HAccRG: Hardware-Accelerated Data Race Detection in GPUs
Anup Holey, Vineeth Mekkat, Antonia Zhai
Department of Computer Science & Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455, USA
{aholey, mekkat, zhai}@cs.umn.edu

Abstract—Modern Graphics Processing Units (GPUs) are capable of supporting thousands of concurrent threads. However, they provide relatively little guarantee with respect to the coherence and consistency of the memory system. Thus, GPUs are prone to multitude of concurrency bugs related to inconsistent memory states. Many such bugs manifest as some form of data races at runtime, and being able to identify these data races can help programmers improve software reliability. Mechanisms that enable efficient and effective data race detection at runtime can form the basis of powerful tools for enhancing GPU software correctness. Most prior works in data race detection for GPU focus on the software-based approaches that incur significant performance overhead. Furthermore, they often focus on the smaller shared memory, while neglecting the larger global memory. We believe that adequate hardware support can enable efficient data race detection in all levels of the memory system for GPUs.

In this paper, we propose a hardware-accelerated data race detection mechanism, HAccRG, for efficient data race detection in GPUs. HAccRG provides hardware support for tracking data dependencies across a large number of threads and detects various forms of data races. We incorporate HAccRG on both the shared and global memory spaces in GPU. Our evaluation shows that, with moderate hardware support, HAccRG can detect data races in GPU kernels with a small overhead: 1% for the shared memory and 27% for combined shared and global memory data race detection.

Keywords—GPUs; data race detection; concurrency bugs

I. INTRODUCTION

Graphics Processing Units (GPUs) are becoming increasingly popular not only across various scientific communities, but also as integrated data-parallel accelerators on existing multicore processors. Support for massive fine-grained parallelism in contemporary GPUs provides a tremendous amount of computing power. Unlike CPUs, GPUs support thousands of lightweight threads to deliver very high computational throughput. GPU is popularized by the easy-to-adapt programming models such as CUDA [19] and OpenCL [10], allowing programmers to port existing CPU applications to the GPU domain with minimal effort. However, designing and implementing correct and efficient GPU programs remains a challenge since programmers must consider interaction between thousands of parallel threads. Many concurrency bugs are results of different threads perceiving memory states differently at runtime. Such differences are often a manifestation of data races present in the programs.

A data race occurs when more than one thread simultaneously accesses the same memory location, and at least one of the accesses is a write. Being able to detect these data races at runtime can facilitate the construction of powerful tools to improve the reliability of GPU applications. While a large body of work exists on detecting data races between CPU threads [15, 16, 20, 21], these techniques cannot be directly extended to GPU threads for both correctness and performance reasons. From a correctness perspective, the causes of data races are multifaceted in contemporary GPUs. Not only do improperly placed synchronizations and critical sections cause data races, but the lack of memory coherence support also introduces data races that otherwise would not occur. Most CPU-based multicore systems are coherent and are not subject to the latter. From a performance perspective, many proposed solutions require mechanisms for tracking memory accesses per-thread to detect data races between CPU threads. For modern GPUs that are capable of executing thousands of threads simultaneously, tracking per-thread accesses poses challenges both in terms of performance as well as hardware overhead. Thus, providing an efficient and scalable data race detection mechanism for the GPU becomes a significant challenge.

There have been several recent efforts addressing software correctness in GPUs [11, 14], including data race detection [6, 8, 12, 13, 26, 27]. These work detect data races statically or at runtime by instrumenting GPU applications. Instrumentation introduces significant performance degradation, rendering such techniques inefficient. Adequate hardware support can potentially improve the effectiveness and efficiency of runtime data race detection for GPUs. In this paper, we design a hardware-accelerated data race detection framework for modern GPUs, referred to as HAccRG. Such hardware support is responsible for tracking and comparing the memory accesses from all threads to detect data races. To the best of our knowledge, HAccRG is the first hardware implementation for detecting data races in the GPU.

A straightforward implementation of data race detection requires pairwise comparisons of all memory traces from all threads. The number of comparisons is quadratic relative to the number of threads. Scaling this implementation on GPUs that typically support thousands of threads is impractical. Therefore, HAccRG employs a per-memory access tracking mechanism, instead of tracking per-thread accesses. This method allows us to design a fast hardware data race detector with moderate hardware overhead. The proposed hardware support covers both the shared and global memory spaces of a GPU. We implement HAccRG on a cycle-accurate general purpose GPU (GPGPU) simulator and evaluate its effectiveness using a set of CUDA benchmarks. Our evaluation shows that HAccRG is able to accurately detect real and injected data races in GPU programs with an average performance overhead of only 1% for the shared memory and 27% for combined shared and global memory data race detection. We also compare HAccRG with its software implementation as well as with an existing instrumentation-based runtime data race detection mechanism called GRace [26]. Our experience shows that hardware can accelerate data race detection by order of magnitude than the software approaches. Furthermore, the software implementation of HAccRG is more efficient than GRace in terms of both performance as well as space overhead.

II. BACKGROUND

A. GPU Execution Model

GPUs support thousands of threads executing in parallel. Such computing power is provided by an array of computing cores, often
referred to as streaming multiprocessors (SM). Each SM consists of an array of simple in-order cores that are referred to as streaming processors (SP). SPs located within a single SM execute the same instruction but operate on different data in a given cycle, an execution model known as single instruction multiple data (SIMD).

Work is allocated to the GPU as kernels that contain a large number of threads. Threads within the same kernel are organized into blocks, and blocks are mapped onto different SMs. At execution time, threads within the same thread-block are further partitioned into warps [19] or wavefronts [2]. Threads within the same warp are scheduled to the SIMD computing engine simultaneously, and thus are executed in lockstep. Threads across different warps are executed asynchronously.

GPU contains an on-chip memory in each SM, explicitly managed by a programmer, referred to as the shared memory. The shared memory is banked [19] to enhance throughput. If threads within a warp access different banks, all the accesses are served in parallel. Global memory, a part of the off-chip device memory, is accessible to all threads in a GPU kernel. Local memory is private to individual threads, although, it also resides in the off-chip device memory [19].

