GRace: A Low-Overhead Mechanism for Detecting Data Races in GPU Programs

Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal

Dept. of Computer Science and Engineering
The Ohio State University
2015 Neil Ave, Columbus, OH, USA
{zhengm, raviv, qin, agrawal}@cse.ohio-state.edu

Abstract

In recent years, GPUs have emerged as an extremely cost-effective means for achieving high performance. Many application developers, including those with no prior parallel programming experience, are now trying to scale their applications using GPUs. While languages like CUDA and OpenCL have eased GPU programming for non-graphical 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. Thus, tool support for detecting race conditions can significantly benefit GPU application developers. Existing approaches for detecting data races on CPUs or GPUs have one or more of the following limitations: 1) being ill-suited for handling non-lock synchronization primitives on GPUs; 2) lacking of scalability due to the state explosion problem; 3) reporting many false positives because of simplified modeling; and/or 4) incurring prohibitive runtime and space overhead.

In this paper, we propose GRace, a new mechanism for detecting races in GPU programs that 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, GRace 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, GRace can accurately detect data races with no false positives reported.

Based on the above idea, we have built a prototype of GRace with two schemes, i.e., GRace-stmt and GRace-addr, for NVIDIA GPUs. Both schemes are integrated with the same static analysis. We have evaluated GRace-stmt and GRace-addr with three data race bugs in three GPU kernel functions and also have compared them with the existing approach, referred to as B-tool. Our experimental results show that both schemes of GRace are effective in detecting all evaluated cases with no false positives, whereas B-tool reports many false positives for one evaluated case. On the one hand, GRace-addr incurs low runtime overhead, i.e., 22-116%, and low space overhead, i.e., 9-18 MB, for the evaluated kernels. On the other hand, GRace-stmt offers more help in diagnosing data races with larger overhead.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.

Categories and Subject Descriptors C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures; C.4 [*Performance of Systems*]: Reliability, availability, and serviceability; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Algorithm, Design, Reliability

Keywords CUDA, Concurrency, Data Race, GPU, Multithreading

1. Introduction

1.1 Motivation

Starting from the last 4-5 years, parallel computation has become essential for taking full advantage of Moore's law. Prior to that, chip designers delivered increases in clock rates and instruction-level parallelism, and enabled single-threaded code to be executed faster. This changed trend is probably best reflected in the cost effectiveness and popularity of GPUs. Through a large number of simple (in-order execution) cores, GPUs achieve very high peak performance with low costs and modest power requirements.

Today, a variety of non-graphical 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. On the one hand, GPUs are becoming part of the high-end systems. For example, in the list of top 500 supercomputers released in May 2010, two of the top seven systems were based on GPUs. On the other hand, a single GPU attached to a desktop or a laptop is being used by application developers who never used parallel machines in the past.

Sustaining the trend towards application acceleration using GPUs or GPU-based clusters will require advanced tool support [33]. Though CUDA [1] and OpenCL [23] have been quite successful, they are both explicitly parallel languages, and pose a programming challenge for those lacking prior parallel programming background. As we stated above, it is common for today's desktops and laptops to have low and medium-end GPUs, making a highly parallel environment accessible and affordable to application developers that never used clusters or SMPs in the past. Developing correct and efficient parallel programs, however, is a formidable challenge for this group of users. Similarly, 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 very experienced parallel programmers. Recently, a number of efforts have been initiated with the goal of automatic code generation for GPUs [26, 31, 40, 44], but these efforts are still in early stages.

Any multithreaded program involves the risk of *race conditions*. A data race occurs when two or more threads access the same memory location concurrently, and at least one of the accesses is a write [39]. As GPUs obtain high performance with a large number of concurrent threads, race conditions can very easily manifest [7, 27]. Once a race condition has occurred, it can lead to program crashes, hangs, or silent data corruption [28]. 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 [9, 39], happens-before based techniques [12, 34, 37], and hybrid schemes combining these two [35, 38, 43]. Lockset based methods maintain a set of locks that protect each shared variable and report a data race if a lockset is empty. Happens-before based techniques maintain the Lamport happens-before relation [25] between different accesses to each shared variable by tracking synchronization events and report a data race if there is no strict order between two accesses to a shared variable. 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 obsolete. Furthermore, GPU programs typically have simple happens-before relation through barriers, which makes existing happens-before based techniques unnecessarily expensive.

Recently, two approaches [7, 27] have been proposed to detect data races in GPU programs. PUG [27], proposed by Li *et al.*, symbolically models GPU kernel functions and leverages Satisfiability Modulo Theories (SMT) solvers to detect data races. While effective in finding subtle data races, PUG may also report false positives due to its approximation of the models. Furthermore, the state explosion of thread interleavings remains a problem for PUG, even though partial order reduction methods can mitigate the problem to some degree.

Boyer *et al.* proposed a dynamic tool to detect data races by tracking all accesses to shared variables at runtime [7]. More specifically, for each access to a shared variable, the tool reports a data race if other threads have accessed the same shared variable after last synchronization call. While the tool can detect data races, it incurs prohibitive (several orders of magnitude) 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. This is because the tool accesses the device memory at least tens of thousands of times for each shared memory access from a kernel function. Additionally, as we will show later, this tool can report many false positives since it does not consider all details of GPU thread scheduling.

As we stated above, one of the examples of race conditions we have observed arises when a third party kernel is incorrectly used by an application developer. Detecting such race conditions requires low-overhead 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 GRace, for accurately detecting data races in GPU programs. GRace exploits the advantages of both static analysis (i.e., no runtime overhead) and dynamic checking (i.e., accurate detection). The main idea is to first detect data races and prune addresses that cannot cause data races by statically analyzing the program. For other memory accesses, GRace instruments the corresponding statements to check data races during program execution.

Our idea is based on the observation that many memory accesses in GPU kernel functions are regular and therefore the memory addresses can be pre-determined at compile time. For example, the statement "$Array[tid] = 3$" assigns 3 to the array element indexed with current thread id. Based on this observation, GRace statically detects, or prunes the possibility of, data races for a majority of the memory accesses. This step is expected to significantly reduce the number of statements that need to be monitored by dynamic checking and therefore improve the overall performance of GRace.

