BARRACUDA: Binary-level Analysis of Runtime RAces in CUDA programs

Ariel Eizenberg[1], Yuanfeng Peng[1], Toma Pigli[1], William Mansky[2] *, Joseph Devietti[1]

[1]University of Pennsylvania, USA [2]Princeton University, USA

{arieleiz, yuanfeng, tpigli}@cis.upenn.edu, wmansky@cs.princeton.edu, devietti@cis.upenn.edu

Abstract

GPU programming models enable and encourage massively parallel programming with over a million threads, requiring extreme parallelism to achieve good performance. Massive parallelism brings significant correctness challenges by increasing the possibility for bugs as the number of thread interleavings balloons. Conventional dynamic safety analyses struggle to run at this scale.

We present BARRACUDA, a concurrency bug detector for GPU programs written in Nvidia's CUDA language. BARRACUDA handles a wider range of parallelism constructs than previous work, including branch operations, low-level atomics and memory fences, which allows BARRACUDA to detect new classes of concurrency bugs. BARRACUDA operates at the binary level for increased compatibility with existing code, leveraging a new binary instrumentation framework that is extensible to other dynamic analyses. BARRACUDA incorporates a number of novel optimizations that are crucial for scaling concurrency bug detection to over a million threads.

CCS Concepts **• Software and its engineering → Software maintenance tools**

Keywords data race detection, GPUs, CUDA

1. Introduction

In recent years, graphics processing units (GPUs) have thoroughly permeated consumer processor designs. It is now essentially impossible to find a smartphone, tablet or laptop without a substantial integrated GPU on the processor die. Utilizing these omnipresent GPUs, however, remains a challenge. Writing correct parallel code, a notoriously difficult task, is exacerbated by the high degrees of parallelism that GPUs demand to attain high performance. GPU programming models have also grown more expressive over time to support increasingly general-purpose GPU (GPGPU) programming. This extra expressiveness, unfortunately, allows many kinds of subtle concurrency bugs to arise, several of which are new and particular to GPGPU programming. Such bugs can introduce complicated consistency model semantics [1, 44] – and even undefined behavior – into programs, making debugging difficult.

Our system, BARRACUDA, seeks to provide precise, efficient race detection for the programming idioms used by real-world programs written in CUDA, Nvidia's GPGPU programming language. We focus on races among the threads of a single GPU kernel, which requires us to handle low-level synchronization mechanisms like atomics and memory fences. Previous GPGPU concurrency bug detection work has eschewed handling these low-level constructs because they necessitate tracking precise synchronization relationships between individual threads. While conventional CPU programming models can handle such fine-grained synchronization, they struggle with the massive scale of GPU code. In happens-before race detection, for example, each thread in the program has a vector clock, with the vector size equal to the number of threads (n^2 storage for n threads). GPU programs can easily reach hundreds of thousands of threads, requiring hundreds of gigabytes of storage for these vector clocks alone, leaving aside other race detection metadata. To scale to real-world GPU programs, BARRACUDA provides new lossless compression techniques for vector clocks that leverage the structure of the GPU thread hierarchy.

To further improve compatibility, we implement BARRACUDA in a new binary instrumentation framework. Our framework operates on PTX code, a virtual assembly language for Nvidia GPUs. By operating on PTX code, our framework can be run directly on existing binaries without recompilation. Furthermore, we naturally handle inline PTX

* This work was done while William Mansky was a post-doc at the University of Pennsylvania.

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062342

assembly code, which appears in several of our benchmarks. This paper makes the following contributions:

- We identify a new class of GPU concurrency bugs called *branch ordering races*.
- We have constructed a concurrency test suite of 66 simple CUDA programs. We use this suite to validate the correctness of both BARRACUDA and CUDA-Racecheck, a race detector from Nvidia.
- We present the BARRACUDA dynamic data race detection algorithm, which handles low-level synchronization idioms like atomics and fences, and uses lossless metadata compression to scale to GPU programs with over a million threads.
- We present a proof that the BARRACUDA algorithm correctly tracks our notion of what it means for a CUDA program to be well-synchronized.
- We have implemented BARRACUDA in a new binary instrumentation framework for CUDA programs that operates at the PTX level. Running a program with BARRACUDA incurs runtime overheads comparable to those of Nvidia's CUDA-racecheck race detector. Our binary instrumentation framework can serve as a foundation for other CUDA dynamic analyses as well.

2. CUDA Programming Model

To keep this paper self-contained, we provide a brief primer on the CUDA programming model. A CUDA program that runs on a GPU is called a *kernel*. CUDA uses the *single instruction multiple thread* (SIMT) programming model wherein a programmer specifies the code for a single thread, and at runtime an entire collection of threads are created with each thread executing the kernel's code. The collection of runtime threads is known as a *grid* and has a hierarchical structure. At the highest level of the grid are *thread blocks*, which are further subdivided into *warps*, with each warp consisting of up to 32 threads. Thread blocks, and the threads within them, can be organized into a 1-, 2- or 3-D structure, to simplify mapping each thread to the work it needs to do. For simplicity we discuss only 1-D layouts though BARRACUDA handles 2- and 3-D layouts as well.

CUDA programs are written in a variant of C/C++ and compiled to a high-level assembly language called PTX (Parallel Thread eXecution). All PTX instructions are SIMD instructions executed by an entire warp of threads. Scalar or sub-warp execution can be achieved via branches. If a branch condition does not evaluate the same way for every thread in a warp, *branch divergence* arises. Branch divergence is handled via a SIMT stack ([24],Section 3.3.1) that tracks the active threads within a warp.

CUDA presents a hierarchically-structured set of memory spaces that mirror the thread hierarchy. *Local memory* is private to each thread, *shared memory* is shared by all threads within a thread block (but not accessible across thread blocks) and *global memory* is a single memory space accessible by all threads.

3. BARRACUDA Semantics

In this section we first describe how we convert dynamic PTX instructions into abstract trace operations that are easier to reason about. Next we provide our definition of what it means for a CUDA program to be well-synchronized, and show how to detect races on a trace. We then prove that our algorithm tracks synchronization precisely.

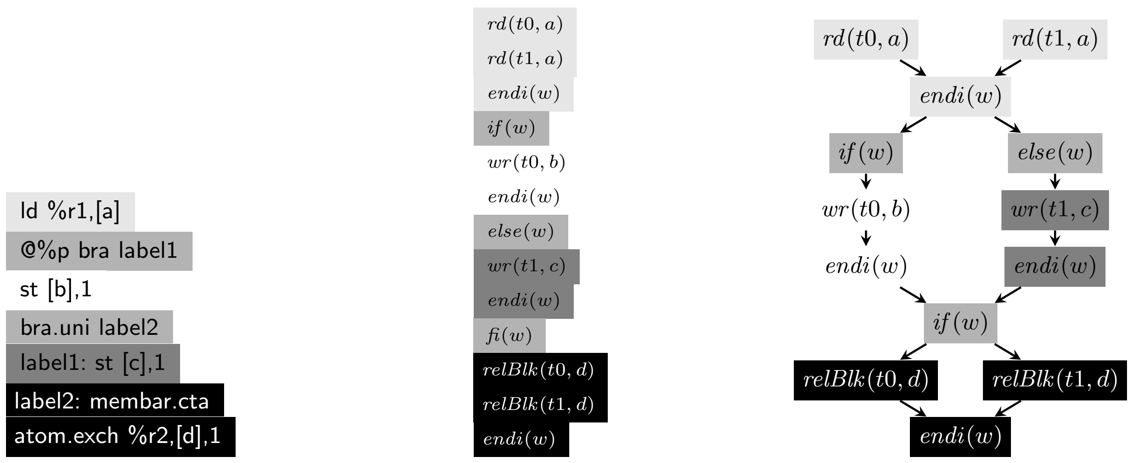
3.1 Modeling a CUDA Execution as a Trace

A program execution is modeled as a *trace*: a sequence of operations performed by a set of threads. Trace operations are an abstraction over the stream of dynamic PTX instructions to facilitate race detection. Our trace operations are:

- $rd(t, x)$ or $wr(t, x)$, in which a thread t reads or writes a location x
- $endi(w)$, used to model a warp w's lockstep execution
- $\mathit{if}(w)$, in which warp w begins executing a branch
- $\mathit{else}(w)$, in which w executes the else path of a branch
- $\mathit{fi}(w)$, in which w concludes its execution of a branch
- $bar(b)$, a barrier for all threads in thread block b
- $atm(t, x)$, in which t performs an atomic read-modify-write operation on a location x
- $\mathit{acqBlk}(t, x)$, $\mathit{relBlk}(t, x)$ or $\mathit{arBlk}(t, x)$, in which t acquires, releases (or both) a synchronization location x with a block-level memory fence
- $\mathit{acqGlb}(t, x)$, $\mathit{relGlb}(t, x)$ and $\mathit{arGlb}(t, x)$ behave like the block-level versions but with a global fence

Our trace includes operations like acquires and releases, similar to high-level language consistency models like the C++ memory model [7]. Inferring trace operations involves heuristics (so that our traces are fundamentally approximations of the synchronization that actually occurred in an execution) and is complicated by the lack of an official CUDA memory consistency model to define illegal behavior. We describe below a useful set of rules for translating PTX instructions into trace operations.

