GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme

Mai Zheng, *Student Member*, *IEEE*, Vignesh T. Ravi, *Member*, *IEEE*, Feng Qin, *Member*, *IEEE*, and Gagan Agrawal, *Member*, *IEEE*

Abstract—In recent years, GPUs have emerged as an extremely cost-effective means for achieving high performance. While languages like CUDA and OpenCL have eased GPU programming for nongraphical applications, they are still explicitly parallel languages. All parallel programmers, particularly the novices, need tools that can help ensuring the correctness of their programs. Like any multithreaded environment, data races on GPUs can severely affect the program reliability. In this paper, we propose GMRace, a new mechanism for detecting races in GPU programs. GMRace combines static analysis with a carefully designed dynamic checker for logging and analyzing information at runtime. Our design utilizes GPUs memory hierarchy to log runtime data accesses efficiently. To improve the performance, GMRace leverages static analysis to reduce the number of statements that need to be instrumented. Additionally, by exploiting the knowledge of thread scheduling and the execution model in the underlying GPUs, GMRace can accurately detect data races with no false positives reported. Our experimental results show that comparing to previous approaches, GMRace is more effective in detecting races in the evaluated cases, and incurs much less runtime and space overhead.

Index Terms—GPU, CUDA, data race, concurrency, multithreading

1 Introduction

1.1 Motivation

Today, a variety of nongraphical applications are being developed on GPUs by programmers, scientists, and researchers around the world [1], spanning different domains, including computational biology, cryptography, financial modeling, and many others. Sustaining the trend toward application acceleration using GPUs or GPU-based clusters will require advanced tool support [2]. Though CUDA [1] and OpenCL [3] have been quite successful, they are both explicitly parallel languages, and pose a programming challenge for those lacking prior parallel programming background. It is common for today's desktops and laptops to have low- and medium-end GPUs, making a highly parallel environment affordable to application developers that never used clusters or SMPs in the past. Developing correct and efficient parallel programs is a formidable challenge for this group of users. On the other hand, when developing high-end applications for a cluster of GPUs, a hybrid programming model combining message passing interfaces (MPI) and CUDA must be used, making the code hard to write, maintain, and test even for experienced parallel programmers. Recently, a number of efforts have been initiated with the goal of automatic code generation for GPUs [4], [5], [6], [7], but these efforts are still in early stages.

- *M. Zheng, F. Qin, and G. Agrawal are with The Ohio State University, 395 Dreese Laboratories, 2015 Neil Avenue, Columbus, OH 43082.*
- *V.T. Ravi is with AMD, 7171 Southwest Pkwy, B200 4A, Austin, TX 78735.*

Manuscript received 16 Mar. 2012; revised 29 Jan. 2013; accepted 31 Jan. 2013; published online 15 Feb. 2013.
Recommended for acceptance by F. Mueller.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-03-0288.
Digital Object Identifier no. 10.1109/TPDS.2013.44.

Any multithreaded program involves the risk of *race conditions* [8]. Once a race condition has occurred, it can lead to program crashes, hangs, or silent data corruption [9]. As GPUs obtain high performance with a large number of concurrent threads, race conditions can easily manifest [10], [11]. Besides the fact that many GPU programmers may lack parallel programming experience, another reason for data races can be programmers' unawareness of implicit assumptions made in third-party kernel functions. For example, one assumption a kernel function developer may make while aggressively optimizing shared memory use in a GPU program is *"the maximal number of threads per block will be 256."* A user of this kernel may be unaware of such an assumption, and may launch the kernel function with 512 threads. This is likely to create overlapped memory indices among different threads and lead to data races.

Many approaches have been developed for detecting data races in multithreaded programs that run on CPUs. These approaches can be classified into three categories: lockset-based methods [8], [12], [13], happens-before-based techniques [14], [15], [16], [17], and hybrid schemes combining these two [18], [19], [20]. While effective in detecting data races for CPU programs, these approaches are mostly inapplicable to GPU programs. This is because GPU programs only use barriers for synchronization instead of locks, which makes lockset-based methods inappropriate. Furthermore, GPU programs typically have simple happens-before relation through barriers, which makes existing happens-before-based techniques unnecessarily expensive.

Recently, two distinct approaches [10], [11] have been proposed to detect data races in GPU programs. PUG [11] symbolically models GPU kernels and leverages satisfiability modulo theories (SMT) solvers to detect data races.

While effective in finding subtle data races, PUG may report false positives or false negatives due to its approximation of the models. Furthermore, the state explosion of thread interleavings remains a problem. Boyer et al. [10] proposed a dynamic tool to detect data races by tracking all accesses to shared variables at runtime. While it can detect data races, the tool incurs several orders of magnitude of runtime overhead since it is designed for execution only in the emulation mode. Even though it may be ported to real GPUs, the runtime overhead is expected to be very large because the tool attempts to detect races for every shared memory access, which renders at least tens of thousands of times device memory accesses. Additionally, the tool may report many false positives since it does not consider all details of GPU thread scheduling.

As we stated above, one race condition we have observed arises when a third-party kernel is incorrectly used by an application developer. Detecting such race conditions requires mechanisms that only incur modest overheads, suitable for an application development environment. While the overheads for any tool that performs runtime logging will clearly be too high for a production run of the application, our target is to keep slowdowns comparable to those of versions compiled with the debugging (`-g -G`) option. Second, the mechanisms must report very few (or preferably none) false positives so that they can guide programmers to quickly fix the data races.

1.2 Our Contributions

In this paper, we propose a low-overhead technique, called GMRace, for accurately detecting data races in GPU programs. GMRace exploits the advantages of both static analysis (i.e., no runtime overhead) and dynamic checking (i.e., accurate detection). Our idea is based on the observation that many memory accesses in GPU kernels are regular and, therefore, they can be predetermined at compile time. For example, the statement "$Array[tid] = 3$" assigns 3 to the array elements indexed with current thread ids. Based on this observation, GMRace statically detects, or prunes the possibility of, data races for a majority of the memory accesses. Certainly, static analysis may not determine all memory accesses in GPU programs. For example, a memory index that depends on user input data cannot be determined at compile time. In this case, GMRace instruments the corresponding statements and detects data races at runtime.

Unlike previous work [10], GMRace's dynamic checking is aware of GPU architecture and runtime systems, i.e., memory hierarchy and thread scheduling. By exploiting GPU memory hierarchy, GMRace can detect part of data races via shared memory efficiently. Similarly, GMRace is aware of GPU's thread scheduling mechanism and execution model. Therefore, GMRace does not report false positives in our experiments.

In summary, we have made the following contributions on detecting data races in GPU programs.

1.2.1 Combining Static Analysis and Dynamic Checking

GMRace utilizes static analysis for improving the performance of dynamic data race detection for GPU programs. Our experimental results show that GMRace can efficiently detect data races and benefits significantly from static analysis.

1.2.2 Exploiting GPU's Thread Scheduling and Execution Model

We have identified the key difference between data race detection on GPU and that on CPU, which lies in GPU-specific thread scheduling and execution model. By exploiting the difference, GMRace can detect data races more accurately in GPU programs than existing work. Furthermore, we have explored two designs of GMRace, including GMRace-stmt with more helpful bug diagnostic information, and GMRace-flag with lower runtime and space overhead.

1.2.3 Exploiting GPU's Memory Hierarchy

By leveraging low-latency memory in GPU, GMRace detects data races more efficiently. Users can decide whether to enable such feature or not based on their own needs.