Consecutive accesses to both global and local memory from different threads in a warp are coalesced, i.e., combined into a single larger access to compensate for higher memory access latency [19]. The latest GPUs also support caching. Global and local memory are cached in per-SM non-coherent L1 data caches and a coherent shared unified L2 cache. Since global memory is accessible to all threads in a GPU and L1 caches are non-coherent, global memory writes to L1 data cache are written through to the corresponding L2 cache banks [18].

GPUs also support atomic operations in hardware that are used to implement critical sections in GPU kernels. CUDA and OpenCL support atomic operations in the shared and global memory spaces [10, 19].

**B. Data Races in GPUs: Case Studies**

In this section, we present two classes of data races in GPUs.

```c
1. __device__ uint count = 0;
2. __device__ void race_example (int *in, int *out)
3. {
4.   int tid = threadIdx.x;
5.   for(int i=1; i<32; i++) {
6.     out[tid] = foo(in, tid, i);
7.   } // missing memory fence
8.   if(blockDim.x-1 == atomicInc(&count, blockDim.x)) {
9.     out[0] = out[0]+out[1]+...+out[blockDim.x-1];
10.    count = 0;
11.   }
12.  // missing barrier synchronization
13. }
14. }
```

**Fig. 1.** Two sources of global memory data races in a GPU kernel: missing memory fence (line 7) and missing barrier synchronization (line 12).

**Lack of correct synchronization:** In the example shown in Figure 1, threads within the same thread-block update the array `out` in global memory at the index `tid` in line 6. Each thread atomically increments a global variable `count` (initialized to 0) in line 8. Every atomic increment operation returns the old value of `count`. The last thread from the block that increments the variable `count` reads the global array modified by all threads in the block and writes the final sum to `out[0]` in line 9. The same thread resets the `count` to 0. The entire process is repeated in a `for` loop.

Because of the `for` loop in line 5, all threads but the last one, after atomically incrementing `count`, immediately loop back and update the `out` array. It can happen that before the last thread reads the `out` array in line 9, the other threads modify `out` again. A barrier synchronization (e.g. `syncthreads` [19]) in line 12 is required to ensure correct execution. Note that, memory accesses from threads within the same warp are always ordered. The barrier synchronization ensures correct ordering of memory accesses across all warps in a thread-block.

**Fig. 2.** Data races in critical sections on a GPU. (a) A data race that occurs in both GPU and systems with coherence support. (b) A data race due to missing fence that only occurs in GPU.

**Incorrect use of locks:** Locks allow multiple threads to read or write shared variables atomically by serializing the accesses from different threads. We describe two data races that can occur while implementing locks. Figure 2(a) shows execution of a critical section in a GPU kernel. Threads `T1` and `T2` acquire locks `L1` and `L2`, respectively. Thread `T1` writes to an address `A` within its critical section, while thread `T2` reads the same address before thread `T1` releases its lock `L1`, resulting in a data race. This type of data race can occur in both GPU and systems with coherence support.

**Fig. 2(b) shows two threads `T3` and `T4` contending for the same lock `L3`, which causes both threads to execute their critical sections sequentially. Thread `T3` writes to an address `A` within its critical section, while thread `T4` reads the same address upon acquiring the lock `L3`. However, since GPU does not guarantee memory access ordering across threads, it is possible that `T4` finds the lock variable updated before the write to the address `A` is complete. Thus, thread `T4` can read the old value at address `A`. This can be avoided by...
calling a memory fence function after write to address A and before T3 releases lock L3. This type of data race can occur only in GPU.

III. DATA RACE DETECTION FRAMEWORK FOR GPU

Efficient and effective runtime data race detection mechanisms for GPU systems are the basis for implementing powerful tools for facilitating software development as well as enhancing software correctness and reliability. In this paper, we propose a Hardware-Accelerated data Race detection tool for GPU (HAccRG), a low overhead, high accuracy data race detection mechanism for GPU. In the rest of this section, we outline the design issues and basic mechanisms of HAccRG.

There are two common techniques for detecting data races at runtime: comparing set of locks held by threads when accessing shared data items or establishing happens-before relationship between accesses. Lockset-based data race detection technique tracks the set of locks held by each thread and reports a data race when different threads access the same memory location without holding a common lock, with at least one write access. In happens-before-based data race detection technique, thread execution is partitioned into epochs using synchronization events. Such epochs are considered concurrent if their execution times overlap, while a data race is reported when concurrent epochs from different threads access the same memory location with at least one write. Although both techniques are powerful, none of them covers all the data races. The lockset-based detection only covers data races caused by improper use of locks. In addition to the runtime data races, it also reports the potential data races that do not occur at runtime. The happens-before-based detection considers all synchronization events, but only reports data races that occur at runtime, while potentially missing some data races that are not exposed due to thread scheduling. HAccRG uses happens-before mechanism to detect data races caused by incorrect barrier synchronizations and fences, and lockset mechanism to detect data races in critical sections.

To detect data races, memory accesses from different threads must be tracked. There are two approaches for recording memory access history: aggregating memory accesses for each thread or tracking memory accesses for each memory location. In the first approach, memory accesses from each thread in a GPU are compared against each other. Since the number of comparisons grow quadratically with the number of threads, this approach is impractical for GPU systems with thousands of concurrent threads. Thus, HAccRG takes the second approach that keeps track of thread(s) that accessed each memory location. This approach is particularly attractive for GPUs, because the number of data race comparisons has a linear relationship with the number of memory accesses and the space overhead grows linearly with the application memory size.

The metadata for data race detection for a shared data is stored in shadow memory. Consider a memory location M: accesses to M are tracked in the shadow memory at location \( M_{\text{shadow}} \). Each access to memory location M invokes a concurrent access to the corresponding \( M_{\text{shadow}} \) entry. In the rest of this section, we describe what information is tracked by the shadow memory; and how different data races are detected using this information. The hardware/software support necessary for implementing the shadow memory depends on the performance characteristics of the shared data, and is described in details in Section IV.