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, GRace instruments the corresponding statements and detects data races at runtime.

Unlike previous work [7], GRace's dynamic checking mechanism is aware of GPU architecture and runtime systems, i.e., memory hierarchy and thread scheduling. By exploiting memory hierarchy, particularly the programmer controlled shared memory, GRace temporarily stores memory access information in GPU's shared memory, which is much faster than GPU's device memory. As a result, GRace can efficiently detect part of data races via faster memory accesses. Similarly, GRace is aware of thread scheduling mechanism and execution model. Therefore, GRace does not report false positives in our experiments.

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

- **Combining static analysis and dynamic checking.** Our proposed GRace utilizes static analysis for improving the performance of a dynamic data race detection tool for GPU programs. Our experimental results show that GRace can efficiently detect data races and benefits significantly from static analysis.
- **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 the execution model. By exploiting the difference, GRace can detect data races more accurately in GPU programs than existing work. Furthermore, we have explored two designs of GRace, including GRace-stmt with more helpful bug diagnostic information, and GRace-addr with lower runtime and space overhead.
- **Exploiting GPU's memory hierarchy.** By leveraging low-latency memory in GPU, GRace detects data races more efficiently. Users can decide whether to enable such feature or not based on their own needs.
- **Implementing and evaluating a prototype of GRace.** We have implemented a prototype of GRace and evaluated it with three GPU kernel functions, including two well-known data-mining algorithms and one used in the previous work in this area. Specifically, we have evaluated the functionality and performance of GRace, as well as benefits from utilizing static analysis and exploiting GPU architecture and runtime systems.

```
1  __device__ void sample_kernel(int* result)
2  {
3        extern __shared__ int s_array[];
4        int tid = threadIdx.x;
5        ...
6        s_array[ tid ] = tid;
7        result[ tid ] = s_array[ tid + 1 ] * tid;
8  }
```

Figure 1. An example of data race in a GPU kernel function.

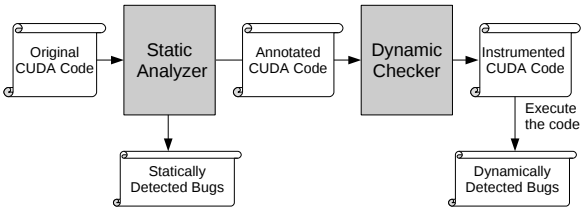

Figure 2. GRace overview. Two shaded blocks are the components of GRace.

2. Background: GPU and Data Race in GPU Programs

2.1 An Example of Data Race on GPU Programs

Figure 1 shows an example of data race in a GPU kernel function. Line 6 writes data to the shared memory address with the index of tid and Line 7 reads the data from the shared memory address with the index of $tid + 1$. This kernel function will be executed by all the threads. It will appear that Threads i and $i+1$ incurs a data race since Thread i reads the shared memory address $s\_array[i + 1]$ at line 7 and Thread $i + 1$ writes the same shared memory address at line 6. However, this is not true for all pairs of threads due to GPU's thread scheduling and the SIMD execution within a warp of threads.

All threads within a warp are scheduled together on one multiprocessor, and executed in the SIMD fashion. Thus, threads within a warp can only cause data races by executing the same instruction. On the contrary, threads across different warps can have data races by executing the same or different instructions. Therefore, in the above case, Threads i and $i+1$ have a data race only if they belong to different warps. This implies that only threads at the boundary of two consecutive warps incur data races. Without such a detailed knowledge of thread scheduling and execution model of GPUs, a tool can easily report many false positives. This, in turn, can make it very hard for a programmer to debug the application.

2.2 GPU Architecture and SIMD Execution Model

A modern Graphical Processing Unit (GPU) architecture consists of two major components, the processing component and the memory component. The processing component in a typical GPU is composed of a certain number of streaming multiprocessors. Each streaming multiprocessor, in turn, contains a set of simple cores that perform in-order processing of the instructions. To support general purpose computations on GPUs, different vendors releases their proprietary software development kits, runtime support, and APIs. These releases include CUDA (from NVIDIA) [1], Stream SDK (from AMD) [2], and OpenCL (from Khronos group) [23]. To achieve high performance, a large number of threads, typically a few thousands, are launched. These threads execute the same operation on different sets of data. A block of threads are mapped to and executed on one streaming multiprocessor. Furthermore, threads within a block are divided into multiple groups, termed as *warp* in CUDA, and *wavefront* in stream SDK and OpenCL. Each warp or wavefront of threads are co-scheduled on a streaming multiprocessor and execute the same instruction in a given clock cycle (SIMD execution). The number of threads/work-items in a warp or a wavefront varies with the design choice. For example, on NVIDIA Tesla cards, the warp size is 32, while the wavefront size on ATI Radeon is 64. The APIs also provide an explicit barrier synchronization mechanism across threads within a block. However, there is no explicit synchronization between blocks, which are implicitly synchronized at the completion of kernel execution.

The memory component of a modern GPU typically contains several layers. One is the *host memory*, which is available on the CPU main memory. This is essential as any general purpose GPU computation can only be launched from the CPU. The second layer is the *device memory*, which resides on the GPU card. This represents the global memory on a GPU and is accessible across all streaming multiprocessors. The device memory is interconnected with the host through a PCI-Express. This interconnectivity enables DMA transfer of data between host and device memory. Note that any access to device memory is of high latency and hence expensive. A scratch memory, which is programmable and of high-speed access, is available privately to a streaming multiprocessor. The scratch memory is termed as *shared memory* on NVIDIA cards and *local data store* on AMD cards. Registers and local memory that is private to each simple core in a streaming multiprocessor is also available.

On NVIDIA GPUs, the size of the shared memory on each multiprocessor was 16 KB till recently (up to the Tesla cards). The latest GPUs from NVIDIA (Fermi) have a slightly different design (with a 64 KB *cache*). This cache can be partitioned into either a 48 KB programmable cache (like shared memory on older cards) and a 16 KB L1 (hardware-controlled) cache, or a 16 KB programmable cache and a 48 KB L1 cache.