1.2.4 Implementing and Evaluating a Prototype of GMRace

We have implemented a prototype of GMRace and evaluated it with five GPU kernel functions from different areas. Specifically, we have evaluated the functionality and performance of GMRace, as well as benefits from utilizing static analysis and exploiting GPU architecture and runtime systems. In addition, we have evaluated the accuracy of GMRace in terms of false positives with 10 GPU applications.

Note that GMRace is an improvement over our previous work GRace [21]. GMRace-stmt scheme solves the performance bottleneck of GRace-stmt [21] by using a whole block of GPU threads to perform detection in parallel, which leads to a 100-fold reduction in runtime overhead. Additionally, GMRace-flag is a new scheme based on the key observation that the counters used in the GRace-addr [21] are unnecessary. Instead, by using simple 1/0 flags, GMRace-flag not only reduces the runtime overhead of GRace-addr by a factor of 2.6, but also reduces the space overhead by a factor of 4.5.

2 GMRace Design and Implementation

2.1 Design Challenges and GMRace Overview

It is challenging to design a low overhead, accurate dynamic data race detector for GPU programs. More specifically, there are three key design challenges which we list below.

2.1.1 How to Handle a Large Number of Shared Memory Accesses in GPU Programs?

Typically, a GPU kernel function launches thousands of threads to achieve the best use of available cores and for masking memory latencies. As a result, a running GPU kernel issues a large number of shared memory accesses from many threads concurrently. We need to monitor all of these memory accesses and check possible data races for each pair of memory accesses from different threads. This can easily incur a prohibitive runtime overhead, which is also the key limitation of the previous work in this area [10].

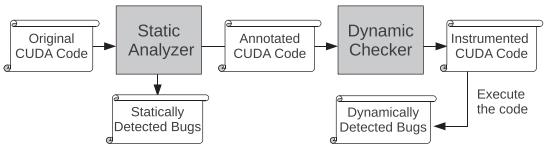


Fig. 1. GMRace overview. Two shaded blocks are the components of GMRace.

2.1.2 How to Detect Data Races in GPU Programs Accurately?

As we explained in the previous section, GPU runtime systems typically schedule a warp (or wavefront in stream SDK and OpenCL) of threads to run together on a GPU processor. The threads within a warp execute the same instruction in a given clock cycle (SIMD model). As a result, different instructions are executed sequentially by all the threads within a warp and thereby cannot cause any data races. Without distinguishing threads based on warps, a detection method can easily introduce false positives.

2.1.3 How to Handle Slow Device Memory in GPUs?

Detecting data races at runtime requires monitoring and storing a large number of memory accesses. The best choice is to store these data in device memory since it is much larger than shared memory, and also faster than the host memory. However, even device memory is much slower than shared memory. As a result, checking data races at each memory access is expected to be very slow if all profiling data is stored in device memory.

To address the above challenges, GMRace exploits static analysis, knowledge of GPU-specific thread scheduling, and the memory hierarchy. As shown in Fig. 1, GMRace consists of two major components: Static Analyzer and Dynamic Checker. More specifically, Static Analyzer first detects certain data races and also prunes memory access pairs that cannot be involved in data races. Other memory access statements, which are neither determined as data races nor pruned by the Static Analyzer, are instrumented by the Dynamic Checker. It then efficiently detects data races at runtime by leveraging both shared memory and device memory.

To detect a data race, both Static Analyzer and Dynamic Checker check one synchronization block at a time. A *synchronization block* contains all the statements between a pair of consecutive synchronization (barrier) calls. The reason for checking one synchronization block at a time is that memory accesses across different synchronization blocks are strictly ordered and, therefore, they cannot cause data races. In the case of no explicit synchronization calls, the end of a kernel function is the implicit synchronization point.

GMRace currently only considers data races in shared memory accesses from GPU kernel functions. Based on our experiences, race conditions are more likely on shared memory accesses, since the use of a small shared memory is aggressively optimized in kernel functions. For example, variables that are updated frequently and/or updated by different threads are usually stored in shared memory. To detect races in device memory, our proposed static analysis (see Section 2.2) and exploitation of thread scheduling knowledge (see Section 2.3.1) are still applicable since these techniques are irrelevant to the locations where races occur.

It should be noted that the latest NVIDIA cards support a traditional L1 cache. However, it is expected that experienced GPU programmers, particularly those developing kernels, will continue to use and aggressively optimize shared memory even on newer cards. This is because a programmer controlled cache can provide better reuse, especially when a developer has a deep understanding of the applications and the architecture.

2.2 Static Analyzer

In this section, we describe the algorithm of static analysis. The goal of our static analysis is to resolve as many memory references as possible, and determine if they could be involved in a data race or not. After our static analysis phase, only the memory references that cannot be resolved completely, or the memory references that could conflict with another unresolvable access, are instrumented. Another key goal of our static analysis is to help further reduce the overhead of dynamic instrumentation. To achieve this goal, our static analysis determines whether the same address is accessed across multiple loop iterations and/or different threads. If so, dynamic checker can perform the instrumentation for only certain iterations and/or threads so that the runtime and space overheads are drastically reduced.

Our overall analysis algorithm for a synchronization block is as follows: The kernel code within a synchronization block may contain nested loops. However, we assume that there is no unstructured control-flow, i.e., the loops are explicit, and are not created with the use of a *goto* statement. We perform the following steps, if the address can be statically determined. All memory accesses are transformed into a linear constraint in terms of the *thread id* (tid) and loop iterator(s) (I). Also, the constraints are parameterized with the range of values that tid and I can possess. We consider all pairs of left-hand-side memory accesses to examine the possibility of *write-write* race conditions. Similarly, we consider all pairs each comprising a left-hand-side and a right-hand-side memory access, to evaluate the possibility of a *read-write* race condition. Integer programming (*linear constraint solver*) is used to determine the existence of combination of tid and I for which the shared memory addresses accessed by distinct threads can be identical. A conflict could be intrawarp only if the conflict arises between the threads with identifiers from $i * warpSize$ to $(i+1) * warpSize - 1$, where $0 \leq i < warpNum$. Similarly, a conflict may lead to *interwarp* race if the conflict arises between threads with identifiers that across different warps.

If the address cannot be determined at compile time, we mark for analysis at runtime (for Dynamic Checker). After this, all such pairs have been determined, we next consider how the overheads of dynamic instrumentation can be reduced. Toward this, we determine if an address potentially involved in a conflict is invariant across threads. If so, the address needs to be recorded during execution of only one thread. Moreover, we can only have an *interwarp* race in this condition, as race condition must arise with execution of a different instruction by another thread. Otherwise,

within an instruction, if the write access (which is a thread invariant) is not protected by atomic operations we report an intrawarp data race.

Also, if a memory access expression is invariant across iterations of one or more loops in which this memory access is enclosed, it only needs to be recorded during one iteration of the loop. The formal description of the algorithm and a case study can be found in Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.44.

2.3 Dynamic Checker

Dynamic Checker detects data races at runtime. Given a GPU kernel function, the Dynamic Checker instruments every memory access statement that is annotated by Static Analyzer. At runtime, the inserted code after these memory access statements records information of the memory access and detects *intrawarp* data races, which are caused by the same statement from multiple threads within a warp. Furthermore, Dynamic Checker instruments every synchronization call to detect *interwarp* data races, which are caused by the same or different statements from multiple threads across different warps. Note that the profiling and checking code will be executed only by one thread and/or at one iteration if Static Analyzer determines the memory address is invariant across threads and/or loop iterations.

