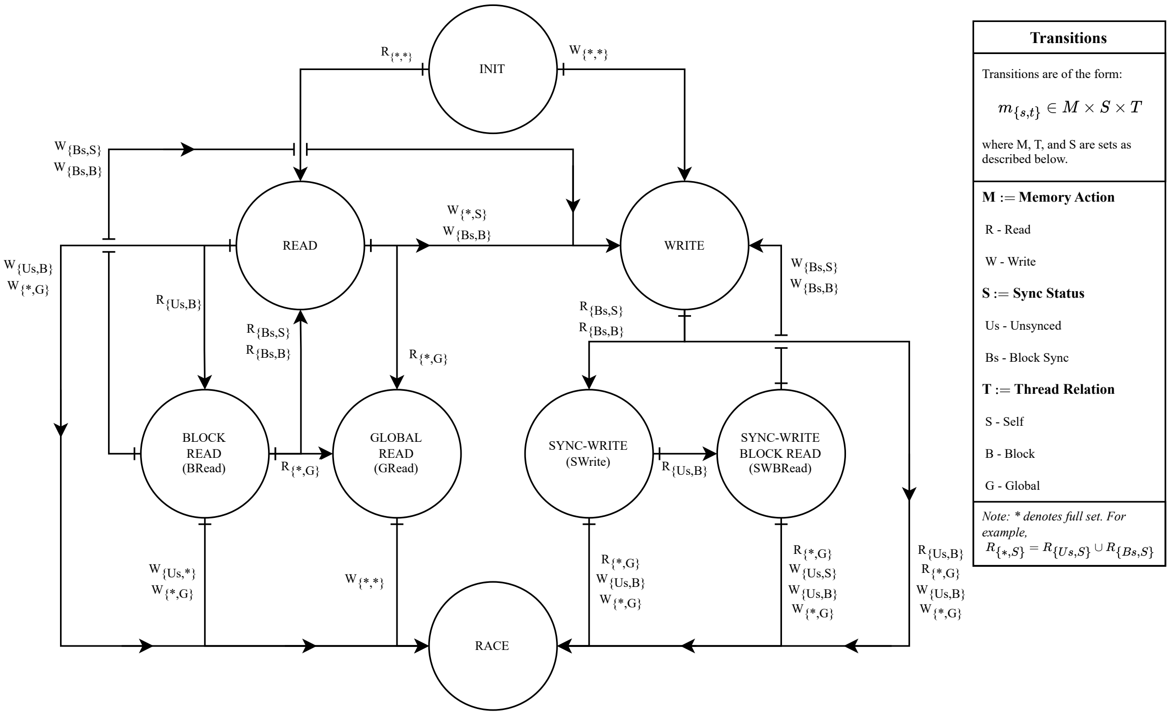
INIT
READ
WRITE
BLOCK READ (BRead)
GLOBAL READ (GRead)
SYNC-WRITE (SWrite)
SYNC-WRITE BLOCK READ (SWBRead)
RACE
$R_{\{*,*\}}$
$W_{\{*,*\}}$
$W_{\{Bs,S\}}$ $W_{\{Bs,B\}}$
$W_{\{*,S\}}$ $W_{\{Bs,B\}}$
$W_{\{Us,B\}}$ $W_{\{*,G\}}$
$R_{\{Us,B\}}$
$R_{\{Bs,S\}}$ $R_{\{Bs,B\}}$
$R_{\{*,G\}}$
$R_{\{*,G\}}$
$W_{\{Bs,S\}}$ $W_{\{Bs,B\}}$
$R_{\{Bs,S\}}$ $R_{\{Bs,B\}}$
$R_{\{Us,B\}}$
$W_{\{Us,*\}}$ $W_{\{*,G\}}$
$W_{\{*,*\}}$
$R_{\{*,G\}}$ $W_{\{Us,B\}}$ $W_{\{*,G\}}$
$R_{\{*,G\}}$ $W_{\{Us,S\}}$ $W_{\{Us,B\}}$ $W_{\{*,G\}}$
$R_{\{Us,B\}}$ $R_{\{*,G\}}$ $W_{\{Us,B\}}$ $W_{\{*,G\}}$
Transitions
Transitions are of the form:
$m_{\{s,t\}} \in M \times S \times T$
where M, T, and S are sets as described below.
M := Memory Action
R - Read
W - Write
S := Sync Status
Us - Unsynced
Bs - Block Sync
T := Thread Relation
S - Self
B - Block
G - Global
Note: * denotes full set. For example,
$R_{\{*,S\}} = R_{\{Us,S\}} \cup R_{\{Bs,S\}}$

Fig. 4: State-machine for code with block synchronization; any unmentioned outgoing transition from a state means we stay in the same state

(indicated by differing block scalar clock, BC, values).[3] The wildcard "$*$" means any applicable letters.

Consider the scenario of T_{011} (the "red" thread of Fig. 1) executing Line 4 in Listing 3, where it reads location 0 of the data array. This access updates the state in $shadow[0]$ to READ, as indicated by the wildcard-subscripted $R_{\{*,*\}}$ transition in Fig. 4. Suppose T_{000}, another thread in the same block, now executes Line 4. This access updates $shadow[0]$ to BREAD (Block Read)—a new state—that captures the fact that two threads from the same block scope have read this location asynchronously, as indicated by the label $R_{\{U_s,B\}}$. As long as other threads from the same block execute Line 4, $shadow[0]$ remains in the BREAD state. As soon as a thread from a different block, for example T_{110}, executes Line 4, we promote $shadow[0]$ to GREAD ("global read") as a read has been performed in the "global scope" ($R_{\{*,G\}}$).

