LD: Low-Overhead GPU Race Detection Without Access Monitoring

PENGCHENG LI, XIAOYU HU, DONG CHEN, JACOB BROCK, and HAO LUO,
University of Rochester
EDDY Z. ZHANG, Rutgers University
CHEN DING, University of Rochester

Data race detection has become an important problem in GPU programming. Previous designs of CPU race-checking tools are mainly task parallel and incur high overhead on GPUs due to access instrumentation, especially when monitoring many thousands of threads routinely used by GPU programs.

This article presents a novel data-parallel solution designed and optimized for the GPU architecture. It includes compiler support and a set of runtime techniques. It uses value-based checking, which detects the races reported in previous work, finds new races, and supports race-free deterministic GPU execution. More important, race checking is massively data parallel and does not introduce divergent branching or atomic synchronization. Its slowdown is less than 5× for over half of the tests and 10× on average, which is orders of magnitude more efficient than the cuda-memcheck tool by Nvidia and the methods that use fine-grained access instrumentation.

CCS Concepts: • **Computing methodologies** → **Graphics processors**; • **Software and its engineering** → **Software performance;**

Additional Key Words and Phrases: GPU race detection, low overhead, value-based checking, instrumentation-free

ACM Reference Format:
Pengcheng Li, Xiaoyu Hu, Dong Chen, Jacob Brock, Hao Luo, Eddy Z. Zhang, and Chen Ding. 2017. LD: Low-overhead GPU race detection without access monitoring. ACM Trans. Archit. Code Optim. 14, 1, Article 9 (March 2017), 25 pages.
DOI: http://dx.doi.org/10.1145/3046678

1. INTRODUCTION

Graphics processing units (GPUs) rely heavily on the programmer to realize their high performance potential. One pitfall that the programmer must be aware of is data races. While this concern exists for parallel programs with a handful of threads running on traditional single-core processors or multicore processors, a GPU programmer must consider the interaction between thousands of threads; any two may access the same memory cell and trigger a data race.

The research is supported in part by the National Science Foundation (Contract No. CCF-1629376, CNS-1319617, CCF-1116104), IBM CAS Faculty Fellowship, and a grant from Huawei. Dong Chen is supported in part by the Chinese Scholarship Council. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.
Authors' addresses: P. Li, X. Hu, D. Chen, J. Brock, H. Luo, and C. Ding, University of Rochester, P.O.Box 270226, CSB bldg, Rochester, NY, 14627; emails: pli@cs.rochester.edu, {hxy9243, jameschennerd}@gmail.com, {jbrock, hluo, cding}@cs.rochester.edu; E. Z. Zhang, Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019; email: eddy.zhengzhang@cs.rutgers.edu.

DOI: http://dx.doi.org/10.1145/3046678

In this article, we present a low-overhead race detector called *LDetector* (LD). LD analyzes a GPU program on a GPU. LD includes compiler support and a runtime library. The main function of the compiler is privatization. When executed, each thread group (a group of threads executing in lockstep, called a *warp* in CUDA) uses a private copy of the shared data, so its writes are not visible to other warps. After execution, the runtime detector examines shared data accesses for data races. LD is a *precise* dynamic race detector, meaning that it may have false negatives but no false positives [Flanagan and Freund 2009].

LD runs a target program twice. In the first pass, it detects the writes of a warp by comparing its private copy with the stale values. The writes of all warps are then compared to find write-write races. In the second pass, it reruns a warp with the new values from other warps and compares the second-run results with those of the first run. The first pass can be adapted to support safe parallelism, by which we mean either race-free or deterministic parallelism.

Existing dynamic detectors rely on access monitoring. They use a compiler to instrument program loads and stores (that may access shared data). During execution, the instrumented accesses are recorded in some form of metadata. The metadata may be shared or private. Shared metadata, for example, states of variables in FastTrack [Flanagan and Freund 2009], can detect concurrent accesses directly. However, they require atomic access, which is costly on GPUs because of the high degree of parallelism. Private metadata, for example, read/write sets in TARDIS [Lu et al. 2014], avoid atomics but still need instrumentation of each load/store. At a minimum, it turns every read into a read and a write, and every write into two writes. Furthermore, the working set is larger, and the reduced locality harms performance more on GPUs than on CPUs, because GPU device memory has less space per thread.

Unlike previous work, LD does not monitor data accesses. Instead, it redirects accesses to a private copy. Unlike access monitoring, access redirection incurs almost no extra operation. While privatization increases total memory usage, it does not increase in working set size of each thread.

More importantly, LD operations are data parallel and specially designed, suited, and optimized for GPU execution. Data-parallel detection differs from task-parallel detection. LD includes a number of novel techniques; two of the most effective are atomic-free conflict detection and kernel fusion. On GPUs, data-parallel detection has at least five important benefits over task-parallel techniques: massive parallelism, maximal locality, absence of atomics, absence of read/write sets, and byte granularity precision.

Full privatization requires memory linearly proportional to the degree of parallelism, which may exceed the memory capacity on GPUs. We give a memory-adaptive extension that uses multiple rounds to operate under a given memory constraint while fully utilizing the available memory.

Our approach exploits the simplicity of the bulk-synchronous parallel (BSP) programming model [Valiant 1990], which is used by most GPU programs. As a limitation of BSP, LD does not handle locks and atomics. In addition, because of value-based detection, LD has false negatives. Section 4.7 discusses these limitations.

Race detection and safe parallel execution are traditionally task parallel. This article explores a data-parallel design for GPU programs. The main contributions are as follows:

—Two-pass race detection (Section 4) and its optimization, especially the combination of kernel fusion and atomic-free conflict detection (Section 4.3)
—Memory-adaptive race detection (Section 4.5)
—Deterministic GPU program execution (Section 4.6)

—Compiler transformation in GPU programs for shared-data privatization and efficient access redirection (Section 5)
—Evaluation on 11 GPU benchmark programs showing low runtime overhead and superior GPU performance of value-based detection over previous location-based techniques and the cuda-memcheck tool (Section 6).

2. PRELIMINARIES

2.1. GPU Execution Model

The processing component of a GPU consists of a set of streaming multiprocessors (SMs). Each SM consists of an array of in-order cores that are referred to as streaming processors (SPs).

A *kernel* in GPU terminology is a function that is executed N times in parallel by N threads [NVIDIA 2016]. The N GPU threads are divided into *thread block*s. A thread block is further divided into thread *warp*s, each of which has 32 threads on Nvidia GPUs. A warp is the smallest scheduling unit on an SM. The threads in a warp run in lock-step, one instruction at a time. Therefore, a thread warp implicitly synchronizes at every instruction. Different thread warps execute asynchronously. This mode of parallel execution is called single-instruction multiple-thread (SIMT).

The GPU memory is hierarchical. It has L1 and L2 caches similar to those of CPUs. It also has on-chip scratch-pad memory, called *shared memory*. The latency of shared memory is close to that of an L1 cache. The shared memory is partitioned to different thread blocks and every shared-memory partition is only visible to the thread block it is assigned to. Global memory is visible to all the threads. Threads within a thread block may communicate through shared memory or global memory. Threads across thread blocks can communicate only through global memory.

The GPU provides an explicit barrier synchronization function for threads within a thread block, named *__syncthreads()*. There is no built-in global barrier synchronization function. However, a thread waits at the end of a kernel function for other threads to finish before moving forward; that is, there is implicit barrier synchronization at the end of a kernel function. The GPU also supports intrinsic atomics that can be used to implement locks, mutexes, and barriers.

Our techniques are developed and implemented based on the Nvidia CUDA model. The approach is applicable to other GPU architectures and programming models that use the equivalent of warps (of any number of threads) and barrier synchronization, for example, the OpenCL programming model [Stone et al. 2010].

2.2. GPU Data Race

The execution model of GPU programs follows the BSP model [Valiant 1990]. There are three components in the BSP model: concurrent execution, communication, and barrier synchronization. In this model, threads carry out local computation on fast local memory asynchronously until they reach a barrier synchronization point, at which one thread waits for all other threads. The concurrent threads communicate with each other between each pair of adjacent barrier synchronization points. We call the code region between two adjacent barrier synchronization points the *asynchronous parallel code region*.

Definition 1 (*Asynchronous Parallel Code Region*). The code region of a GPU program executed between two adjacent barrier synchronization calls is called an asynchronous parallel (AP) code region.

AP code regions are executed sequentially, so a data race may happen only within an AP code region and not across multiple AP code regions.

Potential Data Race	Within local AP region	Across local AP region	Within global AP region
Thread-private data objects	✗	✗	✗
Shared memory data objects	✓	✗	✓
Global memory data objects	✓	✓	✓

Fig. 1. The five types of data races.

We categorize AP code regions into two different types: local AP regions and global AP regions. A local AP code region is the code in between every pair of thread-block level barrier synchronizations—for instance, the *__syncthreads()*. The global AP code region is the code in between every pair of global barrier synchronization points, for instance, the start and the end of a kernel, since a thread waits implicitly at the end of a kernel until all other threads finish. For shared-memory variables, data races can potentially happen only within a local AP code region, since shared-memory variables are only visible within a thread block. A global AP region may contain multiple local AP regions. For global-memory variables, data races can potentially happen within a global AP code region and thus within/across local AP regions. Considering different types of data objects, we summarize the data race scope in Figure 1.

There are two types of data races: write-write data races and read-write data races. For every type of data race, we can further categorize them into *intrawarp data race* and *interwarp data race*. An intrawarp race happens within a single GPU instruction (in the same warp). An interwarp race is more complicated because it may span multiple instructions as different warps execute asynchronously. In this article, we primarily focus on solving interwarp race detection. Intrawarp checking is covered in Section 4.7.

