CURD: A Dynamic CUDA Race Detector

Yuanfeng Peng
University of Pennsylvania
USA
yuanfeng@cis.upenn.edu

Vinod Grover
NVIDIA
USA
vgrover@nvidia.com

Joseph Devietti
University of Pennsylvania
USA
devietti@cis.upenn.edu

Abstract

As GPUs have become an integral part of nearly every processor, GPU programming has become increasingly popular. GPU programming requires a combination of extreme levels of parallelism and low-level programming, making it easy for concurrency bugs such as data races to arise. These concurrency bugs can be extremely subtle and difficult to debug due to the massive numbers of threads running concurrently on a modern GPU.

While some tools exist to detect data races in GPU programs, they are often prohibitively slow or focused only on a small class of data races in shared memory. Compared to prior work, our race detector, CURD, can detect data races precisely on both shared and global memory, selects an appropriate race detection algorithm based on the synchronization used in a program, and utilizes efficient compiler instrumentation to reduce performance overheads. Across 53 benchmarks, we find that using CURD incurs an average slowdown of just 2.88x over native execution. CURD is 2.1x faster than Nvidia's CUDA-Racecheck race detector, despite detecting a much broader class of races. CURD finds 35 races across our benchmarks, including bugs in established benchmark suites and in sample programs from Nvidia.

CCS Concepts • Theory of computation → Program analysis; • Software and its engineering → Concurrent programming languages; Runtime environments; Software defect analysis;

Keywords data race detection, GPUs, Nvidia CUDA

ACM Reference Format:

Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: A Dynamic CUDA Race Detector. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3192366.3192368>

1 Introduction

Graphics processing units (GPUs) are a ubiquitous accelerator in modern processors, in consumer devices and some server chips as well. Interest in general-purpose GPU (GPGPU) programming has skyrocketed accordingly, as programmers look to turn these accelerators to many computational tasks outside their original domain of computer graphics. GPGPU programming models have become steadily more expressive over the years, and we focus on Nvidia's CUDA programming model in this work. Modern versions of CUDA (e.g. CUDA 8.5) support GPGPU programs (referred to as *kernels*) that can contain many interesting classes of data races. Compounded with the fact that modern GPU hardware can sustain millions of concurrent threads, the scope for concurrency errors is broad.

Unfortunately, debugging and correctness tools to help programmers detect and repair such bugs have been slower to develop. Existing tools offer their users an unfortunate trade-off between coverage and performance. Many current race detectors tackle only a narrow class of data races: those that can arise in *shared memory*, a pool of memory accessible by only a small subset of the threads in a kernel. This limitation applies to both academic tools [30, 41, 42] as well as commercial tools like Nvidia's CUDA-Racecheck [33]. These tools ignore *global memory*, a larger pool of memory accessible by all threads within a kernel, though we find several kernels that suffer from global memory races. The Barracuda race detector [16] offers much more complete race detection coverage, covering both shared and global memory and even identifying incorrect use of low-level atomics. Barracuda's dynamic binary instrumentation is quite heavyweight, however, and slowdowns can exceed 1000x on some benchmarks.

With this work, we seek a CUDA race detector that provides both precision and acceptable performance. We leverage the open-source Barracuda design as a starting point, but find that its focus on supporting a wide range of synchronization primitives makes it a poor match for common-case GPU code, which generally uses just simple barriers to synchronize. We describe a new race detector, CURD, that is highly

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-5698-5/18/06...\$15.00

<https://doi.org/10.1145/3192366.3192368>

optimized for this common case but retains the ability to fall back to the Barracuda algorithm to preserve precision. We describe a simple static analysis that can decide, before a kernel runs, which race detection algorithm to use. This hybrid approach provides a significant performance improvement for many programs without compromising precision.

CURD is implemented in LLVM and is more than 2.1x faster (geomean) than Nvidia's CUDA-Racecheck and 17.4x faster than Barracuda. CURD's central approach to race detection is to record the read and write sets of synchronization-free regions (SFRs) of code, and then perform set intersections to detect locations for which a data race occurred, *i.e.*, the location was accessed by two distinct threads, at least one of which wrote to the location. While using read/write-set tracking for race detection has been explored before [21, 31, 41, 42], CURD innovates in several key respects. First, CURD utilizes an efficient set representation that is attuned to the typical access patterns of GPU kernels, ensuring that kernels that are well-optimized for the GPU memory hierarchy have very compact read/write set representations with CURD. Second, CURD aggressively exploits the GPU's parallelism in its internal operations. Third, CURD's hybrid organization allows it to avoid "overpaying" for race detection on programs that eschew complicated synchronization.

This paper makes the following contributions:

- CURD employs a novel scheme for compressing read and write sets based on typical GPU access patterns. These compressed representations save memory and latency during race detection.
- CURD outperforms the CUDA-Racecheck race detector from Nvidia by 2.1x on average, while also detecting more races. CURD outperforms the Barracuda detector by 17.4x on average, while offering equivalent precision.
- CURD detects 35 real data race bugs, including bugs in Gunrock [39], Parboil [38], Rodinia [7] and benchmarks from Nvidia. These bugs have been validated manually and, where possible, with other GPU race detectors to confirm their existence.

The remainder of this paper is organized as follows. [Section 2](#) provides a brief overview of the CUDA programming model. [Section 3](#) discusses how the CURD algorithms work, and the performance optimizations we have implemented. [Section 4](#) describes CURD's implementation in the LLVM compiler. [Section 5](#) evaluates CURD in terms of its runtime performance and the data races it found among our benchmark programs. [Section 6](#) describes related work, and [Section 7](#) concludes.

2 GPGPU Programming with CUDA

Nvidia's CUDA programming model has two core abstractions: one for parallelism and one for the GPU hardware's memory hierarchy ([Figure 1](#)). We discuss the parallelism

abstraction first. Parallel threads are expressed in a hierarchical structure known as a *grid*. A grid is composed of *thread blocks*, which are in turn decomposed into individual threads. The arrangement of blocks within the grid, or threads within a block, can be 1-, 2- or 3-dimensional as desired to fit a specific problem domain. Individual threads within a thread block are also grouped into *warps*, which often execute in lockstep on the GPU's SIMD hardware. Many thread blocks may be scheduled concurrently onto one *streaming multiprocessor* (SM), analogous to a CPU core, though a single thread block is never split across SMs.