While PTX instructions represent warp-level operations, *e.g.*, an entire warp performing a vector read from memory, for simplicity we model memory operations (reads, writes, atomics, acquires and releases) as *thread-level* operations so that we can consider an access to one memory location at a time. However, previous work has taken into account the fact that warps execute in a "lockstep" fashion [47] wherein all operations from warp instruction i complete before instruction $i + 1$ begins. Lockstep warp execution is indirectly acknowledged in the official CUDA documentation, stating


```
ld %r1,[a]
@%p bra label1
st [b],1
bra.uni label2
label1: st [c],1
label2: membar.cta
atom.exch %r2,[d],1
```

$$\begin{tikzcd}
rd(t0,a) \arrow[dr] & & rd(t1,a) \arrow[dl] \\
& endi(w) \arrow[dl] \arrow[dr] & \\
if(w) \arrow[d] & & else(w) \arrow[d] \\
wr(t0,b) \arrow[d] & & wr(t1,c) \arrow[d] \\
endi(w) \arrow[dr] & & endi(w) \arrow[dl] \\
& if(w) \arrow[dl] \arrow[dr] & \\
relBlk(t0,d) \arrow[dr] & & relBlk(t1,d) \arrow[dl] \\
& endi(w) &
\end{tikzcd}$$

(a) PTX instructions **(b) trace operations** **(c) synchronization order**

Figure 1: (a) Sample PTX instructions for a warp w with 2 threads, $t0$ and $t1$. (b) Shading shows the translation from PTX instructions into trace operations. (c) Arrows indicate synchronization order.

that cores perform scheduling at warp granularity, a warp executes only one common instruction at a time, and warps are issued in program order [37, §4]. The treatment of diverging control flow within a warp also gives evidence that warps execute in lockstep (Section 3.3.1). However, the actual size of a warp can change across architectures, so portable CUDA code should eschew assumptions about warp size, or validate these assumptions at runtime. BARRACUDA's dynamic analysis checks for races based on the warp size of the current architecture, though in future we could simulate the behavior of smaller/larger warps to find additional latent bugs.

We encode the end of warp w's instruction explicitly with an $endi(w)$ operation. A warp w performing a read of location x is thus translated into $rd(t, x)$ for each active thread t in w followed by $endi(w)$ (see the top part of Figure 1a & 1b).

Normally, given a dynamic trace, control flow constructs like loops, function calls, branches and so forth are only implicitly represented. However, we are the first to recognize that branches in GPUs have synchronization implications, so we include explicit branch operations in a trace. Unconditional control flow constructs like loops and function calls do not require such handling and are implicitly unrolled/inlined in the trace.

Control-flow operations are modeled at warp level as they manipulate the warp-level stack that tracks branches, and would be awkward to model at the level of individual threads. if, $else$ and fi operations are readily inferred from static PTX code by examining the targets of branch instructions (see Figure 1). All branches are encoded using if, $else$ and fi for simplicity: simpler constructs like an if statement (without an else) can be encoded via an empty else path.

We infer synchronization trace operations as follows. $bar(b)$ represents a block-wide barrier for thread block b, when every thread in b having executed the bar.sync PTX instruction (__syncthreads in CUDA). Atomic instructions (atom.* in PTX, or atomic* functions in CUDA) from a thread t to location x not immediately preceded or followed by a memory fence in static code become standalone $atm(t, x)$ operations (see Section 3.3.2 for more details).

If a store instruction is immediately preceded by a memory fence (membar.cta or membar.gl in PTX, __threadfence_block or __threadfence in CUDA[1]) in static code, the store plus the fence are bundled together into a release operation, with the scope (block or global) determined by the kind of fence used (see the $relBlk$ operation in Figure 1). Acquire operations arise similarly, from a load followed by a fence. An atomic instruction sandwiched between fences acts as both an acquire and a release (like $arBlk$). To identify errors in CUDA lock implementations, we treat the atom.cas and atom.exch PTX instructions specially. atom.cas performs a compare-and-swap, commonly used for obtaining a lock, and atom.exch performs a fetch-and-set, commonly used to free a lock. If atom.cas is followed by a fence, we treat them as an acquire. If atom.exch is preceded by a fence we treat them as a release.

Our inferences of acquire and release operations from PTX code are necessarily approximate, as other CUDA code may compile into something that looks to us like an acquire or release operation. If we infer an acquire/release where none existed in the original code, BARRACUDA may consider an execution safe when it actually contains a race. Interestingly, even the CUDA C/C++ API defines synchronization operations in terms of fences and loads/stores/atomics, instead of with high-level acquires and releases. Thus, some inference is necessary whether performing race detection at the PTX or the CUDA C/C++ level. We tuned our in-

[1] System-level fences are treated as global fences, as we focus on intra-kernel races.

ference of acquire/release operations based on litmus tests, documentation, and sophisticated code examples like threadFenceReduction from the CUDA SDK, and find that our policy avoids any incorrect atomic inference for our benchmarks.

We consider only *feasible* traces: those where (1) a warp-level memory instruction from warp w is represented in the trace as a consecutive sequence of memory operations, one for each active thread in w, (2) each of w's memory instructions is followed by an $endi(w)$ operation, and (3) branches are translated appropriately into $if(w)$, $else(w)$ and $fi(w)$ operations.

3.2 Synchronization Order

While operations appear in a total order in a trace, that order does not imply that the effects of operations can be linearized. Instead, we derive a partial order called *synchronization order* from a trace α, written $<_\alpha$, such that $a <_\alpha b$ when a must occur before b. Synchronization order is the transitive closure of the smallest relation such that $a <_\alpha b$ when a occurs before b in α and either:

- a and b are both performed by thread t (intra-thread program order); or
- one of a or b is $endi(w)$, and the other is by a thread that's both in w and active at the time the $endi$ is performed (intra-warp program order); or
- one of a and b is $bar(k)$, and the other is by a thread in k (barrier synchronization); or
- a and b are operations on the same synchronization location x where a is a release operation and b is an acquire operation, and both operations are either 1) at block scope within the same thread block or 2) at least one operation is at global scope (inter-thread synchronization)

Given this definition, a *data race* occurs when two operations a and b both access the same location, at least one of them is a write, they are not both atm operations (atomic operations do not race with each other, but also do not imply synchronization), and neither $a <_\alpha b$ nor $b <_\alpha a$ holds (so that the operations are seen as *concurrent* in the trace).

3.3 The BARRACUDA Algorithm

BARRACUDA checks for races by maintaining a tuple of metadata based on *vector clocks*. A vector clock V records a timestamp for each thread t in a system, written as $V(t)$. The standard comparison ($\sqsubseteq$), join ($\sqcup$) and increment (inc_t) operations on vector clocks are defined as:

$$
\begin{array}{rcl}
V \sqsubseteq V' & \text{iff} & \forall t.\, V(t) \leq V'(t) \\
V \sqcup V' & = & \lambda t.\, \max(\, V(t), V'(t)\,) \\
inc_t(V) & = & \lambda u.\ \text{if } u = t \text{ then } V(u)+1 \text{ else } V(u)
\end{array}
$$

The minimal vector clock has a 0 timestamp for each thread and is written $\bot_V$.

Just as with the FASTTRACK race detector [20], to save space *epochs* are sometimes used in place of vector clocks; an epoch $c@t$ is a reduced vector clock that holds a timestamp for just one thread, and is treated as a vector clock that is c for t and 0 for every thread other than t. Because epochs have a single non-zero entry, an epoch can be compared with a vector clock, or another epoch, in O(1) time using the $\preceq$ operator. We say $c@t \preceq V$ when $c \leq V(t)$. $\bot_e$ denotes a minimal epoch $0@t0$.

The BARRACUDA analysis state is a tuple (K, C, S, R, W). K_w is the per-warp stack for warp w that tracks branch divergence. Each stack entry is an *active mask* ($amask$ for short) which is the set of threads that are currently active.

C_t is a vector clock for the thread t: $C_u(t)$ records the last time at which the thread t synchronized with u. S_x represents the synchronization location x, which is an ordinary memory location since CUDA programs often use the same location to store data and for coordination. S_x is a map from thread block $\mapsto$ vector clock, recording the most recent logical time at which some thread from each thread block synchronized with x. R_x is the read metadata for a location x recording the most recent reads of x, which can be encoded either as an epoch or as a vector clock. W_x is a tuple of (write epoch, atomic-bit) for a location x, recording the time of the most recent write to x. The atomic-bit records whether the most recent write to x arose from an atomic operation or not. Epoch comparison $\preceq$ with write metadata W_x ignores the atomic bit. We write $E(t)$ for the epoch $C(t)@t$, the current epoch for thread t.