For simplicity, we use the terms from NVIDIA to describe our work in the rest of this paper.

3. GRace Design and Implementation

3.1 Design Challenges and GRace 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:

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 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 involve a prohibitive runtime overhead, which is also the key limitation of the previous work in this area [7].

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.

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 the device memory is much slower than the shared memory. As a result, data race detection for

each memory access is expected to be very slow if all profiling data is stored in device memory.

To address the above challenges, GRace exploits static analysis, knowledge of GPU-specific thread scheduling, and the memory hierarchy. As shown in Figure 2, GRace 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.

GRace currently only considers data races in shared memory accesses from GPU kernel functions. In our experience, 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 (Section 3.2) and exploitation of thread scheduling knowledge (Section 3.3.1) are still applicable since these techniques are irrelevant to the locations where races occur.

It should be noted that the latest NVIDIA cards are also supporting 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 application and the architecture.

3.2 Static Analyzer

In this subsection, we describe the algorithm for 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 access that cannot be resolved, are instrumented. Another key goal of our static analysis is to help further reducing the overhead of dynamic instrumentation. To achieve this goal, our static analysis determines whether the same address is accessed across iterations and/or threads. If so, dynamic checker can perform the instrumentation for only certain threads and/or iterations so that the runtime and space overheads are drastically reduced.

Our overall analysis algorithm for a synchronization block is shown in Algorithm 1. 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, 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 comprising a left-hand-side and a right-hand-side memory accesses, 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 intra-warp 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 *inter-warp* race if the conflict arises between threads with identifiers that are across different warps.

Algorithm 1 Static Analyzer

```
1:  for each synchronizationblock do
2:    for each pair of write − write and read − write accesses
      do
3:      if static_determinable(sh_mem_addr) then
4:        lin_constraints[] = gen_constraints(sh_mem_addr,
          tid, iter_id)
5:        add_params_to_constraints(tid, iter_id,
                max_thread_id, max_iteration)
6:        soln_set[] = constraint_solver(lin_constraints[])
7:        if empty(soln_set[]) then
8:          No data race
9:        else
10:         Extract conflicting tid_x, tid_y from soln_set[]
                and report inter/intra warp data races
11:       end if
12:     else
13:       Mark the pair of accesses as 'Montior at Runtime'
14:       for each of the two accesses do
15:         if check_loop_invariant(sh_mem_addr) then
16:           Mark as loop invariant
17:         end if
18:         if check_thread_invariant(sh_mem_addr) then
19:           if is_write(sh_mem_addr) then
20:             Report Intra-warp data races
21:           else
22:             Mark as thread invariant
23:           end if
24:         end if
25:       end for
26:     end if
27:   end for
28: end for
```

If the address can not 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. Towards 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 *inter-warp* 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 thread invariant) is not protected by atomic operation we report an intra-warp 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. We later demonstrate the benefits of these optimizations in the experimental results section.

Now, we consider a case study with *Co-clustering* [8], a popular data mining algorithm. Co-clustering considers clustering along two dimensions, i.e., original points can be viewed as being in a two dimensional array. We use a GPU implementation of this code that was aggressively optimized for shared memory use in a recent

```
1   void rowClusterCount(float* data, int numRow, int numRowCluster,
          int numColCluster, int numCol, float* Acompressed, float*
          rowQuality4Compressed, int* rowCL, int* colCL, int* rowCS_l)
2   {     //Args of kernel are in device memory

          //Declaration of shared memory arrays
3         extern __shared__ float s_float[];
4         float* Acomp = s_float;
5         float* rowQual = s_float + numColCluster * numRowCluster;
6         float* rowCS = s_float + numColCluster * numRowCluster +
                         numRowCluster;
7         float* sh_rowCL = s_float + numColCluster * numRowCluster +
                            numRowCluster + numRowCluster;

          /* Omitting other declarations and shared memory initialization */

          //Computation and reduction on shared memory
8         for (int r = 0; r < numRow; r += rowcluster_THREADS *
              rowcluster_BLOCKS)
9         {
10          for (int rc = 0; rc < numRowCluster; rc++)
11          {
12              if (rowCS[rc] > 0)
13              {
14                  for (int c = 0; c < numCol; c++)
15                  {
16                      tempDistance += data[(r+n_index)*numCol+c] *
                                        Acomp[rc*numRowCluster + colCL[c]];
17                      tempDistance = -2*tempDistance + rowQual[rc];
18                  }
29              }
20          }
21          sh_rowCL[r+tid] = minCL;
22      }
23  }
```

Figure 3. A simplified code of Co-clustering kernel.

study [32]. A simplified portion of this GPU kernel code is shown in Figure 3. From the code, we can see that there are four arrays declared in the shared memory (lines 3-7) and there is one nested *for* loop (lines 8-22). Looking into the memory accesses within those four arrays on shared memory, three of those arrays (*rowCS* at line 12, *Acomp* at line 16, and *rowQual* at line 17) are read-only and the last one (*sh_rowCL* at line 21) is write only. Thus, there are three pairs of memory addresses that needs to be analyzed for data races. Next, for each of the three pairs of memory addresses, we need to generate linear constraints. These linear constraints are created as a function of *tid* and *loop index*. Note that, all the four shared memory arrays point to the base address of *s_float* with some offsets. Thus, this offset translation is also accounted for when formulating the linear constraints. Additionally, we also provide some ranges for the parameters that are in the linear constraints. For example, let us consider *rowCS[rc]*. First, rowCS is translated based on its offset from the base address of *s_float*. Then, the constraint is generated with *rc* as an unknown. However, the range of values that *rc* can take is known from the loop initialization. Thus, this range will be parameterized into the constraint. The other variables in the subscript, *numColCluster* and *numRowCluster* are constants taken from the user input. Once a constraint is developed and parameterized, it is fed into the linear constraint solver. The solution output is analyzed to detect any inter/intra data races. It turns out, in this example, there cannot be any data races between reads on *rowCS* and writes to *sh_rowCL*, and between reads on *rowQual* and writes to *sh_rowCL*.