CUDA's abstraction of memory provides a collection of memory spaces to each kernel. At the top level, *global memory* (also known as device memory) is accessible to all threads within a kernel. *Shared memory* is accessible only by threads within the same thread block. Though each thread block has access to shared memory, threads within thread block b_1 see a different memory space than those within thread block b_2 . The final memory space is *local memory*, which is private to each individual thread. The semantics of these different memory spaces lead to vastly differing implementations and performance characteristics. Global memory is primarily backed by the GPU's off-chip DRAM, though in modern GPUs it can be cached in on-chip L2 caches and, in some cases, L1 caches as well. Shared memory, in contrast, is backed by on-chip L1 caches that are fast but limited to 48KB per thread block in the current version of CUDA. Local memory is backed by a combination of cacheable global memory and the GPU's register file.

CUDA provides a limited set of synchronization constructs for coordination across threads. The core construct is a thread-block-wide barrier called `__syncthreads`, which is fully supported by all current CUDA race detectors. Lower-level synchronization primitives also exist in the form of atomic instructions and memory fence instructions. While many prior CUDA race detectors [26, 30, 33, 41, 42] do not support these constructs, the Barracuda [16] algorithm does.

2.1 Data Races in CUDA

The CUDA programming model dictates what kinds of data races are possible. Threads within a thread block can race with one another via shared or global memory, but threads in different thread blocks can race with one another only via global memory.

Suppose t_1 and t_2 are two different threads in a GPU kernel. A **data race** arises when t_1 and t_2 access the same memory location m without intervening synchronization, and at least one thread in t_1 and t_2 writes to m . Threads within a block can synchronize via CUDA's `__syncthreads` barrier, or via atomics and memory fences. For example, a write to m from t_1 before a barrier is well-ordered with a write after the barrier to m from t_2 , so long as t_1 and t_2 are in the same thread block.

Figure 1. The CUDA thread and memory hierarchy.

Some CUDA race detectors [16, 41, 42] (including Barracuda) account for a common GPU architecture detail which executes threads within a warp in lockstep on SIMD hardware. As a result, these race detectors treat certain intra-warp access patterns as well-synchronized. Consider the following example, where threads t_i and t_j are part of the same warp: a write to location x by t_i is followed by a read of x by t_j in a subsequent instruction. Because warps execute in lockstep, t_i 's write is in a sense ordered before t_j 's read by program order. However, Nvidia's latest Volta GPUs do not always guarantee lockstep execution [32] and thus synchronization-free intra-warp communication is now considered a data race. CURD detects such races, and we update the Barracuda algorithm to flag them as well.

3 The CURD System

When a programmer decides to instrument a program with CURD to perform race detection, a simple static analysis runs to examine which race detection algorithm to use. This analysis has been implemented in LLVM and examines the bytecodes of a GPU kernel, looking for atomic and fence instructions. If one of these instructions is present, CURD uses the Barracuda algorithm [16] to preserve precision in the face of low-level synchronization. Otherwise, CURD uses an alternative algorithm that is optimized for common-case GPU code that synchronizes only via barriers. In the remainder of this paper, we focus on this alternative algorithm, which we refer to as CURD for simplicity.

Figure 2 shows CURD's high-level operation. To detect intra-block races, CURD maintains two sets, $\text{TRS}(t)$ and $\text{TWS}(t)$, for each thread t to track the locations read and written by t , respectively. CURD tracks accesses to each memory space (shared or global) separately, so there are in fact two read sets and two write sets for each thread. For

simplicity, we do not distinguish between these sets in the discussion below except in the special cases (like barriers) where they are treated differently. To detect inter-block races on global memory, for each block b , $\text{BRS}(b)$ and $\text{BWS}(b)$ sets are maintained to save the aggregated sets of reads & writes performed by the block b .

3.1 Access Logging

When a thread t reads/writes a memory location in shared or global memory, CURD records the memory region being accessed in $\text{TRS}(t)/\text{TWS}(t)$. Each memory region is a pair (a, s) , where a is an address within a particular memory address space (either shared or global memory), and s is an unsigned integer. The pair represents a region of memory starting at address a and extending for the next s contiguous bytes. In addition, CURD can track the source code location (file and line) for each region's most recent read/write.

3.1.1 Per-thread Read/Write Sets

Each read or write set is a collection of regions that obey the following invariant: memory regions stored inside these sets are always disjoint and non-adjacent. This invariant is maintained on insertion operations.¹ If a region being inserted into a set overlaps with, or is adjacent to, an existing region, recursive merging occurs to keep the set elements disjoint and non-adjacent. GPU DRAM and caches provides maximum bandwidth when threads access adjacent data. Thus, by constantly compressing adjacent memory regions, the space consumption of CURD's read/write sets is typically quite small for well-optimized CUDA programs.

A per-thread read or write set is implemented as an array of structs, where each struct represents a memory region. Memory regions within the array are not ordered, so a region containing a particular address may lie anywhere within the array. The memory regions in use are densely packed at the beginning of the array, allowing for simple bump allocation of new regions and efficient clearing of the array.

Figure 3 shows an example of adding a load access into an existing read set. The initial read set contains three memory regions. The incoming load is adjacent to the first region, and overlaps with the second. Thus, the first and second regions can be merged into a larger region in the resulting read set.

Inserting a new memory region r into a set is performed as follows. r is checked against each memory region in the set to see whether r can be merged with an existing region in the set. If r is disjoint and non-adjacent to all existing regions, r is appended to the array of regions. If r can be merged with another region r' (i.e. r is adjacent to or overlaps with r'), we merge r with r' and then recursively explore the remaining regions of the set for further merging opportunities that may have opened up. After merging regions, source information

¹We will actually relax this invariant later for efficiency, but it is convenient for expository purposes to uphold it for now.

Figure 2. High-level operation of CURD in response to various kernel events.

Figure 3. CURD's read and write sets are implemented as an array of disjoint, non-adjacent memory regions.

is updated to reflect this most recent access. Note that regions in the already-explored prefix of the set cannot be candidates for further merging. We know that such “prefix regions” were already disjoint and non-adjacent with respect to both r' (because of the set's invariant) and r (because we did not choose to merge r earlier), so prefix regions must also be disjoint and non-adjacent with respect to $r \cup r'$.

Note that when two regions merge, the source information of one of the merging regions is overwritten. If a race is detected on a memory region r , CURD reports the source-level information of the most recent access to r , which may differ from the actual access involved in the race (though we found this problem rare in practice, see Section 4). However, on a race CURD additionally reports the memory addresses on which the race was detected, and these addresses are always precisely those involved in a race.

In CUDA programs, to fully exploit the available memory bandwidth, threads within a warp need to access adjacent memory locations. This motivates a “striped” organization of the per-thread sets as shown in Figure 4. The first elements of each per-thread set, for each thread t within a block b , appear first. They are followed by the second elements for each thread, the third elements for each thread, and so on.

Figure 4. CURD organizes per-thread read/write sets to maximize global memory coalescing by interleaving entries from each thread. In the example shown, there are three threads per block.

Figure 5. The steps involved in intra-block race detection and adding per-thread sets to per-block sets.