Our initial analysis state σ_0 is the tuple ($\lambda w.[initActive]$, $\lambda t.inc_t(\bot_V), \lambda x, b.\bot_V, \lambda x.\bot_e, \lambda x.(\bot_e, false)$). Each warp's initial active mask takes account of the number of threads requested for the grid, as the last warp of each thread block may be only partially full. Each thread initially has an empty vector clock with its own entry incremented, all synchronization locations have empty vector clocks for all blocks, and all memory locations have empty read epochs and empty write epochs without any previous atomic operations.

3.3.1 Basic Operations

Figure 2 gives the operational semantics for BARRACUDA for non-synchronization memory accesses and branches. For thread-level memory accesses, if a thread t is not active (due to a branch), t's operation is a NOP – no analysis state is updated. To avoid clutter, each rule implicitly checks that the current thread is active.

Read and write operations are handled essentially as with FASTTRACK. Totally-ordered reads can use a compact epoch representation (rule READEXCL), while concurrent reads require a vector clock (rule READSHARED). The first concurrent read, which necessitates a transition from an epoch to a vector clock, is handled by the READINFLATE rule. The WRITEEXCL rule handles totally-ordered writes, and WRITESHARED the first write after concurrent reads. Because the ENDINSN rule increments per-thread logical time after every instruction, the "same epoch" rules of FASTTRACK are not needed.

$$\frac{\begin{array}{l} R_x \in \mathit{VectorClock} \qquad W_x \preceq C_t \\ R' = R_x[t := C_t(t)] \end{array}}{(K, C, S, R, W) \Rightarrow^{rd(t,x)} (K, C, S, R', W)}\ \textsc{ReadShared}$$

$$\frac{\begin{array}{l} R_x \in \mathit{Epoch} \\ R_x \preceq C_t \qquad W_x \preceq C_t \\ R' = R[x := E(t)] \end{array}}{(K, C, S, R, W) \Rightarrow^{rd(t,x)} (K, C, S, R', W)}\ \textsc{ReadExcl}$$

$$\frac{\begin{array}{l} R_x \in \mathit{Epoch} \\ W_x \preceq C_t \qquad R_x = \mathit{clock}@t' \\ vc = \bot_v[t := C_t(t), t' := \mathit{clock}] \\ R' = R[x := vc] \end{array}}{(K, C, S, R, W) \Rightarrow^{rd(t,x)} (K, C, S, R', W)}\ \textsc{ReadInflate}$$

$$\frac{\begin{array}{l} R_x \in \mathit{Epoch} \\ W_x \preceq C_t \qquad R_x \preceq C_t \\ R' = R[x := \bot_e] \\ W' = W[x := (E(t), \mathsf{false})] \end{array}}{(K, C, S, R, W) \Rightarrow^{wr(t,x)} (K, C, S, R', W')}\ \textsc{WriteExcl}$$

$$\frac{\begin{array}{l} R_x \in \mathit{VectorClock} \\ W_x \preceq C_t \qquad R_x \sqsubseteq C_t \\ R' = R[x := \bot_e] \\ W' = W[x := (E(t), \mathsf{false})] \end{array}}{(K, C, S, R, W) \Rightarrow^{wr(t,x)} (K, C, S, R', W')}\ \textsc{WriteShared}$$

$$\frac{\begin{array}{l} \mathit{amask} = K_w.\mathsf{peek}() \\ vc = \bigsqcup_{t \in \mathit{amask}} C_t \\ \forall t \in \mathit{amask}\ .\ C'_t = \mathsf{incr}_t(vc) \\ \forall t \notin \mathit{amask}\ .\ C'_t = C_t \end{array}}{(K, C, S, R, W) \Rightarrow^{endi(w)} (K, C', S, R, W)}\ \textsc{EndInsn}$$

$$\frac{\begin{array}{l} \mathit{amask}_1, \mathit{amask}_2 = \mathsf{splitActive}(K_w.\mathsf{peek}()) \\ \mathit{stack}_1 = K_w.\mathsf{push}(\mathit{amask}_1) \\ \mathit{stack}_2 = \mathit{stack}_1.\mathsf{push}(\mathit{amask}_2) \\ K' = K[w := \mathit{stack}_2] \\ vc = \bigsqcup_{t \in \mathit{amask}_2} C_t \\ \forall t \in \mathit{amask}_2\ .\ C'_t = \mathsf{incr}_t(vc) \\ \forall t \notin \mathit{amask}_2\ .\ C'_t = C_t \end{array}}{(K, C, S, R, W) \Rightarrow^{if(w)} (K', C', S, R, W)}\ \textsc{If}$$

$$\frac{\begin{array}{l} \mathit{stack} = K_w.\mathsf{pop}() \\ K' = K[w := \mathit{stack}] \\ \mathit{amask} = \mathit{stack}.\mathsf{peek}() \\ vc = \bigsqcup_{t \in \mathit{amask}} C_t \\ \forall t \in \mathit{amask}\ .\ C'_t = \mathsf{incr}_t(vc) \\ \forall t \notin \mathit{amask}\ .\ C'_t = C_t \end{array}}{(K, C, S, R, W) \Rightarrow^{else(w), fi(w)} (K', C', S, R, W)}\ \textsc{ElseEndif}$$

Figure 2: Barracuda basic operational semantics

To faithfully model lockstep warp execution, the individual thread memory operations within a warp instruction run concurrently. This allows us to detect **intra-warp races**.[2] With intra-warp races, according to Nvidia, "the number of serialized writes that occur to that location varies depending on the compute capability of the device ... and which thread performs the final write is undefined" [37, §4.1]. Thus, intra-warp races can result in architecture-specific behavior and nondeterminism. If all active threads within the warp write the same value to a location, we do not consider this a race as the documentation is clear that the outcome is well-defined. Our implementation detects and filters such "same-value" intra-warp races.

The $endi$ rule is used to join all active threads together after all thread-level memory operations complete, and then to fork the active threads again to support detection of future intra-warp races. Note that $endi$ operates only on active threads within a warp w; inactive threads (*e.g.*, those following a different control flow path) are logically concurrent with the active threads, as we explain next.

Branches on GPUs are handled via a hardware SIMT stack [24]. The top of the stack tracks which threads are active along the current control-flow path, and deeper entries support nested control flow. We explain the operation of the SIMT stack along with our semantics. When an if operation is encountered, the set of currently-active threads is split according to the branch condition into two sets: those threads active on the *then* path and those active on the *else* path. One of these sets may be empty due to the branch condition. The IF rule uses the splitActive function to capture the actual active masks. The *then* and *else* sets are represented as active masks that are pushed onto the stack K_w for the current warp w. The order in which they are pushed is arbitrary, but determines the order in which the paths will execute. In our IF rule, the *else* path is pushed first so the *then* path executes first. While Nvidia states that "the different executions paths have to be serialized" [37, §5.4.2] they do not define the order in which the serialization occurs. These semantics are similar to the way event handlers are treated in event-based concurrency systems like Android and JavaScript [6, 29, 42]. Our semantics treats different paths as concurrent so that we can identify **branch ordering races** between paths, though our modeling is conservative in that we do not exempt commutative paths. Branch ordering races are a new class of bugs not identified in previous work, and represent a subtle way in which a GPU program's correctness can implicitly rely on a given architecture and its SIMT stack implementation. Once the *then* and *else* active masks are determined, the IF rule joins and forks the *then* threads, capturing the fact that they are now concurrent with the *else* threads.

$else$ and fi operations are handled the same in our semantics. First we pop the SIMT stack to discard the *then*/*else* active mask, respectively, and perform a join and fork of the newly-active threads. For an $else$ operation, this models the beginning of the *else* path's execution which is logically concurrent with the *then* path. For a fi operation, this models threads from both the *then* and *else* paths restarting lockstep execution after their branching is complete.

[2] An intra-warp race is always a write-write race, as all active threads within the warp execute the same instruction and reads cannot race.

3.3.2 Barriers and Atomic Operations

Figure 3 presents BARRACUDA's operational semantics for synchronization operations. $bar(b)$ is the simplest operation, representing a barrier for all threads within a thread block b. The BAR rule has an explicit predicate that all threads in b are active, as otherwise the Nvidia documentation states that "the code execution is likely to hang or produce unintended side effects" [37, §B.6]. Executing a bar operation with inactive threads, known as a **barrier divergence bug**, is detected as an error by BARRACUDA.

The INITATOM* rules handle an atomic operation on location x where the preceding write to x was non-atomic. These rules check for ordering with previous reads and the previous non-atomic write, as Nvidia states in the PTX documentation that "atomic operations on shared memory locations do not guarantee atomicity with respect to normal store instructions to the same address." [38, §8.7.12.3]. While the documentation leaves the door open for stronger semantics for atomics on global memory, recent work [1] recommends that programmers avoid making both atomic and non-atomic accesses to the same global memory location, as doing so can exhibit relaxed consistency effects. We adopt a similar approach, and do not consider atomic and non-atomic accesses to synchronize with one another.

The ATOM* rules handle an atomic operation on x when the preceding write to x was another atomic. These rules check for ordering with preceding reads, but elide checks of the previous atomic write. Nvidia states that "Atomic functions do not act as memory fences and do not imply synchronization or ordering constraints for memory operations" [37, §B.12]. We capture this constraint by avoiding checks between atomic operations and also avoiding additions to synchronization order. Thus, atomics alone cannot be used to synchronize between threads.