A specific case arises with respect to a variable like *Acomp* where the subscript variable contains *indirect accesses* (*colCL[c]*). In this case, the static analysis cannot resolve the memory access, and we mark the address to be instrumented by the dynamic checker. Since this memory access can have a conflict with *sh_rowCL*, it is also marked for instrumentation.

It appears that our static analysis can at most reduce the accesses to be instrumented from four to two. However, static analysis can extract other information that can be used by the dynamic checker.

Algorithm 2 Intra-warp Race Detection

```
1:  if accessType is read then
2:      Stop execution of this algorithm
3:  end if
4:  for targetIdx = 0 to warpSize − 1 do
5:      sourceIdx ← tid % warpSize
6:      if sourceIdx ≠ targetIdx then
7:          if warpTbl[sourceIdx] =
              warpTbl[targetIdx] then
8:              Report a Data Race
9:          end if
10:     end if
11: end for
```

For example, accesses to *Acomp* are invariant across all the threads in the block. Thus, it is sufficient to monitor *Acomp* for only one thread. Furthermore, *Acomp* needs to be monitored only once in the outer loop, i.e., when $r = 0$, since accesses to *Acomp* are invariant across iterations of the loop.

3.3 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 *intra-warp* data races, which are caused by the same statement from multiple threads within a warp. Furthermore, Dynamic Checker instruments every synchronization call to detect *inter-warp* 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 intra-warp and inter-warp races, GRace improves detection accuracy, i.e., no false positives caused by different instructions from intra-warp threads accessing the same address. Another benefit of this separation is that GRace can perform fast intra-warp race detection after each memory access and delay the slow inter-warp race detection to each synchronization call. Furthermore, GRace can utilize a small chunk of shared memory to temporarily store memory accesses from a warp of threads and thereby speed up intra-warp race detection.

3.3.1 Intra-warp Race Detection

GRace 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, GRace 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, GRace performs intra-warp race detection as described by Algorithm 2. More specifically, it first checks whether the access type is read (line 1). If yes, the checking process stops (line 2) since it is impossible to have races only through reads. Otherwise, GRace scans the table (line 4) to check whether two threads access the same memory address (line 5-7). If so, GRace reports a data race with the executed statement and racing thread ids (line 8). All threads within a warp execute the above steps in parallel. Note that GRace requires no explicit synchronization between updating a warpTable and detecting intra-warp races since both operations will be executed se-

Algorithm 3 Inter-warp Race Detection by GRace-stmt

```
1: for stmtIdx1 = 0 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 15
5:     end if
6:     if BlkStmtTbl[stmtIdx1].accessType is read and
         BlkStmtTbl[stmtIdx2].accessType is read then
7:       Jump to line 15
8:     end if
9:     for targetIdx = 0 to warpSize − 1 do
10:      sourceIdx ← tid % warpSize
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
```

quentially (SIMD) by all threads within a warp. This is important because inserted synchronization may lead to deadlock if the statement is a conditional branch executed by a subset of threads within a warp.

After performing intra-warp race detection, GRace transfers memory access information from the warpTable, fully or partially, to device memory for future inter-warp race detection, which is discussed in Section 3.3.2. As a result, GRace 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 intra-warp race detection minimal. Our experimental results have shown that typically 1 KB can hold the warpTables for all the warps on Tesla C1060 (More details in Section 5). Thus, GRace only incurs 6% 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%. If running legacy GPU kernel functions that assume 16 KB shared memory, GRace 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, GRace can store the warpTables in device memory and performs intra-warp race detection there.

3.3.2 Inter-warp Race Detection

GRace periodically detects inter-warp races after each synchronization call. More specifically, GRace transfers the memory access information from a warpTable to device memory after each intra-warp race detection. After each synchronization call, GRace identifies inter-warp races by examining memory accesses from multiple threads that are across different warps. After detecting inter-warp races at one synchronization call, GRace 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 inter-warp detection schemes. One scheme 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 organizes memory access information by 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.

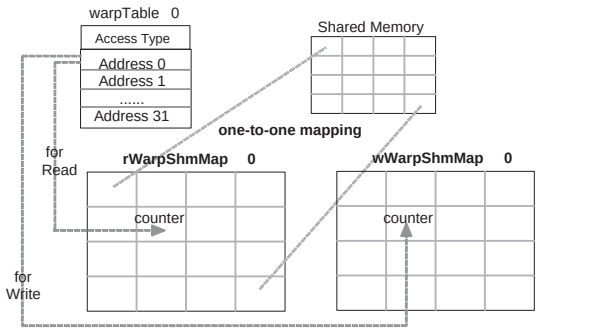

Figure 4. Data structures of a rWarpShmMap and a wWarpShmMap. Each WarpShmMap or BlockShmMap entry is corresponding to one shared memory address in one-to-one mapping. Each address stored in a warpTable is used as the index to increase the corresponding counter in the rWarpShmMap and rBlockShmMap, or wWarpShmMap and wBlockShmMap, depending on the access type. Note that rBlockShmMap and wBlockShmMap are not shown here due to space limit.

The statement-based scheme (GRace-stmt). This scheme of GRace (referred to as GRace-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, GRace-stmt maintains a BlockStmtTable in device memory for the threads from all the warps that can access the same shared memory. Each entry 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. Essentially, GRace-stmt organizes a BlockStmtTable by memory access statements from all the threads.

At each synchronization call, GRace-stmt scans the entire BlockStmtTable and identifies inter-warp data races as described in Algorithm 3. More specifically, GRace-stmt checks two BlockStmtTable entries at a time throughout the entire table (line 1-2). For each pair of the entries, GRace-stmt checks both the warp IDs and access types (line 3-8). If the warp IDs are the same or both access types are read, GRace-stmt skips this pair since any pair of memory accesses from both entries cannot cause inter-warp data races. Otherwise, GRace-stmt utilizes a warp of threads in parallel to check whether there are two addresses, one from each entry, are the same (line 9-14). Once the same addresses are found, GRace-stmt reports a data race (line 12).