3. A RUNNING EXAMPLE

We use a running example to explain the techniques to be presented in the following sections. Figure 2(a) shows an example program, which is a simplified version of an real benchmark (EM). The entire body of the function `M_count` is an AP code region.

A *target variable* is one that may incur a data race, that is, the variable may be concurrently accessed in an AP code region, and the accesses are not read only. A target variable may be a shared-memory variable or a global-memory variable. We use a compiler to first identify a set of target variables and then privatize them by creating a copy of them for every warp and redirecting the memory access of each warp to its private copy. In the transformed code in Figure 2(b), the variable declarations are changed to effect data privatization and access indirection, and new function calls are inserted for two-pass race checking. We next describe the two-pass race checking. The compiler support is described in Section 5.

4. TWO-PASS RACE DETECTION

We run each AP code region twice. The first pass detects write-write conflicts. If a conflict is found, `LD` terminates the program and generates a report to the user; otherwise,

```
1 __device__ void M_count( float * data, … )
2 {
3       extern __shared__ float s_float[];
        ...
4       float * Rp = s_float+ index2 + 1536 ;
5       float * C = s_float + 3840 ;
6       float * temp1 = s_float + index0;
7       float * dist = s_float + index1+768;
        ...
8       for(int j=0;j<9;j++)Rp[j]=0;
9       for (int i=0; i<n; i+=THREADS*BLOCKS)
10      {
11        for (int j=0; j<K; j++)
12        {
13          float x_s = x[j*n+(i+n_index)];
14          for (int cnt = 0; cnt<DIM; cnt++)
15          {
16            dist[cnt] = data[(i+n_index)*DIM+cnt]-
                                  C[j*DIM+cnt];
17            temp1[cnt] = dist[cnt] * x_s;
18          }
19          for (int cnt = 0; cnt<DIM; cnt++)
20          {
21            for(int in=0;in<DIM;in++)
22            {
23              Rp[cnt*DIM+in]+=temp1[cnt]*dist[in];
24            }
25          }
26        }
27      }
28      __syncthreads();
29}
```

(a) the kernel of the EM program

```
__global__ float s_float_pri[BLOCKS][BLOCK_SIZE/WARP_SIZE][2048];

1 __device__ void M_count( float * data, … )
2 {
3       extern __shared__ float s_float[];
4       __shared__ float s_float_union[2048];
5       int block_id = blockIdx.x;
6       int warp_id = threadIdx.x  / WARP_SIZE;
7       float * s_float_new = &s_float_pri[block_id][warp_id][0];
8       parallel_memcpy( s_float_new, s_float, 2048*4);
        ...
9       float * Rp = s_float_new + index2 + 1536 ;
10      float * C = s_float_new + 3840 ;
11      float * temp1 = s_float_new + index0;
12      float * dist = s_float_new + index1+768;
13      replica from line 8 to line 27 of (a)
14      __syncthreads() ;

15      ww_check( s_float, s_float_new, s_float_union, 2048 * 4) ;
16      replica from line 7 to line 13
17      __syncthreads();

18      rw_check( s_float_union, s_float_new, 2048 * 4 ) ;
19      parallel_memcpy( s_float, s_float_union, 2048*4 );
20}
```

(b) the transformed kernel

Fig. 2. A simplified version of the EM program is used as a running example: (a) shows the original code, and (b) shows the transformed code.

LD runs the second pass to detect read-write conflicts. For ease of understanding, we first describe the checking for accesses to shared-memory variables. Then we extend this approach for global-memory variables. Figure 3 shows the detailed algorithm, with definitions first and then the procedure for each of the two passes.

4.1. Write-Write Race Detection

Let the state S be the set of (location, value) pairs for all target variables. Let warps be ordered. Before the first-pass run, we create a private copy of S for every thread warp and denote the copy for the ith warp as the private state P_i^1. The privatization is performed by replicating S (which is fully data parallel as discussed in Section 4.3). The superscript 1 indicates the private state used in the first pass. In the first pass, each warp executes exactly the same as in the original program except that it accesses P_i^1 instead of S. The warps may run in parallel as they may in the original code.

After the execution, we create a union state U, initialize it $U = S$, and then check all warps. For each warp i, we check every byte and compare its private state P_i^1 against the original values in S. If they differ, there is a write by warp i. We update the changed value in the union copy U. Before we update, we compare S and U and see whether there is already a write to this location by another warp (already checked earlier). Once we find that two warps modify the same byte, we record the variables and threads involved in the write-write race. After the checking, we report all detected write-write races and terminate the program if any race is found; otherwise, the union copy U contains all the changes made by all warps.

(a) Notations

n: number of thread warps
S: original set of target var
U: union copy of S
P$_i$: warp i's private copy of target var
diff(M',M): difference between two target var sets M' and M
apply(d,M): apply difference d to a target var set M.
note that apply(diff(M',M),M) = M'
modSet(M',M): the set of indices **i** such that M'[i] != M[i]

(b) Algorithm

w-w race check

1: P_i^1 = S, for i = 1 … n
2: Every thread warp i runs the AP code region on P_i^1
3: **if** $\exists\, i,j, i \neq j$, modSet(P_i^1, S) $\cap$ modSet(P_j^1, S) is not empty
4: **then** report write-write conflicts and exit
5: **else** proceed to check read-write conflicts
6: U = S
7: U = apply($\cup_{i=1\ldots n}$ diff(P_i^1, S), U)

r-w race check

8: P_i^2= S, for i = 1 … n
9: P_i^2 = apply($\cup_{j\neq i}$ diff(P_j^1, S), P_i^2), for all i = 1…n
10: Every thread warp i re-runs the AP code region on P_i^2
11: **if** ($\exists$ i, k, P_i^2[k] $\neq$ U[k])
12: **then** report read-write conflicts and exit
13: **else** S = U
14: proceed to check the next AP code region

Fig. 3. The algorithm of two-pass race detection.

The data copying, checking, and updating operations are fully data parallel, not only within but also across warps. They use no fine-grained synchronization and no divergent branches (unless there is a race). We will describe their implementation in Section 4.3.

In the running example in Figure 2(b), *parallel_memcpy* in line 8 shows the initialization of the first pass (i.e., the creation of the private state), and *ww_check* in line 15 shows the checking of write-write races. The seven steps of *ww_check* are shown by lines 1 through 7 in Figure 3.

4.2. Read-Write Race Detection

Before the second pass, we initialize another private state P_i^2 for each warp. The initial values in P_i^2 include all the changes made by all the warps in the first pass except by warp i. P_i^2 is created by copying the union U and erasing the changes made in P_i^1 (and reverting them to the original value in S). The second pass starts from a different private state, where each warp sees all data changes made by all other warps. The idea is that if a warp is completely independent from others, then this change to the private state should not affect its results.

In the second pass, each warp executes exactly the same as in the original program except that it accesses P_i^2 instead of S. After the execution, we compare the modifications by the two passes by comparing the two private states P_i^1, P_i^2 to find a difference.

This is performed by comparing each byte of P_i^2 and U. If there is a difference, there must be a read-write data race, because the effect of another warp must have mattered. Otherwise, we copy the contents of the union copy U back to the original state S. The checking of this AP region finishes, and the program proceeds to execute the next AP code region.

Rationale for Comparing P_i^2 with U. Let's use W_i^1 and W_i^2 to denote the sets of (location, value) pairs modified by the first pass and the second pass. Each set of (location, value) pairs maintains the invariance that in the same set, there is only one value for each distinct location. Let $W_i^1 = P_i^1 - (P_i^1 \cap S)$, and $W_i^2 = P_i^2 - (P_i^2 \cap ((U - W_i^1) \cup (S - (P_i^1 \cap S))))$, where $U - W_i^1$ shows the changes made by all other warps in the first pass, and $S - (P_i^1 \cap S)$ shows reversion of the changes made by warp i. In the reversion formula, $P_i^1 \cap S$ is the set of location-value pairs not modified by i, so $S - (P_i^1 \cap S)$ is the set of locations modified by i but reverted to the original values.

If $W_i^1 \neq W_i^2$, there is a read-write race. Mathematically, $W_i^1 = W_i^2$ if and only if $P_i^2 = U$, as proved next:

$$
\begin{aligned}
P_i^2 = U &\Rightarrow W_i^2 = U - \big(U \cap \big((U - W_i^1) \cup \big(S - \big(P_i^1 \cap S\big)\big)\big)\big) \\
&\Rightarrow W_i^2 = \big(W_i^1 - (U \cap S)\big) \cup \big(U \cap P_i^1 \cap S\big) \\
&\Rightarrow W_i^2 = \big(W_i^1 - (U \cap S)\big) \cup (U \cap S) \\
&\Rightarrow W_i^2 = W_i^1 \\
W_i^2 = W_i^1 &\Rightarrow P_i^2 = \big(W_I^2 \cup U\big) - W_I^1 \Rightarrow P_i^2 = U.
\end{aligned}
$$

Since we compare P_i^2 and U for read-write race detection, P_i^2 can reuse the space of P_i^1. Note that we have false negatives by comparing P_i^2 with U, if P_i^2 writes to the locations, which are not written by P_i^1, with values of U. One might suggest that we should create a second, private copy for a warp, which is initialized with S, to record writes in the second pass, and compare the second copy with P_i^1. However, this does not remove false negatives if P_i^2 writes with values of S but also consumes more space. Actually, the false negatives are inevitable in value-based checking, which we will discuss in Section 4.7.