3.2 Intra-block Race Detection

When the threads in a block reach a barrier operation, we can perform intra-block race detection using the $TRS(t)$ and

Figure 6. An example of how per-thread sets are compressed in parallel.

TWS(t) for each thread t in a block B . We write $a \cap b$ to denote the intersection of two sets a and b . If two sets have a non-empty intersection, we say that the sets **overlap**. For each thread in B , we logically intersect read and write sets to detect read-write races, and then we intersect write sets with one another to detect write-write races. In a race-free CUDA program, these intersections should all be empty. Once intra-block race detection is finished, each thread t can safely discard its read and write sets for shared memory, as no inter-block races are possible in shared memory and any future shared memory accesses from the block will be well-ordered with t 's current accesses due to the barrier. Global memory TRS(t)/TWS(t) need to be added to BRS(t)/BWS(t), respectively, and then they, too, are cleared.

While intra-block race detection and adding to the block sets are logically two separate steps, their implementation is integrated for efficiency. Figure 5 gives an overview. We first discuss adding to the block sets, then RW intra-block checks, and then WW intra-block checks.

3.2.1 Per-thread Set Compression

Per-thread sets are compressed before being added to the per-block sets to reduce time and space overheads. TRS(t) and TWS(t) are treated identically, so we refer just to TRS(t) in this section.

Compression produces a compressed read set CRS(b) from the read sets for all threads in the block b , i.e. $\text{CRS}(b) = \forall t \in b \cup \text{TRS}(t)$. CRS(b) is a new set, so the original TRS(t) are preserved, which is important later for intra-block race detection. Compression operates directly on the striped TRS(t) layout (Figure 4), and runs in parallel via the following algorithm (Figure 6). First, an *ordering bitmap* is computed, identifying whether each region is located in memory after its left neighbor. Next, a *mergeability bitmap* is computed, identifying whether each region is adjacent to or overlaps with its left neighbor. Neighboring regions are often mergeable because, for well-optimized kernels, adjacent threads (more precisely, threads within the same warp) often access

adjacent memory regions to ensure coalesced global memory accesses. Next, a parallel scan of the mergeability bitmap computes the indices to use in the output set CRS(b). Finally, using the output indices and the ordering bitmap, we identify the beginning and end of each mergeable sequence of input regions, and compress them into a single output region.

Once CRS(b) is computed, it is inserted into BRS(b). This insertion does not exhaustively search BRS(b) for potential merges, instead examining just k regions. We found that bounded search yielded better performance in practice, even though it makes BRS(b) larger. While bounded search also violates the disjoint and non-adjacent property for region sets, it does not impact correctness because set intersections still operate on all regions in a set.

3.2.2 Checking for Read-Write Intra-Block Races

A sufficient, but not necessary, condition for the absence of intra-block read-write races within a block b is $\text{CRS}(b) \cap \text{CWS}(b) = \emptyset$. As the compressed sets are small, this check is much faster than checking for intersections among TRS(t) and TWS(t) for each pair of threads. Moreover, the ordering bitmap from Section 3.2.1 and bit-twiddling instructions quickly reveal whether CRS(b) and CWS(b) are sorted, which then permits linear-time set intersection.

If $\text{CRS}(b) \cap \text{CWS}(b) \neq \emptyset$, it may be because of a true intra-block race or because a thread t both read and wrote some location x . Thus, we must go back and perform intersections among the individual TRS(t) and TWS(t). To make these intersections faster, we first sort any unsorted TRS(t) and TWS(t) sets in-place. With n threads in a block, this requires $O(n^2)$ intersections, each of which can run in linear time. However, the optimistic RW check is effective at reducing the need for this more expensive check.

3.2.3 Checking for Write-Write Intra-Block Races

To check for write-write races in a SPMD fashion, each write set is compared with every other write set. However, half of these intersections are unnecessary as set intersection is commutative. Considering a matrix with each row and column representing a write set, the strict upper triangular matrix represents the necessary intersections. Performing only these intersections improves performance as both memory traffic and intersections are reduced by half. This “triangular check” design also minimizes branch divergence [23] in the code, which saps performance on GPU SIMD hardware. Since the TWS(t) are sorted from read-write checking, individual set intersections can run in linear time.

3.3 Inter-Block Race Detection

For detection of inter-block races, an important design decision is when to do the checks. The CURD design explores two options: a *lazy* scheme (CURD-Lazy) which defers inter-block race detection until the kernel terminates, and an *eager*

Figure 7. CURD combines per-thread sets into per-block sets differently in the lazy (top) versus eager (bottom) schemes. The two threads shown here are part of the same block. Write sets are elided for simplicity but are handled analogously.

scheme (CURD-Eager) which performs inter-block race detection at each barrier. Note that intra-block race detection always occurs eagerly, at each barrier. Both lazy and eager inter-block race detection are fully precise, in the sense that no data race on location x will ever be missed, nor will a false race ever be reported. Both schemes report the precise addresses involved in any data races. The lazy approach has lower implementation complexity and can be faster in some circumstances, but has less-timely detection because races are not reported until the end of the kernel when source information may have been overwritten during previous merges. In contrast, performing inter-block checks eagerly allows races to be caught earlier, potentially giving users more accurate source-level information about the race.

We discuss the lazy scheme first, as it is conceptually simpler than the eager detection scheme. The inter-block detection scheme also affects how the results of per-thread set compression (Section 3.2.1) are used, as we explain below.

3.3.1 CURD-Lazy

In the lazy scheme, inter-block race checks are not conducted until the target kernel terminates. As the kernel executes, at each barrier, a block b unions all of its per-thread sets with each other (Section 3.2.1) and with a single cumulative $BRS(b)$ (or $BWS(b)$) as shown in the top of Figure 7. $BRS(b)$

and $BWS(b)$ are implemented exactly as the per-thread sets are, maintaining disjointedness and non-adjacency among all constituent regions. The location A , which is accessed both before and after a barrier in Figure 7, is represented only once in the final per-block set.

Checks for inter-block races are done in a separate checker kernel launched after the termination of the target kernel. After the checker kernel terminates, global memory used for block read/write sets can be freed. Similar to intra-block race checks, in the lazy scheme, inter-block races are detected in a read-write pass and a write-write pass for each block b in the grid G .

Each inter-block check involves $O(B^2)$ intersection operations of block read/write sets in total. To parallelize these operations, each block is responsible for checking its own read/write set with the write set of all other blocks, involving $O(B)$ intersections. Within each block, the operations are distributed evenly among N threads, with each thread performing $O(B/N)$ intersections.

One benefit of the lazy strategy is a simplified implementation. As all block sets are read-only after the target kernel terminates, no synchronization is necessary inside the inter-block checker kernel.

