LDetector: A Low Overhead Race Detector For GPU Programs

Pengcheng Li, Chen Ding
Department of Computer Science,
University of Rochester,
Rochester, NY, USA
{pli, cding}@cs.rochester.edu

Xiaoyu Hu, Tolga Soyata
Department of Electrical and Computer Engineering,
University of Rochester,
Rochester, NY, USA
{x.hu, tolga.soyata}@rochester.edu

Abstract

Data race detection becomes an important problem in GPU programming. The paper presents a novel solution mainly aimed at detecting data races happening in shared memory accesses with no use of atomic primitives. It makes use of compiler support to privatize shared data and then at run time parallelizes data race checking. It has two distinct features. First, unlike previous existing work, our work gets rid of per memory access monitoring by data privatization technique, which brings a very low performance overhead and also well scalability. Second, data race checking utilizes massively parallel resources on GPU. Preliminary results show two orders of magnitude performance improvement over an existing work.

1. Introduction

Despite its significant undeniable potential for high performance, Graphics Processing Units (GPUs) rely heavily on the programmer for properly executing parallel code. While understanding the underlying processor architecture will empower the developer to write highly efficient code, an increased burden is also placed on the programmer to ensure correctness of the code. One criteria that the developer must be aware of is data races. While this phenomenon also exists in the traditional parallel programming environment, its impact is significantly more emphasized due to the thousands of active threads executing inside a GPU, potentially interacting with shared memory resources.

A rich body of literature exists on race detection for CPUs [14–17]. However, this existing work cannot be really applied to GPU environments due to the significantly different architecture. GPU programs use the SIMD execution model and massive parallelism. They use synchronization but very little locking. The past solutions on the CPU side often focused on the problem of complex locking and required monitoring data accesses to shared memory. Access monitoring is used in recent GPU studies [9, 10, 19, 20]. Since hundreds of thousands of threads may run concurrently, the overhead of access monitoring is significant. In this paper, our goal is data race detection without access monitoring by expanding shared data and directing concurrent writes to private copies of shared data.

Our approach relies on the compiler's determination of which variables should be expanded. We call them *target variables*. Unlike CPU programs, GPU programs have reduced pointer-induced aliasing. We use a set of data structure expansion rules to create multiple copies of each target variable (one for each warp). In addition, we keep one original copy as comparison.

We use a two-pass approach to detect data races. The first pass detects write-write races. Each warp operates on its own private copy of data, followed by a check for overlapped writes by multiple warps. If no write-write race is detected, we commence to the second pass to detect read-write races by restoring the values of non-target variables to the value before the first pass. We copy the changed values from other warps, so the warp is to run after all other warps have finished. After second pass, if the output differs from the first output by any wrap, we report a read-write race. Otherwise, there is no data race. Our two-pass approach is aimed at detecting data races in shared memory accesses with no use of atomic primitives.

Contributions

- This paper presents a compiler technique for data privatization for GPU programs. It identifies target variables and expand data structures automatically.
- This paper presents ***LDetector***, a two-pass approach to detect write-write and read-write races. By relying on data privatization, it does not have to monitor per shared memory access. The runtime design is light weight, using massive parallelism, memory coalescing to fully leveraging the high performance of GPU in race detection.
- This paper presents ***LDetector-spec***, a tool that detects data races with a new speculative parallelization technique in GPU. ***LDetector-spec*** can greatly increase performance in utilizing massive computing core resources, but achieve the same soundness and convergence and precision as ***LDetector***.
- This paper compares our tools with existing tools over a range of applications. While supporting the same detection precision, our tools outperform others by more than an order of magnitude in speed and less memory consumption.