Once we are in the GREAD state, any type of write by any thread ($W_{\{*,*\}}$) results in a RACE as this memory location has witnessed a read from at least two blocks. This is important because syncthread barriers do not synchronize threads across blocks, so any writing thread must be executing asynchronously to at least one of the prior readers.

Instead of thread T_{110} performing the aforesaid access, let $shadow[0]$ remain in BREAD and let the red thread T_{011} execute `__syncthreads` on Line 5. Because of the barrier semantics, all threads must cross the barrier before they can execute any following instruction. Therefore, eventually, the thread-private block-scalar clock (BC) updates to 1 in all threads in this block. At that point, a write by any thread in block B_0 sends the line to the WRITE state as indicated by the transitions $W_{\{B_s,S\}}$ and $W_{\{B_s,B\}}$. Even so, T_{110} eventually performs a read of this location, thus transition to the RACE state. The remaining transitions of the FSM in Fig. 4 may be similarly explored.

B. Extending to Atomics and Warp Scope

This section provides an informal view of how our FSM works. In a similar manner to what is described here, the HiRace FSM has been extended beyond what is shown in Fig. 4 and fully supports:

- Reads, writes, global atomics, and block-scope atomics,
- Global and shared memory regions,
- Device synchronization (including implicit synchronization after kernel execution), all variations of syncthreads, syncwarps, and warp-voting primitives.

Scoped atomic operations are represented as two additional classes of memory action (beyond Read and Write) and require new states that describe the corresponding access histories (e.g., a synchronized atomic access is distinct from a synchronized write access) as well as the relationship between different scopes.

Warp-scoped barriers require expanding the transition function with a new synchronization status and new thread relations. They also require additional states to capture the warp-scoped synchronization history while retaining the higher-scoped synchronization history (e.g., a thread that executes `syncthreads` followed by `syncwarp`).

[3] The size of these clocks is configurable, and race detection is discontinued with a warning if a clock overflows (after reporting identified races).

In summary, the current HiRace FSM includes 4 read-tracking states, 7 write-tracking states, 12 atomic-tracking states, and the initial and race states. Transitions are based on 4 memory access types (reads, writes, block-scoped atomics, and global-scoped atomics), 3 synchronization statuses (unsynchronized, warp synchronized, and block synchronized), and 4 thread relations (same thread, warp-scope, block-scope, or grid-scope). This results in $4 \cdot 3 \cdot 4 = 48$ transitions per state.

C. *Correctness*

To improve confidence in the correctness of the HiRace state machine, we encoded it as a Murphi model [11]. Murphi is an explicit-state model checker that takes a model written in the Murphi specification language and compiles it into a C++ program (called a 'verifier of the model') that explores the state space defined in the original model for invariant properties and assertions. Our Murphi model consists of a driver model, which simulates GPU memory interactions and synchronization, and an analysis model, which models the HiRace algorithm. The model is a bisimulation of these two components where each GPU action initiates a transition of the HiRace FSM. This model preserves the invariant that the analysis model transitions to the RACE state if and only if the driver model witnesses a data race.

Threads are uniquely identified by a combination of thread, warp, and block scalar sets. Actions are constrained to those relevant to the HiRace algorithm and include read, write, atomics, syncblock, and syncwarp. The `GlobalMem` record represents a single global memory location as an array of actions performed by each thread on that location as well as the Race flag, which indicates whether a race has been witnessed in the current state. The `SharedMem` record is similar: it contains an array of one shared memory location per block.

The driver model's actions are formed by rule sets representing a unique rule for each thread and GPU action described above on a single memory location. For example, there is one rule representing thread 1, warp 0, block 1 performing a shared memory write in `SharedMem[1]`. After updating its global state (including maintaining a history of operations), a *Turn* bit is flipped indicating that the analysis model should perform a transition (by enabling only rules that update the analysis model's state). The analysis model is a direct translation of the actual HiRace FSM. The model state contains the current FSM state (enumeration type) for each memory location and scoped clocks (a finite integer range) and a record of the prior accessing thread. The current driver action enables an associated analysis rule, and each such rule enumerates the transitions from each FSM state. For example, a rule for the WRITE action switches on the current FSM state, and then the current and prior threads and clocks are compared to determine the next FSM state. Finally, that state and the thread metadata are stored and the *turn* bit is flipped to return control the driver. The model is defined with the invariant that the analysis model enters the RACE state if and only if the driver model exhibits a data race.

IV. IMPLEMENTATION

HiRace's configuration is exposed through a *toml* interface, allowing easy specification of which kernels and variables to monitor for races, which thread groups to monitor, and bounds for shadow value metadata to reduce memory overhead.

A. *Memory Access Monitoring*

To capture memory accesses, HiRace utilizes a templated wrapper class to monitor user data structures. For each shared memory variable, a shadow value data structure is allocated. Both the original shared memory variable and the shadow value pointer are tracked within a thread-local instance of the HiRace wrapper. The wrapper then overrides relevant operators (for example, the subscript or array index operator `operator[]`) to intercept memory access events and update the associated shadow value transparently. This approach also requires overriding other relevant functions, such as atomic functions and synchronization primitives. This methodology allows flexible management of the underlying test data structures as well as their associated shadow structures and thread metadata. Individual threads or test data addresses may have their instrumentation enabled or disabled dynamically. The memory footprint can be reduced by tracking only representative threads from a symmetric group.