3.3.2 Eager Scheme

To detect inter-block races before kernel termination, CURD also offers an eager detection strategy. In this scheme, checks for inter-block races are performed at the end of each synchronization-free region.

The simplest version of the eager scheme would intersect all the per-block sets at each barrier, exactly as the lazy scheme does. However, this would entail a significant number of redundant comparisons as the per-block sets change only partially after each barrier. Thus, we adopt a more sophisticated incremental detection scheme that reduces redundancy by handling the updates to a per-block set separately.

A key challenge in the design of eager inter-block detection is coordinating concurrent access to per-block sets across blocks. While all threads in b_0 are at a barrier when b_0 begins inter-block detection, another block b_i may be actively executing and as a result $BRS(b_i)$ and $BWS(b_i)$ may be changing. To avoid the need for complex concurrency control, $BRS(b_0)$ is organized as a list of read subsets $BRS(b_0)[0] \dots BRS(b_0)[n]$ (and $BWS(b_0)$ similarly). Within each read subset, memory regions are disjoint and non-adjacent. However, there are no such guarantees across subsets. At each barrier, b_0 compresses its threads' read sets into a compressed read subset (Section 3.2.1) and appends this subset to $BRS(b_0)$ as shown in the bottom part of Figure 7. The same memory location A is accessed before and after a barrier so A is stored twice, within each subset of the per-block set.

Once a read subset is appended to $BRS(b_0)$, it becomes immutable. Thus, when a block b_i wants to check against $BRS(b_0)$, b_i records the current number of subsets in $BRS(b_0)$

Figure 8. CURD-Eager inter-block race checks involve three set intersections between each distinct pair of blocks. This example illustrates the checks between blocks b_0 and b_i .

and intersects with them. b_0 may concurrently append additional subsets, but these will be checked at b_i 's next barrier (or at the end of the kernel). A per-block spinlock is used to protect the size of each block's $BRS(b)$ and $BWS(b)$ sets, to ensure that remote threads see a consistent view of the set during their checks. Performing intersections with a read subset does not require synchronization, however, as these subsets are immutable.

To avoid redundant intersections, when b_0 reaches a barrier we partition $BRS(b_0)$ in two: a prefix $BRS(b_0)[0 : -1]$ of read subsets from before the current SFR began, and a suffix $BRS(b_0)[-1]$ containing just the read subset from the current SFR.² $BWS(b_0)$ can be split similarly. $BRS(b_i)$ for a remote block b_i can be partitioned at some index k , where $BRS(b_i)[0 : k]$ is a prefix of b_i 's read subsets up to but not including subset k , and $BRS(b_i)[k :]$ is a suffix of the read subsets from index k onward through the end of the list. After performing an inter-block check, a block b_0 remembers the prefix $BRS(b_i)[0 : k]$ of read subsets that have been checked so as to avoid redundant intersections with these subsets in the future. We will explain how k is tracked for each block shortly, but first we illustrate how eager inter-block race checks work assuming k is already known.

When b_0 reaches a barrier, the race checking required between b_0 and some other block b_i is shown in Figure 8. The checks proceed in three steps. **Step 1:** We intersect b_0 's current read subset with all writes from b_i . **Step 2:** We intersect all of b_0 's previous read subsets with the suffix write subsets from b_i , eliding $BRS(b_0)[-1] \cap BWS(b_i)[k :]$ which was performed in Step 1. **Step 3:** We intersect b_0 's current writes with all writes from b_i . Note that $BRS(b_0)[0 : -1] \cap BWS(b_i)[0 : k]$, and $BWS(b_0)[0 : -1] \cap BWS(b_i)[0 : k]$, have been performed by b_0 during some previous barrier. At b_0 's first barrier, $BRS(b_0)[0 : -1]$ and $BWS(b_0)[0 : -1]$ are empty so no intersection is necessary. The intersection $BWS(b_i)[k :] \cap BWS(b_0)[0 : -1]$ will be performed by b_i at its next barrier. We use multiple threads within a block to

accelerate these set intersections, by having each thread read a different element of, say, $BWS(b_i)$ in parallel and compare it with each other element of $BRS(b_0)[-1]$.

To determine the pivot k which divides subsets we have checked already from those we have not yet checked, each block maintains an array `alreadyChecked` with an element for each other block in the grid. These elements contain the appropriate k value for each other block. Because read subsets are immutable once appended to some $BRS(b)$, the length of $BRS(b)$ is non-decreasing over time and thus the k values are as well. `alreadyChecked` can also be thought of as a kind of vector clock, tracking the time at which a block last coordinated with each other block.

3.4 Additional Optimizations

In this section we describe additional optimizations we performed that are less central to the core CURD idea than those described previously.

3.4.1 Caching Thread-Set Insertions

Recall that when the program accesses a memory region r , CURD first searches for any mergeable region in the corresponding thread set before inserting r into the set. Typically, programs exhibit temporal and spatial locality when accessing their data, so the efficiency of the lookups into the thread sets can be improved if each search starts from the most recently added/merged region. Therefore, we optimized the thread-set insertion routine such that 1) when a merging of memory regions occurs, move the merged region to the last position of the thread-set and 2) each search in a thread set starts from the last element of the thread-set and moves forwards and 3) only a bounded number of elements are examined in each search. These modifications make thread-set insertion faster because the newly merged/inserted region is more likely to be reused in subsequent insertions and the insertion procedure can return within a bounded number of steps, even if no mergeable region exists in the thread sets.

Note that since insertions into the thread sets now only examine a bounded number of elements, it is possible for a thread set to have overlapping/adjacent elements. This does not hurt soundness, but might make the thread sets larger than necessary and potentially introduces more work during intra-block checks and compression into block sets. We found in our experiments that the benefit of cached set insertions outweighs the cost of the potential extra work, as the cached thread-set insertions speed up all benchmarks used in our evaluation.

3.4.2 Triangular Checks

This triangular check optimization presented in Section 3.2.3 is implemented not just in intra-block write-write checking, which is common to both CURD-Eager and CURD-Lazy, but also for CURD-Lazy's inter-block write-write race checking. It is also possible to imagine implementing triangular checks

²We adopt Python's array slicing syntax for identifying sub-lists of per-block sets. $[a : b]$ indicates the slice from index a up to, but not including, index b . The special index -1 indicates the last index of a list. Thus, the slice $[0 : -1]$ contains all elements except the last element in the list. A slice may omit the second index, e.g., $[a :]$, in which case the slice begins at index a and continues through the end of the list.

within CURD-Eager's write-write inter-block checks. Consider Figure 8, when b_i performs its write-write inter-block check it may redundantly intersect some part of $BWS(b_i)[k:]$ with $BWS(b_0)[-1]$, even though b_0 has already performed this intersection previously. The payoff for triangular checks is likely to be small, however, as $BWS(b_0)[-1]$ is small. Given the extra complexity and synchronization that would be required to track this redundancy, we have opted not to implement triangular checks for CURD-Eager's inter-block detection.