2. Background

2.1 Data Race in GPUs

The architecture of the Graphical Processing Unit (GPU) consists of two major components: the processing component, often referred to as streaming multiprocessors (SM) and the hierarchical memory. A GPU program is referred to as a *kernel*, composed of a large number of *thread*s. Threads within one kernel are organized into *block*s. A block of threads are divided into several groups, called *warp*s in CUDA. Warps are the smallest scheduling units on SM. A warp of threads execute the same instruction in a clock cycle, called the SIMD execution model, by which GPU data races are different from CPU races. According to the GPU execution model, data races can be divided into intra-warp and inter-warp data races. An intra-warp data race happens when more than one thread in a warp write to the same memory location, that is only write-write data race. Detecting this type of race is trivial. One can only use a bitmap shadow memory for each warp to record writes of each shared access, which can be reused for the following accesses. In addition, most of them can be decided at compile time. In this paper, we primarily focus on inter-warp data races.

2.2 A Motivation Example

```
__global__ void Jacobi(int * data)
{
      extern __shared__ int A[];
      int tid = threadIdx.x;
      ...
      if ( tid < BLOCK_SIZE-1 && tid > 0)
             A[tid] = ( A[tid-1] + A[tid] +
                              A[tid+1] ) / 3;
      __syncthreads();
}
```

In above we show the GPU kernel for the Jacobi computation, in which all threads in a thread-block compute the average of itself and two adjacent values. If the code is multi-threaded for CPU, there is a race between threads *i* and *i+1*. However, due to GPU's SIMD execution model. Within a warp, all threads are scheduled together, thus no data race happens within a warp. The threads at the boundary of two consecutive warps incur a read-write race due to out of order execution of warps.

3. Data Structure Expansion

This section introduces the rules to expand data structures and to redirect memory accesses. Figure 2(b) shows a code snippet of the EM benchmark program [19] for illustration.

3.1 Privatizing Data Copies

Privatization can be performed on all variables with sharing property in a GPU program, i.e. the *target variables*, including shared and global variables. This is achieved by promoting the type declaration of data structures. Table 1 shows the expansion rules. A related technique is Yu et al. [18] for multi-threaded programs. They use dynamic memory allocation to expand global variables. In comparison, we expand them into (higher-dimensional) arrays. For general purpose CPU code, it is difficult to determine which variable is shared, so Yu et al. relies on a dependence profiling to expand all global variables. For GPU programs, our compiler expands only the variables shared by different warps.

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	cudaMalloc(size)	cudaMalloc(N*size)

Table 1: Variable Expansion Rules. The "Type" column shows all possible sharing types, "Declaration" native type declaration and "Expansion" the expanded types. N is number of warps in a block. All above declarations without "shared" keyword denote declarations of global memory variables.

3.2 Memory Access Redirection

Table 2 shows redirection rules to guide different warps to redirect their access to their private copy. The variable `warpId` denotes the index of different warps. Note that in dynamically allocated objects, we need to know the original size, denoted by the symbol $span$, so as to index warp private copies. We know it is difficult to infer $span$ having only the pointers. We use shadow variables to record the $span$ at allocation, and use a compiler to compute the $span$ and insert it in the right place. Figure 2(b) shows a code example where the addresses to privatized arrays have been re-directed using these rules.

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]
Pointer deref	*p	*(p+warpId*span/sizeof(*p))

Table 2: Redirection Rules. The column "Type" shows all shared types, "Before" the native memory addressing format and "After" the redirected memory address. `warpId` is the index of the warp in a block. "Span" denotes the original size before expansion.

4. Shared-array Oriented Race Detection

The method we introduce is based on performing two passes on the same program: In the first pass, write-write conflicts are detected. If this pass fails (i.e., when there is a write-write conflict), the program terminates and reports write-write conflicts. Upon successful completion of the first pass, our tools move onto the second pass, where read-write conflicts are detected. Our tools are primarily aimed at detecting data races in shared memory accesses not using atomic primitives. We will talk a little about applying this method to global array and drawbacks of it later. Both of these passes are explained in details in the following subsections.

