2018 IEEE 29th International Symposium on Software Reliability Engineering

Bugaroo: Exposing Memory Model Bugs in Many-core Systems

Mohammad Majharul Islam
Intel Corporation
mohammad.majharul.islam@intel.com
Abdullah Muzahid
University of Texas at San Antonio
abdullah.muzahid@utsa.com

***Abstract*—Modern many-core architectures such as GPUs aggressively reorder and buffer memory accesses. Updates to shared and global data are not guaranteed to be visible to concurrent threads immediately. Such updates can be made visible to other threads by using some fence instructions. Therefore, missing the required fences can introduce subtle bugs, called *Memory Model Bugs*. We propose Bugaroo to expose memory model bugs in any arbitrary GPU program. It works by statically instrumenting the code to buffer some shared and global data for as long as possible without violating the semantics of any fence or synchronization instruction. Any program failure that results from such buffering indicates the presence of subtle memory model bugs in the program. Bugaroo later provides detailed debugging information regarding the failure. Bugaroo is the *first* proposal to expose memory model bugs of GPU programs by simulating memory buffers. We present a detailed design and implementation of Bugaroo. We evaluated it using seven programs. Our approach uncovers new findings about missing and redundant fences in two of the programs. This makes Bugaroo an effective and useful tool for GPU programmers.**

I. Introduction

A. Memory Model Bugs

With the widespread adoption of parallel architectures such as many-core and multi-core machines, programmability becomes a pressing concern for today's computing world. A memory model directly affects programmability, performance, and portability of a parallel architecture. Among various memory models, Sequential Consistency (SC) [18] is the most intuitive one. It guarantees a total global order among the memory operations where each thread maintains its program order. However, most commercial architectures do not implement SC because of its prohibitively large performance overhead. For example, many-core architectures such as Graphics Processing Units (GPUs) from NVIDIA implement some form of Relaxed Memory Ordering (RMO) memory model [4]. RMO [24] allows any later memory access of a thread to bypass any earlier memory access. The aggressive buffering and reordering of memory accesses in GPUs can lead to incorrect behavior of a program unless programmers use sufficient *fence* instructions in the code. We refer to such bugs caused by missing fences as *Memory Model Bugs*.

Figure 1 shows a code snippet from the book "CUDA by Example" [21]. It shows how we can implement lock and unlock operations using `atomicCAS` and `atomicExch` respectively. Unfortunately, the implementation suffers from a memory model bug that occurs due to a missing fence (i.e., `__threadfence`) in the unlock function. Imagine that there is no `__threadfence` before the `atomicExch` (which sets the `mutex` to 0) and we are accessing some global data in a critical section protected by lock and unlock operations. In that case, accesses to the global data could potentially get reordered beyond `atomicExch` and become unprotected by the critical section. Thus, the critical section fails to provide mutual exclusion to those memory accesses. Sorensen et al. [22] detected this bug and suggested to use `__threadfence` before `atomicExch`. However, Sorensen et al. also suggested to use another `__threadfence` (after `atomicCAS` in the lock function), which we find to be unnecessary during our experimental evaluation (Section IV-D). This *new* finding is acknowledged in the NVIDIA Developer Forum [20].

```
__device__ void lock(…) {
  while( atomicCAS( mutex, 0, 1 ) != 0 );
  // __threadfence() not needed
}


__device__ void unlock(…) {
  // Missing __threadfence()
  atomicExch( mutex, 0 );
}
```

Fig. 1: A memory model bug in unlock operation.

B. Why Important?

Absence of any memory model bug is a fundamental correctness criteria. Therefore, detecting memory model bugs is crucial for any parallel program. This is particularly important in the context of GPUs because most GPUs lack adequate formalism and documentation in memory model specification. On top of that, there are inconsistencies and even incorrectnesses in manuals such as NVIDIA PTX manual (e.g., regarding `.volatile`) and CUDA manual (e.g., regarding lock & unlock operation in Figure 1) [4]. Despite the importance of memory model bugs, there is hardly any research to detect these bugs for GPUs. The most relevant one was proposed by Sorensen et al. [22] which stresses the memory system

2332-6549/18/$31.00 ©2018 IEEE
DOI 10.1109/ISSRE.2018.00028

to expose various memory model bugs. However, our experiments have shown (Section IV-D) that there is still room for significant improvement and uncover new findings.