By separating intrawarp and interwarp races, GMRace improves detection accuracy, i.e., no false positives caused by different instructions from intrawarp threads accessing the same address. Another benefit of this separation is that GMRace can perform fast intrawarp race detection after each memory access and delay the slow interwarp race detection to each synchronization call. Furthermore, GMRace can utilize a small chunk of shared memory to temporarily store memory accesses from a warp of threads and thereby speed up intrawarp race detection.

It is worth mentioning that the inserted code by Dynamic Checker may affect register assignment for each thread, which in turn may alter the scheduling of warps on the GPU. However, such warp scheduling change does not affect the detection capability of GMRace. First, intrawarp races detection only concerns the threads within a warp, which is irrelevant to scheduling between warps. Second, since GPUs use barriers for synchronization, GMRace examines memory accesses from different warps of threads within a synchronization block (i.e., statements between two consecutive barriers) for detecting potential racing memory accesses. In other words, if two memory accesses within a synchronization block are racing, no matter how the threads are scheduled, GMRace will detect the race at the synchronization point. This is true for both GMRace-stmt and GMRace-flag, which means the two schemes will detect the same race.

2.3.1 Intrawarp Race Detection

GMRace maintains a table, called warpTable, to store memory access information from every statement executed by each warp of threads. More specifically, after each instrumented memory access in the kernel function, GMRace records the access type (read or write) and the memory addresses accessed by all the threads within a warp into the corresponding warpTable. A warpTable has one address entry for each thread, which allows all the threads within a warp to write the accessed memory addresses into the table in parallel.

After recording one warp of memory accesses in a warpTable, GMRace performs intrawarp race detection as follows: More specifically, it first checks whether the access type is read. If yes, the checking process stops since it is impossible to have races only through reads. Otherwise, GMRace scans the table to check whether two threads access the same memory address. If so, GMRace reports a data race with the executed statement and racing thread ids. All threads within a warp execute the above steps in parallel. Note that GMRace requires no explicit synchronization between updating a warpTable and detecting intrawarp races since both operations will be executed sequentially (SIMD) by all threads within a warp. This is important because inserted synchronization may lead to deadlock if the statement is a conditional branch that will be executed by a subset of threads within a warp. The formal description of the algorithm can be found in Appendix C, which is available in the online supplemental material.

After performing intrawarp race detection, GMRace transfers the necessary information to device memory for future interwarp race detection, which is discussed in Section 2.3.2. As a result, GMRace can recycle the warpTable for next memory access and race detection for the same warp of threads. This design choice keeps the memory footprint of intrawarp race detection minimal. Our experimental results have shown that typically 1 KB can hold the warpTables for all the warps on Tesla C1060 (see Section 4). Thus, GMRace only incurs 6 percent space overhead for 16-KB shared memory in Tesla cards. With the trend of increasing size of shared memory, the relative space overhead will become even smaller. For example, the latest release of GPU chip, Fermi, gives the option of having 48-KB shared memory, which reduce the relative space overhead of our approach to 2 percent. If running legacy GPU kernel functions that assume 16-KB shared memory, GMRace can enjoy plenty of shared memory. The extreme case is that a kernel function uses up shared memory for its own benefits. In such case, GMRace can store the warpTables in device memory and performs intrawarp race detection there.

2.3.2 Interwarp Race Detection

GMRace periodically detects interwarp races after each synchronization call. More specifically, GMRace transfers the memory access information from a warpTable to device memory after each intrawarp race detection. After each synchronization call, GMRace identifies interwarp races by examining memory accesses from multiple threads that are across different warps. After detecting interwarp races at one synchronization call, GMRace reuses the device memory for next synchronization block.

By exploring the design space along two dimensions, i.e., accuracy of bug reports and efficiency of bug detection, we propose two interwarp detection schemes. One scheme, called *GMRace-stmt*, organizes memory access information by the executed program statements. This scheme reports data races with more accurate diagnostic information while

incurring time and space overheads that are quadratic and linear with regard to the number of executed statements, respectively. The other scheme, called *GMRace-flag*, records memory access information using 0/1 flags based on shared memory addresses. This scheme incurs constant time and space overhead while reporting aggregated diagnostic information on data races. We present both schemes in the rest of this section.

The statement-based scheme (GMRace-stmt). This scheme of GMRace (referred to as GMRace-stmt) literally stores all the memory addresses that have been accessed from all the threads in device memory and identifies two threads from different warps for accessing the same memory address. More specifically, GMRace-stmt maintains a BlockStmtTable in device memory for the threads from all the warps that can access the same shared memory. As shown in Fig. 2, each entry (i.e., row) of the BlockStmtTable stores all the content of a warpTable (all memory addresses accessed from one statement executed by a warp of threads) and the corresponding warp ID and statement number. Essentially, GMRace-stmt organizes a BlockStmtTable by memory access statements from all the threads. Note that in this scheme the BlockStmtTable is shared among all warps of threads. However, different warps can write to different rows of the BlockStmtTable concurrently. As a result, we only need to use atomic operations when updating the row index, which is not a significant source of overhead (see Section 4).

At each synchronization call, GMRace-stmt scans the entire BlockStmtTable for identifying interwarp data races as described in Algorithm 1. Specifically, it checks two BlockStmtTable entries at a time throughout the entire table (lines 1 and 2). Note that each thread starts from a different entry to check the table in parallel. For each pair of the entries, GMRace-stmt checks both the warp IDs and access types (line 3-8). If the warp IDs are the same or both accesses are read, GMRace-stmt skips this pair since any pair of memory accesses from both entries cannot cause interwarp races. Otherwise, GMRace-stmt checks whether there are two addresses, one from each entry, are the same (line 9-15). Once the same addresses are found, GMRace-stmt reports a data race (line 12).

Algorithm 1. Interwarp Race Detection by GMRace-stmt.

1: **for** $stmtIdx1 = tid$ to $maxStmtNum - 1$ **do**
2: **for** $stmtIdx2 = stmtIdx1 + 1$ to $maxStmtNum$ **do**
3: **if** $BlkStmtTbl[stmtIdx1].warpID = BlkStmtTbl[stmtIdx2].warpID$ **then**
4: Jump to line 17
5: **end if**
6: **if** $BlkStmtTbl[stmtIdx1].accessType$ is $read$ **and** $BlkStmtTbl[stmtIdx2].accessType$ is $read$ **then**
7: Jump to line 17
8: **end if**
9: **for** $targetIdx = 0$ to $warpSize - 1$ **do**
10: **for** $sourceIdx = 0$ to $warpSize - 1$ **do**
11: **if** $BlkStmtTbl[stmtIdx1][sourceIdx] = BlkStmtTbl[stmtIdx2][targetIdx]$ **then**
12: Report a Data Race
13: **end if**
14: **end for**
15: **end for**
16: **end for**
17: $stmtIdx1 += threadNum$
18: **end for**

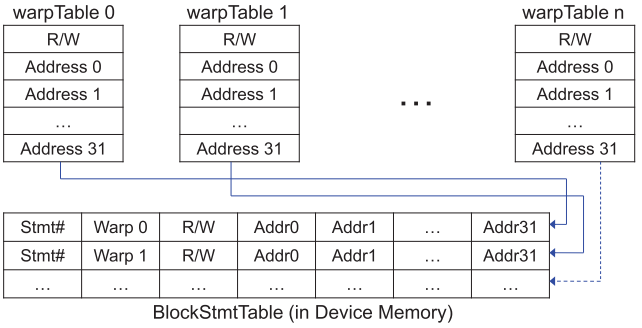


Fig. 2. Data structure of a BlockStmtTable. Each entry in the BlockStmtTable stores all the content of a warpTable, in addition to the warp ID and the statement number.