In Figure 2(b), the *ww_check* initializes the second private state after write-write race checking (with checking and initialization performed simultaneously through kernel fusion as discussed in Section 4.3). The *rw_check* shows the read-write race checking. After the read-write race checking, line 19 updates the original state S by copying the changes from the union state U. These steps are shown by lines 8 through 13 in Figure 3.

Illustration of the Two-Pass Checking. Figure 4 illustrates the steps for two warps. Initially, we have the set S of target variables and we create a copy for each warp, P_1 and P_2. After the first run, the writes in each private copy are identified by comparing with the original copy. Overlapping writes are indicators of write-write races, as shown in Figure 4. If there are no overlapping writes, the changes from both warps are merged into one union copy, $U = apply(W(P_1^1) \cup W(P_2^1), S)$, at the top of the second pass in Figure 4. The union copy is used to reinitialize each private copy before the second run of the AP code region, so that each warp re-executes with its own copy of target variables. Finally, the second-run results, P_1^2 and P_2^2, are compared with U for read-write race detection.

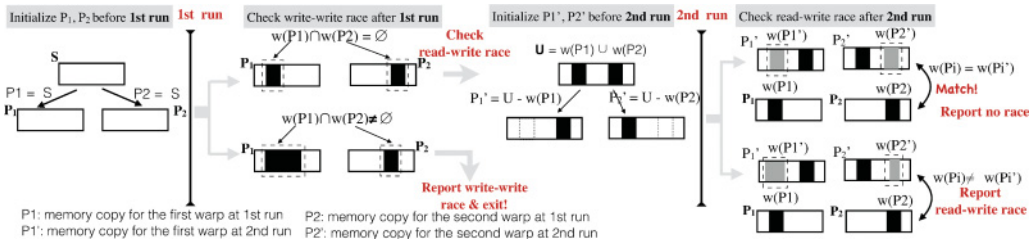

Fig. 4. A pictorial view of race checking through two running passes: initialization of the first pass (left), write-write race checking and initialization of the second pass (middle), and read-write race checking (right). The symbols have the same meaning as they do in Figure 3.

4.3. The Data-Parallel Implementation

As far as we know, LD is the first dynamic value-based race detector on GPUs. The new approach offers two important benefits: massive parallelism and locality optimization, which are critical to overcome the high cost of race checking for thousands of threads.

First, all LD steps are data parallel, including replicating data in private copies and comparing the private copy for changes, except for one step, the write-write race checking, which we discuss later in this section.

Second, we fuse the three operations, write detection, conflict detection, and write combination (for a union copy), into one kernel. These data-parallel operations are fused because they operate on the same data. *Kernel fusion* maximizes the temporal locality because a fused kernel loads data just once rather than once for each operation. A benefit of kernel fusion is the elimination of intermediate metadata. For example, in write-write race checking, if we were to detect writes first and then find conflicts, we would have to record the writes of each warp in a bitmap. Kernel fusion avoids bitmaps entirely.

In comparison, bitmaps or read/write sets are often used in previous work. For example, a safe parallelization system called BOP used bitmaps initially [Ding et al. 2007] and addressed ranges later [Ke et al. 2011] to record read/write sets. More recently, a deterministic parallel system, TARDIS, used intersecting sets for task-level access race detection [Ji et al. 2013; Lu et al. 2014]. LD does not maintain any form of access sets during program execution.

The idea of write-diffing was pioneered by Treadmarks [Amza et al. 1996] and used extensively to implement software distributed shared memory (DSM), including the use in race checking to distinguish between false sharing and a true race [Perkovic and Keleher 2000]. The technique by Perkovic and Keleher relies on a software implementation of shared memory and its lazy release consistency. In addition, DSM programs have loosely coupled MIMD parallelism. GPUs differ in that they have massive data (SPMD) parallelism, and the memory is physically shared. The shared memory is necessary for atomic-free checking, which we describe next.

Atomic-Free Conflict Checking. Write-write race checking requires comparing the write sets between every pair of warps for overlap. This quadratic cost can be reduced to linear by replaying all the writes by all the threads on a single copy of data, that is, the union copy. However, when we replay warps in parallel, they may write to the same location, which requires atomic operations to avoid data races (in race checking). A common atomic operation is Compare-and-Swap (CAS) [Scott 2013]. However, atomic operations are costly when used by thousands of threads at the same time. In addition, CAS on GPUs (as on CPUs) is based on 32-bit or 64-bit data, not 8-bit data. The coarse granularity is a problem for value-based checking because it cannot detect false sharing and as a result generates false positives.

LD uses a novel technique for *atomic-free* conflict checking. The CAS-based solution was task parallel, where each warp checked its own accesses and updated them in the union copy. The atomic-free solution performs the same work but converts it to data parallel.

Instead of each warp replaying its writes over the entire space of shared data, we partition the shared data among warps so each warp checks only the writes in its assigned region and detects conflicts just in that region. For each data item, the writes by all warps are replayed sequentially by a warp. In the data-parallel solution, different warps check different data regions. They no longer share data and hence require no atomic operations. There is no branch divergence unless there is a conflict. Last but not least, the granularity of race checking is a byte. The byte granularity eliminates false positives, which makes LD a precise race detector [Flanagan and Freund 2009].

Memory Access Coalescing. When we place privatized data in GPU global memory, our algorithms are implemented so that consecutive data accesses in a warp are coalesced into contiguous memory chunks. The GPU memory controller fetches a contiguous chunk of memory each time. If threads in the same warp access scattered data items, the memory controller might need to fetch multiple memory chunks before the thread warp can start running. This effect is caused by noncoalesced memory accesses on GPUs. In the implementation of LD, memory accesses are all coalesced.

Thread Divergence Elimination. As often reported (e.g., Zhang et al. [2011]), performance drops if different threads in the same warp execute different code paths. It may serialize the operations because a thread warp is issued one instruction at a time (i.e., the lock-step behavior discussed in Section 2). This effect is called *thread divergence*. For all race-checking operations, we pad data (and add threads) to a size that can be divided by the warp size. Although padding adds unnecessary operations, it eliminates conditional checks and hence any thread divergence caused by different code paths.

Memory Consumption. Privatization and race checking require additional memory. If a thread block has 512 threads in 16 warps, and the thread block uses a 1KB array, the first pass by LD will create 17 additional copies of the array. If privatization requires more space than available in shared memory, global memory is used. Because the shared memory is of a bounded size (e.g., 48KB per SM on Nvidia Kepler), privatization for shared-memory data can always be achieved in global memory if needed. We will discuss the memory problem for global-memory data in Section 4.4.

4.4. Interblock Data Race Detection

As explained in Section 2, there is only one synchronization point at the termination of the kernel for thread blocks. Hence, the entire kernel is one AP code region for interblock race detection. Interblock race checking operates on two levels. First, we apply the two-pass approach to check for races between concurrent thread blocks. Privatization is performed at the thread-block level, and a private copy is created for each thread block. If no race is found, we then apply the two-pass approach within each thread block to check for interwarp races as described in the last section. Altogether, four passes are needed to check for both interblock and interwarp races.

Interblock races may happen for only global-memory variables, while interwarp races may happen for shared-memory variables in addition to global-memory variables. There is no procedural difference when checking for the two types of variables. However, the memory consumption of privatization may be very different. As just discussed in Section 4.3, shared-memory variables can always be privatized in shared memory or global memory when needed. We next solve the problems of privatization and race checking for global-memory variables.

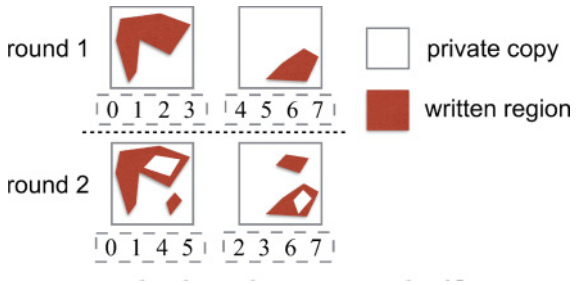

Fig. 5. Memory-adaptive race checking by warp reshuffling, assuming $N = 8, M = 2$. Only the first two rounds are shown.

4.5. Memory Adaptivity by Warp Reshuffling

In this section, we extend our race detection algorithm to overcome the problem when the available memory is too small for full privatization. We call this *memory-adaptive race checking*.

To motivate, let's consider the memory requirement. On Nvidia GPUs, an array in shared memory is not large due to the shared-memory size constraint, for example, at most 48KB on Nvidia K40c. The maximal parallelism is 960 warps on the fly. The maximal memory requirement of privatization is 48KB times 960, which is 46MB and can easily fit in global memory. If a global-memory array is 1GB, its privatization for 960 warps would require almost 1TB of memory, which is nearly impossible even for modern CPUs. Our memory-adaptive extension supports race checking under a given memory constraint.

Memory-adaptive checking uses multiple rounds of checking to trade time for space. We call each round a *memory-constrained round*. With infinite memory, it takes just one round of two-pass checking, which is the solution as presented so far. With finite memory, the number of rounds is a logarithmic function of the total memory requirement and the physical memory constraint.

We use M to denote the largest number of private copies of target variables according to the constraint. In the first round, we divide all warps into M groups and assign every group a private memory copy. Then we check races between groups as if each group were a single warp, when in reality a group may consist of multiple warps.

If there is no data race between these M groups, in the subsequent rounds, we check races within each group. For every group, we further partition the thread warps into M subgroups and assign every subgroup a private memory copy.