Algorithm 1 Lock-Free Shadow Value Update

```
Require:
$sAddr$ : Shadow value address
$access$ : Access type (read/write/atomic)
$tid$ : Global thread ID
$bc$ : Block-scope scalar clock
$wc$ : Warp-scope scalar clock
procedure UPDATESHADOW($sAddr, access, tid, bc, wc$)
    repeat
        $oShadow \leftarrow$ ATOMICREAD($sAddr$)

        $\langle oState,$
        $oTid,$
        $oBC,$
        $oWC\rangle \leftarrow$ UNPACKSHADOW($oShadow$)

        $tRel \leftarrow$ COMPARETIDS($tid, oTid$)
        $sRel \leftarrow$ CHECKSYNC($bc, oBC, wc, oWC$)

        $trans \leftarrow$ GETTRANS($oState, access, sRel, tRel$)
        $nState \leftarrow$ STATEMACHINELOOKUP($trans$)

        $nShadow \leftarrow$ PACKSHADOW($nState, tid, bc, wc$)
    until ATOMICCAS($sAddr, oShadow, nShadow$)
end procedure
```

B. *Race Detection*

The core of HiRace's detection logic is embedded in the shadow state update algorithm (Algorithm 1) built around the state machine described in Section III. As previously described, our FSM contains 25 states and 1200 transitions. It requires only 5 bits of memory to record the FSM state per monitored memory address, which is combined with other metadata (as described in Fig. 3) resulting in 8 bytes of memory per address by default (though this is configurable and can be reduced). In contrast, the state of the art uses 16 bytes per monitored address to track 2 prior accessors with no ability to reason about any earlier accesses in general.

When a thread accesses a monitored memory address through the HiRace wrapper, it calls Algorithm 1. This causes the thread to (atomically) check the metadata for the memory location it is accessing, compare it against its own current metadata, and use the result to update the state to a new shadow value (again atomically).

In more detail, the first step is entering the lock-free loop in Line 7 of Algorithm 1. Any accessing threads may concurrently compare the old shadow value ($oShadow$) to their own metadata, but a thread may only store its new shadow value ($nShadow$) if it verifies, via the `atomicCAS`, that $oShadow$ has not changed. If any other thread updates the shadow value in the meantime, the current thread detects the change, starts over, and re-calculates $nShadow$ relative to the newly read $oShadow$. All shadow-value accesses in shared memory are atomic and guaranteed to match FSM transitions. Since most of the race-detection logic is contained within the FSM, the amount of computation in the `atomicCAS` loop is minimal. Each shadow value is a bit array as outlined in Fig. 3. $UnpackShadow$ splits the shadow value into individual "old" metadata values from the prior access. $CompareTids$ and $CheckSync$ compare both old and new metadata values to determine the Thread Relation and Sync Status labels as described in Fig. 4.

The HiRace FSM is encoded as a flat array indexed by a concatenation of $oState$ and the transition labels $m_{\{s,t\}}$ shown in Fig. 4, which map to a new state. $GetTrans$ creates an integer index into the state-machine array using bit operations to combine $oState$ with the current Memory Action ($access$), Thread Relation, and Sync Status. Afterwards, $StateMachineLookup$ is just an array access to index $trans$.

Finally, the newly calculated state is concatenated with the current metadata, and the thread attempts to store the result.

C. Source Instrumentation

To facilitate instrumentation with our wrappers, we provide a standalone Clang tool for AST parsing and rewriting. The Clang LibTooling C++ interface facilitates parsing source code into AST as well as walking and rewriting the AST and/or source code through a robust API.

As HiRace is a header library designed to wrap around existing data structures and override relevant CUDA language primitives, it can be utilized through manual instrumentation fairly easily. However, the HiRace Clang tool automates this instrumentation. The Clang AST allows us to mechanically identify and wrap monitored data pointers and to allocate and deallocate necessary metadata.

V. Evaluation

Our experimental platform is as follows: (1) CPU: AMD Ryzen 5 3600; (2) GPU: NVIDIA Geforce RTX 2070 Super; (3) OS: Ubuntu 22.04; and (4) CUDA version: 11.7.

A. Section: Indigo Benchmarks

We use programs and inputs from the recently released Indigo benchmark suite [9] to drive our experiments. Unlike conventional suites, Indigo contains scripts and configuration files that allow the user to generate the desired codes and inputs. It is also the only labeled data-race detection benchmark suite with GPU codes.

There are 14 graph generators to choose from for creating parameterizable inputs of arbitrary size. There are also 21 CUDA kernel generator patterns, which generate up to 580 distinct CUDA programs operating on a given input word size. One example kernel is shown in Listing 1, which contains a data race on line 13 to the `data1` array. This code is labeled as an *atomicBug* in Indigo (the full list of descriptors being `push_node_neighbors`, `persistent`, `cond`, `atomicBug`), as use of CUDA's `atomicMin` would remove this race. More importantly for us, the code generators are *not only able to synthesize hundreds of different versions* of common graph-processing code patterns but also to *systematically insert various bugs*. We make extensive use of this latter capability as it enables us to control precisely which bugs, if any, are present. This provides a reliable ground truth valuation, which is typically not available in other suites or codes. Each benchmark kernel comes paired with a sequential implementation of the same algorithm as a baseline verification of the benchmark's correctness (or lack thereof).

```
1  __global__ void
2  test_kernel(int* nindex, int* nlist,
3              data_t* data1, data_t* data2, int numv)
4  {
5    __shared__ data_t s_carry[1024];
6    int tid = threadIdx.x;
7    s_carry[tid] = 0;
8
9    int i = blockIdx.x;
10   if (i < numv) {
11     __syncthreads();
12     int beg = nindex[i];
13     int end = nindex[i + 1];
14     data_t val = 0;
15     for (int j = beg + threadIdx.x; j < end; j += blockDim.x) {
16       int nei = nlist[j];
17       val += data2[nei];
18     }
19     s_carry[tid] = val;
20
21     for (int stride = blockDim.x / 2; stride > 0 ; stride >>= 1) {
22       if (tid < stride) {
23         s_carry[tid] += s_carry[tid + stride];
24       }
25       __syncthreads();
26     }
27     if (tid == 0) data1[i] += s_carry[0];
28   }
29 }
```

Listing 4: Indigo Kernel - Sync Bug (Lines 20, 24)

Lastly, graph codes are generally more complex than (and a superset of) matrix- and vector-based programs. In particular, graph analytics codes tend to be data dependent and exhibit irregular control-flow and memory-access patterns [20] in addition to the behaviors already found in matrix- and vector-based applications (such as schedule-dependent behavior).