4.1 First Pass: Write-Write Race Detection

DEFINITION 1. **Synchronized Block** *A code region between two adjacent synchronization statements is called a synchronized block. Inter-warp data races on a shared array happen only in the same synchronized block.*

The detailed steps of write-write race detections for each synchronized block are provided below:

Initialization Our tools first expand target shared arrays in current synchronized block. In addition, we keep an extra copy used for comparison after the first run. For example, if a thread-block consists of 512 threads and is using a 1KB shared array, since the warp size is 32 threads, this implies 16 warps, each with a private replica of the 1KB array. Therefore, it will require 16KB additional shared array, along with the 1KB for the original copy. If an array expansion takes space larger than the shared memory size, 48KB on Nvidia Fermi and Kepler, the global memory will be used to keep the private copies of the shared array for each individual warp.

Block Start Two-level parallelism is used to copy all data from original copy to warps' own copies. On the first level parallelism, all warps copies data in parallel to its private copy. On the second level parallelism, all intra-warp threads, 32 in CUDA, divide the work of copying. So *two-level parallelism* refers to as warp-level parallelism and intra-warp thread-level parallelism. A synchronization is needed for each warp (but not between warps). After a warp finishes copying, its threads begin to execute the synchronized block.

Execution After redirecting memory addresses, every warp executes on its own copy of data as original. Each thread costs one register variable to representing base address pointer of the warp's private copy. Other than the redirection, all threads execute the program exactly the same as the original of program.

Block Commit After executing synchronized block independently, two-level parallelism is used to compare warps' private copies of data with the original copy of data for potential write-write race conditions (at a byte granularity) As in the initial copy-

ing, all warps work in parallel at the first level. At the second level, every thread compares data in the granularity of one byte. A bitmap of shadow memory is used to record comparison results. Each one byte use one bit. Just one bitmap is used for the shared array for write-write conflict detection. We go through the warps one by one. If a warp writes, we check the corresponding bits. If the bits are set, we report write-write data races. Otherwise, we set the bits. If write-write races are found, we report the races and stop program execution. Otherwise, all warps combine the written parts of an array into to an union. Because there is no overlap in writes, two-level parallelism can be used for union.

Discussion Compared with existing work, our tool has zero memory access monitoring overhead. Additionally, the GPU massive parallelism is utilized three times to perform the copying, comparison and union operations.

4.2 Second Pass: Read-Write Race Detection

If there are no write-write data races, we start the second pass to detect read-write races. The following subsection walks through the steps of the second pass.

Initialization We classify memory loads and stores using compiler analysis:

DEFINITION 2. **Upward-exposed load** *A memory load is said to be upwards-exposed in a synchronized block if the value may be defined before this synchronized block.*

DEFINITION 3. **Downwards-exposed store** *A memory store is said to be downwards-exposed in a synchronized block if the value may be used after this synchronized block.*

If any variables other than target shared array have upwards-exposed loads and are modified in target synchronized block, we must store the values of these variables at the beginning of the synchronized block for second-pass and stored them before the first pass.

Block Start Every warp first merges the modified data from the private copy of all other warps to its own copy and then copies the original values to its own copy for array that is not modified by other warps. Between the two copy operations, a synchronization statement is inserted to ensure that all warps finish the first copying before the second.

Execution Every warp executes the synchronized block using its private copy independently.

Block Commit After the second pass, every warp compares the values of its own copy between the second pass and the first pass in modified array cells by comparing the warps' private copy with the union copy. If a different value is found, we report the presence of a read-write race and stop program execution. If there is no race, the union copy will be used as the original copy, and the program continues.

One may ask that the second pass can overwrite the downwards-exposed variables. The answer is, if we find data races, although the second pass may overwrite downwards-exposed variables, the program stops. If no data race is found, the downwards-exposed variables must have the correct values. Program execution remains correct.

Review of Data Copying To illustrate, we use Figure 1 to review the series of data copying in the two-pass detection. For the target array, there are three copies and a bitmap. The original copy keeps the original values. Each warp has a private copy in the expanded array. Finally, the union copy combines the changes from all warps. To detect write-write data races, we use the bitmap. One bit represents the writing records of one byte of shared array.