A. Races Between Barrier Synchronizations

In the GPU, barrier instructions ensure that memory accesses issued before the barrier are completed before instructions following the barrier are executed. Hence, memory accesses across barriers are ordered, however, all accesses between two barrier synchronizations are concurrent. Between two barriers, a data race can occur when different threads read and write shared data. The start and the end of a kernel are implicit barriers.

In HAccRG, when a thread accesses a memory location, the corresponding \( M_{\text{shadow}} \) entry is also checked and updated. For each memory location, the corresponding shadow entry has three fields: the \( \text{tid} \) field containing the thread ID of the first thread that accessed the given memory location; the \textit{modified} (M) field indicating whether the memory location has been written to by a thread; and the \textit{shared} (S) field indicating whether the memory location has been read by more than one thread.

Figure 3 shows the state transitions of the \( M_{\text{shadow}} \) entries. At the beginning of the execution, the shared and modified fields in the shadow entries are set to true, indicating that there has been no access to the memory locations. When a thread-block reaches barrier synchronization, the modified and shared fields in the shadow memory entries are set to true. When a memory access is issued, a shadow entry can be in one of the four states:

- State 1 (M=true, S=true): There is no prior access to the corresponding memory location. HAccRG performs the following operations for memory access from thread T: i) reset S to false because this is the first access; 2) reset M to false if the access is a read; and 3) set the \textit{tid} to the thread identifier of T.
- State 2 (M=false, S=false): There have only been read accesses from a thread with its ID stored in \textit{tid}. For a read access, no action is necessary if the same thread accesses the location; otherwise set S to true. The shared field (S) is required because we only store the ID of the thread that has first accessed the given memory location. For a write access, set M to true for access from the thread \textit{tid}; otherwise report a data race of type write-after-read (WAR).
- State 3 (M=true, S=false): There have been at least one write access from a thread with its ID stored in \textit{tid}. For a read access, no action is necessary for access from the thread \textit{tid}; otherwise report a data race of type read-after-write (RAW). For a write access, no action is necessary for access from the same thread \textit{tid}; otherwise report a data race of type write-after-write (WAW).

![Data race detection process](image)
• State 4 (M=false, S=true): There have been read accesses from more than one thread. For a read access from any thread, no action is necessary. For a write access from any thread report a data race of type write-after-read (WAR). The shared field is set to true when more than one thread reads the location; hence, any write to the same location is a potential data race.

Impact of Warps on Reporting Races: After analyzing the shadow entries, HAccRG reports data races for instructions between two barrier synchronization points only if the two threads belong to different warps. The threads within the same warp do not create a data race condition since all threads in a warp are synchronized. Therefore, for every memory access by a warp, it is guaranteed that the previous memory accesses by the same warp are complete. The shared field in the shadow entry is set to true when a read occurs from a different warp than that of thread tid. However, HAccRG does detect write-after-write violations within the same warp before the memory request is issued.

When warp re-grouping occurs dynamically at runtime [7], threads that originally belonged to different warps are merged into a single warp. Consequently, for these threads some memory accesses would come from different warps. Thus, when warp re-grouping is enabled, HAccRG reports data races regardless of the warp considerations.

B. Races in Critical Sections

GPUs support atomic operations for implementing critical sections to ensure sequential access to shared data. When a memory location is accessed by different threads holding a common lock, the accesses to that memory location are serialized. To detect data races related to incorrect acquisition and release of locks, HAccRG supports lockset-based data race detection [23].

In HAccRG, a per-thread register, referred to as the atomic ID, records the set of locks held by each thread. When a thread acquires a lock, the lock variable is entered into the atomic ID; when the lock is released, the corresponding entry is removed from the atomic ID. The register is a small Bloom filter [5] that keeps track of lock variables, similar to the prior work for detecting CPU-based data races [28].

A Bloom filter signature is a bit vector divided into multiple bins. When an address is added into the signature, one bit selected using a hash function in each of the bins is set to 1. To remove addresses, we simply clear the signature when a thread releases all the lock variables held. Most of the CPU applications use single level locks, while the ones that use nested locks have very small nesting levels [22, 28]. The GPU applications also follow similar trends. Therefore, clearing the Bloom filter signatures provides a low overhead mechanism to remove lock variables from atomic IDs. Atomic ID is attached to each memory request that is issued within a critical section, and is stored in the shadow entry along with the other fields. A more accurate look-up table based approach for tracking lock variables can also be adopted, however we choose Bloom filter due to its low hardware overhead. The effect of Bloom filter signature size and number of bins on data race detection accuracy is discussed later in Section VI.

To determine whether two or more threads have accessed a shared variable without holding a common lock, HAccRG tracks the set of common locks protecting each shared variable at runtime. This is obtained by intersection of the set of locks protecting a shared variable so far, with the set of locks held by the thread accessing the shared variable. The intersection of Bloom filter signatures is a simple bitwise AND operation. The atomic ID field in the shadow entry indicates the set of locks protecting the shared variable so far, while the atomic ID register of the accessing thread represents the current set of locks held by that thread.

In critical sections, lockset-based detection has priority over barrier synchronizations. For the first access to a memory location (M=true, S=true), the lock variable is stored in the atomic ID if the access is protected, otherwise the atomic ID is set to 0 in the shadow entry. For later protected accesses, the intersection of the atomic ID in the shadow entry and the atomic ID of the accessing thread is stored in the shadow entry. A data race can occur in two different scenarios when using locks.

• Accesses using different locks: For a protected access by a thread other than tid, if the shadow entry shows a non-zero atomic ID, a data race is reported if the modified field is true (write by tid) or the current access is a write, and if the intersection of the atomic ID in the shadow entry and the atomic ID of the accessing thread is null. A null intersection indicates that no common locks are used to access a shared variable.

• Unprotected accesses: When accesses to a memory location involve protected and unprotected accesses from different threads, a data race can occur if one of the accesses is a write. When the atomic ID in the shadow entry is non-zero and the current access is unprotected, or the atomic ID in the shadow entry is 0 and the current access is protected, a data race is reported when either the modified field is true or the current access is a write.