3.3.3 Memory Fence Litmus Tests

To explore the semantics of inter-thread synchronization, we conducted a series of litmus tests on two Nvidia GPUs: a GRID K520 Kepler GPU, obtained via Amazon AWS, and a GTX Titan X Maxwell GPU on a local machine. We ran variations of the message-passing (**mp**) litmus test from [1], with different combinations of fences in each thread. The variables x and y reside in global memory, the default cache operator is .cg (skipping the incoherent L1 cache), and each test thread runs in a distinct thread block. We utilized [1, 44]'s memory stress and thread randomization strategies to provoke weak consistency behavior.

Our results are presented in Figure 4, which shows that using a membar.cta in each thread allows non-sequentially-consistent (non-SC [30]) behavior to arise on the K520 GPU, though not on the GTX Titan X. Using a membar.gl in either thread resulted in SC behavior across all our tests on both GPUs. Our results are consistent with [1], which ran only tests with the same fence in each thread. Of course, testing

$$\frac{\begin{array}{l}\forall t \in b \,.\, \mathsf{active}(t)\\ vc = \bigsqcup_{t \in b} C_t\\ \forall t \in b \,.\, {C'}_t = \mathsf{incr}_t(vc)\\ \forall t \notin b \,.\, {C'}_t = C_t\end{array}}{(K, C, S, R, W) \Rightarrow^{bar(b)} (K, C', S, R, W)}\ \textsc{Bar}$$

$$\frac{\begin{array}{l}W_x = (-, \mathsf{false}) \qquad R_x \in \mathit{Epoch}\\ W_x \preceq C_t \qquad R_x \preceq C_t\\ R' = R[x := \bot_e]\\ W' = W[x := (E(t), \mathsf{true})]\end{array}}{(K, C, S, R, W) \Rightarrow^{atm(t,x)} (K, C, S, R', W')}\ \textsc{InitAtomExcl}$$

$$\frac{\begin{array}{l}W_x = (-, \mathsf{false}) \qquad R_x \in \mathit{VectorClock}\\ W_x \preceq C_t \qquad R_x \sqsubseteq C_t\\ R' = R[x := \bot_e]\\ W' = W[x := (E(t), \mathsf{true})]\end{array}}{(K, C, S, R, W) \Rightarrow^{atm(t,x)} (K, C, S, R', W')}\ \textsc{InitAtomShrd}$$

$$\frac{\begin{array}{l}W_x = (-, \mathsf{true}) \qquad R_x \in \mathit{Epoch}\\ R_x \preceq C_t\\ R' = R[x := \bot_e]\\ W' = W[x := (E(t), \mathsf{true})]\end{array}}{(K, C, S, R, W) \Rightarrow^{atm(t,x)} (K, C, S, R', W')}\ \textsc{AtomExcl}$$

$$\frac{\begin{array}{l}W_x = (-, \mathsf{true}) \qquad R_x \in \mathit{VectorClock}\\ R_x \sqsubseteq C_t\\ R' = R[x := \bot_e]\\ W' = W[x := (E(t), \mathsf{true})]\end{array}}{(K, C, S, R, W) \Rightarrow^{atm(t,x)} (K, C, S, R', W')}\ \textsc{AtomShared}$$

$$\frac{C' = C[t := C_t \sqcup S_x[\mathsf{block}(t)]]}{(K, C, S, R, W) \Rightarrow^{acqBlk(t,x)} (K, C', S, R, W)}\ \textsc{AcqBlock}$$

$$\frac{\begin{array}{c}S' = S_x[\mathsf{block}(t) := C_t]\\ C' = C[t := \mathsf{incr}_t(C_t)]\end{array}}{(K, C, S, R, W) \Rightarrow^{relBlk(t,x)} (K, C', S', R, W)}\ \textsc{RelBlock}$$

$$\frac{\begin{array}{l}C' = C[t := C_t \sqcup S_x[\mathsf{block}(t)]]\\ S' = S_x[\mathsf{block}(t) := {C'}_t]\\ C'' = C'[t := \mathsf{incr}_t({C'}_t)]\end{array}}{(K, C, S, R, W) \Rightarrow^{arBlk(t,x)} (K, C'', S', R, W)}\ \textsc{AcqRelBlk}$$

$$\frac{\begin{array}{l}vc = \bigsqcup_{b \in grid} S_x[b]\\ C' = C[t := C_t \sqcup vc]\end{array}}{(K, C, S, R, W) \Rightarrow^{acqGlb(t,x)} (K, C', S, R, W)}\ \textsc{AcqGlobal}$$

$$\frac{\begin{array}{l}\forall b \in grid \,.\, {S'}_x[b] = C_t\\ C' = C[t := \mathsf{incr}_t(C_t)]\end{array}}{(K, C, S, R, W) \Rightarrow^{relGlb(t,x)} (K, C', S', R, W)}\ \textsc{RelGlobal}$$

$$\frac{\begin{array}{l}vc = \bigsqcup_{b \in grid} S_x[b]\\ C' = C[t := C_t \sqcup vc]\\ \forall b \in grid \,.\, {S'}_x[b] = {C'}_t\\ C'' = C'[t := \mathsf{incr}_t({C'}_t)]\end{array}}{(K, C, S, R, W) \Rightarrow^{arGlb(t,x)} (K, C'', S', R, W)}\ \textsc{AcqRelGlb}$$

Figure 3: Semantics for synchronization operations

```
      init: x = y = 0          final: r1=1 ∧ r2=0
  1.1   st.global.cg [x],1     2.1   ld.global.cg r1,[y]
  1.2   fence1                 2.2   fence2
  1.3   st.global.cg [y],1     2.3   ld.global.cg r2,[x]
```

observations per 1 million runs

fence1	*fence2*	K520	GTX Titan X
membar.cta	membar.cta	7,253	0
membar.cta	membar.gl	0	0
membar.gl	membar.cta	0	0
membar.gl	membar.gl	0	0

Figure 4: Memory fence litmus tests

cannot prove the absence of weak behavior, but our results demonstrate that membar.cta is insufficient to implement synchronization between thread blocks.

3.3.4 Inter-thread Synchronization

To realize inter-thread synchronization, release and acquire operations must be used. These rules update the S_x metadata, a map from thread blocks to vector clocks for a location x, which is used to propagate synchronization order. The AcqBlock rule is similar to a lock acquire in a CPU program in that the current thread t joins its vector clock C_t with the vector clock for the synchronization location, but scoped to the particular thread block in which t resides. The RelBlock rule accordingly updates S_x only for the current block. A *relBlk* in block b_1 followed by an *acqBlk* in block b_2 thus does *not* contribute to synchronization order, as non-SC behavior is possible in this case (Section 3.3.3).

Our litmus tests show that a global fence in just one message-passing thread results in SC behavior. The AcqGlobal rule thus joins the vector clocks for all blocks in S_x, while RelGlobal sets the vector clocks for all blocks in the grid. This ensures that a global release/acquire in one block can synchronize with an acquire/release in any other block, even if the latter operation is at block scope.

3.4 Correctness

We can show that the rules of Barracuda precisely model the synchronization order relation, as indicated by the following theorem.

Theorem 1 (Correctness)**.** *For any feasible trace α, there is some σ' such that $\sigma_0 \Rightarrow^{\alpha} \sigma'$ iff there are no races in α.*

As in FastTrack, the key invariant is that a thread's timestamp for itself is greater than any other component's timestamp for it. In terms of the Barracuda analysis state, this means that for each thread t, $C_t(t) > C_u(t)$ for any $u \neq t$, $R_x(t) \leq C_t(t)$, $W_x(t) \leq C_t(t)$, and $S_x(t) \leq C_t(t)$. The proof of correctness consists in showing that 1) this invariant is maintained and 2) as long as it is maintained, the $\sqsubseteq$ relation on vector clocks precisely captures the happens-before relation.

The new synchronization operations that appear in our system—$endi$, bar, if, $else$, and *fi*—are all "barrier"-style operations, taking a set of threads and synchronizing with all operations performed by all active threads in the set. Thus, instead of referring to the thread id of an operation, we write $tids(a)$ to refer to the set of threads involved in operation a (if a is not a barrier operation, then $tids(a)$ is a singleton set). We write $a.\alpha$ for the trace beginning with the single operation a followed by the sequence α, and likewise $a.\alpha.b$ for the trace that starts with operation a, followed by the sequence α, followed by the operation b. We write C^a_t for the vector clock of t before the operation a is performed, and C'^a_t for the vector clock of t after a is performed. Our first lemma is needed to show that a trace in which no races is detected is in fact race-free:

Lemma 1 (Clocks Imply Synchronization)**.** *Suppose that $\sigma_a \Rightarrow^{a.\alpha} \sigma_b \Rightarrow^{b} \sigma'_b$. For any $t \in tids(a)$ and $u \in tid(b)$, if $C^a_t(t) \leq C^b_u(t)$ then $a <_{a.\alpha.b} b$.*

Proof. By induction on the length of α. If $t = u$, then $a <_{a.\alpha.b} b$ by intra-thread program order. Otherwise, we know that $C^a_t(t) > C^a_u(t)$, so there must be some operation e in $a.\alpha$ that increases the value of $C_u(t)$; let e be the last such operation. This operation must have been either a bar, $acqBlk$ or $acqGlb$ operation. For bar, there is some u' such that $C^a_t(t) \leq C^e_{u'}(t)$, and u and u' are both in the synchronizing set for the barrier; thus we know that $e < b$ and by the inductive hypothesis $a < e$, allowing us to conclude that $a < b$. If e is an acquire operation, it must have been performed by u (so $e < b$), and there must be some earlier operation d by a thread u' such that $d < e$ and $C^a_t(t) \leq C^d_{u'}(t)$, so by the inductive hypothesis we have $a < d$ and so $a < b$. □

Our second lemma is needed to show that if a trace is race-free, then no race will be detected. As in FastTrack, we use the abbreviation K^a to refer to C'^a when a is an acquire operation, and C^a otherwise.