On the one hand, GRace-stmt provides accurate diagnostic information about a detected race, including the pair of racing statements (i.e., the indexes of the BlockStmtTable entries), 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 algorithm complexity of GRace-stmt 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 GRace-stmt is linear to the number of BlockStmtTable entries. Although this indicates that GRace-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 5).

The address-based scheme (GRace-addr). This scheme of GRace (referred to as GRace-addr) stores summarized information of the memory addresses that have been accessed from all the threads and detects data races based on the summarized information. More specifically, GRace-addr maintains two tables for each warp, one

Algorithm 4 Inter-warp Race Detection by GRace-addr

```
1: for idx = 0 to shmSize − 1 do
2:     if wBlockShmMap[idx] = 0 then
3:         Jump to line 15
4:     end if
5:     if rWarpShmMap[idx] = 0 and
         wWarpShmMap[idx] = 0 then
6:         Jump to line 15
7:     end if
8:     if wWarpShmMap[idx] ≤ wBlockShmMap[idx] and
         wWarpShmMap[idx] > 0 then
9:         Report a Data Race
10:    else if wWarpShmMap[idx] = 0 then
11:        Report a Data Race
12:    else if rWarpShmMap[idx] ≤
             rBlockShmMap[idx] then
13:        Report a Data Race
14:    end if
15: end for
```

for read access from threads within the warp (referred to as rWarpShmMap) and the other for write access (referred to as wWarpShmMap). Additionally, for all the warps that can access the same shared memory, GRace-addr maintains two tables, one for read access from all such warps of threads (referred to as rBlockShmMap) and the other for write access (referred to as wBlockShmMap). Each entry in any of these tables maps to one shared memory address linearly. Specifically, each entry stores a counter that records the number of accesses to the corresponding shared memory address from all the threads within a warp (for rWarpShmMap or wWarpShmMap) or across all warps (for rBlockShmMap or wBlockShmMap). Figure 4 shows the data structures of rWarpShmMap and wWarpShmMap, which are the same as rBlockShmMap and wBlockShmMap.

After performing each intra-warp race detection, GRace-addr transfers memory access information in the warpTable to the corresponding rWarpShmMap and rBlockShmMap for read, or to the corresponding wWarpShmMap and wBlockShmMap for write. More specifically, GRace-addr scans through the warpTable and, for each warpTable entry, adds one to the value of the corresponding counter in the rWarpShmMap and rBlockShmMap, or wWarpShmMap and wBlockShmMap. Essentially, these tables keep the number of accesses to each shared memory address for different warps, individually and aggregately.

At each synchronization call, GRace-addr detects inter-warp races as described in Algorithm 4. More specifically, GRace-addr scans through all the counters stored in the rWarpShmMap and the wWarpShmMap for each warp in parallel (line 1). For each shared memory address, GRace-addr first rules out the cases of all read accesses (line 2-4) and no accesses from the local (or current) warp (line 5-7). Then GRace-addr detects data races of write-write conflict, i.e., writes from both local and remote warps (line 8-9), followed by read-write conflict, i.e. read from local warp and write from remote warp (line 10-11). Lastly, GRace-addr detects data races of write-read conflict, i.e., write from local warp and read from remote warp (line 12-13).

On the one hand, the time and space complexities of the GRace-addr algorithm are linear to the size of shared memory, which is constant for a given GPU. Therefore, GRace-addr 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, GRace-addr provides aggregated information about a data race, which is less accurate than GRace-stmt. For example, GRace-addr reports racing memory address and the pairs of racing warps instead of racing statements or racing threads. However, the bug information provided by GRace-addr is still useful. For example, programmers can narrow down the set of possibly racing statements based on a racing memory address reported by GRace-addr. Similarly, programmers can derive racing threads based on the ranges of reported racing warps.

4. 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. This GPU was connected to a machine with two AMD 2.6 GHz dual-core Opteron CPUs and 8 GB main memory. We have implemented the prototype of GRace. Static Analyzer utilizes the linear constraint solver [16], and Dynamic Checker is built on CUDA SDK 3.0. Note that we do not see any particular difficulty to port GRace to other GPU environments such as stream SDK or OpenCL.

We have evaluated GRace with three applications, including Co-clustering [8] (referred to as `co-cluster`), EM clustering [10] (referred to as `em`), and scan code [7] (referred to as `scan`). Among these applications, `co-cluster` and `em` are both clustering (data mining) algorithms. We have used GPU implementations of these applications that were aggressively optimized for shared memory use in a recent study [32]. `scan` is the simple code used in previous work on GPU race detection [7].

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 [7].

We have designed four sets of experiments to evaluate the key aspects of GRace:

- The first set evaluates the functionality of GRace in detecting data races of GPU kernel functions. We compare GRace with the previous approach [7], referred to as B-tool in this paper, 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 GRace in terms of execution time of kernel functions. We also compare GRace with B-tool. Additionally, we evaluate the space overhead caused by GRace and compare it with B-tool. 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 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 GRace with and without Static Analyzer.
- The fourth set evaluates the benefit of shared memory. We measure the runtime overhead of GRace 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 GRace-stmt and GRace-addr, both with the same intra-warp detection scheme and the same Static Analyzer.

Apps	GRace-stmt				GRace-addr				B-tool		
	R-Stmt#	R-Mem#	R-Thd#	FP#	R-Stmt#	R-Mem#	R-Wp#	FP#	R-Stmt#	RP#	FP#
co-cluster	1	10	1,310,720	0	-	10	8	0	-	1	0
em	14	384	22,023	0	-	384	3	0	-	200,445	45,870
scan	3 pairs of racing statements are detected and all addresses are resolved by Static Analyzer								-	Err*	Err*

Table 1. Overall effectiveness of GRace for data race detection. We compare the detection capability of GRace-stmt and GRace-addr with that of B-tool, the tool proposed by previous work [7]. 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, FP means false positives, and RP means the race number reported by B-tool. '-' means the data is not reported by the scheme. * 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.