In HAccRG, we distinguish between memory accesses issued within and outside critical sections. To detect the start and end of a critical section, we insert marker instructions after lock acquire and before lock release operations in the kernel code. These operations in GPU are often implemented using atomic compare-and-swap or atomic exchange instructions [10, 19]. Compiler support can also be used to insert the marker instructions automatically. Only for accesses related to critical sections, lockset-based detection is performed as described above. For other accesses, happens-before detection is used to detect data races.

C. Races Due to Improper Memory Fencing

In GPU, memory access ordering across threads is non-deterministic, as discussed earlier in Section II-B. Programmers use memory fencing functions to impose explicit ordering on memory requests. A fence operation in CUDA ensures that modifications made by a thread are visible to other threads in a GPU before the instructions following the fence are executed by that thread [19].

Fig. 4. Detecting memory fence races in HAccRG. (a) Potential data race as thread T1 consumes T0’s updates without fence execution by thread T0. (b) Safe access as thread T0 executes fence before performing atomic operation.
To understand importance of fences, consider two threads T0 and T1 that have producer-consumer relationship as shown in Figure 4. In Figure 4(a), thread T0 writes to location X, while thread T1 reads from location X. The order between T0 and T1 is ensured through an atomic operation over variable A. Such interaction between threads can be implemented without using explicit critical sections, similar to the example shown in Figure 1 where an atomic increment operation over the count variable creates a synchronization point for all threads. At the time of read access by thread T1, HAccRG determines that thread T0 has not executed memory fence function by monitoring T0’s fence epoch f0, and flags the access as data race. If a fence call is inserted as shown in Figure 4(b) after write to X, HAccRG flags the read access by T1 as safe since T0’s fence epoch changed from f0 to f1. Similarly, HAccRG can also detect data races occurring in critical sections due to missing fences (Figure 2(b)).

To monitor fence calls in GPU, HAccRG maintains a per-warp fence ID that is incremented by one when a warp completes a memory fence call. Fence ID is a per-warp logical clock that keeps track of execution of fences by a warp, which is sent along with the memory requests. The fence ID of a thread’s warp is stored in the shadow entry with the other fields. For the first access to a memory location (M=true, S=true), the shadow entry fields are set along with the fence ID. For all other accesses, if the modified field is set and the new access is a read, the fence ID stored in the shadow entry is compared with the current fence ID of the warp that belongs to the thread indicated by tid. A match indicates that the thread tid has not yet executed a fence instruction since its last write to the same memory location. Therefore, there is a data race since a different thread is reading the same memory location. A mismatch means the thread tid has completed its fence call since its last write to the same memory location, which indicates that other GPU threads can safely consume tid’s updates before the fence call.

IV. HAccRG IMPLEMENTATION

In this section, we describe Race Detection Unit (RDU), the hardware support necessary for accelerating the detection of data races in GPU. Since GPUs have two separate memory modules, the shared and the global memory, two independent RDUs are designed to match the performance and bandwidth requirements of these memory modules. While they do not alter memory accesses originated from the cores, they are responsible for generating shadow memory accesses and matching these accesses with the corresponding memory accesses.

A. Data Race Detection in Shared Memory

Since the shared memory is private to each SM, the shared memory RDU is present in each SM for detecting data races among threads in the same SM. The shared memory is small and fast, and hence the shared memory shadow entries are stored in hardware for faster accesses. This is achieved by extending each shared memory entry with its shadow memory information, as shown in Figure 5.

For every shared memory access, the RDU accesses the corresponding shadow entries and performs data race detection. When a thread-block reaches barrier synchronization, RDU resets all the shadow entries that belong to the block by setting their shared and modified fields to true. Note that the shared memory RDU accesses fence IDs in its own SM since shared memory is accessed only within an SM.

B. Data Race Detection in Global Memory

The global memory is off-chip and is much larger than the shared memory in size. Thus, it is impractical to provide explicit hardware support to track memory accesses. We reserve a section in the global memory, referred to as global shadow memory, to record the metadata necessary for data race detection. The shadow entries in the global shadow memory have a one-to-one correspondence with the entries in the kernel data structures. The global shadow memory is allocated when the kernel is launched using the cudaMalloc [17] API. At execution time, the global memory RDUs automatically generate requests and updates to the shadow memory. When the kernel terminates, the cudaMemset [17] API is invoked to invalidate all global memory shadow entries. Since the global memory is accessible to all threads across all thread-blocks, we augment the shadow memory entries with the bid field for the block ID, and the sid field for the originating SM of the access. Since multiple thread-blocks can access the global memory, the shared field in the global shadow entries is set to true if two different warps or thread-blocks read the same global memory location. Consequently, a data race is reported if different thread-blocks perform read-write accesses to a shared data without using locks or memory fences.

Since global memory shadow entries are part of device memory, setting the modified and shared fields to true after barrier synchronization has additional performance overhead. To avoid this overhead, we add one more field in the global memory shadow entries, referred to as sync ID, to notify execution of barrier instructions. Each SM maintains per-thread-block sync ID, and increments a thread-block’s sync ID when the thread-block reaches barrier synchronization and only if the block has accessed the global memory since its last barrier call. This avoids unnecessary increments to sync IDs after every barrier call. The sync ID is a logical clock which keeps track of barriers executed by a thread-block. It is sent with each global memory request and stored in the shadow entry. The sync ID in the shadow entry and the one sent with the memory request are compared when the request comes from the same thread-block as that of tid. If the sync IDs match, HAccRG checks the shadow entry for data race as shown in Figure 3. If the sync IDs are different, the shadow entry is updated with the current access information. When a memory request comes from a thread-block other than that of tid, sync ID check is not required because scope of barrier is limited to a thread-block. For such accesses, the shadow entry is analyzed for detecting a data race and then updated.

Hardware support for global memory data race detection are shown in Figure 6. The global memory RDU, a hardware module present in
Systems with coherent caches do not face such issues. HAccRG is capable of reporting them when L1 caching is enabled.

At the time of data race detection, a fence ID is read replicated in every global memory RDU for faster access, as shown in Figure 6. Global memory RDU detects a read-after-write access across SMs and data races can be avoided by disabling caching of global memory data if the read is an L1 hit, HAccRG reports a data race. Although such requests are checked associatively to detect a possible data race. When multiple requests accessing the same global memory location enter a global memory RDU simultaneously, the requests are checked associatively to detect a possible data race.