For instance, in Figure 5, we have eight warps, and in Round 1 we split them into two groups, $\{0, 1, 2, 3\}$ and $\{4, 5, 6, 7\}$. Then we consider the two groups as two warps and apply our write-write checking and read-write checking to the two groups. If a race is found, we report it and abort. Otherwise, we go to Round 2. In Round 2, we split each aforementioned group into two subgroups, resulting in $\{0, 1\}$ and $\{2, 3\}$, $\{4, 5\}$, and $\{6, 7\}$. Then we consider each group and apply write-write checking and read-write checking to every two subgroups: $\{0, 1\}$ and $\{2, 3\}$, $\{4, 5\}$ and $\{6, 7\}$.

A naive solution would have to serialize the checking of the two 4-warp groups; otherwise, they would require four private copies of data, one for each two-thread group. Fortunately, such serialization is unnecessary.

After Round 1, we know there is no write-write or read-write conflict between $\{0, 1, 2, 3\}$ and $\{4, 5, 6, 7\}$. Thus, $\{0, 1\}$ and $\{4, 5\}$ can reuse one private copy in Round 2, as can $\{2, 3\}$ and $\{6, 7\}$. As Figure 5 shows, Round 2 uses the same amount of memory as Round 1 but checks races for twice as many groups.

In this technique, a thread warp may use different private copies in different rounds. We call it *warp reshuffling*. The implementation redirects each thread warp to its private copy between rounds. By subdividing warp groups and leveraging the conclusion in successive rounds, warp reshuffling uses the same amount of memory in each round and eventually is able to check races among all warps in the last round.

Memory-Adaptive Checking. For N warps and a memory limit of M private copies, warp reshuffling takes $\log_M N$ rounds. It checks for data races among M (sub)groups at each round. For example, in Figure 5, $N = 8$, $M = 2$; we need $\log_2 8 = 3$ rounds. In fact, in some cases we might need less than $\log_M N$ rounds since the data race might be detected in the ith round ($i < \log_M N$), at which point we would stop the checking process and report the race.

Consider two cases where the two conflicting threads belong to the same or different subgroups. For instance, in Figure 5, Round 1, if threads 1 and 5 have a data race, warp reshuffling finds the race in Round 1. However, if threads 1 and 3 have a race, warp reshuffling will have to at least be performed in Round 2.

Assuming the two racing thread warps are distributed randomly in subgroups, let's analyze the probability that memory reshuffling detects the race in no greater than k rounds, where $k \in 1 \ldots log_M N$.

The probability of finding the race in Round 1 is the probability that the two thread warps are not in the same subgroup, that is,

$$1 - \frac{\binom{N/M}{2} \times M}{\binom{N}{2}} = \frac{N \times (1 - \frac{1}{M})}{N - 1}.$$

The probability of finding the race in no greater than two rounds can be written as

$$1 - \frac{\binom{N/M^2}{2} \times M^2}{\binom{N}{2}} = \frac{N \times (1 - \frac{1}{M^2})}{N - 1}.$$

Repeating this, we have the probability of finding this race in no greater than k rounds, shown in Equation (1):

$$P_k = \frac{N \times (1 - \frac{1}{M^k})}{N - 1}. \tag{1}$$

The tradeoff between efficiency and accuracy in memory-adaptive checking is computed by Equation (1) as P_k, the probability of finding a race in k rounds. The equation shows precisely how the probability depends on the memory limit M and the number of warps N. Warp reshuffling guarantees to find a race in at most $log_M N$ rounds. As a sanity check, we can let $k = log_M N$ and see that indeed P_k equals 1 according to Equation (1).

Since our test programs do not exceed the memory capacity on our test platform, there is no need for memory-adaptive checking. Here we give an analytical evaluation. Let the number of warps be 960, the maximum number of active warps on nvidia k40c. Table I shows the memory overhead in the number of private copies (at least two), performance slowdown (of checking) in the number of passes (at least two for a single round), and the detection probability when the racing thread warps are randomly distributed. If all warps share 1GB of target variable data, we can use 6GB of memory and finish the full detection in eight runs (four rounds). The last five rows show the probability of finding races in k rounds with respect to different memory constraints. With 2× memory overhead, the detection requires 20 runs for 100% probability of finding the race, and with six runs we can find the data race with near 90% probability (88%). Currently, memory-adaptive checking is a theoretical contribution.

Table I. The Time-Space-Accuracy Tradeoff for Race Checking Among 960 Warps (the Memory Overhead Is the Number of Private Copies)

# Warps	# Passes	Mem. Overhead	Probability
960	2	960×	100%
960	4	31×	100%
960	6	10×	100%
960	8	6×	100%
960	10	4×	100%
960	20	2×	100%
960	10	2×	97%
960	8	2×	94%
960	6	2×	88%
960	4	2×	75%

4.6. Safe Parallel Execution on GPUs

A by-product of our approach is that our techniques can enable safe parallelism, even for a racy program. Here safe parallelism means race-free or deterministic parallel execution. We use the same privatization approach. At the end of the first pass, instead of checking for write-write races, we merge the data writes from private copies. Race-free merging is faster than deterministic merging since the latter has to follow a pre-determined order, for example, the increasing order of warp ID. The merged copy becomes the original copy, and the program proceeds to execute the subsequent code. There is no need for a second pass. To extend from interwarp to interblock determinism, we add one level to the merge process. After merging the warps of each block, we then merge blocks in a sequential order.

Safe execution incurs half of the memory cost and needs just one pass. Since we do not rerun a program, we cannot use the warp reshuffling to solve the problem of memory capacity. Safety requires sufficient memory for privatization. In addition, it cannot handle programs with atomic operations.

The principal design ideas—first privatization and then sequential merge—are the same as those of previous work on CPUs, including process-based copy-on-write [Ding et al. 2007; Berger et al. 2009; Raman et al. 2010; Veeraraghavan et al. 2011; Bai et al. 2015; Ding et al. 2014] and compiler or hardware data versioning [Tian et al. 2010; Burckhardt et al. 2010; Bergan et al. 2010]. Privatization is also used in speculative program optimization by guaranteeing safe execution in the presence of aggressive and possibly unsafe optimization [Kelsey et al. 2009]. The novelty here is the data-parallel design and optimization on GPUs, including atomic-free conflict checking (needed for parallel merging), kernel fusion and other optimizations described in Section 4.3 to provide massive parallelism, maximal locality, no read/write sets, and byte granularity for GPU programs. Support for safety is another advantage of `LD` over other GPU race-checking techniques.

4.7. Discussions

Intrawarp Race Checking. Since threads within a warp execute one instruction at a time and implicitly synchronize at every instruction (SIMD), the AP code region for an intrawarp race is just one instruction. Hence, no intrawarp races exist across instructions.[1] We use the approach of GRace [Zheng et al. 2011, 2014]. Every thread

[1]Compiler optimization may break the warp-level synchronization, leading to intrawarp races across instructions. Nevertheless, existing work often assumes SIMD execution and implicit synchronization at every instruction, as does our analysis.

logs its reads and writes in every instruction. Then each thread checks the logs to see whether another thread has accessed the same location.

False Negatives by Value-Based Checking. Value-based checking has been used extensively [Amza et al. 1996; Perkovic and Keleher 1996; Ding et al. 2007]. For race checking, however, it inevitably has false negatives, for example, the well-known ABA problem. We categorize the types of false negatives as follows:

—In the first pass of write-write data race detection, if two warps write to the same location but one of them restores the original value, LD cannot detect the write-write race.
—If one warp restores the original value at a location and another warp reads this location, there is a read-write race. LD cannot detect the read-write race. However, this is a benign race.
—In the second pass of read-write data race detection, if a warp, after reading other warps' writes (read-write races), writes with values of the union copy, LD cannot detect the read-write race.

False Negatives by Atomics and Locks. LD does not detect races on variables protected by *atomic*s or *lock*s. This is a major difference between GPU race detection and CPU race detection. GPU programs rarely use atomics or locks. For high performance, GPU programs typically have massive parallelism and use only collective synchronization [Valiant 1990], while multithreaded CPU programs may use atomics and locks extensively. None of our GPU benchmarks, which are also tested in existing work [Zheng et al. 2011; Boyer et al. 2008; Holey et al. 2013], uses atomics. As far as we know, the existing GPU race-checking work [Zheng et al. 2011; Boyer et al. 2008; Leung et al. 2012; Li et al. 2012; Li and Gopalakrishnan 2010; Betts et al. 2012] either does not check for races on atomics or locks or provides limited support [Chiang et al. 2013; Bardsley and Donaldson 2014]. In programs that have them, atomics are rare, and lock-based critical sections are short. It is efficient to use access monitoring (e.g., Chiang et al. [2013] and Bardsley and Donaldson [2014]) or shared state tracking (e.g., HAccRG [Holey et al. 2013]).

5. GPU DATA PRIVATIZATION

An effective technique was developed by Yu et al. for data privatization on CPUs [Yu et al. 2013]. This section shows how LD adapts their technique for use on GPUs, which have a different task model and memory architecture.

5.1. Data Expansion

A *target variable* is one that may be concurrently accessed in an AP code region. The target variable analysis is to identify the variables whose accesses can potentially cause data races so that we do not have to check every variable in the program. This helps minimize the overhead of race checking. Maximal precision would require advanced alias and points-to analysis, and context and flow sensitivity. Not all programs are amenable to such compiler analysis. In this work, we skip the variables that are read-only or with affine array indices that can be easily analyzed statically.