One additional example of a buggy Indigo kernel is shown in Listing 4. This code is labeled as `pull_node_neighbors_block` with a `syncBug`. In this kernel, a data race exists on the shared memory array `s_carry` between the write on line 20 and read on line 24. A `__syncthreads` call on line 21 would remove this data race. Notably, Listings 1 and 4 are examples of Indigo benchmarks for which data races are found by HiRace when they are missed by IGUARD and/or Compute Sanitizer.

B. Results

For our tests, we used Indigo to generate 580 CUDA kernels, of which 346 contained data races. We also generated a number of small input graphs for testing the accuracy of HiRace and used 6 graphs ranging between 5 and 200 nodes to evaluate both HiRace and IGUARD. The results are displayed in Table I.

We also compared our results to Compute Sanitizer's RACECHECK tool, but since it only checks for CUDA block-shared-memory races, there were relatively few examples to compare against, and both IGUARD and HiRace found all shared-memory races identified by Compute Sanitizer.

We found that, on the 3480 Indigo benchmark comparisons between IGUARD and HiRace, neither tool provided any false positives. While IGUARD performed well in identifying data races given the constraints of classical race detection algorithms, HiRace was able to find 100% of the injected data races in the Indigo benchmarks (when provided with inputs that exhibited the injected data race).

Notably, the columns of Table I show that each tool misses races on some inputs that were found on other inputs. This is due to the input-dependent nature of dynamic analysis, i.e. certain inputs may not reach some conditional code paths.

For example, note that the inputs for which tools miss additional races here are the smallest inputs tested (5-node graphs). If a particular graph pattern executes one thread per node and a data race only exhibits across blocks, these graphs with very few nodes may not execute code across multiple blocks, and thus racy accesses will not occur in the tested execution. This constitutes a missed race as the code semantics still contain a data race under some execution parameters (such as when provided larger input graphs).

Aside from input size, other input properties can cause certain code patterns to miss conditional code paths. In this case, the two 5-node graphs differing edge sets cause differing behavior in a number of code patterns.

Further, we tested the same codes on eleven large-scale real-world graph inputs (as described in Table II) to test performance, as shown in Figure 5. IGUARD struggled with large grid dimensions, and the timing results were very sporadic (occasionally requiring multiple hours, yielding over 1000x slowdown over baseline code execution). We were able to compare 441 executions across these large graphs to IGUARD, which demonstrated an overall average overhead of more than 30x slowdown, against HiRace's average of less than 3x slowdown across the same code/input combinations.

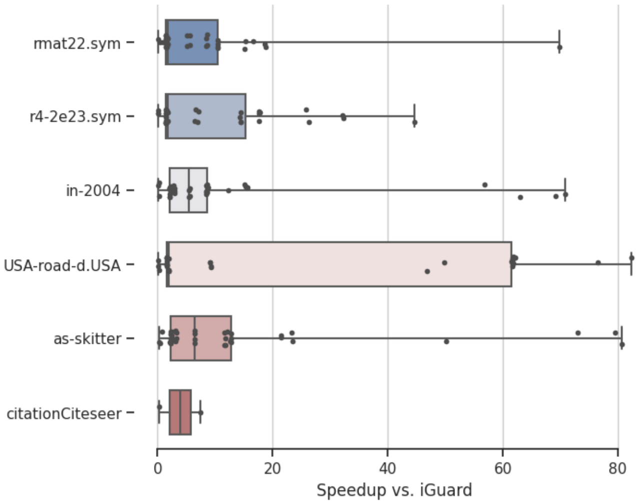

Fig. 5: HiRace speedup vs. iGuard. Each point represents execution of one Indigo benchmark kernel on the associated input graph.

With HiRace's low overhead, we were able to more thoroughly test its performance against baseline codes. Across 5,105 executions (464 Indigo codes across the 11 large graphs in Table II), HiRace demonstrated 7.5x average slowdown while still identifying 100% of the injected data races, with no false positives.

HiRace also has a few outliers in runtime, primarily due to its race reporting method, which requires significantly more I/O on some code patterns to report races on unique memory addresses. We intend to address this (and provide configuration options for reporting) in the future.

For additional perspective, we compared the same three

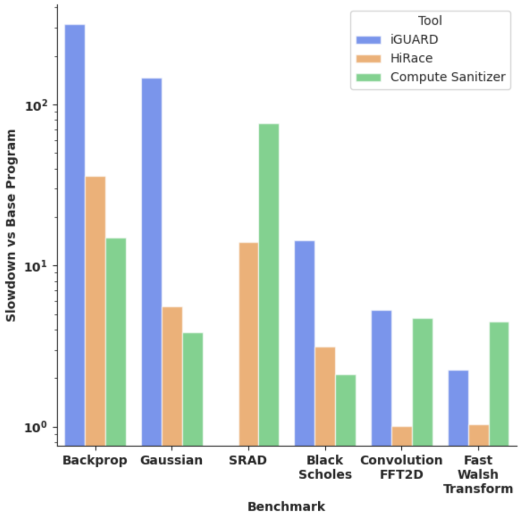

Fig. 6: Race-Detection Slowdown of Rodinia and Cuda-Samples Benchmarks.