On the one hand, GMRace-stmt provides accurate diagnostic information for each detected race, including the pair of racing statements (i.e., the statement numbers), the pair of racing threads (i.e., the indexes of both memory addresses in the BlockStmtTable entries), and racing memory address. This is very helpful for developers to quickly locate the root cause and fix the data race. On the other hand, the time complexity of Algorithm 1 (i.e., the detection part of GMRace-stmt scheme) is quadratic with regard to the number of BlockStmtTable entries, i.e., the number of instrumented statements that are executed. Furthermore, the space overhead incurred by GMRace-stmt is linear to the number of BlockStmtTable entries. Although this indicates that GMRace-stmt may not be scalable, it is expected to perform well with a small number of statements being instrumented and executed (see our experimental results in Section 4).

The flag-based scheme (GMRace-flag). This scheme of GMRace (referred to as GMRace-flag) stores summarized (i.e., using 0/1 flags) information of the memory addresses that have been accessed from all the threads and detects data races based on the summarized information. More specifically, GMRace-flag maintains two tables for each warp, one for read accesses from threads within the warp (referred to as rWarpShmMap) and the other for write accesses (referred to as wWarpShmMap). Each entry in any of these tables maps to one shared memory address linearly. Specifically, each entry stores a 0/1 flag that records whether the corresponding shared memory address has been accessed by the warp or not. Fig. 3 shows the data structures of rWarpShmMap and wWarpShmMap.

After each monitored memory read access, GMRace-flag set the flag in the corresponding rWarpShmMap to 1. Similarly, for each write access, the flag in the corresponding wWarpShmMap is set to 1. Essentially, the rWarpShmMaps and the wWarpShmMaps keep the memory footprints of different warps. Note that we do not count the number of accesses here. The flag will be set to 1 as long as the corresponding shared memory address

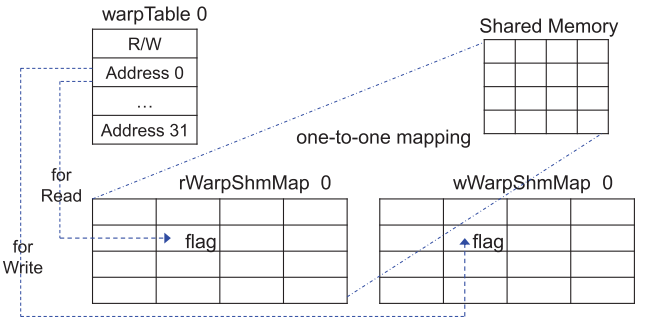


Fig. 3. Data structures of a rWarpShmMap and a wWarpShmMap. Each WarpShmMap entry is corresponding to one shared memory address in one-to-one mapping. Each address stored in a warpTable is used as the index to update the corresponding flag in the rWarpShmMap or wWarpShmMap, depending on the access type.

is accessed, regardless how many times it is accessed. This design choice simplifies the recording operations while keeping enough information for race detection. The flags for all these tables are reset to 0s after race detection at each synchronization call.

At each synchronization call, GMRace-flag detects interwarp races as shown in Algorithm 2. Specifically, GMRace-flag scans through all the flags stored in the wWarpShmMaps and the rWarpShmMaps in parallel. For each shared memory address, GMRace-flag sums up the corresponding flags in the wWarpShmMaps and the rWarpShmMaps, respectively (line 4-7). The first sum (*writeSum*) denotes the number of warps that have written to the shared memory address before the synchronization call. If the *writeSum* is zero, which means no warp has written to the address, then no race could have happened within this synchronization block (line 8-9). If the value is equal to or greater than 2, which means at least two different warps have accessed the address, then GMRace-flag reports races (line 10-11). If the *writeSum* equals to 1, GMRace-flag further checks the second sum (*readSum*), which indicates the number of warps that have read from the address. Given that the *writeSum* equals to 1 (i.e., only one warp have written to this address), a zero value of *readSum* indicates no race (line 13-14), while a value equal to or greater than 2 guarantees races (line 15-16). If the *readSum* also equal to 1, GMRace-flag further locates the wWarpShmMap and the rWarpShmMap that containing the nonzero flag and checks whether they are set by the same warp. If yes, which means the write and read are performed by the same warp, there is no race condition. Otherwise, GMRace-flag reports a race (line 17-23). Note that Algorithm 2 is described in sequential for simplicity, while the detection in GMRace-flag is performed in parallel, i.e., multiple threads check different *idx* simultaneously.

Algorithm 2. Interwarp Race Detection by GMRace-flag.

1: **for** $idx = 0$ to $shmSize - 1$ **do**
2: $writeSum \leftarrow 0$
3: $readSum \leftarrow 0$
4: **for** $warpID = 0$ to $warpID = warpNum - 1$ **do**
5: $writeSum+ = wWarpShmMaps[warpID][idx]$
6: $readSum+ = rWarpShmMaps[warpID][idx]$
7: **end for**
8: **if** $writeSum = 0$ **then**
9: Jump to line 25
10: **else if** $writeSum >= 2$ **then**
11: Report Data Races
12: **else if** $writeSum = 1$ **then**
13: **if** $readSum = 0$ **then**
14: Jump to line 25
15: **else if** $readSum >= 2$ **then**
16: Report Data Races
17: **else if** $readSum = 1$ **then**
18: $wWarpID = getWarpIDofNonZeroFlag(wWarpShmMaps, idx)$
19: $rWarpID = getWarpIDofNonZeroFlag(rWarpShmMaps, idx)$
20: **if** $wWarpID! = rWarpID$ **then**
21: Report a Data Race
22: **end if**
23: **end if**
24: **end if**
25: **end for**

On the one hand, the time and space complexities of the Algorithm 2 (i.e., the detection part of GMRace-flag scheme) are linear to the size of shared memory, which is constant for a given GPU. Therefore, GMRace-flag is scalable in terms of the number of instrumented statements, although it may not be a better choice for a kernel with a small number of instrumented statements. On the other hand, GMRace-flag provides aggregated information about a data race, which is less accurate than GMRace-stmt. For example, GMRace-flag reports racing memory address and the pairs of racing warps instead of racing statements or racing threads. However, the bug information provided by GMRace-flag is still useful. For example, programmers can narrow down the set of possibly racing statements based on a racing memory address reported by GMRace-flag. Similarly, programmers can derive racing threads based on the ranges of reported racing warps.

3 Evaluation Methodology

Our experiments were conducted using a NVIDIA Tesla C1060 GPU with 240 processor cores (30×8), a clock frequency of 1.296 GHz, and 4-GB device memory. The GPU was connected to a machine with two AMD 2.6-GHz dual-core Opteron CPUs and 8-GB main memory. We have implemented a prototype of GMRace based on ROSE compiler infrastructure [22]. Static Analyzer utilizes the linear constraint solver [23], and Dynamic Checker is built on CUDA Toolkit 3.0. We do not see any particular difficulty to port GMRace to other GPU environments such as stream SDK or OpenCL.