4.3 A Lightweight Runtime Library

In Figure 2, we use the EM benchmark [19] to show not just the program transformation but also the API of our runtime library. As we can see, there are only three runtime functions, *region_diff*, *region_union* and *compare_output* functions.

Our lightweight runtime library mainly include four algorithms, including *diff*, *union*, *combination* and *comparison* operations. *diff* is responsible for write-write conflict checking among warps. *union* is used to union all warps' copies of shared array to one union copy if there are no write-write conflicts. In *combination*, each warp copies modified array regions from other warps to its own copy. *comparison* is used to compare results of the two runs. In Figure 2(c), *region_diff* is the implementation of *diff*. *region_union* function includes *union* and *combination*. *compare_output* is the implementation of *comparison*. To make it as efficient as possible, we make several following considerations:

1. All four libraries are implemented using two-level parallelism, including warp-level parallelism and intra-warp thread-level parallelism. Different warps work on their own data copies in parallel. Each warp divides its working set among the intra-warp threads, so that intra-warp threads work on different data elements in parallel. Therefore massive parallel resources on GPU are well utilized.
2. In GPU programming, thread divergence, which refers to as different threads execute different paths at a given clock cycle, is harmful to performance due to the limitation of simple in-order cores. To prevent thread divergence, we pad data to a size that can be divided by the number of threads in a warp, 32 in CUDA, so that no thread is idle when other threads have data to operate. Although computing padded data are meaningless, this optimization smartly prevents different threads from executing different paths.
3. Consecutive accesses to global memory from different threads in a warp are coalesced, that is, combined into a single larger access to shorten high memory access latency. Our algorithms are designed to meet the requirement for coalesced global memory access.

4.4 Correctness, Extension and Limitation

Correctness The data races detected by our tools are always sound inter-warp race. Our two-pass approach is based upon an assumption that there are no atomic operations in synchronized blocks. Overall, we have two chances to report data races, after the first run and the second run, respectively. After the first run, the races reported by our tool are write-write conflicts, based upon value-based diff operations. After the second run, our tool reports if there are inconsistent outputs. The inconsistent outputs suggest read-write conflicts. If they are caused by write-write conflicts, a contraction happens because the second run is only executed provided that the first run does not find any data races. So our two-pass approach is sound. Although our tools in the first run only find out write-write races probably, after modifying the bugs and several later runs, our tools can find out the remaining read-write races.

Extension to Global Array and Other Data Structures The above description of detection process assumes shared array as target variables. Detection for a shared array subsumes the problem of detection for a scalar variable or a record. The same technique can be extended to a global array, which subsumes the problem of detection on global scalars and global records.

Data races on a global array can happen among thread-blocks, not just warps of the same thread block. We use a similar two-step approach to defect data races. First, we detect data races on a global array among thread blocks, using the previous approach.