TABLE I: Number of Indigo benchmark codes tested by each tool on the listed input.
Here, the column "Sequential Comparison" indicates benchmarks for which sequential execution and parallel execution did not produce matching results—indicating that the parallel execution exhibited a data race. These results are discussed further in Section V-C.

Input Graph	Total tests	HiRace Races Found	HiRace Races Missed	iGUARD Races Found	iGUARD Races Missed	Compute Sanitizer Races Found	Compute Sanitizer Races Missed	Sequential Comparison Races Found	Sequential Comparison Races Missed
DAG_100n_200e	580	346	0	298	48	92	254	301	45
DAG_200n_400e	580	346	0	298	48	92	254	301	45
DAG_5n_5e	580	182	164	118	228	92	254	44	302
counterDAG_200n_1000e	580	346	0	298	48	56	290	282	64
counterDAG_5n_5e	580	295	51	148	198	92	254	194	152
power_law_200n_1000e	580	346	0	297	49	128	218	306	40

TABLE II: Real-world graph inputs

Graph	Type	Vertices	Edges
rmat16.sym	RMAT	65,536	967,866
internet	Internet topology	124,651	387,240
USA-road-d.NY	road map	264,346	730,100
citationCiteseer	publication citations	268,495	2,313,294
amazon0601	product co-purchases	403,394	4,886,816
2d-2e20.sym	grid	1,048,576	4,190,208
in-2004	web links	1,382,908	27,182,946
as-skitter	Internet topology	1,696,415	22,190,596
rmat22.sym	RMAT	4194304	65,660,814
r4-2e23.sym	random	8,388,608	67,108,846
USA-road-d.USA	road map	23,947,347	57,708,624

tools on more traditional codes, including some[4] benchmarks from the Rodinia [21] suite, as well as some codes from NVIDIA's cuda-samples repository [22]. These suites contain a number of CUDA codes representing common GPU code patterns (see Figure 6).

SRAD [23] (Speckle Reducing Anisotropic Diffusion) is a diffusion algorithm for ultrasonic and radar imaging applications. Backprop is a machine learning back-propagation algorithm used to train neural-network layers. This algorithm consists of forward and backward phases updating a network of user-defined depth. The Gaussian program is a simple Gaussian Elimination algorithm for 2D matrices. Black Scholes is an algorithm for determining the optimal values of a particular financial market model. ConvolutionFFT2d calculates large 2-dimensional convolutions using fast Fourier transform. FastWalshTransform computes the Walsh-Hadamard transform for arbitrary length vectors.

None of these programs are known to have data races, and none of the tools tested reported any races. We could not get iGUARD to run on SRAD on our test system. Also, only the Backprop, SRAD, and FastWalshTransform programs used thread-block shared memory, so Compute Sanitizer's performance is not entirely representative.

Lastly, using our default shadow value footprint of 8 bytes per tracked memory address, we performed all of our tests with less than half of the memory overhead of iGUARD. In a real application, we would be able to reduce this footprint further by configuring HiRace's settings to more efficiently track the parallel patterns of a given test application, for example by removing memory allocated for clocks in those programs that do not utilize synchronization.

C. Discussions

Much of the literature on data race detection surrounds methods of handling traditional CPU parallel programming patterns (such as lock-based programming). As we described in Section II, these patterns are *not* a component of current GPU programming models and, therefore, not a direct target of HiRace's design. This approach is in line with other state-of-the-art tools such as Compute Sanitizer for GPUs (no support for inference) and Google's ThreadSanitizer (supports locks through explicit lock APIs but requires user annotation for many synchronization patterns). However, although there is no explicit API provided, CUDA does provide atomic and memory fence instructions through which developers may create other forms of thread communication. Prior work focused on these forms of fine-grained thread synchronization has found very few benchmarks utilizing these patterns [24], [25].

Nonetheless, some existing work has focused on these patterns. In particular, iGUARD supports the automatic inference of some lock patterns and uses a lock table to prevent reporting false positives in some codes in which shared memory is protected by correctly implemented locks. However, inferring a correct lock implementation is an undecidable problem as implementations may be arbitrarily complex [26] [27]. iGUARD's lock inference identifies a specific pattern of memory fences and `atomicCAS` instructions to infer a lock. Even if this pattern is observed, it is not necessarily the case that it implements a lock (for example, if the atomically updated values are not compared by accessing threads), so the inference may be incorrect and yield missed races. It also does not account for the many other patterns that may be used to correctly implement locks, thus still resulting in false positives.

Modern NVIDIA GPUs also support cooperative groups and dynamic parallelism. These patterns are conceptually the same as those already encapsulated by HiRace's FSM. To be more specific, HiRace defines thread hierarchies abstractly within its FSM, so the names "block" and "warp" could equally represent AMD wavefronts, CUDA cooperative thread groups, and dynamic sub-grids. As outlined in Section III, HiRace's FSM is straightforward to extend to additional hierarchical layers and corresponding synchronization constructs.

[4] The Rodinia codes are old, and most of them contain features that are no longer supported by recent versions of the nvcc CUDA compiler.

One of the biggest weaknesses of other dynamic race detection tools is the need for arbitrary length access histories. Maintaining a full access history is infeasible, so these tools must balance the history length against memory overheads.