We have evaluated GMRace's functionality and efficiency with five applications, including coclustering [24] (referred to as `co-cluster`), EM clustering [25] (referred to as `em`), Scan Algorithm [10] (referred to as `scan`), Sparse Matrix-Vector Multiplication (referred to as `spmv`), and Binomial Options (referred to as `bo`). Among these applications, `co-cluster` and `em` are both clustering algorithms. We have used GPU implementations of these

TABLE 1
Overall Effectiveness of GMRace for Data Race Detection

Apps	GMRace-stmt				GMRace-flag			B-tool		PUG
	Found?	R-Stmt#	R-Mem#	R-Thd#	Found?	R-Mem#	R-Wp#	Found?	RP#	Found?
co-cluster	Yes	1	10	1,310,720	Yes	10	8	Yes	1	No
em	Yes	14	384	22,023	Yes	384	3	Yes	200,445	Yes
scan	Yes*	-	-	-	Yes*	-	-	Err**	N/A	Yes
spmv	Yes	5	4	113,528	Yes	4	4	Err**	N/A	No
bo	Yes*	-	-	-	Yes*	-	-	Err**	N/A	Yes

*We compare the detection capability of GMRace-stmt and GMRace-flag with that of B-tool and PUG, the techniques proposed by previous works [10] and [11], respectively. "Found" indicates whether a tool can detect the data races or not. R-Stmt is pairs of conflicting accesses, R-Mem is memory address invoked in data races, R-Thd is pairs of threads in race conditions, R-Wp is pairs of racing warps, and RP means the race number reported by B-tool. "-" means the data is not reported by the scheme. * Three pairs of racing statements are detected and all addresses are resolved by Static Analyzer. ** B-tool leads to an error when running with `scan` on the latest versions of CUDA and Tesla GPUs, because of hardware and software changes.*

applications that were aggressively optimized for shared memory use in a recent study [26]. `scan` is the code used in the previous work on GPU race detection [10]. `spmv` is a stencil computation application, and `bo` is a financial modeling algorithm.

In developing GPU *kernels* for `co-cluster` and `em`, i.e., in creating GPU implementations of the main computation steps, certain implicit assumptions were made. For example, `co-cluster` assumes that the initialization values of a particular matrix should be within a certain range, whereas `em` assumes that the maximum thread number within a block is 256. If these kernels are used by another application developer, these assumptions may be violated, and data races can occur. We create invocations of these kernels in ways such that race conditions were manifested. Additionally, to trigger data races in `scan`, we remove the first synchronization call as was done in the previous work on GPU race detection [10]. Similarly, we remove a synchronization call in `bo` to trigger the data races. As for `spmv`, we inject an additional shared memory read to introduce race conditions.

More specifically, we have designed five sets of experiments to evaluate the key aspects of GMRace:

- The first set evaluates the functionality of GMRace in detecting data races of GPU kernel functions. We compare GMRace with two previous approaches, one is referred to as B-tool [10] in this paper and the other is PUG [11], in terms of reported number of races and false positives. In this set, we use bug-triggering inputs and parameters to trigger data races.
- The second set evaluates the runtime overhead incurred by GMRace in terms of execution time of the kernel functions. We also compare GMRace with B-tool. Additionally, we evaluate the space overhead caused by GMRace and compare it with B-tool. Moreover, we compare GMRace with GRace [21]. In this set, we use normal inputs and parameters to launch the kernel functions so that data races do not occur.
- The third set evaluates GMRace's effectiveness in terms of false positives using 10 GPU kernels that have been used in prior work [11].
- The fourth set evaluates the benefit of Static Analyzer. We measure the instrumented statements and memory accesses statically and dynamically in two configurations, i.e., with Static Analyzer and without Static Analyzer. Furthermore, we compare the runtime overhead of GMRace with and without Static Analyzer.
- The fifth set evaluates the benefit of shared memory. We measure the runtime overhead of GMRace in two configurations, i.e., warpTables stored in shared memory and warpTables stored in device memory.

Note that all the above experiments evaluate two inter-warp detection schemes GMRace-stmt and GMRace-flag, both with the same intrawarp detection scheme and the same Static Analyzer. Due to space limit, we put the space overhead in Appendix D, which is available in the online supplemental material, and the last three sets of experimental results in Appendices E, F, and G, which are available in the online supplemental material, respectively.

4 Experimental Results

4.1 Overall Effectiveness

Table 1 demonstrates the overall effectiveness of GMRace. Specifically, we evaluate four schemes, including GMRace-stmt, GMRace-flag, the previous work B-tool [10], and a recently proposed technique PUG [11]. For each of these schemes, we measure whether it can detect the race conditions in the applications or not. Moreover, for GMRace-stmt, we measure another three metrics, including the number of pairs of racing statements, the number of memory addresses involved in data races, and the number of pairs of threads in race conditions. Similarly, for GMRace-flag, we measure the number of pairs of warps in race conditions besides the number of memory addresses involved in data races. For B-tool, we present the number of data races reported by the tool. Unlike GMRace-stmt or GMRace-flag, B-tool reports a data race whenever the current thread accesses a memory address where other threads have conflicting accesses before. It does not report pairs of statements, threads, or warps involved in race conditions. For PUG, we only present whether it can detect the races or not since it does not provide other information to users explicitly. We use bug-triggering parameters or

inputs to launch the kernel functions. For scan and bo, we do not execute it since Static Analyzer detects the races and does not annotate any statement for runtime checking.

As shown in Table 1, both GMRace-stmt and GMRace-flag can effectively detect data races. They correctly identify the race conditions in all of the five evaluated applications. On the contrary, although B-tool can detect races in cocluster and em, it may report many false positives, as discussed in Appendix E, which is available in the online supplemental material. Due to B-tool's incorrect use of inserted synchronization calls for the instrumentation code, it could not run for scan, spmv, and bo on the new hardware and software. PUG can detect races in three of the five applications. However, it reports no errors for cocluster and spmv. This is because there is an indirect shared memory reference in cocluster, while in spmv the racing memory accesses depends on input, both of which cannot be solved by PUG.

Table 1 indicates that GMRace-stmt provides more accurate information about data races than GMRace-flag, B-tool do. Since GMRace-stmt logs memory accesses at the program statement level, it can report the pair of racing statements once a bug is found. On the contrary, GMRace-flag and B-tool cannot report the pair of statements involved in a race, since they do not keep information of the statements involved in the races. Furthermore, GMRace-flag reports only the pairs of racing warps, which are coarser-grained than what is available from GMRace-stmt and B-tool. However, diagnostic information provided by GMRace-flag is still useful to locate the root causes. For example, based on memory addresses involved in a race and the corresponding pair of racing warps, programmers can narrow down the search range of possible statements and threads responsible for the data race and further identify the root causes.

Table 1 further shows that Static Analyzer not only reduces runtime overhead of dynamic checking, can it also detect data races. For example, Static Analyzer detects the data races in scan and bo and resolves all memory addresses. Therefore, it totally eliminates the overhead of running Dynamic Checker for the two applications.

It is worth mentioning that, although GMRace outperforms the other two tools in race detection, B-tool and PUG have other usage that GMRace does not have. To the best of our knowledge, B-tool is the first dynamic analysis tool that can detect races in GPU programs. Besides data races, it detects shared memory bank conflicts as well. Similarly, PUG is able to prove the functional correctness of GPU programs to certain extent besides detecting race conditions.

4.2 Runtime Overhead

We measure the execution time for co-cluster, em, and spmv in four configurations: executing the kernels on GPU natively without any instrumentation, executing the kernels with GMRace-stmt on GPU, executing the kernels with GMRace-flag on GPU, and executing the kernels with B-tool in the device emulation mode provided by the CUDA SDK. We run B-tool in emulation mode as it is not designed to run on an actual GPU [10]. B-tool cannot run for spmv due to its incorrect use of synchronization calls. We use *normal inputs*, i.e., those that do not trigger data races, for these experiments. Note that GMRace does not have runtime overhead for scan and bo since the Static Analyzer did not annotate any statements.