3.4.3 Metadata Reuse Across Kernel Launches

Many GPU programs launch a large number of kernels, each of which does a relatively small amount of work. The Caffe machine learning framework is an excellent example of this design as it uses frequent kernel launches (and terminations) as a form of global synchronization in lieu of atomics. CURD incurs non-trivial startup costs to allocate and initialize its metadata, which would normally be required before every kernel launch. As an optimization, we allow CURD's metadata (which resides in global memory) to persist across kernel launches, avoiding constant deallocation and reallocation. This change improves the performance of Caffe by 7x.

4 Implementation

Like many other compiler-based race detectors (e.g., LLVM's ThreadSanitizer race detector for CPU programs [37]), CURD consists of two major parts: an instrumentation pass and a detection library. CURD uses LLVM to instrument memory accesses and barriers in kernel code, as well as kernel launches in host code. These locations are instrumented with calls into the CURD runtime library, which is linked together with the application.

Shadow Memory management At present, CURD allocates its shadow memory on the host side, via `cudaMalloc()`. The main reason why the allocation was not done dynamically on the device is that CURD needs a global memory region visible to all blocks, and across kernels. It is simpler to set this up before the kernel launch to ensure that the memory region is communicated to all threads and blocks. An additional reason to avoid device-side allocation is that we need a handle to the shadow memory on the host side, and memory dynamically allocated on the device is not accessible to host code. Besides the shadow memory allocated in global memory, several read-only configuration parameters which CURD reads during online race detection (e.g., the sizes of shared memory per block and total global memory), are copied into constant memory.

It is non-trivial to decide the size of shadow memory to be allocated. To keep things simple, the current version of CURD allocates each data structure (e.g., $BRS(b)/BWS(b)$, $TRS(t)/TWS(t)$) with a fixed size, which can be configured at runtime via environment variables. We will investigate more

sophisticated ways of preallocating the shadow memory in future work. After a kernel terminates, all detected races are reported to the user, after which the shadow memory allocated is freed.

Source-level information To provide useful information about the location of data races, CURD stores the source file name and line number with each memory region in the thread and block sets. Since only one piece of the source information can be kept per region, region merges cause information to be lost about one of the regions involved in the merge. To track whether such information loss about any source locations has occurred, each memory region structure maintains a *complete* bit that indicates whether the stored source level information is complete. When a memory region is initially constructed, this bit is set. When two regions are merged into a single region, the bit of the resulting region is set if and only if 1) the bit is set in both merging regions and 2) the source-level information stored in the two regions is identical. When a race is detected when comparing two regions, CURD reports to the user whether it has information about all the source file locations that may be involved in the race, by inspecting the *complete* bits.

We verified the races found on the benchmarks in our evaluation of CURD (Section 5), and the source-level information for all reported races are complete. This is intuitive as 1) CUDA has a SPMD programming model, where many threads execute the same program statement and 2) mergeable regions are often accessed at the same program statement, as these accesses are often to adjacent global memory regions to permit coalescing.

Reducing impact on occupancy One potential side effect of instrumenting CUDA programs is that the instrumented kernel may have reduced achievable occupancy on the GPU, as calls to functions in CURD may increase the maximum number of registers and amount of shared memory a thread may use. Reduced occupancy can slow the instrumented program. To minimize this effect, functions in the CURD library avoid the use of shared memory when possible, and set a limit for the maximum number of registers when compiling the CURD library.

Pruning instrumentation to local and constant memory In addition to shared and global memory, CUDA programs also frequently use another two address spaces: local and constant memory. We found in the development that accesses to local and constant memory can dominate all accesses on some programs (e.g. heartwall from Rodinia), so we develop a data-flow analysis pass that tries to statically identify local and constant memory pointers, to avoid unnecessarily instrumenting these memory locations.

5 Evaluation

5.1 Experimental Setup

To evaluate how effective and efficient CURD is in terms of detecting global and shared memory races in GPU programs, we test CURD on a wide range of 53 benchmarks, as shown in Table 1. We include programs from the NVIDIA CUDA SDK version 7.5 samples (indicated by an *N* after the benchmark name in Table 1), Rodinia 3.1 (*R*) [7], NVIDIA's CUB SDK 1.4.1 samples (*C*), Parboil 2.5 (*P*) [38], Gunrock 0.4 (*G*) [39], and Caffe, a widely-used machine learning framework, evaluated by running the AlexNet and OverFeat models. Of these benchmarks, only threadFenceReduction uses low-level atomics and fences for synchronization, necessitating the use of the Barracuda algorithm to maintain precision. The remaining 52 benchmarks run with the CURD algorithm described in Section 3.

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 3.0 x16. The machine uses Ubuntu 16.04 (Linux 4.4.0), Nvidia CUDA Toolkits 8, and version 367.48 of the Nvidia drivers. All benchmarks were built with Nvidia's nvcc compiler included in CUDA 8.0, using the flags `-cudart=shared -arch=sm_50 -O3`. During compilation, the NVVM IR generated by nvcc was intercepted when a special function call is done by the libNVVM library, which enables us to invoke our instrumentation pass developed based on LLVM 3.4 (which is compatible with the NVVM IR). After that, the instrumented IR is sent back to the toolchain of nvcc. This way we can utilize the handy support for separate compilation and linking of CUDA files, which is not yet supported by LLVM. Performance results shown are the geomean of 5 runs.

5.2 Performance

Across all workloads, CURD-Eager incurs an average slowdown of 2.89x, whereas CURD-Lazy incurs 2.88x slowdown over native execution. CURD-Eager's overhead ranges from 1.1x (device_scan) to 111x (heartwall), while CURD-Lazy slows the programs by 1.1x (device_scan) to 37.9x (heartwall). heartwall is something of an outlier for CURD, as the next slowest benchmark (leukocyte) incurs just a 23.9x slowdown with CURD-Lazy and 24.8x with CURD-Eager. We investigate heartwall's performance in more detail in Section 5.3. Of particular note is CURD's performance on the Caffe machine learning framework, where CURD incurs an acceptable overhead of 14x for this important class of workload.

Overall, CURD-Lazy and CURD-Eager perform similarly but CURD-Lazy has a noticeable performance edge on several programs. We envision programmers first running with

CURD-Lazy to quickly discover if a program has any races, and then, if necessary, using CURD-Eager to get more precise debugging information.