Privatization is applied to all target variables. This is achieved by promoting the type declaration of a variable to expand the data size. The transformation is straightforward, as summarized in Table II. For statically allocated variables, we change the declaration to expand them into arrays or higher-dimensional arrays. Note that static shared-memory variables may be allocated either in shared memory or in global memory, depending on their sizes.

Table II. Data Expansion Rules. *Type* Denotes All Possible Shared Types. *Declaration* Denotes Native Type Declaration. *Expansion* Denotes Corresponding Expansions Based on Native Types. *a* Denotes a Shared Variable. *b* Denotes a Global Variable. *N* Denotes the Number of Warps in a Thread-Block

Type	Declaration	Expansion
__shared__ scalar	__shared__ int a	__shared__ int a[N]
__shared__ record	__shared__ struct S a	__shared__ struct S a[N]
__shared__ array	__shared__ int a[M]	__shared__ int a[N][M]
global scalar	int b	int b[N]
global record	struct S b	struct S b[N]
global array	int b[M]	int b[N][M]
heap object	v = cudaMalloc(size)	v[i] = cudaMalloc(size)

Table III. Rules for Access Redirection. *Type* Denotes All Possible Shared Types. *Before* Denotes a Memory Reference Before Redirection. *After* Denotes the Redirected Memory Reference. *a* Denotes a Shared Variable. *b* Denotes a Global Variable. *warpId* Denotes the Index of a Warp in a Thread-block

Type	Before	After
__shared__ scalar	a	a[warpId]
__shared__ field	a.field	a[warpId].field
__shared__ array	a[i]	a[warpId][i]
global scalar	b	b[warpId]
global field	b.field	b[warpId].field
global array	b[i]	b[warpId][i]

Dynamic allocation happens only in global memory. We let each warp allocate its own copy but record the base pointer in an array (created by the compiler). This design makes access redirection more efficient.

In Figure 2(b), the array *data* is a read-only global array, and hence not a target variable. The shared-memory array *s_float* is the only target variable. The declaration of *s_float_pri* implements the data expansion.

5.2. Access Redirection

Table III shows the redirection rules for accessing privatized variables. For each warp, redirection uses `warpId` to index the extra dimension of the expanded data.

Access redirection could increase the cost of access because of its indexing into the extra data dimension. However, the extra cost is actually negligible. In each warp, the index of the extra dimension is invariant, and the compiler uses a scalar variable to store the base address of its portion in the expanded data. The only overhead comes from the base address calculation, which is done just once at the start of a warp. Furthermore, the access to the dynamically allocated data is redirected directly by the way it is allocated and hence has no extra overhead from access redirection. In Figure 2(b), a local variable *s_float_new* stores the base address of the private portion. Lines 9 through 12 show the access redirection.

The two transformations, variable expansion and access redirection, are implemented in Clang based on *scout* [Krzikalla 2011] using algorithms in Algorithm 1 and Figure 6. They are adapted from the technique by [Yu et al. 2013]. To adapt their technique for use in GPU programs, we need the type promotion rules in Figure 6. For example, the rule of *Decl Heap* is for variables dynamically allocated and passed

$$[\textbf{Decl Var}]\ \frac{\Gamma \vdash v : T}{\Gamma' \vdash v : T[N]} \qquad [\textbf{Decl Array}]\ \frac{\Gamma \vdash v[M] : Array}{\Gamma' \vdash v[N][M]} \qquad [\textbf{Decl Record}]\ \frac{\Gamma \vdash v : struct(f_1 : T_1, \ldots, f_m : T_m)}{\Gamma' \vdash v : struct(f_1 : Decl(T_1), \ldots, f_m : Decl(T_m))}$$

$$[\textbf{Decl Heap}]\ \frac{\Gamma \vdash v = cudaMalloc(sizeof(T))}{\Gamma' \vdash v[i] = cudaMalloc(sizeof(T)),\ i \in 1 \ldots N} \qquad [\textbf{Ref Var}]\ \frac{\Gamma \vdash v : T \quad \Gamma \vdash v = val}{\Gamma' \vdash v[warpId] = val}$$

$$[\textbf{Decl Function}]\ \frac{\Gamma \vdash f_m : T_m func(f_1 : T_1, \ldots, f_{m-1} : T_{m-1})}{\Gamma' \vdash f_m : T_m func(f_1 : T_1, \ldots, f_{m-1} : T_{m-1})} \qquad [\textbf{Ref Array}]\ \frac{\Gamma \vdash v : T[M] \quad \Gamma \vdash v[i] = val}{\Gamma' \vdash v[warpId][i] = val}$$

$$[\textbf{Ref Field}]\ \frac{\Gamma \vdash v : struct(f_1 : T_1, \ldots, f_m : T_m) \quad \Gamma \vdash v.f_i = val}{\Gamma' \vdash Ref(v.f_i) = val} \qquad [\textbf{Ref Function}]\ \frac{\Gamma \vdash f_m = func(f_1, \ldots, f_{m-1})}{\Gamma' \vdash Ref(f_m) = func(Ref(f_1), \ldots, Ref(f_{m-1}))}$$

Fig. 6. Type promotion rules. The *Decl* and *Ref* rules are for shared-memory and global variables. Γ is the typing environment that contains the original type bindings for all variables and functions. Γ' is the typing environment that contains new type bindings after data privatization for race checking. Type T represents all primitive types and pointer types. *Decl* and *Ref* are recursive functions that define type promotion rules of *Decl* and *Ref* uses, respectively. For *Ref*, we only show store operations to illustrate this idea. Load operations are processed similarly. N denotes the number of warps. M denotes an original array dimension. v denotes a variable. f_i denotes a field of a struct or a parameter of a function. *val* denotes a value.

ALGORITHM 1: Type Promotions

```
1:  procedure DECL(t)
2:      switch t
3:      case local:
4:       return t
5:      case shared or global:
6:       Decl(t) based on type promotion rules in Figure 6
7:  end procedure
8:  procedure REF(v)
9:      switch v
10:     case local:
11:      return v
12:     case shared or global:
13:      Ref(v) based on type promotion rules in Figure 6
14: end procedure
```

to a kernel code as parameters. These transformations apply only to target variables. The main novelty of LD is a use of privatization, not the technique itself. For more implementation details, please refer to Yu et al. [2013].

Program transformation has to consider more than just target variables, in particular, variables whose values have either an upward-exposed definition (i.e., assignment before an AP code region) or downward-exposed uses (i.e., live after the AP code region). We use standard compiler def-use analysis to identify and transform upward-exposed loads and downward-exposed stores (e.g., the analysis used in scalar replacement) [Allen and Kennedy 2001; Cooper and Torczon 2010; Li et al. 2014, 2015].

Table IV. GPU Platform Configurations

Configuration	Parameter
# SMs / GPU	15
# Cores / SM	192
SIMD Width / Warp Size	8 / 32
Warp Size	32
Capability	3.5
Shared Memory per SM	48KB
L1 Data Cache per SM	16KB
L2 Cache	1.5MB
Memory Controller	Out-of-Order
Constant Memory	65K
Global Memory	11,520MB
SDK	5.5

6. EVALUATION

6.1. Implementation and Experimental Setup

Implementation. We implemented the compiler support based on an open-sourced source-to-source translator, namely, *scout* [Krzikalla 2011]. We extended scout to support the translation of GPU programs. For the GPU programs that have nested synchronizations, if-branch statements, and loops, we naively unroll the code structures and reorder their inside instructions to generate a target code region.[2] The generated code region has a set of continuous instructions. Then we clone the code region for a two-pass race checking. The current compiler handles GPU programs with a restricted syntax and does not support warp reshuffling, which we leave as future work. The runtime library comprises around 800 lines of code written in CUDA.

Machine Platform. We evaluate using an Nvidia Tesla K40c GPU card with the configuration shown in Table IV. The CPU host is Intel Xeon CPU E5-2620 2.10GHz, running Linux OS 2.6.32.

Benchmarks. We tested 11 programs, including three real-world applications, *coclustter* [Zheng et al. 2011], *em* [Zheng et al. 2014], and *kmeans* [Holey et al. 2013]; three programs from the *Rodinia* 2.4 benchmark suite, *bfs*, *backprop*, and *b+tree*; and five from the CUDA SDK 2.0 benchmark suite. We include the *backprop* and *b+tree* programs here since our work is the first to find races in them. *Cocluster* and *kmeans* are clustering programs used in data mining, and *em* (expectation maximization) is used in machine learning. These programs have been carefully optimized [Ma and Agrawal 2010]. Table V shows the input parameters and the consumption of both shared and global memory. The current LD implementation was designed to handle these programs. However, we believe that LD is able to detect shared-memory races for many other programs. Li et al. [2014] tested the Lonestar [Kulkarni et al. 2009] and Parboil [UIUC 2012] benchmark suites for their race checkers. We did not test those benchmarks. Except for them and *backprop* and *b+tree*, nine of the 11 tests we have include all available test programs gathered from all the other existing papers on GPU data race checking. In our tests, we report the average result over 20 runs.

6.2. Effectiveness and Precision

LD reports which AP code regions have data races and which memory addresses and which warps are involved, so that users can focus on specific statements and threads to

[2]A related problem is the unique worker model when used in OpenMP loops, which has been solved by Aloor and Nandivada [2015] by transforming it into multiple work share loops.