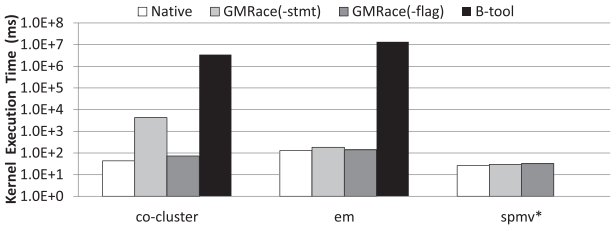


Fig. 4. Runtime overhead of GMRace and B-tool. Note that the y-axis is on a logarithmic scale. *B-tool cannot run on spmv due to its incorrect use of barriers.

Fig. 4 shows that GMRace-flag and GMRace-stmt incur lower runtime overhead than the B-tool. For example, GMRace-flag and GMRace-stmt slow down em by 8 and 40 percent, respectively. On the contrary, B-tool incurs several orders of magnitude higher runtime overhead, i.e., slowing down em by 103,850 times. There are several reasons for the big performance gap between GMRace and B-tool. First, GMRace-flag and GMRace-stmt utilize static analysis to significantly reduce the number of memory accesses that need to be checked dynamically. Second, both GMRace-flag and GMRace-stmt delay interwarp race detection to synchronization calls, while B-tool checks data races for each memory access, which requires scanning of four bookkeeping tables after each memory access. Third, emulation mode further adds to the slow-down.

Fig. 4 also indicates that GMRace-flag is significantly more efficient than GMRace-stmt in some cases. For example, GMRace-flag slows down em and co-cluster by 8 and 67 percent, respectively, while GMRace-stmt slows down em and co-cluster by 40 percent and 98 times, respectively. This is mainly because the profiling and race detection in GMRace-flag is much more efficient than those in GMRace-stmt. While both schemes need to profile the same number of memory accesses, GMRace-flag only modifies a flag in WarpShmMap and GMRace-stmt dump much more relevant information into BlockStmtTable for each memory access. As for interwarp race detection, GMRace-flag's algorithm (i.e., Algorithm 2, the detection part of the GMRace-flag scheme) runs in a constant amount of time, i.e., it does not depend on the execution number of instrumented statements. Whereas, the complexity of GMRace-stmt's interwarp race detection algorithm (i.e., Algorithm 1, the detection part of the GMRace-stmt scheme) is quadratic with respect to the execution number of instrumented statements. As a result, GMRace-stmt takes significant larger amount of time for detection if the number of instrumented statements is large. On the other hand, if the number of instrumented statements is very small, GMRace-stmt requires less time for detection than GMRace-flag, which is the reason why GMRace-stmt outperforms GMRace-flag in spmv. Both GMRace-flag and GMRace-stmt use the same intrawarp race detection method, whose complexity is linear with respect to the number of dynamic memory accesses. Note that, intrawarp race detection, which is performed in shared memory, is much faster than interwarp race detection, which is performed in device memory.

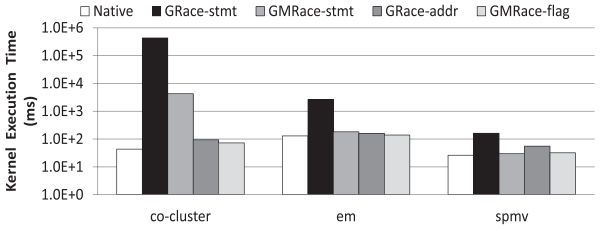


Fig. 5. Runtime overhead of different schemes of GMRace and GRace. Note that the y-axis is on a logarithmic scale.

Overall, we can see that GMRace-flag's runtime overheads are very modest, making it suitable for invocation by an end-user, who is testing a full application. If a race condition is detected in a specific kernel, the user can trigger GMRace-stmt, and collect more detailed information to help debugging.

4.3 GMRace versus GRace

This section compares GMRace with GRace [21], the first version of our tool. By combining static analysis and dynamic analysis, both GMRace and GRace are accurate in detecting data races. However, in terms of efficiency, GMRace improves GRace drastically.

As shown in Fig. 5, GRace-stmt incurs more than 9,872 times overhead on `em`, while GMRace-stmt only incurs about 97 times overhead, which is a 100-fold reduction. This is because GMRace-stmt makes use of a whole GPU thread block to perform the interwarp race detection in parallel (Algorithm 1), which solves the performance bottleneck of GRace-stmt. Similarly, GMRace-flag also improves GRace-addr significantly. For example, the overhead of GRace-addr on `spmv` is 1.1 times, while the overhead of GMRace-flag on `spmv` is only 22.9 percent. This is because GMRace-flag use simple 1/0 flags to mark the memory accesses, while GRace-addr must keep counting the accesses.

On average, GMRace-stmt reduces the overhead of GRace-stmt by a factor of 100.9, while GMRace-flag reduces the overhead of GRace-addr by a factor of 2.6. In addition, GMRace-flag scheme reduces the space overhead of GRace-addr by a factor of 4.5.

5 Issues and Discussion

We discuss three additional issues related to the usage of GMRace in this section.

5.1 Static Analysis

Our static analysis is currently simple and conservative. It only reports invariants or races if it can guarantee that these properties or conditions exist. Any invariants or memory access addresses that it cannot determine is annotated to be monitored at runtime. In the presence of more complex language features, it cannot determine properties like loop-invariance or thread-invariance. Thus, we will have to track more references at runtime if such features are used. Based on our experience, most numerical kernels involve loops on arrays, which can be analyzed by our static analysis methods.

5.2 False Negatives

Similar to other dynamic tools, the Dynamic Checker of GMRace detects bugs that manifest themselves in the exercised paths during program execution. In other words, the control flow paths that are never executed are not checked by the Dynamic Checker. Therefore, GMRace may miss some data races in GPU programs, which is a common problem for all dynamic tools for software bug detection. How to improve the path coverage by generating different test inputs is an interesting research topic garnering much research attention [27], [28], [29], [30]. Research advances along this direction can help improve GMRace.

5.3 Application of GMRace

The current implementation of GMRace focuses on data race detection on GPU programs. While this is an important topic for GPU programs, other issues such as buffer overflow, warp divergence, and shared memory usage may affect the correctness and/or the performance of GPU programs. Although GMRace cannot directly address these problems other than data races, some of the underlying ideas in GMRace are still applicable. For example, the idea of combining static and dynamic analysis can offer significant help to dynamic profiler that collects shared memory usage of GPU programs by reducing runtime/space overheads.

6 Related Work

GMRace is related to previous studies on data race detection, detection for other types of concurrency bugs, bug detection for parallel and distributed programs, tool development for GPU programming, and optimizations of GPU programs.

6.1 Data Race Detection

As discussed before, many dynamic race detectors [8], [12], [13], [14], [15], [16], [17], [18], [19], [20], which are designed for CPU programs, are not suitable for GPU programs. For example, the lockset-based race prediction [13] cannot handle GPU's barrier-based synchronization, and the thread rescheduling [13] does not apply to GPU's SIMD execution model. Besides, researchers proposed static methods for race detection, including static lockset algorithm [31] and race type-safe systems [32], [33]. Without runtime information, static methods may generate many false positives. Additionally, researchers also proposed to detect races using model checking [34], which has the limitation of state explosion problem in general. Furthermore, happens-before relation has also be applied to detect races in OpenMP programs [35]. Unlike these approaches, our work focuses on detecting races in GPU programs, which have different characteristics to deal with. To manage contention of shared resources, new OS schedulers have been proposed [36].

6.2 Detection for Other Types of Concurrency Bugs

In addition to data races, researchers have conducted studies on other types of concurrency bugs such as atomicity violation, deadlock, and typestate violation [37]. Atomizer [38], SVD [39], AVIO [40], and Kivati [41] are proposed to detect or prevent atomicity violation bugs. Moreover, tools using static analysis [31], [42], [43], model checking [44], and dynamic checking [45], [46], [47] can detect or prevent deadlocks. Unlike these approaches, our work focuses on data race detection.