As the Barracuda system is the most related work to CURD, we evaluate Barracuda and CURD under the same environment. However, Barracuda only succeeds in running 24 programs among all the benchmarks we evaluate; for other benchmarks, it fails either due to runtime error or insufficient memory. Figure 10 shows CURD's speedup over Barracuda. Comparing their geomeans, CURD outperforms Barracuda by 17.4x. CURD sees a maximum speedup of 1435x on DWT2D. On 21 programs, CURD runs faster than Barracuda, with a minimum 1.8x speedup (nw). Barracuda is 6.4x faster than CURD on hotspot, where Barracuda consistently outperforms even native execution due to Barracuda's inadvertent effects on the PTX JIT compiler.

To put CURD's performance in an appropriate context, we also run Nvidia's commercial race detector CUDA-Racecheck (from CUDA SDK 8) and measure its performance on our benchmarks. Figure 11 shows CURD's speedup over CUDA-Racecheck. Since CUDA-Racecheck only checks for races on shared memory, we exclude programs that do not use shared memory, and compare the performance on the 35 benchmarks that use shared memory. For these benchmarks, CURD-Lazy is on average 2.1x faster than CUDA-Racecheck, with a maximum 461x speedup on dxtc. CURD runs slower than CUDA-Racecheck on block_reduce, bc, cc, dwt2d, pr, salsa, sssp. On these benchmarks, CURD typically spends a significant amount of time in tracking races on global memory (see Figure 12), while CUDA-Racecheck ignores global memory entirely. Some of CUDA-Racecheck's overhead is likely due to its use of dynamic binary instrumentation instead of CURD's compiler-based approach. However, CURD currently employs essentially no static analysis and focuses on algorithmic optimizations instead, which are likely quite applicable to the CUDA-Racecheck implementation as well.

5.3 Additional Characterization

In this section, we evaluate some additional aspects of CURD's performance. First, Figure 12 shows the breakdown of CURD's overhead from four sources: 1) instrumentation; 2) shared memory checking; 3) intra-block global memory checking; and 4) inter-block global memory checking. We measure the cost of each component by comparing performance after disabling each component (except instrumentation, which is always required). We excluded programs where CURD incurs negligible overhead, as the breakdown of overhead is difficult to measure using this method on such programs.

The instrumentation overhead measures the cost of instrumenting loads and stores with an empty callback. On heartwall, CURD's slowest benchmark, this is 84% of its total overhead. Shared memory checking adds an additional 4% overhead for heartwall, intra-block global memory checks add 1%, and the remaining 11% results from inter-block global

Figure 9. Slowdown over native execution (lower is better) for CURD-Lazy (light bars) and CURD-Eager (dark bars). Note the log scale for the y-axis.

Figure 10. CURD-Lazy's speedup over Barracuda (higher is better). Note the log scale on the y-axis.

memory checks. After investigating the instrumented bit-code of heartwall, we found that most instrumented accesses are to local or constant memory, which introduces unnecessary overhead. CURD includes a dynamic check to ignore local and constant memory accesses, as these cannot be involved in data races, but the cost of these checks accumulates rapidly. Although the data-flow analysis we mentioned in Section 4 already prunes some of local or constant memory accesses, the address space of some pointers cannot be statically decided. One direction to improve this is to do a more aggressive inter-procedural analysis to decide the address space of more pointers statically. We leave this as a future work.

On 21 programs in all, the main source of CURD's overhead is instrumentation. These programs may also benefit from the address space analysis that would help heartwall. Monitoring shared memory is the most significant source

of overhead on eight programs. On eight other programs, checking intra-block races on global memory contributes the most to the overall overhead, while the cost of checking inter-block races dominates on just three programs (myocyte, spmv and streamcluster).

Table 1 shows detailed characterization data for our benchmarks. Column 2 shows the lines of CUDA code in each benchmark. Column 3 shows the maximum number of threads for each program, which occasionally exceeds 1M and is reliably in the hundreds of thousands. Column 4 shows the maximum number of entries ever found in a block set or thread set accordingly; if only one number is shown, the max block set and thread set sizes are the same. These sets are typically just tens of regions, and never more than a few thousand. Column 5 shows the percentage of all per-thread sets that were naturally sorted, of all the sets encountered across barriers and kernel exits. The vast majority naturally appear in sorted order due to the predominant GPU memory access patterns. Column 6 shows the “high-water mark” amount of shadow memory used by CURD, which is typically small but occasionally reaches into the GBs.

5.4 Data Races Detected

Across our benchmark programs, CURD detects 35 races in total (Column 7 in Table 1). Among them are 24 shared memory races, 5 intra-block global memory races, and 6 inter-block global memory races. The shared memory races have been validated with CUDA-Racecheck. For benchmarks that Barracuda can successfully run, we have also cross-validated the shared memory and global memory races detected by CURD and Barracuda. Benchmarks are run with both CURD-Eager and CURD-Lazy, and the two schemes detect the same set of races. After investigating the races reported by CURD, we group the races into the following categories:

- **Shared memory races without any synchronization:** CURD finds 15 such shared memory races in 13 programs. Among these races, three, including one

Figure 11. CURD-Lazy's speedup over CUDA-Racecheck (higher is better). Note the log scale on the y-axis.

Figure 12. How much different parts of CURD-Lazy contribute to its performance overhead.

in leukocyte and two in hybridsort, are write-write conflicts involve multiple threads writing to the same shared memory location within a barrier-free-region. According to the CUDA language specification, the order of these writes are undefined, so such conflicts form data races. The rest of the 12 races are write-read conflicts between threads in different warps within the same barrier-free-region, such as the 5 races shown in leukocyte. Depending on the actual execution order of the read and write operations, the read may return different values.

- **Warp-level synchronization:** As discussed in [Section 2.1](#), CURD issues warnings for fragile code that relies on intra-warp synchronization. We found 9 such issues in our benchmarks. In future GPU architectures, such code may break if the warp size changes or additional warp scheduling flexibility is introduced [20, 29]. In fact, with the release of CUDA 9.0 and introduction of the Volta architecture, threads within a warp are no longer executed in lock-step unconditionally; as a result, code relying on warp-level synchronization

now needs to be modified in order to behave correctly on the new generations of GPU hardware.

- **Intra-block global memory races:** One such race is detected in heartwall and one in cc, while 3 others are found in bfs (from Rodinia). Among these races, The ones in bfs are particularly interesting in that they involve indirect memory accesses: the index of a write into an array a is obtained by first reading an element from another array b ; unfortunately, two different threads can read the same value x from elements of array b , then concurrently write $a[x]$, forming a race condition. These kinds of indirect memory accesses are particularly challenging to reason about, for both humans and static analysis techniques.
- **Inter-block global memory races:** In total, CURD finds 6 such races in the benchmarks we used, including three in bfs, and one each in myocyte, cc and particlefilter_float. These races were easy to verify as these three programs do not have any inter-block synchronization, so these conflicting global memory accesses are clearly race conditions.