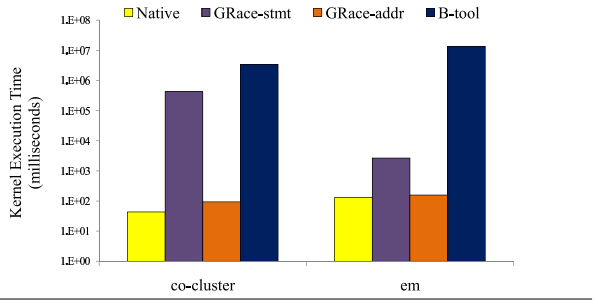

Figure 5. Runtime overhead of GRace. Note that the y-axis is on a logarithmic scale.

5. Experimental Results

5.1 Overall Effectiveness

Table 1 demonstrates the overall effectiveness of GRace. Specifically, we evaluate three schemes, including GRace-stmt, GRace-addr, and the previous work B-tool [7]. For GRace-stmt and GRace-addr, we measure four metrics, including the number of pairs of racing statements, the number of memory addresses involved in data races, the number of pairs of threads or warps in race conditions, and the number of false positives within the reported pairs of threads or warps. For B-tool, we present the number of data races reported by the tool. Unlike GRace-stmt or GRace-addr, 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 the kernel functions, we use bug-triggering parameters or inputs to trigger the data races. For `co-cluster`, we launch 8 blocks with 256 threads per block. For `em`, we launch 8 blocks with 320 threads per block. For `scan`, 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 GRace-stmt and GRace-addr can effectively detect data races. For example, among the reported data races, there are no false positives for both GRace-stmt and GRace-addr. On the contrary, B-tool generates 45,870 false positives, among the reported 200,445 data races for `em`. GRace-stmt and GRace-addr are accurate because both schemes leverage the knowledge of GPU's thread scheduling and SIMD execution model. As a result, GRace does not check memory accesses issued from different instructions that are executed by different threads within a warp, which are the sources of false positives reported by B-tool. Due to B-tool's incorrect use of inserted synchronization calls for the instrumentation code, it could not be run for `scan` on the new hardware and software.

Table 1 indicates that GRace-stmt provides more accurate information about data races than GRace-addr and B-tool do. Since GRace-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, GRace-addr and B-tool cannot report the pair of statements involved in a race, since they do not keep information of the statements that have accessed the conflicting memory addresses before. Furthermore, GRace-addr reports only the pairs of racing warps, which are coarser-grained than what is available from GRace-stmt and B-tool. However, diagnostic information provided by GRace-addr 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.

Apps	GRace-stmt		GRace-addr		B-tool
	ShM	DM	ShM	DM	Mem*
co-cluster	1.1 KB	43 MB	1.1 KB	9 MB	257 MB
em	1.1 KB	0.8 MB	1.1 KB	18 MB	514 MB

Table 2. Space overhead of GRace. ShM means shared memory, DM is device memory, and Mem is (host) memory. * B-tool is running in emulation mode, which does not require shared memory or device memory.

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 resolves all memory addresses. Therefore, it totally eliminates the overhead of running Dynamic Checker for this application.

5.2 Runtime Overhead

We measure the execution time for `co-cluster` and `em` in four configurations: executing the kernels on GPU natively without any instrumentation, executing the kernels with GRace-stmt on GPU, executing the kernels with GRace-addr 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 [7]. For both kernels, we use *normal inputs*, i.e. those that do not trigger data races, for these experiments. Note that GRace does not have runtime overhead for `scan` since the Static Analyzer did not annotate any statements.

Figure 5 shows that GRace-addr and GRace-stmt incur lower runtime overhead than the B-tool. For example, GRace-addr and GRace-stmt slow down `em` by 22% and 19 times, 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 GRace and B-tool. First, GRace-addr and GRace-stmt utilize static analysis to significantly reduce the number of memory accesses that need to be checked dynamically. Second, both GRace-addr and GRace-stmt delay inter-warp 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.

Figure 5 also indicates that GRace-addr is significantly more efficient than GRace-stmt. For example, GRace-addr slows down `em` and `co-cluster` by 22% and 116%, respectively, while GRace-stmt slows down `em` and `co-cluster` by 19 times and 9,862 times, respectively. This is mainly because GRace-addr's inter-warp race detection algorithm runs in a constant amount of time, i.e. it does not depend on the execution number of instrumented statements. Whereas, the complexity of GRace-stmt's inter-warp race

Apps	w/o Static Analyzer				with Static Analyzer			
	Static #		Dynamic #		Static #		Dynamic #	
	Stmt	MemAcc	Stmt	MemAcc	Stmt	MemAcc	Stmt	MemAcc
co-cluster	8	8	10,524,416	10,524,416	2	2	41,216	41,216
em	7	13	19,070,976	54,460,416	7	13	20,736	10,044

Table 3. Benefits from Static Analyzer, which applies to both GRace-stmt and GRace-addr. Stmt means statements and MemAcc means memory access. Static# means the number of instrumented statements or memory accesses at compile time. Dynamic # means the number of instrumented statements or memory accesses that are executed at program runtime.

Apps	Native (ms)	GRace-stmt (ms)		GRace-addr (ms)	
		w/o SA	w/ SA	w/o SA	w/ SA
co-cluster	43.5	Abt*	429,502	10,889	93.9
em	130	Abt*	2,655	47,861	158

Table 4. Runtime overhead of GRace-stmt and GRace-addr with and without applying Static Analyzer. SA means Static Analyzer. * Jobs for GRace-stmt without Static Analyzer were terminated after several hours due to out of memory.