6.3 Bug Detection for Parallel and Distributed Programs

Many approaches monitor program execution to detect bugs in parallel and distributed programs [48], [49], [50], [51], [52], [53], [54], [55]. Various techniques [56], [57] have been proposed to reduce the cost of program monitoring. Additionally, interactive parallel debuggers [58], [59], [60], [61], [62] help programmers to locate the root causes of software bugs in parallel programs by collecting, aggregating, and visualizing program runtime information. Our work can be integrated with these debuggers to help programmers quickly identify root causes.

6.4 Tool Development for GPU Programming

In the area of general purpose computing on GPUs, there have been numerous application development studies on GPUs over the last 3-4 years. We only focus on efforts on tool development for GPU programming. As we stated earlier, there have been two very distinct efforts on race detection for GPUs [10], [11]. In [63], a tool called GKLEE that employs concolic execution-based verification and test-case reduction heuristics has been presented. This tool has recently been extended to scale using the technique of Parameterized Flows [64]. Leung et al. [65] combine static information flow analysis with dynamic logging. They obtain a trace from one execution in emulation mode, and then use static information flow analysis to amplify the possible memory accesses. This approach can explore all possible memory access addresses of the kernel, if the kernels exhibit the access invariant property, i.e., the memory access pattern do not change across different inputs. However, this property may not hold for complex kernels. In addition, there have been recent efforts on performance measurement and profiling on CUDA programs [2], [66]. However, these tools are not designed for race detection.

6.5 Optimizations of GPU Programs

There have been many efforts on optimizations of GPU programs and code generation for GPU programs [4], [5], [6], [7], [67], but none of them have focused on program correctness issues.

7 Conclusions

In this paper, we have presented GMRace, a low-overhead approach for detecting data races in GPU programs. Our experimental results have shown that comparing to previous work GMRace can detect data races more effectively and more efficiently. Particularly, compared with GRace [21], GMRace-stmt scheme improves the performance of GRace-stmt [21] by about 100 times. Moreover, GMRace-flag scheme not only reduces the runtime overhead of GRace-addr by a factor of 2.6, but also reduces the space overhead by a factor of 4.5.

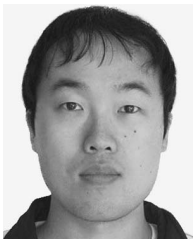
Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback. This work was supported in part by an allocation of computing time from the Ohio Supercomputer Center, and by the US National Science Foundation grants #CCF-0833101 and #CCF-0953759 (CAREER Award).

References

[1] "CUDA Community Showcase," http://www.nvidia.com, 2013.
[2] A.D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam, "An Experimental Approach to Performance Measurement of Heterogeneous Parallel Applications Using Cuda," *Proc. 24th ACM Int'l Conf. Supercomputing (ICS)*, 2010.
[3] Khronos Group, "OpenCL: The Open Standdard for Heterogeneous Parallel Programming," http://www.khronos.org/opencl, 2008.
[4] S. zee Ueng, M. Lathara, S.S. Baghsorkhi, W. mei, and W. Hwu, "CUDA-Lite: Reducing GPU Programming Complexity," *Proc. Int'l Workshop Languages and Compilers for Parallel Computing (LCPC)*, 2008.
[5] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," *Proc. 14th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '09)*, 2009.
[6] N. Sundaram, A. Raghunathan, and S. Chakradhar, "A Framework for Efficient and Scalable Execution of Domain-Specific Templates on GPUs," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS)*, 2009.
[7] W. Ma and G. Agrawal, "A Translation System for Enabling Data Mining Applications on GPUs," *Proc. 23rd Int'l Conf. Supercomputing (ICS)*, 2009.
[8] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Computer Systems*, vol. 15, no. 4, pp. 391-411, 1997.
[9] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics," *Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.
[10] M. Boyer, K. Skadron, and W. Weimer, "Automated Dynamic Analysis of CUDA Programs," *Proc. Third Workshop Software Tools for MultiCore Systems (STMCS)*, 2008.
[11] G. Li and G. Gopalakrishnan, "Scalable SMT-Based Verification of GPU Kernel Functions," *Proc. 18th ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (FSE)*, 2010.
[12] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2002.
[13] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu, "Efficient Data Race Detection for Distributed Memory Parallel programs," *Proc. ACM/IEEE Conf. Supercomputing (SC '11)*, 2011.
[14] A. Dinning and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *Proc. Second ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 1990.
[15] R.H.B. Netzer and B.P. Miller, "Improving the Accuracy of Data Race Detection," *Proc. Second ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 1991.
[16] D. Perkovic and P.J. Keleher, "Online Data-Race Detection via Coherency Guarantees," *Proc. Second USENIX Symp. Operating Systems Design and Implementation (OSDI)*, 1996.
[17] C. Flanagan and S.N. Freund, "Fasttrack: Efficient and Precise Dynamic Race Detection," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2009.
[18] R. O'Callahan and J.-D. Choi, "Hybrid Dynamic Data Race Detection," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 2003.
[19] E. Pozniansky and A. Schuster, "Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 2003.
[20] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: Efficient Detection of Data Race Conditions via Adaptive Tracking," *Proc. 12th ACM Symp. Operating Systems Principles (SOSP)*, 2005.
[21] M. Zheng, V.T. Ravi, F. Qin, and G. Agrawal, "GRace: A Low-Overhead Mechanism for Detecting Data Races in Gpu Programs," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 2011.
[22] "ROSE Compiler Infrastructure," http://www.rosecompiler.org, 2013.
[23] P. Feautrier, "Parametric Integer Programming," *RAIRO Recherche Opérationnelle*, vol. 22, no. 3, pp. 243-268, 1988.