Accessing Fence IDs: Memory fences in the GPU provide memory consistency across threads by imposing memory access ordering. RDUs access the fence IDs maintained by the SMs, to detect data races as described in Section III-C. Global memory RDUs require access to fence IDs from all the SMs in GPU. Therefore, the fence IDs from all the SMs are stored in the race register file, which is replicated in every global memory RDU for faster access, as shown in Figure 6. At the time of data race detection, a fence ID is read only if the access checked for race is a read and the corresponding shadow entry has the modified field set. A data race is reported if the read fence ID matches with the one stored in the shadow entry. We discuss the sizes of sync IDs, fence IDs, and atomic IDs in Section VI-A2.

Effect of L1 Caches on Detecting Global Memory Races: Non-coherent L1 data caches have been introduced in recent GPUs [18], enabling caching of global memory data in SMs. For global memory reads that hit in L1 data cache, HAccRG sends data race detection requests to the corresponding global memory RDU. When thread-blocks executing on different SMs communicate through the global memory, it is possible that an SM does not get the global memory updates from other SMs because of the stale data in its own L1 data cache. HAccRG handles this case by sending the read hit information with the data race detection request to the global memory RDU. When global memory RDU detects a read-after-write access across SMs and if the read is an L1 hit, HAccRG reports a data race. Although such data races can be avoided by disabling caching of global memory data in the L1 caches or by declaring the shared variables as volatile [19], HAccRG is capable of reporting them when L1 caching is enabled. Systems with coherent caches do not face such issues.

Supporting Virtual Memory: Virtual memory simplifies programing by assigning separate address space to each process, thus offloading memory management overhead from the applications. Although virtual memory has traditionally been implemented only for CPUs, recent GPUs have also deployed virtual memory [1, 9]. To track global memory accesses using shadow memory in such architectures, HAccRG should be able to support virtual memory.

Virtual memory systems use page tables for translating virtual addresses to physical addresses. In systems like Intel Sandy Bridge and AMD Fusion, page tables for CPU and GPU are separately maintained [1, 9]. We propose an on-demand paging for shadow memory in HAccRG. Shadow memory pages are allocated when GPU’s application memory pages are generated, i.e., on-demand. However, shadow memory pages are allocated only for global memory space. A one-bit field in the GPU page table entry can indicate if the page belongs to the global memory space.

Modern processors rely on translation lookaside buffer (TLB) to speed-up the virtual address translation mechanism. Intel Sandy Bridge and AMD Fusion provide independent TLBs for GPU address translation. In HAccRG, when GPU’s global memory accesses go through TLB, they should be able to get shadow memory locations along with the application memory. We propose two mechanisms to incorporate dual address translations in the TLB. A TLB is a cache like structure. The first mechanism appends 1-bit to the tag fields in the GPU TLB. This bit is set to 1 when a TLB entry points to a shadow memory page. During address translation, two tags are searched in the TLB: with the appended bit 0 and 1. This approach can potentially reduce the effective TLB capacity for regular (non-shadow) memory entries. In the second mechanism, we propose a separate TLB structure for shadow memory pages. GPU memory accesses go through regular as well as shadow memory TLB. Shadow memory TLB can be smaller than the regular TLB since all GPU pages do not belong to the global memory space. This approach provides faster TLB accesses than the first one. With the aforementioned modifications, HAccRG can be easily adapted for GPUs having virtual memory support.

C. Accuracy Trade-offs in HAccRG

Accurate runtime data race detection is often costly in terms of additional resources required, such as hardware and memory. To tackle limited resources, data race detection accuracy can be reduced. By mapping one shadow entry to one or more consecutive elements in the program’s memory space, the storage overhead of HAccRG can be adjusted. We refer to such mapping as tracking granularity of HAccRG. One-to-one correspondence between shadow entries and elements in the kernel data structures does not report any false positives, while one-to-many correspondence can report false positives. However, varying the tracking granularity changes the hardware as well as the memory requirements of HAccRG. Making the granularity coarser reduces these overheads significantly. We evaluate the data race detection trade-offs in HAccRG in Section VI.

V. EVALUATION METHODOLOGY

We implement the proposed HAccRG changes by extending a detailed GPGPU simulator (GPGPU-Sim 3.0.2 [3]). GPGPU-Sim is configured to model NVIDIA Quadro FX5800 GPU. L1 data caches and a unified L2 cache have been modelled after NVIDIA Fermi GPUs [18]. The L1 data caches are non-coherent, while the unified L2 cache is coherent. GPGPU-Sim simulates timing for the SMs, the interconnection network, the memory controllers, and the GDDR3 memory. The GPU hardware parameters are provided in Table I.
TABLE I

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Inputs</th>
<th>Shared Memory</th>
<th>Global Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCARLO</td>
<td>256 elements, 64K paths</td>
<td>0.3% Inst. 0.3% Shared Reads</td>
<td>6.5% Inst. 6.5% Shared Reads</td>
</tr>
<tr>
<td>SCAN</td>
<td>512 elements</td>
<td>10.7% Inst. 10.7% Shared Reads</td>
<td>1.1% Inst. 1.1% Shared Reads</td>
</tr>
<tr>
<td>FWALSH</td>
<td>Data length 512K, Kernel length 32</td>
<td>9.6% Inst. 9.6% Shared Reads</td>
<td>7.5% Inst. 7.5% Shared Reads</td>
</tr>
<tr>
<td>HIST</td>
<td>Byte count 16M</td>
<td>22.9% Inst. 22.9% Shared Reads</td>
<td>1.4% Inst. 1.4% Shared Reads</td>
</tr>
<tr>
<td>SORTNW</td>
<td>12K elements, 2K values</td>
<td>17.1% Inst. 17.1% Shared Reads</td>
<td>2.6% Inst. 2.6% Shared Reads</td>
</tr>
<tr>
<td>REDUCE</td>
<td>1M elements</td>
<td>19.6% Inst. 19.6% Shared Reads</td>
<td>18.6% Inst. 18.6% Shared Reads</td>
</tr>
<tr>
<td>PSUM</td>
<td>16K elements</td>
<td>0.4% Inst. 0.4% Shared Reads</td>
<td>87.2% Inst. 87.2% Shared Reads</td>
</tr>
<tr>
<td>OFFT</td>
<td>meshW=256, meshH=256</td>
<td>8.5% Inst. 8.5% Shared Reads</td>
<td>2.6% Inst. 2.6% Shared Reads</td>
</tr>
<tr>
<td>KMEANS</td>
<td>mesh=100, dx=10</td>
<td>0.04% Inst. 0.04% Shared Reads</td>
<td>17.4% Inst. 17.4% Shared Reads</td>
</tr>
<tr>
<td>HASH</td>
<td>256K-entry table, 16K elements</td>
<td>0% Inst. 0% Shared Reads</td>
<td>17.8% Inst. 17.8% Shared Reads</td>
</tr>
</tbody>
</table>