Apps	GRace-stmt (ms)		GRace-addr (ms)	
	w/o ShM	w/ ShM	w/o ShM	w/ ShM
co-cluster	433,056	429,502	120.8	93.9
em	2,764	2,655	158.5	158.3

Table 5. Runtime overhead of GRace-stmt and GRace-addr with and without utilizing shared memory. ShM means shared memory.

detection algorithm is quadratic with respect to the execution number of instrumented statements. Both GRace-addr and GRace-stmt use the same intra-warp race detection method, whose complexity is linear with respect to the number of dynamic memory accesses. Note that, intra-warp race detection, which is performed in shared memory, is much faster than inter-warp race detection, which is performed in device memory.

Overall, we can see that GRace-addr'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 GRace-stmt, and collect more detailed information to help debugging the application.

5.3 Space Overhead

Table 2 presents the memory space overheads incurred by GRace-stmt, GRace-addr, and B-tool. GRace-addr incurs much smaller space overhead than B-tool. For example, GRace-addr requires around 9 MB and 18 MB for `co-cluster` and `em`, respectively, while B-tool requires 257 MB and 514 MB for the two kernels, respectively. The reason is that B-tool maintains two bookkeeping tables for each thread, where each table is four times the size of the shared memory, and another two tables for each block. Unlike B-tool, GRace-addr maintains two bookkeeping tables for each warp, which has 32 threads in our platform, and two tables for each block. Therefore, GRace-addr incurs a factor of 28 less space overhead than B-tool.

Table 2 shows that GRace-stmt incurs lower space overhead than GRace-addr when the execution number of instrumented statements is small. For example, GRace-stmt requires 0.8 MB for `em`, while GRace-addr needs 18 MB for the same kernel function. This is because the space required by GRace-stmt increases linearly with the execution number of instrumented statements, while the space required by GRace-addr depends on the number of warps and the size of shared memory. Therefore, we can choose GRace-stmt for detecting data races in kernel functions where the instrumented statements are expected to execute for only a few times.

5.4 Benefits from Static Analysis

To evaluate benefits from static analysis, we use four metrics, which are the number of instrumented statements, the number of instrumented shared memory accesses (one statement could result in multiple memory accesses), the number of dynamically-executed instrumented statements, and the number of dynamically-executed instrumented shared memory accesses. Note that the above numbers are aggregated from all the threads within a block.

Table 3 shows that GRace benefits significantly from static analysis, which applies to both GRace-stmt and GRace-addr in the same way. For example, Static Analyzer reduces the number of statements and memory accesses that need to be instrumented for `co-cluster` from 8 to 2. More importantly, the execution numbers of instrumented statements and memory accesses for both `co-cluster` and `em` are drastically reduced, which in turns reduces the runtime overhead for GRace.

There are three main sources for the runtime overhead reduction. The first one is memory accesses that are statically determined as races or involving in no races. GRace does not need to instrument such memory accesses. The second source is memory addresses that are invariant across threads. In this case, GRace requires only one thread to instrument and check data races for relevant memory accesses. This, for example, reduces the total number of `co-cluster`'s dynamic execution of instrumented memory accesses by a factor of 256. The third one is memory addresses that are invariant to loop iterators. This means GRace only needs to instrument the relevant memory accesses for one iteration. This helps particularly with `em`, where we reduce the total number of dynamic execution of instrumented memory accesses by a factor of 5,422.

Table 4 further indicates the benefits from static analysis. It shows the runtime overhead incurred by GRace-stmt and GRace-addr with and without applying Static Analyzer. For example, GRace-addr incurs 302 times overhead for `em` when Static Analyzer is not used, while the slowdown is only by 22% when Static Analyzer is used.

5.5 Benefits from Shared Memory

To evaluate benefits from shared memory, we run the kernels with two versions of GRace-stmt and GRace-addr, respectively. One version logs memory accesses for a warp, and detects intra-warp races, in shared memory. The second version performs logging and detection for intra-warp races in the device memory.

Table 5 shows that shared memory improves the performance of race detection for both GRace-stmt and GRace-addr. For example, GRace-addr in shared memory slows down `co-cluster` by 116%, while slowing down the same kernel by 177% with device memory only. This is an expected result, since shared memory is much faster than the device memory.

Table 5 also shows that the performance gains for different applications or different race detection schemes varies. For example, performance improvement caused by shared memory for GRace-addr on `em` is negligible, while the improvement for GRace-addr on `co-cluster` is 22%. This is mainly because shared memory is only used for intra-warp race detection for both GRace-addr and GRace-stmt, and inter-warp race detection is still in device memory. As a result, the benefit from shared memory becomes more pronounced with larger number of instrumented statement being executed.

6. Related Work

GRace is related to previous studies on data race detection, bug detection for parallel and distributed programs, tool development for GPU programming, and optimizations of GPU programs. Due to space limit, this section briefly describes these studies.

Data race detection. Besides the dynamic race detection techniques discussed in previous section [9, 12, 34, 35, 37–39, 43], researchers proposed static methods for data race detection, including static lockset algorithm [13] and race type-safe systems [6, 18]. Without runtime information, static methods may generate many false positives. Additionally, researchers also proposed to detect data races using model checking [20], which has the limitation of state explosion problem in general. Furthermore, happens-before relation has also be applied to detect races in OpenMP programs [22]. Unlike these approaches, our work focuses on detecting races in GPU programs, which have different characteristics to deal with. To manage race or contention of shared resources, new OS schedulers have been proposed [17].

Bug detection for parallel and distributed programs. Many approaches monitor program execution to detect bugs in parallel and distributed programs [11, 15, 19, 21, 24, 29, 41, 42]. Various techniques [36, 45] have been proposed to reduce the cost of program monitoring. Additionally, interactive parallel debuggers [3–5, 14, 30] 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.

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 [7, 27]. In addition, there has been one recent effort on performance measurement on CUDA programs [33]. Malony *et al.* have developed TAUcuda, which captures CUDA performance events in profile and trace forms. The distinct feature of this tool is that it does not require instrumentation, i.e., neither the source code nor the binary code is modified. However, it is based on an experimental version of CUDA driver, which is not widely available. This tool is also not designed for race detection.

Optimizations of GPU programs. There have been many efforts on optimizations of GPU programs and code generation for GPU programs [26, 31, 40, 44, 46], but none of them have focused on program correctness issues.