Table V. Benchmark Size, Input, and Memory Consumption. *#LOC* is the Number of Program Lines. *Conf.* Shows the Grid and Thread-Block Sizes Used in Evaluation. They May Be Changed from the Originals in Order to Expose Races. *Shm* Shows the Shared-Memory Usage per Thread Block. *GM* Shows the Global-Memory Usage

Programs	#LOC	Conf.	Shm(KB)	GM(MB)
scan	694	256,512	4.22	0.002
histogram64	482	2,067,192	11	4.11
bisect small	8,162	1,512	10	0.008
nbody	2,365	60,256	4	0.96
radix sort	2,368	1,32	0.26	0.13
bfs	282	16,256	0	0.14
backprop	898	8,256	0	2.19
b+tree	3,717	8,1024	0	15.58
kmeans	2,688	139,128	4.63	0.70
cocluster	10,878	8,256	15.57	32.02
em	3,983	8,320	15.10	63.35

Table VI. Detection Results. The *Type* Column Uses *Rw/Ww* to Denote Interwarp Read-Write/Write-Write Races. *Scope* Shows Shared Memory Races or Global-Memory Races. *#Ana,#Race* Are the Number of Code Regions Analyzed and the Number with Races. #Wps and #Mem Are the Number of Warps and Memory Addresses That Harbor Races. Note that *Scan, Kmeans* Have Injected Races

Program	Race?	Type	Scope	#Wps	#ana, #race	#Mem
scan	yes *	rw	shared	13	1, 1	2,880
histogram64	no	-	-	-	1, 0	-
bisect small	no	-	-	-	2, 0	-
nbody	yes	ww	global	8	1, 1	128
radix sort	no	-	-	-	1, 0	-
bfs	yes	rw,ww	global	128	1, 1	4,012
backdrop	yes	ww	global	5	4, 2	2,446
b+tree	yes	ww	global	6	2, 1	14
kmeans	yes *	rw	shared	2	2, 1	32
cocluster	yes	rw	shared	8	1, 1	896
em	yes	ww	shared	10	1, 1	1,536

debug and remove the races. Table VI summarizes the detection results of LD. It shows which program has what type of data races in which warps, code blocks, and memory addresses. For instance, in one code block of *cocluster*, eight warps and 896 memory addresses have interwarp read-write data races in shared memory. Note that *scan* and *kmeans* have no data races in the original version. For comparison, we injected the same races that prior work did [Zheng et al. 2011, 2014; Holey et al. 2013].

Comparison with Other Methods. GPU race checking has been extensively studied in recent years [Zheng et al. 2011, 2014; Holey et al. 2013; Li et al. 2012; Jooybar et al. 2013; NVIDIA 2014]. The result of LD has not just verified these, but also identified two data races in *backprop* and *b+tree*, which were not reported in prior work.

Much previous work [Zheng et al. 2011, 2014; Betts et al. 2012; Boyer et al. 2008] heavily relied on compile-time analysis or symbolic analysis. However, static analysis and symbolic analysis report false alarms. These include two races reported by Li et al. [2012] in *bisect small* and *radix sort* programs. In *bisect small*, the race appears only when the thread-block size is greater than 32. However, the code path that incurs the race is executed when the thread-block size equals 32. This is a drawback of static or symbolic analysis due to lack of flow sensitivity. In *radix sort*, their analysis does not

consider that threads within a warp respect SIMD execution. Other static techniques, B-tool [Boyer et al. 2008] and GPUVerify [Betts et al. 2012], also report false alarms. GKLEE [Li et al. 2012] used dynamic symbolic analysis. They found an intrawarp write-write race in *histogram64* (with thread-block size of 32), which the current LD can detect with the extension of intrawarp race checking (Section 4.7).

Zheng et al. [2011, 2014] developed four runtime techniques: GRace-stmt (GRS), GRace-addr (GRA), GMRace-stmt (GMS), and GMRace-flag (GMF). They all record and compare read-write sets for race checking. GMS and GRS record read-write sets for every statement in an AP region, while GRA and GMF do so collectively for an AP region. GMF uses an address bitmap, while GRA uses a counter-map. A common trait is that the runtime checking is address based and requires instrumenting memory references. For efficiency, they rely on compiler analysis to prune the amount of instrumentation. Not all programs are amenable to compiler analysis. In this section, *we compare the performance of GRace and GMRace with a manual implementation and without compiler optimization*. GRS and GMS could identify the exact pairs of statements causing races with per-statement instrumentation, but the cost was high. To improve speed, they were simplified to record less information. GMS utilizes more parallelism than GRS does. LD can provide precise statement-level information but will require rerunning a warp after a race is detected on a memory address. The techniques of GRS and GMS are sufficient for rerun. In LD, the cost of instrumentation is incurred only after a race is detected. Since the four tools are not publicly available, we made our best effort to implement their runtime systems. We manually instrument the accesses of shared variables to record the accessed locations for runtime checking. We made the four tools as efficient as we could, for example, by placing as many shared variables as possible in GPU shared memory and instrumenting a minimal number of program statements.

Cuda-memcheck [NVIDIA 2014] (version 5.5) is a vendor-provided tool in Nvidia's CUDA development kit. It only detects race conditions in shared memory. As tested, it could find some of the races found by LD, in *scan*, *bisect small*, *kmeans*, and *cocluster*. Cuda-memcheck provides detailed diagnostic information about detected races, but at a higher cost both on the CPU side and on the GPU side. We will show its overhead later.

LD detects races on both shared memory and global memory. All the other tools, including cuda-memcheck, do not detect races in global memory. Three programs in our test suite, *bfs*, *backprop*, and *b+tree*, use only global memory and have races. HAccRG [Holey et al. 2013] uses hardware support for detecting data races in all levels of the memory system on GPUs, and its overhead is only 1% for shared memory and 27% for global memory. However, we cannot conduct a comparison, because HAccRG requires special hardware support.

6.3. Performance Overhead

For evaluation, we measure and compare only the cost of interwarp race detection. Figure 7 shows the time comparison. Compared to the four instrumentation-based tools and cuda-memcheck, LD is more than an order of magnitude faster. On average, LD has 10× slowdown, while GMS, GMF, GRS, GRA, and cuda-memcheck have 6.89e+3×, 3.59e+3×, 1.40e+4×, 1.00e+3×, and 3.96e+3× respectively. For example, *scan* runs in 0.04 second. The cost is 1.16 seconds for LD, which is 29× slower. The 14× slowdown for *bfs* is similarly tolerable; so are those of *backprop* and *b+tree*. Excluding these four and *histogram64* (discussed next), the average slowdown is under 3× on average for the other six programs. Cuda-memcheck does not add much overhead to the programs that have no shared memory accesses, such as *bfs*, *b+tree*, and *backprop*, because it only checks the races in shared memory.

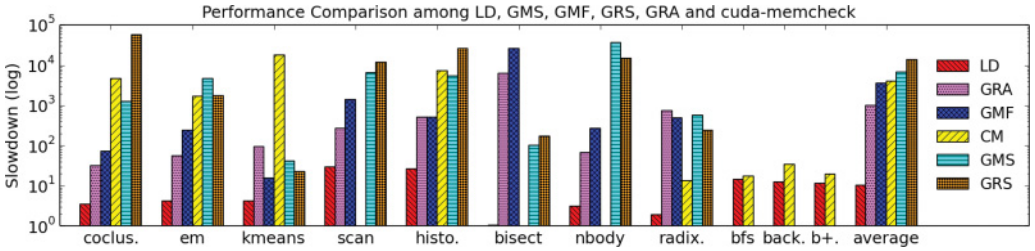

Fig. 7. The comparison of performance slowdowns among LD, GMS, GMF, GRS, GRA, and CM (cuda-memcheck). For *scan*, *bisect small*, and *nbody* programs, cuda-memcheck did not finish within 10 minutes, so we omit these running times.

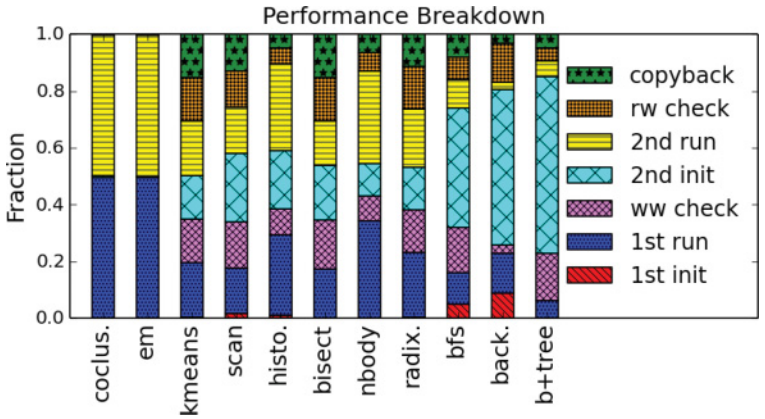

Fig. 8. Performance breakdowns of benchmark programs. *1init* denotes initialization of the first pass. *2init* denotes initialization of the second pass. *copyback* denotes copying the union copy to original copy after race checking.

Figure 8 shows breakdowns of the running time of LD. While the race-checking operations have significant costs, they do not far outweigh the cost of running the kernel itself (twice). The 27× slowdown incurred in *histogram64* is caused in part by its shared-memory data being privatized in global memory. All programs that use shared memory are impacted by this in some degree. For *bisect small* and *radix sort*, although their native running times are short, the AP code regions chosen by our static analysis are also short. Except for the two, race checking happens on almost the entire kernel code. In *nbody*, *kmeans*, *cocluster,* and *em*, the relative slowdown is low because some of the data are privatized in shared memory, and the checking is very fast (less than 3ms, see Section 6.5). Figure 8 divides the runtime of a target AP code region into seven parts. Each part is measured separately in different runs to reduce the noise of the time measurement and measured over 10 times to take the average. We use the first thread of each thread block to record the start and end times of each part by calling the "clock()" function. Then we compute the average cost across all thread blocks. Before reading the start and end times, we synchronize all threads in a thread block by calling "__syncthreads()".