Table 1. Details of each benchmark and its behavior with CURD-Lazy.

program Suite	2	3	4	5	6	7
	CUDA LoC	#max threads	max BS/TS	sorted PTS %	SMem (MB)	races
b+tree R	874	2.6M	33	71	832	
backprop R	337	1M	52	80	375	
bc G	885	98K	117 / 12	50	36	1s
bfs R	349	66K	1 / 11	68	294	6g
bfs G	855	1M	23 / 14	100	427	2s
block_radix_sort C	327	128	2 / 21	99	1	1s
block_reduce C	294	1K	2 / 64	100	2	
block_scan C	348	1K	29 / 34	100	4	1s
caffe	6856	10K	4110 / 121	98	36	
cc G	678	123K	324 / 8	100	30	1s,1g
cutcp P	1280	15K	32 / 241	92	114	2s
dev_part_flagged C	246	128	34 / 10	77	<1	
dev_part_if C	257	128	34 / 10	67	<1	
dev_radix_sort C	226	128	2 / 4	99	<1	
dev_reduce C	193	128	1	100	<1	
dev_scan C	199	128	34 / 12	97	<1	1s
dev_sel_flagged C	246	128	34 / 3	90	<1	
dev_sel_if C	255	128	34 / 3	88	<1	
dev_sel_unique C	234	128	34 / 3	90	<1	
dev_sort_fnt_runs C	384	128	23	95	<1	
dwt2d R	2559	37K	544	100	288	
dxtc N	820	688K	2 / 16	100	528	1s
gaussian R	470	7K	16	100	6	
heartwall R	2171	16K	2605	94	2327	1s,1g
hits G	431	25K	80 / 50	100	38	1s
hotspot R	339	473K	28	99	109	
hotspot3D R	324	66K	81	100	1095	
huffman R	1185	16K	16 / 8	79	4	
hybridsort R	1145	16K	1025	100	1456	2s
kmeans R	518	496K	34	100	423	
lavaMD R	541	128K	126	99	63	
lbm P	1018	2.2M	20	100	1318	
leukocyte R	657	12K	176	100	14	5s
lud R	401	58K	48	98	333	
mri-q P	376	66K	3 / 5	99	10	
myocyte R	5413	64	6 / 8	54	2	1g
nn R	322	43K	1	99	3	
nw R	453	2K	34	99	134	
particlef_float R	870	1K	622	100	8	2g
particlef_naive R	696	1K	6	99	<1	
pathfinder R	236	119K	439	99	358	
pr G	1116	246K	18 / 52	100	390	1s
scalarProd N	271	33K	4 / 32	52	31	1s
sad P	1440	13K	360 / 90	99	35	
salsa G	456	131K	189 / 33	100	132	
sample G	496	33K	2	50	2	
sgemm P	417	1K	1 / 16	100	7232	
sradi R	566	2.1M	113	99	2209	
spmv P	977	12K	3407 / 131	63	48	
sssp G	747	49K	69 / 14	99	21	2s
streamcluster R	1518	66K	2631 / 516	50	1032	
threadFenceRed N	1021	8K	66 / 128	98	32	1s
topk G	449	33K	2 / 49	100	49	

6 Related Work

The Barracuda [16] and CUDA-Racecheck race detectors are the most relevant pieces of related work. Barracuda provides high precision but at high cost, while CUDA-Racecheck is somewhat faster but only detects races in shared memory. We

provide an extensive accuracy and performance comparison with both race detectors in Section 5.

Boyer et al. [30] explored CUDA race detection early on, though their approach runs a GPU kernel on a GPU emulator instead of on real GPU hardware, incurring prohibitive slowdowns. The proposed analysis is also limited to shared memory.

The GRace [41] and GMrace [42] race detectors are, like CURD, based on compiler instrumentation. Combined with static analysis to identify race-free locations, they achieve low performance overheads on a small set of benchmarks. Like CUDA-Racecheck, GRace and GMrace ignore global memory, but have no false positives or false negatives for shared memory.

LDetector [26] is a compiler-based race detector that detects races by using diffs between memory snapshots to determine which locations were written. While this avoids the cost of per-access instrumentation, diffs based on memory contents miss “silent stores” where a thread writes the same value to memory that is already there. LDetector thus cannot detect races involving silent stores.

The GPUVerify system [1, 2, 4, 5, 9, 10, 12] uses SMT solving to find data races and barrier divergence. As a static tool, GPUVerify can detect bugs that dynamic analyses like CURD miss, but it has some assumptions that may not always hold in real programs. For instance, GPUVerify assumes that input pointers to a kernel do not alias. In contrast, CURD does not make any pointer-aliasing assumptions, and thus does not suffer from this precision issue. The GKLEE [25] and KLEE-CL [11] systems use dynamic symbolic execution to find bugs in GPU kernels, and GKLEE has been extended to handle atomic operations [8], but it is difficult to scale symbolic execution beyond small kernels.

Leung et al. [24] propose a static analysis based on information flow to prune race detection and determinism instrumentation. They identify the slice of a kernel that is input-dependent and instrument only that slice. Input-independent parts of a kernel also need to be checked for races once, but not on each additional input. This pruning technique is highly complementary to CURD’s dynamic analysis, and could help reduce its performance overheads further.

Many race detectors have been proposed for CPU programs. A hybrid approach to race detection algorithms, as CURD adopts, would likely be useful for CPU race detection as well. A hybrid detector could choose among algorithms for fork-join programs [3, 13, 18, 22, 35], programs with locks [6, 14, 15, 17, 19, 28, 34, 36, 37, 40] and programs with sophisticated atomics and lock-free synchronization [27].

7 Conclusion

We have presented the CURD system, a new data race detector for CUDA programs. CURD has been implemented in LLVM and applied to a wide range of benchmark kernels.

CURD detects data races in both global and shared memory, unlike many previous race detectors that focus exclusively on shared memory. Simultaneously, CURD provides a substantial performance boost over previous work such as Nvidia's CUDA-Racecheck system. The key to CURD's performance advantages are its compiler-based instrumentation, an efficient representation of read/write sets that is compact in the common case of well-tuned GPU code, and GPU-specific optimizations to maximize memory bandwidth utilization. We believe CURD is robust enough and fast enough to have a positive impact on how CUDA developers debug their code.