7. Conclusions

In this paper, we have presented GRace, a low-overhead approach for detecting data races in GPU programs. GRace first utilizes static analysis to prune memory accesses that are either definitely involved in data races, or can never be in any data race. After this, GRace instruments the unresolved memory accesses for detecting intra-warp and inter-warp data races, exploiting the knowledge of the GPU architecture and the execution model. Furthermore, to detect inter-warp data races, we have explored two designs, i.e. GRace-stmt with more accurate bug diagnostic information, and GRace-addr with lower runtime and space overhead.

We have built a prototype of GRace and have evaluated GRace with three data race bugs in three GPU kernel functions. We have also compared both schemes of GRace, i.e., GRace-stmt and GRace-addr, with an existing dynamic method, B-tool. Our experimental results have shown that both GRace-stmt and GRace-addr can detect data races effectively and efficiently without reporting any false positives, while B-tool slows down the program significantly and can report many false positives. Furthermore, the experimental results have demonstrated that GRace-addr incurs low runtime (22-116%) and space overhead (9-18 MB), while being able to detect race conditions in target programs. The only limitation is that it provides less accurate bug reports. GRace-stmt, in comparison, incurs larger runtime (19-9,862 times) and space (0.8-43 MB) overheads, while providing more accurate bug reports. Moreover, the experimental results have indicated the benefits of the use of static analysis and careful use of memory hierarchy at runtime, for both GRace-stmt and GRace-addr.

Overall, as GRace-addr's runtime overheads are very modest, it is suitable for an end-user to test a full application. GRace-stmt can be invoked when a race condition is detected in a specific kernel, and can provide more detailed information for debugging.

8. Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback. We appreciate the useful discussion with Wenjing Ma, Xin Huo, Wenbin Zhang and Zhezhe Chen. This work was supported in part by an allocation of computing time from the Ohio Supercomputer Center, and by the NSF grants #CCF-0833101 and #CCF-0953759 (CAREER Award).

References

[1] CUDA Community Showcase. http://www.nvidia.com/object/cuda_apps_flash_new.html.

[2] ATI Stream Technology. http://www.amd.com/stream.

[3] 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. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2009.

[4] 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. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[5] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden. Extending a traditional debugger to debug massively parallel applications. *J. Parallel Distrib. Comput.*, 64(5), 2004.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2002.

[7] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of CUDA programs. In *Proceedings of the Third Workshop on Software Tools for MultiCore Systems (STMCS)*, 2008.

[8] H. Cho, I. S. Dhillon, Y. Guan, and S. Sra. Minimum sum-squared residue co-clustering of gene expression data. In *Proceedings of the 4th SIAM International Conference on Data Mining (SDM)*, 2004.

[9] 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. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*, 2002.

[10] A. Dempster, N. Laird, and D. Rubin. Maximum Likelihood Estimation from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[11] 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. In *Proceedings of the 2nd International workshop on Software engineering for high performance computing system applications (SE-HPCS)*, 2005.

[12] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the ACM*

SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP), 1990.

[13] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[14] Etnus, LLC. TotalView. http://www.etnus.com/TotalView.

[15] C. Falzone, A. Chan, E. Lusk, and W. Gropp. A portable method for finding user errors in the usage of MPI collective operations. *International Journal of High Performance Computing Applications*, 21(2):155–165, 2007.

[16] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[17] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2): 49–57, 2010.

[18] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[19] Q. Gao, F. Qin, and D. K. Panda. DMTracker: Finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Proceedings of the ACM/IEEE Annual Conference on Supercomputing (SC)*, 2007.

[20] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[21] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller. A graph based approach for MPI deadlock detection. In *Proceedings of the 23rd International Conference on Supercomputing(ICS)*, 2009.

[22] M.-H. Kang, O.-K. Ha, S.-W. Jun, and Y.-K. Jun. A tool for detecting first races in openmp programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[23] Khronos Group. OpenCL: The Open Standdard for Heterogeneous Parallel Programming. http://www.khronos.org/opencl, 2008.

[24] B. Krammera, K. Bidmona, M. S. Muller, and M. M. Rescha. MARMOT: An MPI analysis and checking tool. In *Parallel Computing (PARCO)*, 2003.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[26] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[27] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2010.

[28] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[29] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurr. Comput. Pract. Exp.*, 15(2), 2003.

[30] S. S. Lumetta and D. E. Culler. The mantis parallel debugger. In *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, 1996.

[31] W. Ma and G. Agrawal. A translation system for enabling data mining applications on GPUs. In *Proceedings of the International Conference on Supercomputing(ICS)*, 2009.

[32] W. Ma and G. Agrawal. An Integer Programming Framework for Optimizing Shared Memory Use on GPUs. In *Proceedings of International Conference on High Performance Computing(HiPC)*, 2010.

[33] A. D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam. An experimental approach to performance measurement of heterogeneous parallel applications using cuda. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2010.

[34] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1991.

[35] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[36] J. Odom, J. K. Hollingsworth, L. DeRose, K. Ekanadham, and S. Sbaraglia. Using dynamic tracing sampling to measure long running programs. In *Proceedings of the ACM/IEEE Annual Conference on Supercomputing (SC)*, 2005.

[37] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[38] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[39] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[40] N. Sundaram, A. Raghunathan, and S. Chakradhar. A framework for efficient and scalable execution of domain-specific templates on GPUs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[41] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2000.

[42] S. Yang, A. R. Butt, Y. C. Hu, and S. P. Midkiff. Trust but verify: monitoring remotely executing programs for progress and correctness. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[43] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[44] S. zee Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu. CUDA-lite: Reducing GPU Programming Complexity. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2008.

[45] A. Zhai, G. He, and M. Heimdahl. Hardware and compiler support for dynamic software monitoring. In *Proceedings of the International Workshop on Runtime Verification (RV)*, 2009.

[46] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining gpu applications on the fly: thread divergence elimination through runtime thread-data remapping. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2010.