Listing 5 demonstrates a simple pattern in which 2 distinct threads read the same location in succession. Though some CPU race detection tools can afford longer histories, which may capture both reads, iGUARD in particular relies on a single prior reader and a single writer (due to the higher cost of GPU thread metadata). Because of this, iGUARD fails to identify the data race in Listing 5.

```
1  __global__
2  void multiRead_Race(int* data, int len)
3  {
4    int tid = threadIdx.x;
5    int local_sum = 0;
6
7    if (tid < len − 2) {
8      local_sum = data[tid]; // evicted
9      local_sum += data[tid + 1];
10
11     data[tid+1]=local_sum; // race!
12   }
13 }
```

Listing 5: Finite-history litmus test

For example, a read from $data[1]$ by thread 1 on Line 8 is lost if thread 0 later reads the same location on Line 9. Thus, when thread 0 writes to $data[1]$ on Line 10, although the access is concurrent to the read by thread 1 on Line 5, this history has been lost (in this particular thread schedule) and the race goes undetected. Note that this example can be adapted to thwart detection on any race checker that only supports a bounded access history. With HiRace's FSM design, any number of reads and any scheduling may be represented entirely within the Read, BRead, GRead, and Write states and must eventually reach the Race state.

It should be noted, however, that HiRace's method relies on maintaining clocks to determine happens-before relations when updating the FSM state (in a similar manner to other prior works we have discussed). Although a single FSM state may represent an arbitrary history of memory interactions, if any of the scoped clocks exceeds the allocated size, then future accesses cannot be compared and HiRace's analysis will be disabled. In practice, HiRace statically allocates a user-defined amount of memory for each shadow value, which determines the number of bits available for the clocks. If the number of barriers witnessed by any thread exceeds the maximum, the user can increase the amount of memory per shadow value.

Additionally, masked warp synchronization primitives may be used to implement non-hierarchical barrier points, which are not amenable to representation within HiRace's FSM. In these cases, HiRace devolves to the traditional method of maintaining multiple access records to approximate full warp-scope access history. Specifically, HiRace tracks a bounded history of sub-warp synchronizations (essentially a warp-level vector clock) and opportunistically prunes the history.

Overall, the constant-space analysis guarantee only holds for programs with bounded occurrences of barriers and that do not excessively utilize non-hierarchical fine-grained synchronization.

All dynamic race checkers suffer from incompleteness due to schedule generation, that is, if the program contains data-dependent control flow, one must drive the code with sufficiently different test inputs to cover these paths. Another approach to offer more completeness is based on *predictive race checking* [28], where one checks for races in one thread interleaving and predicts whether other interleavings contain races. These works largely apply to *mutex-based* synchronization patterns and do not apply to GPU codes adhering to the basic GPU programming model discussed here, though they may apply to codes that implement CUDA global barriers [29], which is of interest to us as future work. In regards to accuracy, there are many takeaways from Table I:

- We evaluated these tools on $580 \times 6 = 3480$ tests with 2076 injected races. HiRace caught all but 215 of the races—a "*completeness rate*" of 93%.
- All races missed by HiRace happen in executions with the smallest test graph used (5 nodes and 5 edges), where the control-flow does not reach the data race. As discussed in section I, Indigo's irregular code patterns are particularly well suited for testing this weakness of dynamic analysis tools.
- The "Sequential Comparison" results indicate the success rate of comparing GPU kernel results against a sequential implementation of the same algorithm. The fact that this naive testing method yields better results than state-of-the-art detection tools is distressing but further supports the value of HiRace's methodology.
- The reasons why other tools miss races in specific cases are unknown to us. Part of the reason is likely due to the eviction of access records as discussed above.
- In general, a compiler is *not required to compile a high-level program containing data races in any predictable manner* [1] (also known as "catch-fire semantics"). This raises doubts on whether tools that instrument compiled code are operating on valid programs when asked to check for races at the assembly level.

VI. Additional Related Work Discussions

The idea of maintaining multiple access records for each word in memory originated in FastTrack and is utilized in ThreadSanitizer and its derivatives as well as in iGUARD. FastTrack showed formally that, while writes in a data-race-free program are totally ordered, reads are not. As a result, the algorithm requires an unbounded read history for each word to be complete. In practice, the access history generally contains only 2 or 4 prior accessors in tools that implement this algorithm on the CPU. iGUARD is only able to store a single prior accessor due to the size of each access record on the GPU. In contrast, HiRace summarizes this history by tracking *groups of threads* as a single identifier, along with a small state tag (discussed in Section III). This allows us to achieve a much smaller memory footprint and a more complete analysis since HiRace does not need to evict access records.

The design of IGUARD is based on the same logic presented in FastTrack but optimized for execution on CUDA devices. Unfortunately, this design relies on an unbounded access history, and, due to the scale of CUDA programs, only two prior accessors (one writer and one reader) can be tracked in 16 bytes of metadata per monitored word of memory.

One of the earliest efforts addressing runtime race-checking in the context of fork/join parallelism is by Mellor-Crummey [30] who introduced the idea of Offset-Span Labeling. This idea is employed by recent OpenMP race-checkers [31], [32] that naturally exploit Offset-Span Labels in the context of barrier-separated regions of concurrency. While Offset-Span-Label-based tools maintain state from a thread-centric point of view, HiRace's approach can be regarded as one that records a summary of the thread access history per location, that is, it maintains state from a location-centric point of view.

The Faial [8] static race checker holds significant promise in terms of arbitrary scalability—an inherent property of static checkers, in general. Recently, an extension called FaialAA [33] that offers a sound and partially-complete static analysis approach has been presented. While these tools offer an exciting and complementary direction to dynamic analysis, their ability to handle irregular codes is as yet unknown to us. Unless specific input-dependent encodings are incorporated into a static analysis tool, false race reports are likely. However, combining static checkers such as Faial and dynamic tools such as HiRace may help obtain the best features of both approaches (e.g., as done in [34]), and forms a promising avenue for future work.