6.4. Memory Overhead

LD uses additional memory for private copies and the union copy. The other four techniques use extra space primarily for two types of metadata tables, warp tables and thread-block tables [Zheng et al. 2011, 2014], which record instrumented memory

Table VII. Memory Overhead of Benchmark Programs. *Shm* Denotes Shared-Memory Usage. *GM* Denotes Global-Memory Usage. In *Nbody*, a Race Happens on Global Memory, but for Fairness We Measure the Overhead of LD Only on the AP Code Regions That Have Only Shared-Memory Accesses

	LD		GMS/GRS		GMF	GRA
Prog.	**Shm**	**GM**	**Shm**	**GM**	**GM**	**GM**
scan	1.1M	17.4M	2.1M	71M	>1G	>1G
histogram	25.3M	145.5M	6.34M	>1G	>1G	>1G
bisect	2K	34K	8.4K	1.59K	6M	6.4M
nbody	240K	1.9M	251K	251M	160M	202M
radix	0.6K	0K	0.54K	82K	384K	768K
kmeans	653K	2.6M	291K	1.2M	208M	260M
cocluster	16K	1M	33.5K	>1G	24M	27M
em	16K	1.3M	41.8K	>1G	30M	33M

access logs. Each warp table or thread-block table has a read table and a write table. When implementing these techniques, we made the tables as small as possible, for example, by using shared memory.

Table VII reports the memory overhead. The memory overhead is the same for GMS and GRS. The shared memory part is identical for all GRace and GMRace techniques. LD uses much less memory in most programs, both in shared and in global memory. The memory overhead of LD is proportional to the number of warps and thread blocks and the size of target variables. The memory overhead of GMF or GRA is proportional to the number of both warps and thread blocks. But instead of the target data size, their overhead is proportional to the capacity of shared memory. LD outperforms GMF and GRA for two reasons. One is that the target data size is always smaller than shared memory capacity. The other is that every warp table or thread-block table in GMF and GRA has a read table and a write table, while in LD, each warp has one private copy. GRA costs more than GMF due to the overhead of an additional thread-block table. The memory overhead of GMS and GRS is proportional to the number of warps, thread blocks, and memory references. Programs that have few memory references use less memory than LD, for example, *bisect small* and *kmeans*. Not reported in Table VII, *bfs*, *backprop,* and *b+tree* have only global-memory accesses, and their global-memory usages are 4.4MB, 10.4MB, and 19.8MB and shared-memory usages are all zero.

LD adds shared-memory consumption for two reasons. One is privatization. The second is the union copy, which is stored in the shared memory whenever possible. Using shared memory helps performance enhancement, and thus we use it as much as possible.

6.5. Scalability and Optimization

Figure 9 shows how the cost of race checking increases with the data size and the number of threads. All checking operations are fully parallel and scale linearly. In Figure 9(a), for data sizes no greater than 1MB and thread-block size of 256, the overhead is below 10ms. *2Init*'s cost is 12ms when data size is 256KB, but we show it to compare with *opt2Init*. We use *opt2Init* in our race checking. In Figure 9(c), when data size is 48KB, the maximal size allowed in shared memory, all operations take less than 3ms. This shows that LD has a negligible cost when checking races in shared memory. Figure 9(a) and Figure 9(b) show the effect of kernel fusion. By comparing *2init* and *opt2init* at 256 threads at the largest data size, 32MB, we see that kernel fusion reduces the time cost by 42% and increases the speed by 72%.

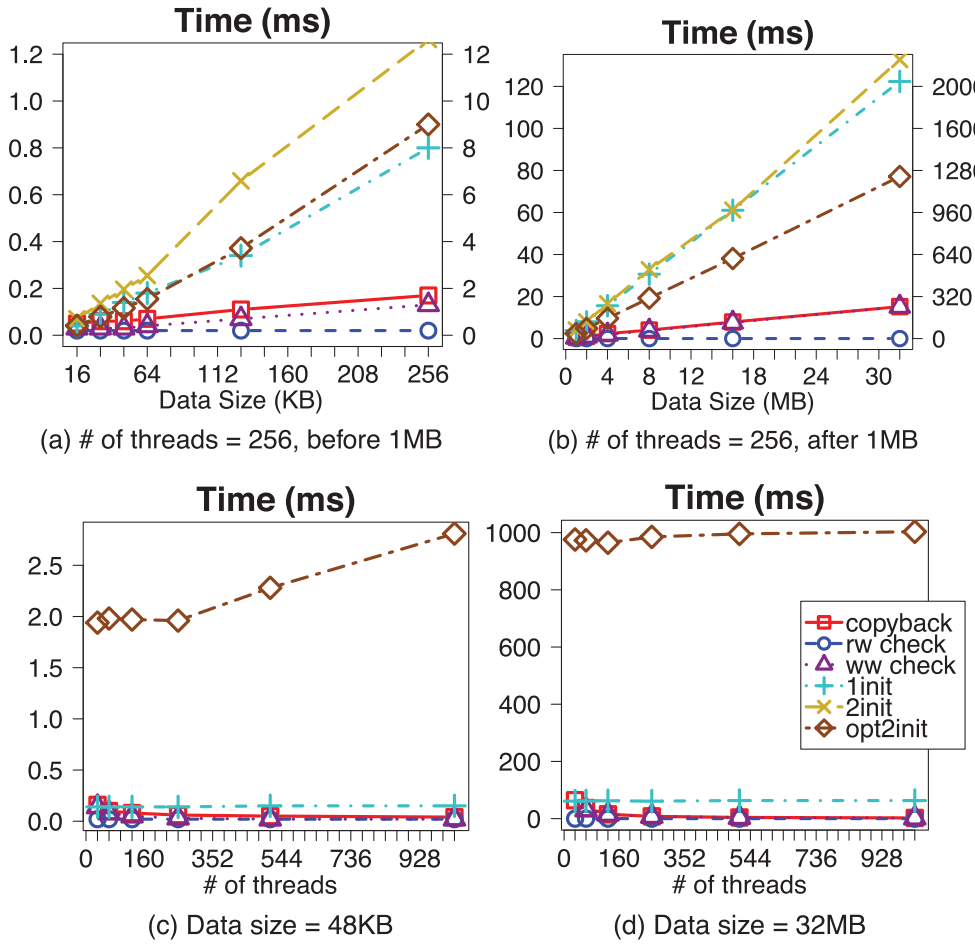

Fig. 9. The cost of checking operations as a function of the data size and thread count. The y-axis is time measured in milliseconds. *1init, 2init, copyback* have the same meanings as in Figure 8. Note that *2init* denotes the nonoptimized version of the second-pass initialization, and *opt2init* is optimized by kernel fusion. In (a) and (b), only *2init* and *opt2init* use the y-axis on the right-hand side.

7. RELATED WORK

Dynamic Race Detection on GPUs. Prior techniques used instrumentation to record all runtime memory accesses from different threads and different warps to detect data races [Boyer et al. 2008; Hou et al. 2009; Zheng et al. 2011, 2014]. The instrumentation could cause orders of magnitude performance degradation compared with un-instrumented versions. Grace [Zheng et al. 2011] and GMRace [Zheng et al. 2014] reduced the instrumentation overhead drastically using static analysis. However, static analysis is not always effective in all programs. LD does not use instrumentation. In addition, some of the previous techniques were limited to shared-memory data to ensure memory efficiency. The space overhead in the metadata tables in GMS/GRS and the read/write bitmaps in GMF and GRA would have become significant if global memory were considered. LD provides a memory-adaptive solution that can handle limited memory capacity.

The technique of Hardware-Accelerated data Race detection for GPU (HAccRG) uses (shared) shadow states to monitor data races in not only BSP-style computations but also programs with fences and atomics [Holey et al. 2013]. The memory overhead is linear to the size of data and independent of the number of warps. The tradeoff is time—the access to shared states requires serialization, which is efficient in HAccRG with special hardware support. Its overhead is 1% for shared memory and 27% for global memory. For the same amount of memory overhead (2× to 3×), For the same memory overhead, LD requires 10 to 20 passes in memory-adaptive checking to achieve 100% coverage (when a program has 960 warps).

Static and Hybrid Race Checking on GPUs. GPUVerify [Betts et al. 2012; Chong et al. 2014] develops new programming semantics on GPUs, under which data races can be detected while programming. PUG uses Satisfiability Modulo Theories [Li and Gopalakrishnan 2010]. GKLEE extends symbolic analysis for correctness checking [Li et al. 2012]. In GKLEE$_p$, Li et al. [2012] developed the concept of *parametric flows*, which divide threads into equivalence classes by their control flow, and race checking was performed on a pair of threads per parametric flow. It uses thread symmetry to analyze GPU kernels with a large number of threads more efficiently than the original GKLEE [Li et al. 2012]. GKLEE$_p$ suffers from search-space explosion in the worst case, that is, 2^n flows from n branches in one thread. Li et al. [2014] drew support from static analysis and built a practical symbolic race checker. All these techniques use symbolic analysis and have false alarms. An example was given in Section 6.2 where a false alarm was raised in GKLEE.