Lemma 2 (Synchronization Implies Clocks)**.** *Suppose that $\sigma \Rightarrow^{\alpha} \sigma'$ and $a \in \alpha$. For any $t \in tids(a)$ and $u \in tids(b)$, if $a <_{\alpha.b} b$ then $K^a(t) \sqsubseteq K^b(u)$.*

Proof. By induction on the derivation of $a <_{\alpha.b} b$. In particular, when synchronization follows from a barrier e, then $C^e_t \sqsubseteq C'^e_u$ for all t and u in the synchronization set. □

Once we have shown that the vector clock $\sqsubseteq$ relation accurately captures the $<$ relation, the rest of the proof of Theorem 1 follows the same argument as the proof of correctness of FastTrack.

Although we have proven that Barracuda correctly detects races according to our definition of synchronization order, recall from Section 3.1 that synchronization order is derived from an event trace that must be inferred from

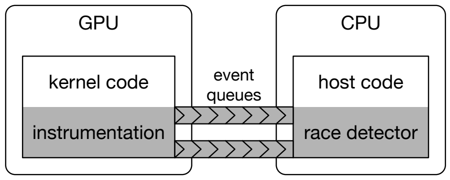


Figure 5: System overview: shading indicates the components of Barracuda.

an actual execution. Because this inference is approximate, BARRACUDA may not detect all real races in practice.

4. Implementation

The BARRACUDA implementation takes advantage of the structure of modern heterogeneous systems by offloading much of the race detection analysis to the host CPU, instead of performing race detection directly on the GPU device which would substantially worsen the performance of the target kernel. There are two benefits to our hybrid GPU+CPU approach. First, the host is typically underutilized during kernel execution, as it waits for the results of the kernel. Second, the host is better suited to the memory-intensive work of race detection as a modern multicore can easily have 1-2 orders of magnitude more DRAM than a modern GPU does. A kernel running under BARRACUDA logs all GPU memory accesses, both global and shared, to queues in GPU memory. These queues are then consumed by host-side threads which do the actual race-checking.

We describe the implementation of BARRACUDA in three steps. First, we describe our dynamic instrumentation framework. Then, we explain the GPU memory access logging mechanism. Finally, we describe the implementation of the host-side race detector.

4.1 Dynamic Instrumentation

BARRACUDA supports all modern versions of CUDA, including 7.5 and 8. We implemented our own binary instrumentation framework because existing frameworks, such as GPU Ocelot [17] and GPU Lynx [18], do not support CUDA SDK versions 5.0 or newer. We also considered the SASSI machine-level instrumentation framework [40] but opted against it because it is closed-source and did not support adequate hooks on synchronization instructions.

BARRACUDA is implemented as a shared library injected into the target process using LD_PRELOAD. It intercepts the __cudaRegisterFatBinary() function call, loads the embedded CUDA fat binary, strips out any architecture-specific binary entries, and extracts and decompresses the architecture-neutral PTX assembly code contained in the fat binary. This PTX code is then fed into the instrumentation engine which performs three operations:

- **Merging the GPU-side logging framework**. The GPU-side logging framework is written in regular CUDA. This code is compiled into PTX at build time and stored inside the BARRACUDA library. At runtime, the logging code is merged with the application's PTX code.
- **Unique thread id calculation**. We add PTX code to the beginning of every kernel to combine the three-dimensional block id and thread id's into a globally unique value. For the rest of the paper we will refer to this 64-bit value as the TID. All device functions are modified to accept this TID as an additional argument so that the TID is always available for logging calls.
- **Memory and synchronization logging**. We scan the PTX source code and add logging calls to all load, store, atomic, fence, and barrier instructions. As described in Section 3.1, we infer high-level *acquire* and *release* operations from the PTX code. Special care is required for predicated instructions: we transform the predicated instruction into a branch and a non-predicated instruction, so that the logging call is covered by the branch. To detect intra-branch races we also add logging calls to all branch convergence points. To reduce logging overhead we avoid some repeated logging of accesses (within the same basic block) to the same memory location, as in some CPU race detection schemes [22] (in particular, we do not log an access to the address in a register if the value of the register has not been changed since the last logged access).

Once the application PTX code is instrumented, the data structures within the CUDA runtime are modified to point to the newly-generated fat binary that includes only the instrumented PTX code. We then relinquish control back to __cudaRegisterFatBinary() and our modified PTX is loaded, JIT-compiled into machine code, and loaded onto the GPU.

The BARRACUDA shared library is also responsible for initializing the GPU-side memory structures used by BARRACUDA. We reserve a configurable percentage (50% by default) of the GPU's global memory for the shared GPU-CPU queues, and invoke a kernel to initialize this region.

Special care has to be given to device resets, as these will free the memory backing BARRACUDA's queues. When BARRACUDA intercepts a cudaDeviceReset call, it delays the reset until the queues are fully drained. It also raises an internal flag so BARRACUDA is reinitialized the next time any CUDA library call is intercepted.

4.2 Device-side Logging

BARRACUDA's GPU logging facility has been designed to be minimal and fast. The core of the GPU-side logging algorithm is a lock-free queue of fixed-size records (Figure 6). The queue contents are tracked via three pointers: a *write head*, a *commit index*, and a *read head*, which track the next entry available for writing by the GPU logging instrumentation, for transferring from the GPU to the host, and for reading by the host race detector, respectively. The queue

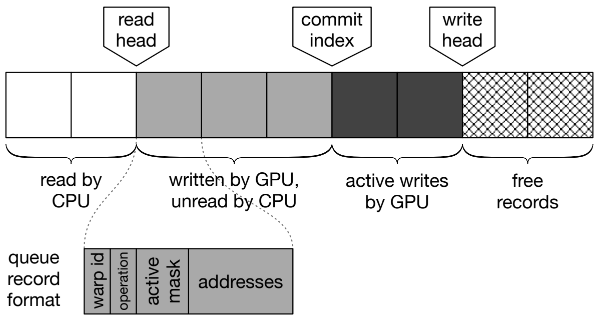


Figure 6: A Barracuda queue, for communicating events from the GPU to the host race detector.

uses a virtual indexing scheme with monotonically increasing indices, which are mapped to physical locations by taking their modulus with the queue size. The queue is considered full when the write head is *queue-size* entries ahead of the read head. Log records are modeled closely on the trace operations given in Section 3.1, except that, for efficiency, a record contains the operation for an entire warp. Each record contains fields identifying the warp, the operation, a 32-bit mask of active threads, and 32 entries for the addresses accessed by each thread in the warp (for memory operations). Records are a fixed $16 + 8 \times 32 = 272$ bytes in size.

To best take advantage of the memory architecture of the GPU we allocate multiple queues, which can achieve orders of magnitude better throughput than using a single queue. Each thread block sends events to a single queue, though multiple thread blocks may use the same queue. We found the optimal organization to be $\approx 1.1 - 1.5$ queues per SM (*i.e.*, GPU core). A significant benefit of mapping each thread block to a single queue is that locking can sometimes be avoided in the host-side race detector. For example, the host race detector uses one CPU thread per queue, so operations on shared memory (which are private to a thread block) will always be processed by the same CPU thread, avoiding the need for locking.

Logging operations on the device are performed cooperatively by all threads within a warp w. To log an operation, the first active thread within w is selected as the leader t_l. t_l reserves an empty slot in the queue, waiting for the CPU to drain queue entries if necessary. Then t_l shares the index of this empty slot with the other threads in w, and all threads record their individual memory addresses in parallel. t_l then fills in the warp ID, operation type, and active mask, and makes the completed record visible to the CPU by bumping the commit index (Figure 6). Logging operations use CUDA's system-level fences to ensure proper memory ordering between the GPU and the CPU.

4.3 Host-side Detector

We implement the BARRACUDA race detection algorithm on the host side. Each GPU queue is allocated a corresponding host thread and GPU stream. Queue draining is the mirror image of the logging algorithm, with the read head used instead of the write head. The detector processes each dequeued event according to the rules given in Section 3.3.