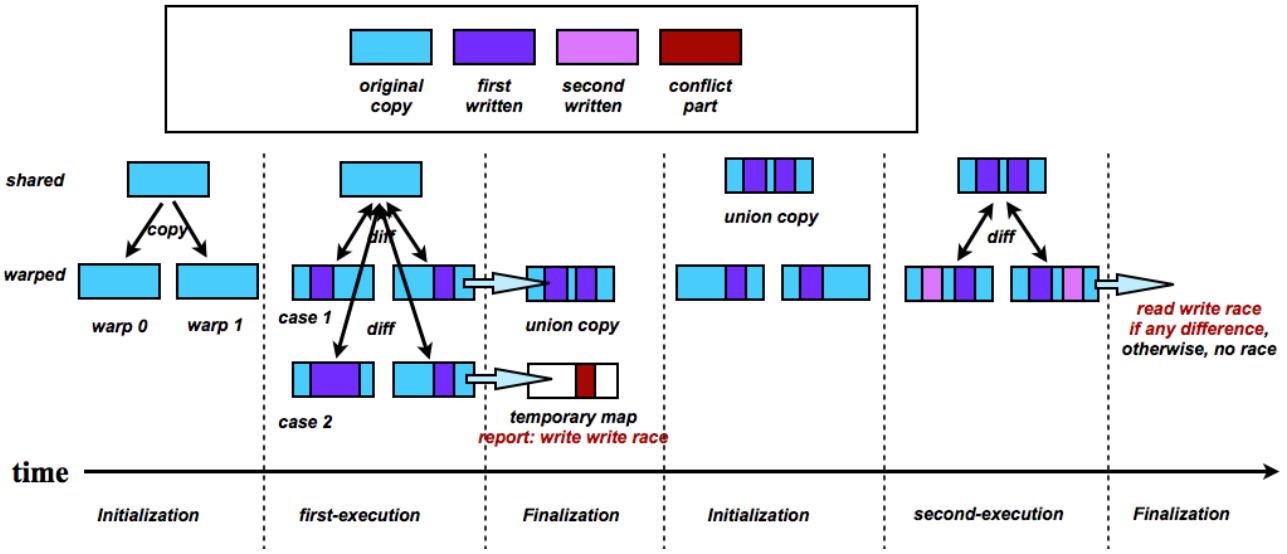

Figure 1: The series of data copying during race detection

```
1 __device__ void M_count(int * ... )
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) original code snippet

```
1 __device__ void M_count(int * ... )
2 {
3       extern __shared__float s_float[];
4       __shared__ float * s_float_rep[BLOCK_SIZE/
32] ;
5       __shared__ float s_float_union[2048] ;
        ...
6       if ( warpId % WARP_SIZE == 0 ) {
7           s_float_rep[warpId] = malloc(2048);
8           mem_cpy(s_float_rep[warpId], s_float,
2048*4) ;
9       }
10      s_float_new = s_float_rep[warpId] ;
11      __syncthreads() ;
        ...
12      float * Rp = s_float_new + index2 + 1536 ;
13      float * C = s_float_new + 3840 ;
14      float * temp1 = s_float_new + index0;
15      float * dist = s_float_new + index1+768;
16      ...
17      if (warpId%WARP_SIZE == 0) {
18          free(s_float_rep[warpId]);
19      }
20      __syncthreads();
21}
```

(b) data structure expansion