Dynamic checking complements static checking in mainly two ways. First, dynamic checking has no false alarms. Second, it is applicable to all programs including those that are not amenable to static analysis. Section 6.2 gives a more detailed comparison based on our test suite. Leung et al. [2012] proposed a combination of static and dynamic analysis based on information flow. Their performance and memory overhead is excellent, but it relies on program analysis. They used program analysis to reduce performance and memory overhead and showed on average 18× slowdown. Our approach is entirely dynamic and has 10× slowdown on average for our test programs.

Data Privatization Techniques. Prior privatization techniques [Gu et al. 1997; Li 1992] were developed for automatic parallelization of scientific code on CPUs. Recently, Yu et al. [2013] gave a general technique for data structure expansion for loop parallelization on CPUs. Our solution is for GPUs and has several differences. First, Yu et al. uses dynamic memory allocation to expand global variables. Earlier, Ding et al. [2007] also allocated global variables dynamically in behavior-oriented parallelization (BOP). Our privatization uses static allocation for static variables. The reason for the difference is partly architectural. On CPUs, dynamic allocation is used because it is not always possible to statically allocate a very large array, but it can always be done in global memory on GPUs. For access redirection after type promotion, our technique avoids shadow variables recording original data sizes and offset calculation. LD shows the significant benefit of privatization in race checking for GPU programs, which have more threads than CPU programs do.

Race Checking on CPUs. As a comparison, we study the existing race detectors on CPUs. Much existing work tracks the happens-before relationship by looking for unordered conflicting accesses [Devietti et al. 2012; Effinger-Dean et al. 2012; Flanagan and Freund 2009]. Radish [Devietti et al. 2012] provides a sound, complete, and always-on race detector accelerated by the hardware. Its overhead is no more than 2×. Eraser [Savage et al. 1997] detects lock-set violation by looking for conflicting accesses not protected by a common lock. HARD [Zhou et al. 2007] is a hardware-based implementation

of the lock-set algorithm. A collection of work explores race checking for structured programs [Raman et al. 2012; Lu et al. 2014]. CPU programs extensively use locks, while GPU programs use bulk synchronizations [Valiant 1990]. The problem of GPU and CPU race detection differs because of the difference in their parallel programming model. Access monitoring is effective in CPU race detectors; LD demonstrates that on GPUs, value-based checking is orders of magnitude faster than access monitoring.

8. SUMMARY

Race checking on GPUs is conventionally task parallel. In this study, we have developed a novel data-parallel detector called LD. We have designed a two-pass detection algorithm that is atomic free and a memory-adaptive solution to reduce the memory overhead. To our knowledge, this is the first value-based race checking on GPUs. Our prototype checker has on average 10× overhead, which shows at least an order of magnitude improvement over fine-grained access monitoring and a state-of-the-art industry tool.

ACKNOWLEDGMENT

The idea was first presented at the 5th Workshop on Determinism and Correctness in Parallel Programming [Li et al. 2014]. The authors wish to thank Sandhya Dwarkardas, Michael Gage, Rahman Lavaee, and TACO reviewers for their comments, which have helped the presentation of the article.

REFERENCES

John R. Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers.

Raghesh Aloor and V. Krishna Nandivada. 2015. Unique worker model for OpenMP. In *Proceedings of the International Conference on Supercomputing*. 47–56. DOI: http://dx.doi.org/10.1145/2751205.2751238

Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Peter J. Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer* 29, 2 (1996), 18–28.

Tongxin Bai, Chen Ding, and Pengcheng Li. 2015. Assessing safe task parallelism in SPEC 2006 INT. In *Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*.

Ethel Bardsley and Alastair F. Donaldson. 2014. Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In *Proceedings of the 6th International Symposium on NASA Formal Methods (NFM'14)*.

Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 53–64.

Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*. 81–96.

Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: A verifier for GPU kernels. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*. 113–132.

Michael Boyer, Kevin Skadron, and Westley Weimer. 2008. Automated dynamic analysis of CUDA programs. In *Proceedings of the 3rd Workshop on Software Tools for MultiCore Systems*.

Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent programming with revisions and isolation types. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*. 691–707.

Wei-Fan Chiang, Ganesh Gopalakrishnan, Guodong Li, and Zvonimir Rakamaric. 2013. Formal analysis of GPU programs with atomics via conflict-directed delay-bounding. In *Proceedings of NASA Formal Methods, 5th International Symposium (NFM'13)*.

Nathan Chong, Alastair F. Donaldson, and Jeroen Ketema. 2014. A sound and complete abstraction for reasoning about parallel prefix sums. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*.

Keith Cooper and Linda Torczon. 2010. *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.

Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. 2012. RADISH: Always-on sound and complete ra detection in software and hardware. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*.

Chen Ding, Brian Gernhart, Pengcheng Li, and Matthew Hertz. 2014. *Safe Parallel Programming in An Interpreted Language*. Technical Report URCS #991. Department of Computer Science, University of Rochester.

Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. 2007. Software behavior oriented parallelization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 223–234.

Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. 2012. IFRit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 121–133.

Junjie Gu, Zhiyuan Li, and Gyungho Lee. 1997. Experience with efficient array data-flow analysis for array privatization. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 157–167.

Anup Holey, Vineeth Mekkat, and Antonia Zhai. 2013. HAccRG: Hardware-accelerated data race detection in GPUs. In *ICPP*.

Qiming Hou, Kun Zhou, and Baining Guo. 2009. Debugging GPU stream programs through automatic dataflow recording and visualization. In *ACM SIGGRAPH Asia 2009 Papers*.

Weixing Ji, Li Lu, and Michael L. Scott. 2013. TARDIS: Task-level access race detection by intersecting sets. In *Proceedings of the Workshop on Determinism and Correctness in Parallel Programming*.

Hadi Jooybar, Wilson W. L. Fung, Mike O'Connor, Joseph Devietti, and Tor M. Aamodt. 2013. GPUDet: A deterministic GPU architecture. In *ASPLOS*.

Chuanle Ke, Lei Liu, Chao Zhang, Tongxin Bai, Bryan Jacobs, and Chen Ding. 2011. Safe parallel programming using dynamic dependence hints. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*. 243–258.

Kirk Kelsey, Tongxin Bai, and Chen Ding. 2009. Fast track: A software system for speculative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*. 157–168. DOI:http://dx.doi.org/10.1109/CGO.2009.18

Olaf Krzikalla. 2011. Scout: A Source-to-Source Translator for SIMD-Optimizations. *Proceedings of the* https://tu-dresden.de/zih/forschung/projekte/scout/.

Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. 2009. Lonestar: A suite of parallel irregular programs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*.

Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU kernels by test amplification. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 383–394.

Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 187–196.

Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic verification and test generation for GPUs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 215–224.

Pengcheng Li, Chen Ding, Xiaoyu Hu, and Tolga Soyata. 2014. LDetector: A low overhead race detector for GPU programs. In *Proceedings of the 5th Workshop on Determinism and Correctness in Parallel Programming*.

Pengcheng Li, Ziang Hu, and Handong Ye. 2015. Compiler and Method for Global-Scope Basic-Block Reordering. https://www.google.com/patents/US20150040106 US Patent App. 14/445,983.

Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2012. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2014. Practical symbolic race checking of GPU programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*.

Pengcheng Li, Hao Luo, Chen Ding, Ziang Hu, and Handong Ye. 2014. Code layout optimization for defensiveness and politeness in shared cache. In *Proceedings of the 2014 43rd International Conference on Parallel Processing*. 151–161.

Zhiyuan Li. 1992. Array privatization for parallel execution of loops. In *Proceedings of the International Conference on Supercomputing*. 313–322.

Li Lu, Weixing Ji, and Michael L. Scott. 2014. Dynamic enforcement of determinism in a parallel scripting language. In *PLDI*.

Wenjing Ma and Gagan Agrawal. 2010. An integer programming framework for optimizing shared memory use on GPUs. In *PACT*.

NVIDIA. 2014. Cuda Memcheck Tool. Retrieved from https://developer.nvidia.com/CUDA-MEMCHECK.

NVIDIA. 2016. CUDA C Programming Guide. Retrieved from http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

Dejan Perkovic and Peter J. Keleher. 1996. Online data-race detection via coherency guarantees. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*.

Dejan Perkovic and Peter J. Keleher. 2000. A protocol-centric approach to on-the-fly race detection. *IEEE Transactions on Parallel and Distributed Systems* 11, 10 (2000), 1058–1072.

Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. 2010. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 65–76.

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin T. Vechev, and Eran Yahav. 2012. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*.

Michael L. Scott. 2013. *Shared-Memory Synchronization*. Morgan & Claypool Publishers.

John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Design Test* 12, 3 (2010), 66–72.

Chen Tian, Min Feng, and Rajiv Gupta. 2010. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 62–73.

UIUC. 2012. The Parboil Benchmark Suite. Retrieved from http://impact.crhc.illinois.edu/parboil/parboil.aspx.

Leslie G. Valiant. 1990. A bridging model for parallel computation. *Communications of the ACM* 33, 8 (Aug. 1990), 103–111. DOI:http://dx.doi.org/10.1145/79173.79181

Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 15–26.

Hongtao Yu, Hou-Jen Ko, and Zhiyuan Li. 2013. General data structure expansion for multi-threading. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 243–252.

Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 369–380.

Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: A low-overhead mechanism for detecting data races in GPU programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 135–146. DOI:http://dx.doi.org/10.1145/1941553.1941574

Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2014. GMRace: Detecting data races in GPU programs via a low-overhead scheme. *IEEE Transactions on Parallel and Distributed Systems* 25 (2014), 104–115.

Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. 2007. HARD: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*.

Received May 2016; revised December 2016; accepted January 2017