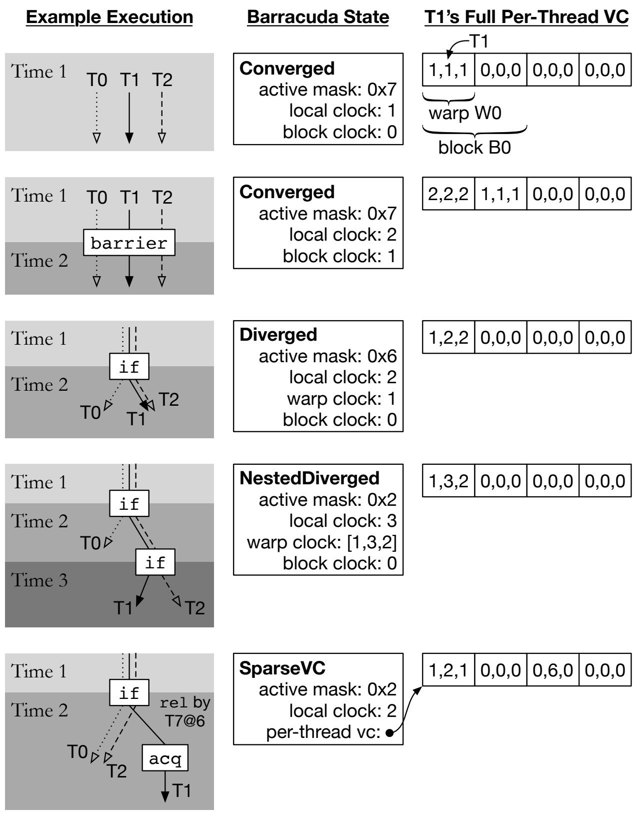


Figure 7: Barracuda's four per-thread VC formats. The example kernel has 3 threads per warp, 2 warps per block, and 2 blocks. Shading of the example execution indicates each thread's logical time. The right side shows the equivalent full per-thread VC that the Barracuda state implicitly represents.

4.3.1 Thread VC Compression

One of BARRACUDA's key innovations is a more efficient mechanism for tracking the per-thread vector clocks (PTVCs) used to record when each thread has synchronized with each other thread in the program (the C_t state from Section 3.3). In a race detector for conventional multithreaded programs, these PTVCs consume O(n^2) space, where n is the number of threads in the program, but n is at most a few tens of threads in practice. With GPU programs, in contrast, kernels can easily utilize *more than one million* threads, which entails crippling space overheads. Fortunately, there is often massive redundancy among the entries in each PTVC. Accordingly, BARRACUDA employs an adaptive scheme for compressing PTVCs, mirroring the GPU thread hierarchy.

In our analysis of CUDA programs, we discovered that roughly 90% of the time PTVCs have the same value for all threads external to a warp and either 1) the same value for all threads in a warp or 2) two distinct values, *e.g.*, along the two branches of an if-else statement. This creates an opportunity for space savings by storing just a few clock values for each warp. BARRACUDA's PTVC compression is lossless, and always functionally equivalent to a full vector clock.

In BARRACUDA, PTVCs are managed at warp granularity because it is often the case that threads in a warp have identical PTVCs. PTVCs can be in one of four formats, as Figure 7 illustrates. The simplest case is the CONVERGED format, used when all threads in a warp are executing in lockstep. Consider the PTVC for thread T1 from warp W0 (top execution of Figure 7). The PTVCs for T0, T1 and T2 are managed collectively at warp granularity. The *active mask* of 0x7 indicates that all 3 threads in W0 are active, and the *local clock* gives the logical time each thread in W0 has for itself. The *block clock* indicates that the threads in W0 have never synchronized with the other threads in B0. All other PTVC entries are implicitly 0, indicating that the threads in W0 have not synchronized with other threads outside their block.

The second execution in Figure 7 shows the impact of a block-level barrier. The block clock for W0 is 1 after the barrier, representing the fact that all threads in W0 synchronized with all other threads in the block at time 1. The threads in W0 then move on after the barrier to time 2.

The third execution illustrates the DIVERGED format, which is used to handle the common case of non-nested control flow. T0 takes one path after the if statement, and threads T1 and T2 the other path. The mask is updated to reflect the active threads (just T1 and T2) along the current path. We introduce a new *warp clock* field to track the last time the active threads in W0 synchronized with the inactive threads in W0, which was at time 1 before the if statement. Synchronization with threads outside the warp is handled via the block clock as in the CONVERGED format.

The fourth execution illustrates the NESTEDDIVERGED format, which is used to handle nested control flow. Here, the warp clock field is generalized to a vector clock to track the precise times at which the active threads synchronized with each other thread in W0. Thus, the warp clock field is a vector clock with the vector size equal to the number of threads in a warp. Due to the nested if statements, T1 is the only currently active thread, and it last synchronized with T0 at time 1 and T2 at time 2. Synchronization with threads outside the warp is handled via the block clock as in the CONVERGED format.

The final execution illustrates the fully-general SPARSEVC PTVC format, which is simply an unordered map from threads to clocks. Using a map instead of a vector clock is more efficient because, typically, the entries for most threads are zero. In this example, T1 is the only thread that acquires a lock l which was previously released by thread T7 at time 6. T7 is in a completely different thread block than T1. This point-to-point synchronization between arbitrary, individual threads requires tracking clock values precisely at thread granularity.

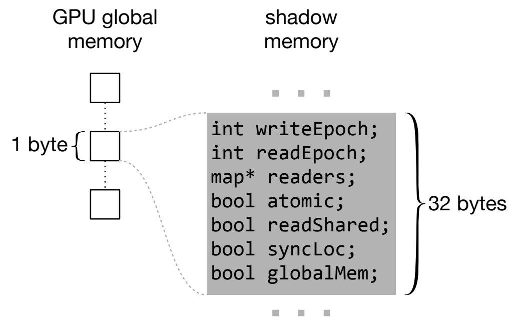


Figure 8: Barracuda shadow memory format. The format is the same for global (shown) and shared memory locations.

BARRACUDA's PTVC management is integrated with a stack that mirrors the GPU's *reconvergence stack* [24], to reduce redundant tracking of the active mask. Each stack entry is 16 bytes and contains the fields listed in the "BARRACUDA State" portion of Figure 7. Whenever a reconvergence operation occurs, we merge the two divergent cases in the stack by joining their PTVCs (the ELSEENDIF rule from Figure 2). BARRACUDA checks for opportunities to use a simpler PTVC format at branches and at reconvergence, as further compression is often possible in these cases.

4.3.2 Barriers

Block-level barriers are the most common CUDA synchronization operation. When a barrier operation occurs, we take a block-wide join of all PTVCs (the BAR rule from Figure 3). We optimize this case by continually tracking the highest clock value for each block, so that at a barrier we can simply broadcast this value to each thread's PTVC.

4.3.3 Shadow Memory

The host race-detector maintains a shadow memory containing per-location race detection metadata (Figure 8). This metadata contains a last-write epoch and a last-read epoch (as in FASTTRACK), and, for locations that have been concurrently read, an unordered map from TIDs to clocks that acts as a sparse vector clock. We do not extensively optimize per-location VCs (unlike per-thread VCs) as the case of shared readers is extremely rare in all the CUDA code we examined. Per-location metadata also contains a spinlock and a set of flags for memory location attributes: whether the location was last accessed by an atomic operation (the *atomic bit* from Section 3.3), has been read concurrently by multiple threads, is used as a synchronization location, and is in global or shared memory. All together, per-location metadata occupies 28 bytes, but 64-bit alignment forces the object to be padded to 32 bytes. Thus, host-side memory usage is 32x that of the GPU, but CPU memory is usually much more abundant than GPU memory. Memory consumption could be substantially decreased if all GPU memory accesses are 2- or 4-byte aligned. Although most of the benchmarks we tested access memory exclusively at 4-byte granularity, BARRACUDA uses 1-byte granularity for generality.

We allocate shadow memory for shared and global memory differently. Shared memory consumption is small (16, 32 or 48KB per thread block in current versions of CUDA) and known at kernel launch, so we preemptively allocate shadow memory for shared memory. A kernel with 512×1024 threads each having 16KB of shared memory requires just $16384 \times 512 \times 32 = 256$MB of CPU memory).

Global memory consumption, on the other hand, is not known at kernel launch as allocations can occur concurrently with kernel execution. Thus, for tracking accesses to global memory, we allocate shadow memory on-demand in response to a kernel's actual global memory accesses. We maintain shadow memory for the GPU's global memory using a page table where each page holds shadow memory to track 1MB of the GPU's global memory. When a global memory location is accessed by the GPU we check if it belongs to an allocated page. If not, we lock the root of the page table and allocate a new page of shadow memory.

When loads, stores and standalone atomic operations are processed, BARRACUDA begins by retrieving the appropriate shadow structures for all the addresses accessed by active threads. The code implements the BARRACUDA algorithm as described in Section 3.3. If a race is detected, the offending TIDs are examined to classify the race as a *divergence race*, an *intra-block race* or *inter-block race*.