At the end of every barrier synchronization, we simulate the extra clock cycles required to invalidate the shared memory shadow entries. For the global memory data race detection, we model interconnection network traffic from the SMs to the global memory RDUs in the memory slices. The network packets carry sync IDs, fence IDs, and atomic IDs along with the other control information.

We evaluate the effectiveness and performance of HAccRG using a set of CUDA applications listed in Table II. Seven of the ten benchmarks are taken from NVIDIA’s CUDA SDK [17]. MCARLO [17] is the Monte Carlo option pricing algorithm. SCAN [17] is the parallel prefix sum algorithm. FWALSH [17] is CUDA implementation of generalized class of Fourier transform, also known as Walsh transform. HIST [17] is a histogram implementation on GPU. SORTNW [17] implements bitonic sort, a special type of sorting algorithm. REDUCE [17] performs parallel reduction operations on an input array to produce a final value. PSUM is a microbenchmark based on the threadfence example explained in the CUDA programming guide [19]. OFFT [17] is an ocean simulation based on fast Fourier transform. KMEANS [24] is a CUDA implementation of the parallel k-means clustering algorithm [25]. HASH is another microbenchmark.

where every thread updates a hash table atomically. REDUCE, PSUM, and KMEANS benchmarks use memory fencing functions in CUDA for inter-thread-block communication. The thread-blocks in other benchmarks work on independent sections in the shared and global memory spaces. All benchmarks are run until completion for detailed evaluation.

VI. EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness and performance overhead of HAccRG.

A. Effectiveness of Data Race Detection

We evaluate the effectiveness of HAccRG in detecting data races for both the shared and global memory accesses in the benchmarks listed in Table II. For this purpose, we track the shared and global memory accesses at the word granularities. Overall, we report four categories of data races: i) races in the shared memory due to incorrect barrier synchronizations; ii) races in the global memory due to incorrect barrier synchronizations; iii) races in the global memory due to the lack of critical sections; and iv) races in the global memory due to missing memory fence instructions. No data race is detected in the shared memory; however, data races are found in the global memory for three benchmarks. In SCAN and KMEANS, the kernels are designed to execute as a single thread-block, but multiple thread-blocks are launched to scale up the workload. Consequently, all thread-blocks operate on the same data, causing data dependences that otherwise would not exist. These bugs are documented by the developers of the benchmark suite. No data race is reported when both SCAN and KMEANS are executed with a single thread-block. In the spectrum generation kernel of OFFT, the input is processed to create a wave spectrum in frequency domain. Multiple thread-blocks process separate data and update the final result. However, the memory address is incorrectly calculated, and two threads accessed the same memory location, causing a write-after-read data race in the global memory space.

Injected Races: To identify the effectiveness of HAccRG in benchmarks that did not show data races, we inject artificial data races in the benchmarks for shared and global memory spaces to verify how well HAccRG performs in detecting them. Data races are injected by removing barrier calls from the benchmarks (23 races), by inserting dummy memory accesses across the thread-block access boundaries (13 races), by removing memory fence calls (3 races), and by inserting dummy memory accesses inside and outside the critical sections (2 races). HAccRG is able to detect all the forty-one injected data races.

1) Impact of Varying Memory Access Tracking Granularity:

Table III shows the number of false shared memory data races detected when shared memory tracking granularity is varied from 4-byte to 64-byte. HAccRG does not detect false data races for five out of nine benchmarks because of their peculiar shared memory access patterns. They all have very regular accesses to the shared memory where all threads in a warp access successive locations. Accesses within a warp always have implicit synchronization because of SIMD execution. Therefore, HAccRG does not report a data race even when the entire warp’s accesses map to a single shadow entry. On the other hand, HAccRG reports high number of false data races for HIST since the benchmark operates on a data structure having element size of one byte, which in turn translates to accesses from multiple warps mapping to the same memory entries.

TABLE II

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Inputs</th>
<th># SMs / GPU Clusters</th>
<th>SIMD Pipeline Width / Warp Size</th>
<th># Threads / Register per SM</th>
<th>Warp Scheduling</th>
<th>Shared Memory per SM</th>
<th>L1 Data Cache per SM</th>
<th>Texture Cache per SM</th>
<th>Constant Cache per SM</th>
<th>Unified L2 Cache</th>
<th># Memory Slices</th>
<th>DRAM Request Queue Size</th>
<th>Memory Controller</th>
<th>GDDR3 Memory Timing</th>
<th>Virtual Channels / Flit Size</th>
<th>Virtual Channel Buffer Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>30 / 10</td>
<td>8 / 32</td>
<td>1024 / 16384</td>
<td>Round Robin</td>
<td>16KB</td>
<td>48KB/6 way/128B line</td>
<td>5KB/5 way/128B line</td>
<td>5KB/2 way/64B line</td>
<td>64KB/Memory Slice: 8 way/128B line</td>
<td>8</td>
<td>32</td>
<td>8</td>
<td>Out-of-Order (F-R-FCFS)</td>
<td>1.4</td>
<td>1 / 32</td>
</tr>
</tbody>
</table>
Table III also reports the number of false global memory data races detected when the global memory tracking granularity is varied. None of the benchmarks have false data race detection for 4-byte granularity since, for all the benchmarks, global memory data structure element sizes are at least 4 bytes. HAccRG does not detect any false data races for four out of nine benchmarks until 64-byte granularity due to very regular memory accesses.