[24] H. Cho, I.S. Dhillon, Y. Guan, and S. Sra, "Minimum Sum-Squared Residue Co-Clustering of Gene Expression Data," *Proc. Fourth SIAM Int'l Conf. Data Mining (SDM),* 2004.
[25] A. Dempster, N. Laird, and D. Rubin, "Maximum Likelihood Estimation from Incomplete Data via the EM Algorithm," *J. Royal Statistical Soc.,* vol. 39, no. 1, pp. 1-38, 1977.
[26] W. Ma and G. Agrawal, "An Integer Programming Framework for Optimizing Shared Memory Use on GPUs," *Proc. IEEE Ann. Int'l Conf. High Performance Computing (HiPC '12),* 2012.
[27] B. Korel, "Automated Software Test Data Generation," *IEEE Trans. Software Eng.,* vol. 16, no. 8, pp. 870-879, Aug. 1990.
[28] P. Godefroid, K. Nils, and K. Sen, "Dart: Directed Automated Random Testing," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* 2005.
[29] P. Godefroid, "Compositional Dynamic Test Generation," *Proc. 34th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL),* 2007.
[30] C. Cadar, G.V., P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically Generating Inputs of Death," *Proc. 13th ACM Conf. Computer and Comm. Security,* 2006.
[31] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP),* 2003.
[32] C. Boyapati, R. Lee, and M. Rinard, "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks," *Proc. 17th ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),* 2002.
[33] C. Flanagan and S.N. Freund, "Type-Based Race Detection for Java," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* 2000.
[34] T.A. Henzinger, R. Jhala, and R. Majumdar, "Race Checking by Context Inference," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* 2004.
[35] M.-H. Kang, O.-K. Ha, S.-W. Jun, and Y.-K. Jun, "A Tool for Detecting First Races in Openmp Programs," *Proc. 10th Int'l Conf. Parallel Computing Technologies (PACT),* 2009.
[36] A. Fedorova, S. Blagodurov, and S. Zhuravlev, "Managing Contention for Shared Resources on Multicore Processors," *Comm. ACM,* vol. 53, no. 2, pp. 49-57, 2010.
[37] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, "2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs," *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '11),* 2011.
[38] C. Flanagan and S.N. Freund, "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs," *Proc. 31st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL),* 2004.
[39] M. Xu, R. Bodík, and M.D. Hill, "A Serializability Violation Detector for Shared-Memory Server Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* 2005.
[40] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting Atomicity Violations via Access Interleaving Invariants," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2006.
[41] L. Chew and D. Lie, "Kivati: Fast Detection and Prevention of Atomicity Violations," *Proc. Fifth European Conf. Computer Systems (EuroSys),* 2010.
[42] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata, "Extended Static Checking for Java," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* 2002.
[43] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective Static Deadlock Detection," *Proc. 31st Int'l Conf. Software Eng. (ICSE),* 2009.
[44] "Java PathFinder," http://javapathfinder.sourceforge.net, 2007.
[45] Y. Nir-Buchbinder, R. Tzoref, and S. Ur, "Deadlocks: From Exhibiting to Healing," 2008.
[46] F. Zeng and R.P. Martin, "Ghost Locks: Deadlock Prevention for Java," *Proc. Mid-Atlantic Student Workshop Programming Languages and Systems,* 2004.
[47] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, "Deadlock Immunity: Enabling Systems to Defend Against Deadlocks," *Proc. Eighth USENIX Conf. Operating Systems Design and Implementation (OSDI),* 2008.
[48] S. Yang, A.R. Butt, Y.C. Hu, and S.P. Midkiff, "Trust But Verify: Monitoring Remotely Executing Programs for Progress and Correctness," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP),* 2005.
[49] J. DeSouza, B. Kuhn, B.R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov, "Automated, Scalable Debugging of MPI Programs with Intel Message Checker," *Proc. Second Int'l Workshop Software Eng. for High Performance Computing System Applications (SE-HPCS),* 2005.
[50] C. Falzone, A. Chan, E. Lusk, and W. Gropp, "A Portable Method for Finding User Errors in the Usage of MPI Collective Operations," *Int'l J. High Performance Computing Applications,* vol. 21, no. 2, pp. 155-165, 2007.
[51] Q. Gao, F. Qin, and D.K. Panda, "DMTracker: Finding Bugs in Large-Scale Parallel Programs by Detecting Anomaly in Data Movements," *Proc. ACM/IEEE Conf. Supercomputing (SC),* 2007.
[52] T. Hilbrich, B.R. de Supinski, M. Schulz, and M.S. Müller, "A Graph Based Approach for MPI Deadlock Detection," *Proc. 23rd Int'l Conf. Supercomputing (ICS),* 2009.
[53] B. Krammera, K. Bidmona, M.S. Muller, and M.M. Rescha, "MARMOT: An MPI Analysis and Checking Tool," *Proc. Advances in Parallel Computing (PARCO),* 2003.
[54] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "MPI-CHECK: A Tool for Checking Fortran 90 MPI Programs," *Concurrency and Computation: Practice and Experience,* vol. 15, no. 2, pp. 93-100, 2003.
[55] J.S. Vetter and B.R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," *Proc. ACM/IEEE Conf. Supercomputing (SC),* 2000.
[56] J. Odom, J.K. Hollingsworth, L. DeRose, K. Ekanadham, and S. Sbaraglia, "Using Dynamic Tracing Sampling to Measure Long Running Programs," *Proc. ACM/IEEE Conf. Supercomputing (SC),* 2005.
[57] A. Zhai, G. He, and M. Heimdahl, "Hardware and Compiler Support for Dynamic Software Monitoring," *Proc. Int'l Workshop Runtime Verification (RV),* 2009.
[58] D.H. Ahn, B.R. de Supinski, I. Laguna, G.L. Lee, B. Liblit, B.P. Miller, and M. Schulz, "Scalable Temporal Order Analysis for Large Scale Debugging," *Proc. Conf. High Performance Computing Networking, Storage and Analysis (SC),* 2009.
[59] D.C. Arnold, D.H. Ahn, B.R. de Supinski, G. Lee, B.P. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Debugging," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS '07),* 2007.
[60] S.M. Balle, B.R. Brett, C.-P. Chen, and D. LaFrance-Linden, "Extending a Traditional Debugger to Debug Massively Parallel Applications," *J. Parallel and Distributed Computing,* vol. 64, no. 5, pp. 617-628, 2004.
[61] Etnus, LLC, "TotalView," http://www.etnus.com/TotalView, 2013.
[62] S.S. Lumetta and D.E. Culler, "The Mantis Parallel Debugger," *Proc. SIGMETRICS Symp. Parallel and Distributed Tools,* 1996.
[63] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S.P. Rajan, "GKLEE: Concolic Verification and Test Generation for GPUs," *Proc. Second ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP),* 2012.
[64] P. Li, G. Li, and G. Gopalakrishnan, "Parametric Flows: Automated Behavior Equivalencing for Symbolic Analysis of Races in CUDA Programs," *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC '12),* 2012.
[65] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner, "Verifying GPU Kernels by Test Amplification," *Proc. 33rd ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '12),* 2012.
[66] M. Zheng, V.T. Ravi, W. Ma, F. Qin, and G. Agrawal, "GMProf: A Low-Overhead Fine-Grained Profiling Approach for GPU Programs," *Proc. 19th Int'l Conf. High Performance Computing (HiPC '12),* 2012.
[67] E.Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, "Streamlining GPU Applications on the Fly: Thread Divergence Elimination through Runtime Thread-Data Remapping," *Proc. 24th ACM Int'l Conf. Supercomputing (ICS),* 2010.
[68] "ATI Stream Technology," http://www.amd.com/stream, 2013.



Mai Zheng received the BS and MS degrees in electronic science and technology from the University of Science and Technology of China and Qingdao University, in 2006 and 2009, respectively. He is currently working toward the PhD degree in the Department of Computer Science and Engineering at the Ohio State University. His research interests include software reliability, high-performance computing, and operating systems. He is a student member of the IEEE.

Vignesh T. Ravi received the PhD degree in computer science and engineering from the Ohio State University in 2012. He is a member of Technical Staff at Advanced Micro Devices. He works on improving programmability, performance and energy efficiency of heterogeneous architectures (mainly CPU-GPU) through system softwares. He is a member of the IEEE.

Feng Qin received the BE, ME, and PhD degrees in computer sciences from the University of Science and Technology of China, Chinese Academy of Sciences, and the University of Illinois at Urbana-Champaign, in 1998, 2001, and 2006, respectively. He is currently an assistant professor in the Department of Computer Science and Engineering at the Ohio State University. He has published papers on top system and architecture conferences in recent years. One of his papers was awarded as best papers in SOSP'05. Two of his papers won IEEE Micro To Picks, in 2004 and 2007, respectively. He received NSF CAREER Award in 2010. His research interests include software reliability, operating systems, and security. He is a member of the IEEE.

Gagan Agrawal received the BS degree from IIT Kanpur, India, in 1991 and the MS and PhD degrees from the University of Maryland, College Park, in 1994 and 1996, respectively. He is a professor of computer science at Ohio State University. His research interests include parallel and distributed computing, cloud computing, and data mining. He has published extensively in these areas.