```
1 __device__ void M_count(int * ... )
2 {
3       extern __shared__float s_float[];
4       __shared__ float * s_float_rep[BLOCK_SIZE/
32] ;
5       __shared__ float s_float_union[2048] ;
        ...
6       if ( warpId % WARP_SIZE == 0 ) {
7           s_float_rep[warpId] = malloc(2048);
8           mem_cpy(s_float_rep[warpId], s_float,
2048*4) ;
9       }
10      s_float_new = s_float_rep[warpId] ;
11      __syncthreads() ;
        ...
12      float * Rp = s_float_new + index2 + 1536 ;
13      float * C = s_float_new + 3840 ;
14      float * temp1 = s_float_new + index0;
15      float * dist = s_float_new + index1+768;
16      ... replica from line 8 to line 27 of (a) ...
17      __syncthreads() ;
18      region_diff( s_float, s_float_rep[warpId], 2048 ) ;
19      region_union( s_float, s_float_rep[warpId], s_float
union, 2048) ;
20      ... replica from line 12 to line 16...
21      compare_output( s_float_union,
s_float_rep[warpId], 2048 ) ;
22      if (warpId%WARP_SIZE == 0) {
23          free(s_float_rep[warpId]);
24      }
25      __syncthreads();
26}
```

(c) data race detection example

Figure 2: A code snippet of EM benchmark program to illustrate the data structure expansion and data race detection. (a) shows an original program. (b) shows how to expand data structure. The size of shared array $float$ is 2048. (c) shows an example of race detection.

There is only one synchronization point at the termination of a kernel for different blocks, so the entire kernel is analogous to a block. Detection among blocks is similar to detection among warps. Then, we detect data races among warps in blocks. The detection process is exactly the same with the one for shared array.

Unlike shared array, the size of global array could be very large, limited by device memory. The data structure expansion for global array could explode memory consumption. So our two-pass approach is not fit for detecting data races in global array accesses. In the future work, we will study specific approaches to detect data races in global array.

A Limitation Our tool can have false negatives. For example, in the first run of write-write race detection, if two warps write to the

same position, but one of them writes with the initial values of the original copy, our tool cannot detect write-write races.

5. Speculative Parallelization

Nvidia's new Kepler architecture supports dynamic parallelism, in particular, launching a kernel inside a kernel asynchronously. It is like a *fork* in Linux and raises the possibility to speculative parallelization in GPU platform in the style of behavior-oriented parallelization on CPU [3, 8]. In the past, speculation happens on the host side. With the new support, it can be done on the device side. It allows GPU work to be speculated on, utilizing the abundant computing resources on GPU. In this section, we use GPU speculative parallelization to further reduce the performance overhead. To our best knowledge, this is the first of such design.

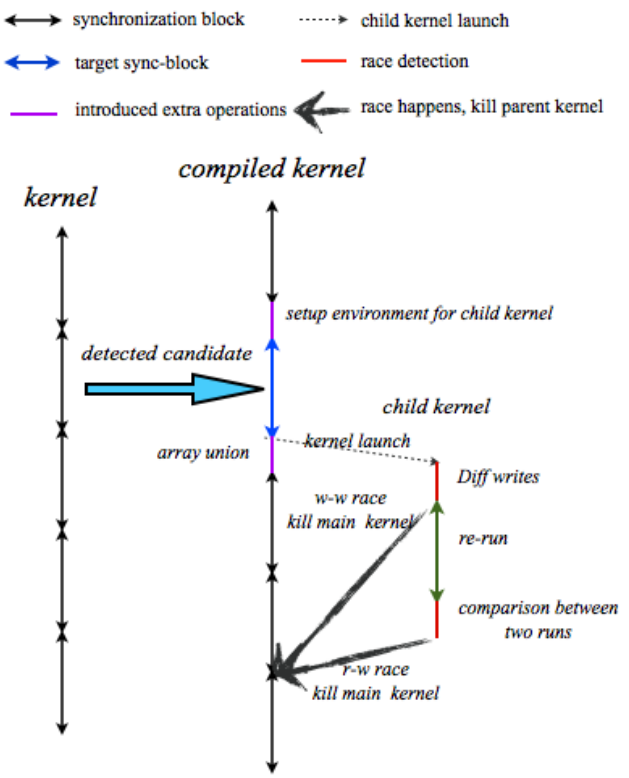

Figure 3: Utilize speculative parallelization to fast kernel execution

Speculative parallelization is only used in detecting inter-warp data races. For inter-block data races, because there is only one "synchronized block", using speculative parallelization cannot overlap computations. Inter-warp data race detection is performed in each thread-block. So launching a child kernel is done by thread 0 in each block, with a configuration *<<<1 , BLOCK_SIZE>>>*. It means that every block launches a block of the same number of warps to check data races. *BLOCK_SIZE* is the same number of threads with the kernel configuration launched by host side.

Figure 3 shows the execution model of speculative parallelization in GPU. The main kernel needs to store upwards-exposed variables for a child kernel, namely *setup environment* in Figure 3. After that, all warps in the main kernel start the first run. At the end of the first run, a child kernel is launched for write-write race detection. The second run and read-write race detection are also done in the child kernel. If a child kernel finds a race, it stops executing and reports it to parent kernel. After launching the child kernel, without any waits, all warps in the parent kernel assume there is no data race, merge warps' copies of target variables to one union copy, namely *array union* in Figure 3, and continue to execute. At the end of the following thread block, the main kernel checks if there are any races reported by a child kernel. If any, main kernel stops execution and reports results to the user. At the end of the kernel code, the main kernel collects child kernel's status and cleans it up. Note that, current Nvidia Kepler architecture does not support transferring shared memory addresses between the parent kernel and the child kernel. All the communication between them is stored in device memory, including the setup environment, data race reports and other data.