To summarize, the impact of access tracking granularity is application-dependent. However, varying the granularity changes the hardware requirements as well as the memory footprint of HAccRG. To reduce hardware overhead of shared memory shadow entries, we decrease the shared memory tracking granularity. We set it to 16 bytes since 7 out of 10 benchmarks do not see any false positives at this granularity. However, since device memory available to the contemporary GPUs is quite large, we keep the global memory tracking granularity to 4 bytes.

2) Impact of Sizes of Sync, Fence, and Atomic IDs: Sync and fence IDs act as logical clocks for barrier and fence calls, respectively, while atomic IDs track the lock variables used in kernels. Sync and fence IDs are counters which are incremented during execution. They can detect false positives when counters reset after overflow, and the overflowed values match the IDs in the shadow entries. However, we believe that such occurrences are very rare in actual execution when sufficiently large counters are provided. Moreover, optimization to increment sync ID only when global memory accesses are made, effectively limits the number of increments. We observe that sync ID increments are very small (maximum 5 for REDUCE) because barrier synchronizations are primarily used only for shared memory accesses in the benchmarks. Although fence ID is incremented when a warp completes a fence call, the number of fences executed is small, maximum being 5 for HASH. Therefore, we set sync and fence ID sizes to 8 bits each which are large enough to avoid overflows.

Atomic IDs, which are Bloom filter signatures, hold the addresses of lock variables protecting critical sections. Since signatures are compact representation of addresses, two different addresses can form the same signature, which is common in signature-based systems. Therefore, HAccRG can miss the actual data races if it is not able to distinguish between different lock variables. We perform a stress test on a microbenchmark by injecting data races for over 1 million addresses to study impact on accuracy. We test 8-bit, 16-bit, and 32-bit signatures with 2 and 4 bins each. Bits in each bin are set through direct indexing by lower order bits of the addresses [28]. We observe that signatures with 2 bins have better accuracy than those with 4 bins for the same signature size. 8-bit, 16-bit, and 32-bit signatures with 2 bins miss 25%, 12.5%, and 6.25% data races, respectively. To trade-off between hardware cost and accuracy, we set the atomic ID size to 16 bits in HAccRG.

B. Performance Impact of Race Detection

Figure 7 shows the performance impact of enabling HAccRG. The bars represent execution time normalized to that of benchmarks executing on unmodified GPU where HAccRG is disabled. The geometric mean across all benchmarks is shown at the end of Figure 7.

We also compare HAccRG with its software implementation and a prior instrumentation-based mechanism, GRace\(^1\) [26]. For SCAN, HIST, and KMEANS, HAccRG incurs a 0.2%, 0.3%, and 22.1% slowdown, respectively, while its software implementation incurs a slowdown of 6.6x, 12.4x, and 18.1x for these three benchmarks. GRace is two orders of magnitude slower than our software implementation and has higher memory overhead.

Performance Impact of Allocating Shared Memory Shadow Entries in Global Memory: Figure 8 shows performance impact in HAccRG when the shared memory shadow entries are split between hardware and software. For this experiment, we enable both shared and global memory data race detection. Software shared memory shadow entries are stored in the global memory and fetched into the L1 data caches for data race detection using shared memory RDUs. In our simulations, L1 data cache is 4KB per SM which is large enough to accommodate shadow memory for 16KB of shared memory. Therefore, most of the benchmarks do not show significant performance degradation, except OFFT. Since shared memory is banked, accesses to different shared memory rows across different

\(^1\)GRace has two versions, GRace-stmt and GRace-add; the former one is more accurate but slow. We compare HAccRG with faster but less accurate GRace-add.
banks are served in parallel. However, for data race detection, this can lead to reading multiple lines from the software shadow memory. OffT suffers from this behavior as single shared memory access translates to multiple shadow memory lines read from the global memory. Thus, if hardware support in the shared memory is not possible, for most kernels, placing the shared memory shadow entries in the global memory incurs only a small performance penalty.

C. Overheads in HAccRG

1) DRAM Bandwidth Utilization: Figure 9 shows per-benchmark average bandwidth utilization reported by the simulator. Y-axis represents the average bandwidth utilization of all DRAM banks over the entire execution. The first bar of each benchmark shows the bandwidth utilization for execution without HAccRG. It is evident from Figure 9 that the DRAM bandwidth utilization is the characteristic of an application, which is dependent on the cache miss rate of L1 data caches and unified L2 cache. The benchmarks which have low L1 or L2 cache miss rate result in much lesser DRAM bandwidth utilization.

Shared memory data race detection does not create memory requests, which is seen as the unchanged bandwidth utilization for all benchmarks. On the other hand, global memory data race detection accesses shadow entries that reside either in L2 cache or DRAM. Therefore, Mcarlo, Fwalsih, Hist, SortNW, Reduce, OffT, and Hash have more bandwidth utilization as they rely on L2 cache performance due to their characteristic higher L1 miss rates. Enabling global memory data race detection degrades the L2 cache performance since the shadow entries pollute the L2 cache, thus increasing the DRAM utilization. However, Scan, Psum, and Kmeans do not depend upon the L2 cache performance because of their higher L1 hit rates. With their low L2 utilization, enabling global memory data race detection hardly affects their L2 cache performance, which in turn keeps the DRAM utilization almost constant for these benchmarks as seen in Figure 9.

To summarize, the applications that effectively exploit the L1 data caches increase the DRAM bandwidth utilization only marginally, while the applications that rely on the L2 cache show higher bandwidth utilization for the global memory data race detection. However, the overall bandwidth utilization is well within the DRAM limits.

2) Hardware Overhead: HAccRG requires additional control logic and storage to perform data race detection in GPU.