A memory location x accessed with acquire and release operations is deemed a synchronization location and tracked specially. GPU code usually has few such synchronization locations, and many programs have none, so storing them in shadow memory would be wasteful. Instead they are stored in their own map (the S_x map from Section 3.3). For each synchronization location x we maintain a collection of VCs, representing the various times at which different thread blocks synchronized on x. Each of these per-block VCs is compressed via the scheme described above in Section 4.3.1. For each synchronization location x, the ACQGLOBAL, RELGLOBAL, ACQBLOCK, RELBLOCK rules are thus implemented via join operations between the PTVCs and the per-block VCs of x.

5. Experimental Setup

Our experimental machine is a dual-socket system with two Xeon E5-2620v4 processors, each with 8 cores running at 2.1GHz, and 128GB of RAM. The machine additionally has an Nvidia GTX Titan X GPU which uses the Maxwell architecture and has 12GB of RAM, and 3072 threads across 24 SMs running at 1GHz. The GPU is connected via PCIe x16. The machine uses Ubuntu 16.04 (Linux 4.4.0), Nvidia CUDA Toolkits 7.5 and 8, and version 367.48 of the Nvidia drivers. All benchmarks were built with Nvidia's nvcc compiler, using the flags -cudart=shared -arch=sm_35 -O.

BARRACUDA is evaluated with the following benchmarks: we use bfs, backprop, dwt2d, gaussian, hotspot, hybridsort, kmeans, lavamd, needle, nn, pathfinder and streamcluster from Rodinia version 3.1 [8]; hashtable from GPU-TM [16, 25]; bfs from SHOC [14]; dxtc and threadFenceReduction from the Nvidia CUDA SDK 7.5 samples; and block_radix_sort, block_reduce, block_scan, device_partition_flagged, device_reduce, device_scan, device_select_flagged, device_select_if, device_select_unique and device_sort_find_non_trivial_runs from Nvidia's CUB SDK 1.4.1 samples. Performance measurements are the average of 10 runs, with a full GPU reset between each run.

6. Evaluation

In this section we evaluate BARRACUDA along two axes: ability to detect races precisely, and performance overhead.

6.1 Concurrency Bug Suite

To evaluate the correctness of BARRACUDA, we constructed a CUDA concurrency bug suite consisting of 66 small CUDA programs that exhibit subtle data races or race-free behavior via global memory, shared memory, within and across warps and blocks, and using a variety of atomic and memory fence instructions to implement locks, whole-grid barriers and flag synchronization. BARRACUDA reports races (or the absence of a race) correctly for all 66 of our tests. Nvidia's CUDA-racecheck [39] reports correctly on only 19 tests, sometimes reporting races where there are none (with intra-warp synchronization), missing races on global memory, and even hanging on the tests involving spinlocks.

6.2 Standard Benchmarks

Table 1 gives more detail about each benchmark used in our evaluation. Column 2 lists the number of static PTX instructions in each program. Column 3 lists the total number of threads used within the largest kernel in each program. Four benchmarks launch more than 1 million threads to run on the GPU simultaneously, and several launch many hundred thousand. Column 4 lists the total global memory used by each benchmark, which is typically small with the exception of DWT2D. There is plenty of space in global memory BARRACUDA to allocate its queues without impinging on the application. Finally, column 5 lists the number of races found by BARRACUDA for each benchmark, and whether the races are in shared memory or global memory. Previous software-based race detectors for CUDA [36, 39, 46, 47] focus on shared memory, and so will not be able to detect the 9 races in global memory that BARRACUDA finds.

Figure 9 shows the fraction of the static instructions in each benchmark that are instrumented by BARRACUDA. Because arithmetic instructions don't require instrumentation

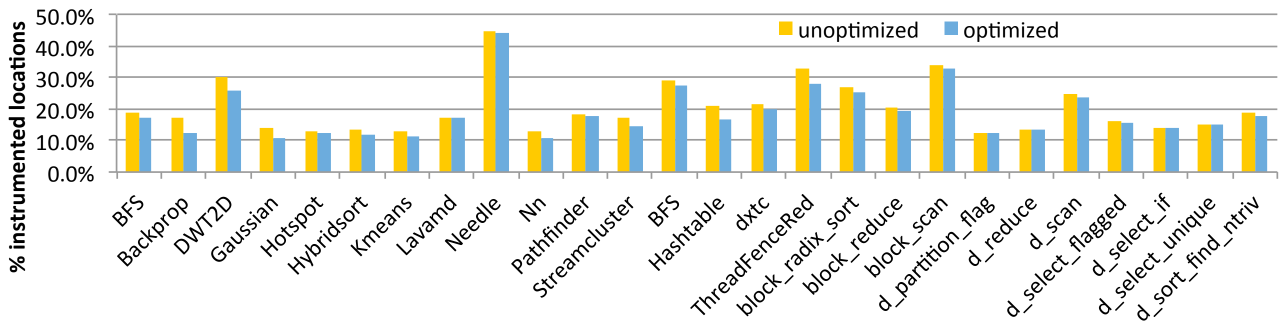


Figure 9: The percentage of static PTX instructions instrumented by Barracuda before (left bars) and after instrumentation pruning (right bars).

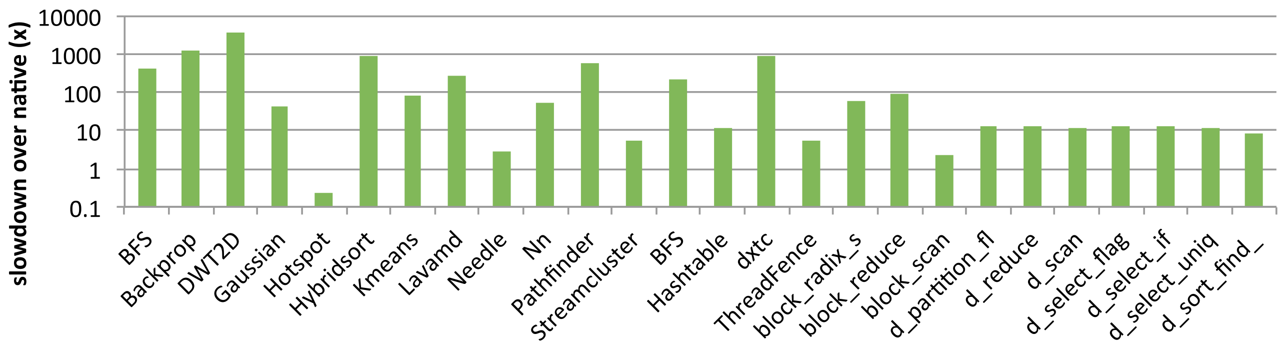


Figure 10: The performance overhead of Barracuda, normalized to native execution. Note the log y-axis.

with BARRACUDA, and they typically comprise the bulk of the instructions in a GPU kernel, BARRACUDA never instruments more than half of the instructions among our benchmarks. The blue bars in Figure 9 show how fewer instructions are instrumented thanks to BARRACUDA's intra-basic-block logging optimizations (Section 4.1).

Figure 10 shows the performance overhead that BARRACUDA incurs, normalized to native execution. BARRACUDA's dynamic binary instrumentation approach, which maximizes compatibility with existing CUDA binaries, can incur significant performance overheads. On DWT2D, BARRACUDA's slowest benchmark, the overhead is 3700x, though the relative overhead for this and many other benchmarks is exacerbated by short running times: DWT2D executes natively in just 90ms. dxtc is BARRACUDA's slowest benchmark in absolute time and completes in 20 minutes, which is too slow for interactive use but more than fast enough for usable debugging.

hotspot with BARRACUDA reliably runs significantly faster than with native execution. We have traced this issue to differing JIT compilation decisions with and without BARRACUDA, but have not yet pinpointed the issue further.

6.3 Bugs Discovered

Here we describe some of the bugs we discovered with BARRACUDA. In the hashtable benchmark, each thread stores a value in a random key in the hashtable. Each hashtable bucket is protected by a fine-grained lock. The program uses an atomicCAS without a fence to synchronize access to each bucket. BARRACUDA detects two bugs: first, since there is no fence for the atomicCAS, it can be reordered with other operations that manipulate the hashtable bucket. Second, releasing the bucket lock occurs via a non-atomic write without a fence. The hashtable data structures reside in global memory, so the bug is not discoverable by tools that focus only on shared memory [36, 39, 46, 47].

Another interesting race comes from the bfs SHOC benchmark [14]. The graph data structure in bfs is stored in global memory. As multiple threads traverse the graph, they track the distance of each node from the starting node. These updates are performed without atomic operations or fences, and the writes can occur concurrently from multiple blocks. A flag is also concurrently set to 1 from multiple threads. While the CUDA documentation states that multiple writes, from threads within a warp, to the same location are serialized [37, §4.1], no such guarantees are stated for writes beyond a warp.