6. Preliminary Results

A preliminary evaluation has been done on a Nvidia Kepler-architecture GPU card, namely K20 [7]. EM and Co-clustering [19] benchmark programs were used to evaluate performance and memory overhead of our tools. They are both data mining algorithms and have been aggressively optimized to use shared arrays. The comparisons between our tools and Grace [19] are done in Table 3 and Table 4.

Programs	EM	Co-clustering
Native	60.27 (1x)	28.91 (1x)
LDetector	250.15 (4.15x)	99.46 (3.44x)
LDetector-spec	177.42 (2.94x)	107.04 (3.70x)
Grace-stmt	104445.9 (1733x)	1643781.88 (56858x)
Grace-addr	18866.83 (313x)	7236.8 (250x)

Table 3: Performance comparison between ***LDetectors*** and Grace running times in millisecond. The values in parentheses are speedups.

Performance Overhead Without zero memory access monitoring, our performance overhead is small, as shown by the running time of a kernel. Our performance overhead includes the memory allocation and release overhead of shared arrays and global arrays, the time spent in the run-time library, and the cost of the setup for the second-run as described in Section 4.2. If we measure the run time of a thread block, the performance overhead is 2x in theory, not including the library costs (which is much less than one millisecond as tested). Table 3 shows our performance in the two programs is less than 5x. Two orders of magnitude faster than the previous techniques, *Grace-stmt,Grace-addr*.

Speculative parallelization improves EM's performance, while slows down Co-clustering. The reason is, EM has three blocks, and the second one needs race detection. The computation of child kernels overlaps with the third synchronized block's execution of the main kernel. However, Co-clustering has only one block. There is no computing overlap when launching a kernel. As a matter of fact, launching a child kernel costs more, which makes the performance of ***LDetector-spec*** is slower than ***LDetector***.

Memory Overhead The memory overhead consists of the three copies of target variables and a bitmap shadow memory. Assume the size of a target variable is N bytes. At most one block has M warps, such as 32 in Nvidia Fermi and Kepler. The memory overhead is $3 * M * N + N/8$. The lifetime of these variables is limited within the block, so the same memory is reused for successive blocks. Table 4 shows our tools have much less memory cost than Grace [19], which shows in all but one case near or over 10 times reduction in the memory consumption. For *Grace-stmt* approach, we didn't instrument all memory accesses for EM and Co-clustering. Instead, we instrument as much as memory accesses

when memory overhead gets up to 1G bytes. As Grace [19] said, the *Grace-stmt* cannot work if instrumenting all memory accesses due to huge memory overhead.

Programs	EM	Co-clustering
LDetector	16K, 1M	16K, 1M
LDetector-spec	0K, 1.3M	0K, 1.3M
Grace-stmt	1.1K, 1000M*	1.1K, 1000M*
Grace-addr	1.1K, 54M	1.1K, 27M

Table 4: Comparing the memory overhead between ***LDetectors*** and Grace in bytes. The left value in table cell denotes shared memory usage volume and the right value denotes global memory usage volume.

7. Related work

A growing number of techniques on data race detection on GPUs [2, 6, 9, 10, 19, 20] have been proposed. [2, 6] used instrumentation to record all runtime memory accesses from different threads and different warps to detect data races. The instrumentation caused orders of magnitude performance degradation. Grace [19] and GMRace [20] tried to reduce the performance overhead of instrumentation through compiler analysis. However, static analysis is not effective in the presence of irregular memory access. The authors only show its effectiveness against three programs. A number of methods were studied for kernel verification [1, 10, 11]. Bretts et al. [1] proposed a new programming semantic in GPU, under which data races can be detected at programming time. Li et al. [10] proposed PUG, using a Satisfiability Modulo Theories-based technique to detect data races. GKLEE [11] extended symbolic analysis in GPU platform for correctness checking. Since SMT solver and GKLEE symbolic analysis are static, they have more false positives than run-time methods. Leung et al. [9] proposed a combination of static analysis and dynamic analysis based on information flow. Their performance and memory overhead are low, about 18x slowdown on average. Our technique, measured on our benchmarks, has 5x slowdown on average.