Control Logic: Fences and atomic operations are evaluated only for the global memory in this paper. Shared memory shadow entries require 12-bit (1-bit modified, 1-bit shared, and 10-bit tid) comparator. For parallel comparison across shared memory banks at 16-byte granularity, HAccRG requires 8 12-bit comparators per SM. Global memory shadow entries are configurable and their size can be 28 bits (1-bit modified, 1-bit shared, 10-bit tid, 3-bit bid, 5-bit sid, 8-bit sync ID). With fence ID (8 bits) or atomic ID (16 bits) added, the size of shadow entries can be 36 bits or 52 bits, respectively. To summarize, for a cache line size of 128 bytes at 4-byte granularity, we need 32 28-bit comparators for basic shadow entries and 16 24-bit comparators for fence and atomic IDs per memory slice.

Storage: For the shared memory data race detection, HAccRG employs 12-bit shadow entries at the granularity of 16 bytes. The recent NVIDIA Fermi GPUs can have up to 48KB of shared memory per SM [18]. HAccRG will require 4.5KB storage per SM on Fermi for the shared memory shadow entries. For the global memory data race detection, each SM maintains a per-block 8-bit sync ID, a per-warp 8-bit fence ID, and a per-thread 16-bit atomic ID. For a single Fermi SM supporting 8 concurrent blocks, 48 warps, and 1536 threads, the storage size for global memory data race detection will be 3KB per SM. The race register file replicated in each memory slice takes 0.75KB per copy.

3) Memory Overhead: HAccRG requires fixed space to store global memory shadow entries. This overhead at the granularity of 4 bytes is shown in Table IV. By changing the global memory tracking granularity, HAccRG can significantly reduce the overhead of shadow entries by trading off the data race detection accuracy.

VIII. RELATED WORK

There have been several previous studies on data race detection in GPUs [4, 6, 8, 12, 13, 14, 26, 27]. Hou et al. [8] presented a framework that instruments the GPU kernels to record all runtime memory accesses in a log file for offline processing. Boyer et al. [6] proposed to instrument the GPU kernels to log and compare shared memory accesses from different threads to detect data races. The evaluation shows that the instrumented kernel is orders of magnitude slower than the un-instrumented version. Due to the inadequate consideration for GPU thread scheduling, false positives are also reported.

GRace [26] and GMRace [27] attempted to reduce the performance degradation associate with instrumentation and meta data manipulation by utilizing software analysis mechanism. However, software analysis is not always effective in the presence of indirect memory

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Shadow Memory Overhead</th>
<th>Benchmark</th>
<th>Shadow Memory Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mcarlo</td>
<td>28KB</td>
<td>Scan</td>
<td>9KB</td>
</tr>
<tr>
<td>Fwalsih</td>
<td>4.7MB</td>
<td>Hist</td>
<td>27.9MB</td>
</tr>
<tr>
<td>SortNW</td>
<td>576KB</td>
<td>Reduce</td>
<td>6.7MB</td>
</tr>
<tr>
<td>Psum</td>
<td>208KB</td>
<td>OffT</td>
<td>1.2MB</td>
</tr>
<tr>
<td>Kmeans</td>
<td>9.5KB</td>
<td>Hash</td>
<td>4.6MB</td>
</tr>
</tbody>
</table>

Table IV: Global Memory Overhead of HAccRG.
accesses which are common in important classes of applications. The authors of GRace only demonstrated the effectiveness of their software analysis tools on three kernels. Furthermore, the software analysis approach requires extra steps in compilation, and thus it is not always a deployable option. On the other hand, with the proposed hardware support in HAccRG, we are able achieve low overhead data race detection on all kernels.

Other than dynamically monitoring kernel execution, various other approaches are proposed. Li et al. [13] proposed PUG, a Satisfiability Modulo Theories (SMT)-based technique for detecting potential data races in the shared memory accesses. GKLEE [14] proposes symbolic virtual machine for correctness checking and test generation for GPU programs. The symbolic analysis based on SMT solver in GKLEE identifies data races, but can result in false alarms. Alan et al. [12] combine dynamic and static analysis based on information flow to amplify the outcome of single input execution over a range of inputs for CUDA programs. This methodology can also report false positives because it does not consider the GPU execution model. Further, the performance and storage overheads in this approach are non-trivial. On average, an 18x slowdown is reported, while some benchmarks show much higher slowdowns.

It is worth pointing out that most existing data race detection mechanism for GPU focus on races that are related with barrier synchronizations. HAccRG, on the other hand, not only detects data races caused by barrier synchronizations, but also data races caused by the mis-usage and/or lack of critical sections and memory fences. Furthermore, HAccRG offers a variety of optimizations that trade off the overhead of data race detection for accuracy.

The data race detection support proposed in HAccRG can also be applied to CPUs. However, since most CPUs have coherence support and stricter memory consistency than GPU, some of the HAccRG techniques may not be required in CPUs. Additionally, the same hardware support can be shared with other functionalities. For example, hardware transactional memory in GPUs can utilize the data race detection support to track dependence violations among concurrent transactions.

VIII. Conclusions

Data race is one of the major concerns in multi-threaded programming. Modern GPUs support thousands of concurrent threads which makes program debugging increasingly difficult when data races occur. In this paper, we propose a hardware-accelerated mechanism (HAccRG), for efficient and accurate data race detection in GPUs. We implement HAccRG support in the shared and global memory of the GPU with moderate hardware overhead. We evaluate the effectiveness and efficiency of the proposed technique using a set of GPGPU benchmarks. HAccRG accurately detects data race bugs in GPU programs with the average runtime overhead of 1% for the shared memory and 27% for combined shared and global memory. We have demonstrated that it is feasible to devote reasonable hardware to facilitate the design and implementation of an efficient data race detection mechanism in GPU. Such a data race detection mechanism can form the basis of powerful tools for easing parallel software development and enhancing software correctness.

Acknowledgment

This work is supported in part by grants from National Science Foundation under CNS-0834599, CSR-0834599 and CPS-0931931, a contract from Semiconductor Research Cooperation under SRC-2008-TJ-1819, and gift grants from HP, IBM, and Intel. We would like to thank all anonymous reviewers for their constructive comments that helped to improve the quality of this paper. We would also like to thank Ragavendra Natarajan and Jieming Yin for their suggestions to improve the paper.

References