VII. Concluding Remarks

We contribute a novel dynamic GPU data-race detection methodology, implemented in a tool called HiRace. Our methodology is based on representing thread access history with a constant-space state machine designed to fit the hierarchical bulk-synchronous GPU programming model. This state representation reduces per-thread memory overhead, as well as per-access computation since state transitions may be computed with a single table lookup.

We tested HiRace on more than 3500 programs, most of which were designed to measure race-finding efficiency, demonstrating significant improvements in bugs found, runtime overhead, and memory overhead compared to state-of-the-art solutions. The underlying state machine was also developed in tandem with a corresponding model tested with the Murphi model checker.

There are a number of exciting directions for future work. First, adding static analysis support can make HiRace more complete and reduce runtime overhead. Second, if sub-warp synchronization sees increased usage in the future, support in HiRace would be a good direction to pursue. Third, and most exciting of all, since our methodology is based on the GPU programming model generally, it may be readily applied to AMD GPUs for which there is currently no race checker.

References

[1] H.-J. Boehm and S. V. Adve, "You don't know jack about shared variables or memory models," *Commun. ACM*, vol. 55, no. 2, p. 48–54, feb 2012. [Online]. Available: https://doi.org/10.1145/2076450.2076465

[2] S. Atzeni, G. Gopalakrishnan, Z. Rakamarić, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "Archer: Effectively spotting data races in large OpenMP applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 53–62.

[3] Nvidia, "cuda-memcheck," Website: https://docs.nvidia.com/cuda/cuda-memcheck/index.html, 2019.

[4] A. K. Kamath and A. Basu, "Iguard: In-gpu advanced race detection," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 49–65. [Online]. Available: https://doi.org/10.1145/3477132.3483545

[5] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "Nvbit: A dynamic binary instrumentation framework for nvidia gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 372–383. [Online]. Available: https://doi.org/10.1145/3352460.3358307

[6] G. Li and G. Gopalakrishnan, "Scalable smt-based verification of gpu kernel functions," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 187–196. [Online]. Available: https://doi.org/10.1145/1882291.1882320

[7] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: A verifier for gpu kernels." Association for Computing Machinery, 2012.

[8] T. Cogumbreiro, J. Lange, D. Liew Zhen Rong, and H. Zicarelli, "Memory access protocols: Certified data-race freedom for gpu kernels," *FMSD*, 2023, 1 of 7 invited publications from CAV21. [Online]. Available: assets/faial-fmsd23.pdf

[9] Y. Liu, N. Azami, C. Walters, and M. Burtscher, "The indigo program-verification microbenchmark suite of irregular parallel code patterns," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 24–34.

[10] C. Lattner, "LLVM and Clang – Advancing Compilers and Tools," in *Proceeding of the Free and Open Source Software Developers' European Meeting*, ser. FOSDEM '11, Feb. 2011.

[11] D. L. Dill, "The murphi verification system," in *International Conference on Computer Aided Verification*, 1996.

[12] J. R. Wilcox, C. Flanagan, and S. N. Freund, "Verifiedft: A verified, high-performance precise dynamic race detector," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 354–367. [Online]. Available: https://doi.org/10.1145/3178487.3178514

[13] R. Netzer and B. P. Miller, *Detecting data races in parallel program executions*. University of Wisconsin-Madison, Computer Sciences Department, 1989.

[14] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, 1979.

[15] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *PLDI*, 2009, pp. 121–133.

[16] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Workshop on Binary Instrumentation and Applications (WBIA)*, 2009, pp. 62–71.

[17] "Archer," https://github.com/PRUNERS/archer, accessed: 2021-6-26.

[18] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "GKLEE: concolic verification and test generation for GPUs," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, J. Ramanujam and P. Sadayappan, Eds. ACM, 2012, pp. 215–224.

[19] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[20] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on GPUs,” in *2012 IEEE International Symposium on Workload Characterization*, 2012, pp. 141–151.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[22] NVIDIA Corporation. (2024) CUDA samples. [Online]. Available: https://docs.nvidia.com/cuda/cuda-samples/index.html

[23] Y. Yu and S. T. Acton, “Speckle reducing anisotropic diffusion,” *Trans. Img. Proc.*, vol. 11, no. 11, p. 1260–1270, nov 2002. [Online]. Available: https://doi.org/10.1109/TIP.2002.804276

[24] T. Sorensen, H. Evrard, and A. F. Donaldson, “GPU schedulers: How fair is fair enough?” in *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, ser. LIPIcs, S. Schewe and L. Zhang, Eds., vol. 118. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 23:1–23:17. [Online]. Available: https://doi.org/10.4230/LIPIcs.CONCUR.2018.23

[25] T. Sorensen and A. F. Donaldson, “Exposing errors related to weak memory in gpu applications,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 100–113. [Online]. Available: https://doi.org/10.1145/2908080.2908114

[26] V. Kahlon, F. Ivančić, and A. Gupta, “Reasoning about threads communicating via locks,” in *Proceedings of the 17th International Conference on Computer Aided Verification*, ser. CAV’05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 505–518. [Online]. Available: https://doi.org/10.1007/11513988_49

[27] A. J. Bernstein, “Analysis of programs for parallel processing,” *IEEE Transactions on Electronic Computers*, vol. EC-15, no. 5, pp. 757–763, 1966.