7. Related Work

Several prior schemes have been proposed for detecting data races in GPU programs. Boyer et al. [36] analyze CUDA programs for data races and inefficient memory accesses. Their race analysis is restricted to shared memory only, and

	2	3	4	5
benchmark	static insns	total threads	global mem MB	races found
BFS	281	1,000,448	155	
Backprop	272	1,048,576	9	
DWT2D	35,385	2,304	6,644	3 global
Gaussian	246	1,048,576	124	
Hotspot	338	473,344	119	
Hybridsort	906	32,768	252	1 shared
Kmeans	384	495,616	252	
Lavamd	1,320	128,000	965	
Needle	1,006	495,616	64	
Nn	234	43,008	188	
Pathfinder	285	118,528	155	7 shared
Streamcluster	299	65,536	188	
BFS	770	1,024	68	3 global
Hashtable	193	64	103	3 global
dxtc	1,578	1,048,576	17	120 shared
ThreadFenceRed	5,037	16,384	787	12 shared
block_radix_sort	2,174	128	66	
block_reduce	2,456	1,024	70	
block_scan	4,451	128	118	
d_partition_flagged	2,834	128	66	
d_reduce	2,397	128	66	
d_scan	1,661	128	65	
d_select_flagged	2,615	128	66	
d_select_if	2,508	128	66	
d_select_unique	2,484	128	66	
d_sort_find_non_triv	16,479	128	66	

Table 1: The benchmarks used with Barracuda.

does not take account of atomics or memory fences. GRace [47] proposed a dynamic analysis to find intra-warp races and inter-warp races via shared memory, using static analysis to prune instrumentation when possible. GMrace [46] detects the same kinds of errors as GRace, but with improved running time. Neither GRace nor GMrace detect any inter-block concurrency bugs, nor bugs related to global memory, atomics or memory fences. LDetector [34] can find concurrency bugs via both shared and global memory, but it uses value-based checking to detect writes so it may miss bugs that involve a thread overwriting a location with the location's existing value. LDetector does not handle atomics or memory fences. HAccRG [27] is a hardware-based data race detector for GPUs. It provides coverage of both shared and global memory and also memory fences. To keep hardware overheads low, however, HAccRG does not track all readers for a given location, which can lead to missed races.

The structured data parallel nature of many GPU kernels makes them well-suited to static verification. The GPUverify system [2–5, 10, 11, 13] uses SMT solving to find data races and barrier divergence. GPUverify is sound (it does not miss real bugs) up to the CUDA features it supports, though it occasionally reports false races and does not support memory fences or indirect memory accesses. PUG [32] also uses SMT solving to find races and barrier divergence bugs, though its abstractions can cause it to be both unsound and incomplete in some cases. The GKLEE [33] and KLEE-CL [12] systems use dynamic symbolic execution to find bugs in GPU kernels, and GKLEE has been extended to handle atomic operations [9], but it is difficult to scale symbolic execution beyond small kernels. Leung *et al.* [31] check for data races and determinism of GPU kernels leveraging the insight that most of a kernel's execution is independent of its input parameters, leaving only a portion of the kernel that requires dynamic checking. Their dynamic analysis does not, however, handle kernels with atomics or memory fences. Though completeness remains a challenge for static analysis techniques, leveraging verification machinery to filter dynamic instrumentation could be a powerful and complementary optimization for systems like BARRACUDA.

Several papers have focused on elucidating the memory consistency models of GPU architectures. Work targeting AMD GPUs [26, 28, 45] has culminated in the Heterogeneous System Architecture (HSA) formal memory consistency model [23] adopted by AMD and other GPU manufacturers. In comparison, work on formalizing CUDA's consistency model is still nascent and driven by 3rd party researchers. Recent work has explored the CUDA memory model via litmus tests: [1] presents an axiomatic memory consistency model for Nvidia GPUs, and [44] identifies fuzz testing strategies to expose concurrency bugs on GPUs. Our definition of synchronization order is informed by this prior work. In contrast to work on litmus testing, we have pursued a new safety property for CUDA that can be dynamically checked without the need for fuzzing or program-specific invariants.

There is a vast literature on data race detection for CPU programming models [15, 19, 35, 41, 43]. These prior schemes, FASTTRACK [20, 21] in particular, have inspired the design of BARRACUDA. However, these CPU-based schemes are not directly applicable to CUDA programs both because of the sheer scale of GPU kernels and because of new GPU programming model features, like scoped fences and branch ordering, that necessitate the development of new algorithms.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback on previous versions of this work. This work was supported by NSF grant XPS-1337174 and a hardware donation from Nvidia. Any opinions, findings or conclusions in this material are those of the authors and do not necessarily reflect the views of these sponsors.

References

[1] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2015.

[2] Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. Engineering a Static Verification Tool for GPU Kernels. In *Proceedings of the International Conference on Computer Aided Verification*, CAV, 2014.

[3] Ethel Bardsley and Alastair F. Donaldson. Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels. In *Proceedings of the 6th International Symposium on NASA Formal Methods - Volume 8430*, 2014.

[4] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: A Verifier for GPU Kernels. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA, 2012.

[5] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The Design and Implementation of a Verification Technique for GPU Kernels. *ACM Transactions on Programming Languages and Systems*, 37(3), May 2015.

[6] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable Race Detection for Android Applications. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA, 2015.

[7] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2008.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization*, IISWC, 2009.

[9] Wei-Fan Chiang, Ganesh Gopalakrishnan, Guodong Li, and Zvonimir Rakamarić. *Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-Bounding*. 2013.

[10] Nathan Chong, Alastair F. Donaldson, Paul H.J. Kelly, Jeroen Ketema, and Shaz Qadeer. Barrier Invariants: A Shared State Abstraction for the Analysis of Data-dependent GPU Kernels. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA, 2013.

[11] Nathan Chong, Alastair F. Donaldson, and Jeroen Ketema. A Sound and Complete Abstraction for Reasoning About Parallel Prefix Sums. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL, 2014.

[12] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. Symbolic Testing of OpenCL Code. In *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'11, 2012.

[13] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. Interleaving and Lock-step Semantics for Analysis and Verification of GPU Kernels. In *Proceedings of the European Symposium on Programming Languages and Systems*, ESOP, 2013.

[14] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, 2010.

[15] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, Jun 2007.

[16] W. W. L. Fung et al. KiloTM Benchmarks, 2013. http://www.ece.ubc.ca/ wwlfung/code/kilotm-gpgpu_sim.tgz.

[17] Naila Farooqui, Andrew Kerr, Gregory Diamos, S. Yalamanchili, and K. Schwan. A Framework for Dynamically Instrumenting GPU Compute Applications Within GPU Ocelot. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, 2011.

[18] Naila Farooqui, Andrew Kerr, Greg Eisenhauer, Karsten Schwan, and Sudhakar Yalamanchili. Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, 2012.

[19] Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8), Aug 1991.

[20] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2009.

[21] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. *Communications of the ACM*, 53(11), Nov 2010.

[22] Cormac Flanagan and Stephen N. Freund. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, 2013.

[23] HSA Foundation. HSA Memory Consistency Model. http://www.hsafoundation.com/html/HSA_Library.htm#-SysArch/Topics/03_Memory/_chpStr_HSA_memory_consistency_model.htm.

[24] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2007.

[25] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Hardware Transactional Memory for GPU Architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2011.

[26] Benedict R. Gaster, Derek Hower, and Lee Howes. HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models. *ACM Transactions on Architecture and Code Optimization*, 12(1), Apr 2015.

[27] Anup Holey, Vineeth Mekkat, and Antonia Zhai. HAccRG: Hardware-Accelerated Data Race Detection in GPUs. In *Proceedings of the International Conference on Parallel Processing*, ICPP, 2013.

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

[29] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race Detection for Event-driven Mobile Applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2014.

[30] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), Sep 1979.

[31] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying GPU Kernels by Test Amplification. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2012.

[32] Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based Verification of GPU Kernel Functions. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, 2010.

[33] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. GKLEE: Concolic Verification and Test Generation for GPUs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2012.

[34] Pengcheng Li, Chen Ding, Xiaoyu Hu, and Tolga Soyata. LDetector: A Low Overhead Race Detector For GPU Programs. In *Proceedings of the 5th Workshop on Determinism and Correctness in Parallel Programming (WODET '14)*, 2014.

[35] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, 1989.

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

[37] Nvidia. CUDA C Programming Guide v7.5. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[38] Nvidia. Parallel Thread Execution ISA Version 4.3. http://docs.nvidia.com/cuda/parallel-thread-execution/.

[39] Nvidia. Racecheck Tool. http://docs.nvidia.com/cuda/cuda-memcheck/index.html#racecheck-tool.

[40] Nvidia. SASSI Instrumentation Tool for NVIDIA GPUs, 2016. https://github.com/NVlabs/SASSI.

[41] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2003.

[42] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective Race Detection for Event-driven Programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA, 2013.

[43] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4), Nov 1997.

[44] Tyler Sorensen and Alastair F. Donaldson. Exposing Errors Related to Weak Memory in GPU Applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2016.

[45] John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson. Remote-scope Promotion: Clarified, Rectified, and Verified. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, OOPSLA, 2015.

[46] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. *IEEE Transactions on Parallel and Distributed Systems*, 25(1), 2014.

[47] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. GRace: A Low-overhead Mechanism for Detecting Data Races in GPU Programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2011.