Acknowledgement

We thank the anonymous reviewers for their valuable feedback on previous versions of this work. We thank Yang Chen, Sean Lee, and Mahesh Ravishankar from NVIDIA for their help on integrating CURD with NVCC. 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] Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. 2014. Engineering a Static Verification Tool for GPU Kernels. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag New York, Inc., New York, NY, USA, 226–242. https://doi.org/10.1007/978-3-319-08867-9_15
- [2] Ethel Bardsley and Alastair F. Donaldson. 2014. Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels. In *Proceedings of the 6th International Symposium on NASA Formal Methods - Volume 8430*. Springer-Verlag New York, Inc., New York, NY, USA, 230–245. https://doi.org/10.1007/978-3-319-06200-6_18
- [3] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '04)*. ACM, New York, NY, USA, 133–144. <https://doi.org/10.1145/1007912.1007933>
- [4] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: A Verifier for GPU Kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 113–132. <https://doi.org/10.1145/2384616.2384625>
- [5] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. 2015. The Design and Implementation of a Verification Technique for GPU Kernels. *ACM Trans. Program. Lang. Syst.* 37, 3, Article 10 (May 2015), 49 pages. <https://doi.org/10.1145/2743017>
- [6] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10*. Toronto, Ontario, Canada, 255. <https://doi.org/10.1145/1806596.1806626>
- [7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 44–54.
- [8] Wei-Fan Chiang, Ganesh Gopalakrishnan, Guodong Li, and Zvonimir Rakamarić. 2013. *Formal Analysis of GPU Programs with Atomics via Conflict-Directed Delay-Bounding*. Springer Berlin Heidelberg, Berlin, Heidelberg, 213–228. https://doi.org/10.1007/978-3-642-38088-4_15
- [9] Nathan Chong, Alastair F. Donaldson, Paul H.J. Kelly, Jeroen Ketema, and Shaz Qadeer. 2013. Barrier Invariants: A Shared State Abstraction for the Analysis of Data-dependent GPU Kernels. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 605–622. <https://doi.org/10.1145/2509136.2509517>
- [10] Nathan Chong, Alastair F. Donaldson, and Jeroen Ketema. 2014. A Sound and Complete Abstraction for Reasoning About Parallel Prefix Sums. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 397–409. <https://doi.org/10.1145/2535838.2535882>
- [11] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. 2012. Symbolic Testing of OpenCL Code. In *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing (HVC'11)*. Springer-Verlag, Berlin, Heidelberg, 203–218. https://doi.org/10.1007/978-3-642-34188-5_18
- [12] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. 2013. Interleaving and Lock-step Semantics for Analysis and Verification of GPU Kernels. In *Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 270–289. https://doi.org/10.1007/978-3-642-37036-6_16
- [13] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. 2015. Race Detection in Two Dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. ACM, New York, NY, USA, 101–110. <https://doi.org/10.1145/2755573.2755601>
- [14] Anne Dinning and Edith Schonberg. 1991. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging (PADD '91)*. ACM, New York, NY, USA, 85–96. <https://doi.org/10.1145/122759.122767>
- [15] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. 2012. IFrit: interference-free regions for dynamic data-race detection. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '12)*. ACM, New York, NY, USA, 467–484. <https://doi.org/10.1145/2384616.2384650>
- [16] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. BARRACUDA: Binary-level Analysis of Runtime RAces in CUDA Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 126–140. <https://doi.org/10.1145/3062341.3062342>
- [17] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. 245–255. <https://doi.org/10.1145/1273442.1250762>
- [18] Mingdong Feng and Charles E. Leiserson. 1997. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures - SPAA '97*. Newport, Rhode Island, United States, 1–11. <https://doi.org/10.1145/258492.258493>
- [19] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09*. Dublin, Ireland, 121. <https://doi.org/10.1145/1542476.1542490>
- [20] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society,

- Washington, DC, USA, 407–420. <https://doi.org/10.1109/MICRO.2007.12>
- [21] Weixing Ji, Li Lu, and Michael L. Scott. 2013. TARDIS: Task-level Access Race Detection by Intersecting Sets. In *Proceedings of the 4th Workshop on Determinism and Correctness in Parallel Programming (WODET '13)*.
- [22] John Mellor-Crummey. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing - Supercomputing '91*. Albuquerque, New Mexico, United States, 24–33. <https://doi.org/10.1145/125826.125861>
- [23] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanovic. 2014. Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 101–113. <https://doi.org/10.1109/MICRO.2014.48>
- [24] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU Kernels by Test Amplification. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 383–394.
- [25] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic Verification and Test Generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 215–224. <https://doi.org/10.1145/2145816.2145844>
- [26] Pengcheng Li, Chen Ding, Xiaoyu Hu, and Tolga Soyata. 2014. LDetect: A Low Overhead Race Detector For GPU Programs. In *Proceedings of the 5th Workshop on Determinism and Correctness in Parallel Programming (WODET '14)*.
- [27] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic Race Detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 443–457. <https://doi.org/10.1145/3009837.3009857>
- [28] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09*. Dublin, Ireland, 134. <https://doi.org/10.1145/1542476.1542491>
- [29] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 235–246. <https://doi.org/10.1145/1815961.1815992>
- [30] Michael Boyer, Kevin Skadron, and Westley Weimer. 2008. Automated Dynamic Analysis of CUDA Programs. In *Workshop on Software Tools for MultiCore Systems*.
- [31] Abdullah Muzahid, Dario Suñáñez, Shanxiang Qi, and Josep Torrellas. 2009. SigRace: Signature-Based Data Race Detection. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*. ACM, New York, NY, USA, 337–348. <https://doi.org/10.1145/1555754.1555797>
- [32] Nvidia. [n. d.]. Nvidia Tesla V100 GPU Architecture. <http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>
- [33] Nvidia. 2016. Racecheck Tool. <http://docs.nvidia.com/cuda/cuda-memcheck/index.html#racecheck-tool>
- [34] Eli Pozniansky and Assaf Schuster. 2003. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)*. ACM, New York, NY, USA, 179–190. <https://doi.org/10.1145/781498.781529>
- [35] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 531–542. <https://doi.org/10.1145/2254064.2254127>
- [36] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* 15, 4 (Nov. 1997), 391–411. <https://doi.org/10.1145/265924.265927>
- [37] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBLA '09)*. ACM, New York, NY, USA, 62–71. <https://doi.org/10.1145/1791194.1791203>
- [38] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, vLi Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign. <http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf>
- [39] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 11, 12 pages. <https://doi.org/10.1145/2851141.2851145>
- [40] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP '05)*. ACM, New York, NY, USA, 221–234. <https://doi.org/10.1145/1095810.1095832>
- [41] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: A Low-overhead Mechanism for Detecting Data Races in GPU Programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 135–146. <https://doi.org/10.1145/1941553.1941574>
- [42] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. 2014. GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. *IEEE Transactions on Parallel and Distributed Systems* 25, 1 (Jan 2014), 104–115. <https://doi.org/10.1109/TPDS.2013.44>