[28] J. Roemer, K. Genç, and M. D. Bond, “High-coverage, unbounded sound predictive race detection,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 374–389. [Online]. Available: https://doi.org/10.1145/3192366.3192385

[29] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamaric, “Portable inter-workgroup barrier synchronisation for gpus,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, 2016, pp. 39–58. [Online]. Available: https://doi.org/10.1145/2983990.2984032

[30] J. Mellor-Crummey, “On-the-fly detection of data races for programs with nested fork-join parallelism,” in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing ’91. New York, NY, USA: Association for Computing Machinery, 1991, p. 24–33. [Online]. Available: https://doi.org/10.1145/125826.125861

[31] Y. Gu and J. Mellor-Crummey, “Dynamic data race detection for openmp programs,” in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 767–778.

[32] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, I. Laguna, G. L. Lee, and D. H. Ahn, “Sword: A bounded memory-overhead detector of openmp data races in production runs,” in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 845–854.

[33] D. Liew, T. Cogumbreiro, and J. Lange, “Sound and partially-complete static analysis of data-races in gpu programs,” in *Proceedings of the 2024 ACM SIGPLAN Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH ’24), Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’24)*, 2024.

[34] D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, “Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis,” *SIGPLAN Not.*, vol. 53, no. 2, p. 348–362, mar 2018. [Online]. Available: https://doi.org/10.1145/3296957.3177153

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 HiRace finds more data races than competing tools.

C_2 HiRace has lower runtime overhead than competing tools, on average.

B. Computational Artifacts

A_1 https://zenodo.org/doi/10.5281/zenodo.13334326

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2	Table 1 Figures 5-6

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This artifact executes all included tools on all included test codes and inputs, measuring races detected by each tool (C_1) and runtimes for each tool (C_2).

Expected Results

For every input, HiRace should find more (and thus miss fewer) data races than both iGUARD and Compute Sanitizer's Racecheck when tested against all Indigo CUDA benchmarks included.

Expected Reproduction Time (in Minutes)

The overall expected reproduction time on an NVIDIA 2080 should be approximately 2500 minutes.

Artifact setup should take 10 minutes. Execution should take 2480 minutes. Analysis should take 10 minutes.

Artifact Setup (incl. Inputs)

Hardware: The artifact should be run on an NVIDIA 2070 GPU. Other GPU's may work and should produce similar results, but we have not tested other hardware and iGUARD in particular may not function on other GPU's.

Software: The CUDA toolkit is required, and includes Compute Sanitizer's Racecheck tool; our tests were run on version 11.7, but newer versions may also work. different versions may be obtained at https://developer.nvidia.com/cuda-toolkit-archive. The artifact may be obtained at https://github.com/JohnJacobsonIII/hirace. This includes scripts to download iGUARD (from https://github.com/csl-iisc/iGUARD-SOSP21) and Indigo (from https://github.com/burtscher/IndigoSuite/).

Datasets / Inputs: The test datasets used in original experiments are included in the HiRace repository, but additional inputs (including these) may be generated from the Indigo input generators.

Installation and Deployment: All tests can be executed from the HiRace root directory with the `make` command, or different components may be executed independently as outlined in the README through additional `make` targets.

Artifact Execution

The artifact requires a directory containing CUDA source files, as well as a directory of input graph files.

The artifact selects a single source file, compiles it, then selects a single input graph, and tests the selected source file with the selected input with each tool iteratively (HiRace, iGUARd, and Compute Sanitizer).

Each tool outputs a log file which is parsed by the artifact, and the results are written to a sqlite3 database for the analysis phase.

This process repeats for all test source files and file inputs.

Artifact Analysis (incl. Outputs)

Executing `hirace_analysis.py` will parse the sqlite3 database generated during artifact execution, and generate LaTeX for table 1, as well as both figures 5 and 6.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

The main libraries needed are a CUDA installation and Python 3. We've included snapshots of iGUARD and Indigo to save time and setup headaches, though some larger test files will be downloaded in the test scripts.

Our experiments were tested on CUDA 11.7, though other versions may work. You can test your CUDA configuration with `cuda_test.sh` to ensure necessary commands and scripts will work correctly.

In Python, you'll need packages `matplotlib`, `seaborn`, `pandas`, and `plotly`.

Artifact Execution

The evaluation has been divided into two work flows within the Makefile in the root directory. They can be performed with `make <target>` for each target described below.

- `make test_all`: loops through each test program, compiles the program, then executes it 4 times for each tested input. For each program input a baseline execution is performed, followed by an execution for each compared tool (HiRace, iGUARD, and ComputeSanitizer). Each execution is timed, and its output is piped to a file which is parsed for error reports from the respective test tools. The time and error reports for each execution are then recorded in a sqlite database. Afterwards, the next input is tested on the same test program.
  This step comprises the entire testing process and thus virtually all of the test time. All of the code for performing this process is contained within the `all_tests.py` file (except for some sqlite utilities in `util.py`.
  After this step you should have a database `results.sqlite3` containing the results of all tests within separate tables.
- `make generate_figures`: queries the results database and generates figures to be compared to the papers results. `hirace_analysis.py` will execute queries on the result database and generate figures in the `evaluation_results` directory which can be directly compared to paper figures.

Artifact Analysis (incl. Outputs)

The evaluation should result in a set of files in the `evaluation_results` directory which are directly comparable to those found in the `paper_figures` directory.

- Table 1 should show proportionally similar quantities of found (and missed) races when compared in each dimension to the table in the paper. These values represent data races detected by each tool, and we expect these results to be nearly bitwise reproducible.
- Figure 5 should show similar statistical distribution (as shown by the box plots) to those in the original. These values represent run-times which will be variable, but the figure should support median speedups at least around 5-10x.
- Figure 6 should show the same relationship between tools as those displayed in the original. Meaning, for each test we expect the tools to be ordered the same from fastest to slowest, and by approximately similar magnitudes. These are again determined by run times and will be variable.