Data privatization is standard in auto-parallelization of sequential programs. It replicates shared data and creates a private copy for each thread [4, 5, 12, 13]. Recently, Yu et al. [18] proposed a general approach for privatization to parallelize sequential loops.

8. Conclusion and Future Work

This paper presented new data race detection solutions optimized for high performance, because it has zero memory access monitoring overhead and fully utilizes the massive parallelism of the GPU in race checking. The tools use compiler support of data structure expansion for shared variables, a new two-pass approach for race detection, and the speculation to overlap race checking and program execution. Preliminary results show significant performance improvement and space overhead reduction over a recently published technique. Here are several opportunities of future work:

- Extending our tool to support atomic operations in GPU programming model is one future direction. Atomics allows safe inter-warp data sharing. Our current tools are designed based upon an assumption that no atomic operations exist in the synchronized blocks. We plan to propose a more generic data race detection approach.
- We have to face the problem of memory consumption explosion when detecting data races in global memory. In the future, we will investigate specific approaches for data races in global memory.

Acknowledgments

We thank Li Lu for helpful discussions and the anonymous reviewers for insightful feedback on this work.

References

[1] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. Gpuverify: A verifier for gpu kernels. In *Proceedings of OOPSLA*, pages 113–132, 2012.

[2] M. Boyer, K. Skadron, and W. Weimer. Automated Dynamic Analysis of CUDA Programs. In *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[3] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of PLDI*, pages 223–234, 2007.

[4] J. Gu, Z. Li, and G. Lee. Experience with efficient array data-flow analysis for array privatization. In *Proceedings of PPoPP*, pages 157–167, 1997.

[5] M. Gupta. On privatization of variables for data-parallel execution. In *Proceedings of the 11th International Symposium on Parallel Processing*, pages 533–541, 1997.

[6] Q. Hou, K. Zhou, and B. Guo. Debugging gpu stream programs through automatic dataflow recording and visualization. In *ACM SIGGRAPH Asia 2009 Papers*, 2009.

[7] N. K. K20. http://en.wikipedia.org/wiki/nvidia_tesla.

[8] C. Ke, L. Liu, C. Zhang, T. Bai, B. Jacobs, and C. Ding. Safe parallel programming using dynamic dependence hints. In *Proceedings of OOPSLA*, pages 243–258, 2011.

[9] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner. Verifying gpu kernels by test amplification. In *Proceedings of PLDI*, pages 383–394, 2012.

[10] G. Li and G. Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. 2010.

[11] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. Gklee: Concolic verification and test generation for gpus. In *Proceedings of PPoPP*, pages 215–224, 2012.

[12] Z. Li. Array privatization for parallel execution of loops. In *Proceedings of ICS*, pages 313–322, 1992.

[13] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 2–15, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. . URL `http://doi.acm.org/10.1145/158511.158515`.

[14] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of ASPLOS*, pages 235–244, New York, NY, USA, 1991.

[15] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. Sigrace: Signature-based data race detection. In *Proceedings of ISCA*, pages 337–348, 2009.

[16] M. Prvulovic. Cord: cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proceedings of HPCA*, pages 232–243, 2006.

[17] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of ISCA*, pages 110–121, 2003.

[18] H. Yu, H.-J. Ko, and Z. Li. General data structure expansion for multi-threading. In *Proceedings of PLDI*, pages 243–252, 2013.

[19] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. Grace: A low-overhead mechanism for detecting data races in gpu programs. In *Proceedings of PPoPP*, pages 135–146, 2011. .

[20] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. Gmrace: Detecting data races in gpu programs via a low-overhead scheme. *IEEE Trans. Parallel Distrib. Syst.*, 25:104–115, 2014.