C. Our Approach

We propose a novel approach, called *Bugaroo*, to expose memory model bugs in a GPU program. At the high level, Bugaroo works by emulating memory buffering and reordering at various degrees. Bugaroo instruments GPU code to buffer various writes to global or shared data for as long as possible (without violating any constraint of fence instructions). In other words, Bugaroo makes the writes visible to other threads whenever the first thread (i.e., the thread whose writes are buffered) executes any fence instruction. If the program fails (e.g., deadlocks, crashes, or produces incorrect results), Bugaroo detects a memory model bug. In that case, Bugaroo provides detailed debugging information related to the location of the bug. On the other hand, if the program continues to execute without any failure, we conclude that the buffering has not exposed any memory model bug.

D. Contributions

We make the following contributions:

1) Bugaroo is the *first* proposal to expose memory model bugs by emulating memory buffering in GPUs. It does not require any formal specification of memory model or modification to compiler.
2) We implemented Bugaroo in NVIDIA Tesla K80 GPU using SASSI [23]. SASSI is a low level assembly language instrumentation tool for GPU.
3) We evaluated Bugaroo using seven GPU applications from various reference manuals as well as Rodinia benchmark suite [11]. Bugaroo detected existing as well as injected memory model bugs in those applications. Compared to Sorensen et al. [22], we uncovered two *new* findings in two applications.

E. Organization

The rest of the paper is organized as follows: Section II provides some background related to GPUs and their programming model; Section III explains the main idea of Bugaroo; Section IV provides the experimental results; Section V points out some limitations; Section VI discusses related work, and finally, Section VII concludes.

II. Background

Here, we provide necessary background on memory models, CUDA programming model, and a brief overview of the instrumentation framework, SASSI [23].

A. Memory Models

A memory model of a multiprocessor system is an architectural specification of how memory operations of a program will execute. In other words, the memory model specifies the values that memory read operations of a program executed on the multiprocessor system will return [3]. The strongest memory model, SC, only allows executions that correspond to an interleaving of different thread's instructions [18]. However, due to SC's extremely high performance overhead, GPUs from NVIDIA implement some form of RMO memory model [4] and allows aggressive buffering and reordering of memory accesses. The buffering and reordering can lead to incorrect behavior of a program. Such bugs are called Memory Model Bugs. The bugs can be prevented by placing fence instructions between some memory access instructions [4].

B. The CUDA Programming Model

In CUDA programming [1], a program consists of host code that executes on the CPU and device code that executes on the GPU. The device code is called a kernel, and is executed by many threads. Threads are grouped into 32-element vectors, called warps, to improve efficiency. The threads in each warp execute in SIMT (single instruction, multiple thread) fashion, all fetching from a single Program Counter (PC) in the absence of control flow divergence. Warps are grouped into disjoint sets called blocks; the number of threads (and by extension, warps) in a block is a parameter for the kernel. Collectively, the blocks that execute a kernel form a grid; the grid size is also a parameter for the kernel. Threads in the same block can communicate using shared memory. A single global memory region is accessible to all threads in the grid.

1) Fence Instructions in CUDA: The CUDA programming model assumes a device with a weakly-ordered memory model. In other words, the order in which a CUDA thread writes data to shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the order in which the data is observed being written by another CUDA or host thread. Ordering can be enforced by calling memory fence operations such as: __threadfence_block(), __threadfence(), and __threadfence_system() [1].

Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the __syncthreads() function; __syncthreads() waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to __syncthreads() are visible to other threads in the block.

Memory fence operations only affect the ordering of memory operations by a thread. They do not ensure that these memory operations are visible to other threads. This is where __syncthreads() comes into play. It ensures that memory operations are visible to other threads within a block [1].

C. SASSI

SASSI is a compiler-based instrumentation framework that runs as the final pass in NVIDIA's production backend compiler and assembler, `ptxas` [23]. Because SASSI is invoked after the original, uninstrumented SASS (NVIDIA's ISA) has already been finalized, the injected instrumentation does not disrupt the perceived final instruction schedule or register usage.

SASSI must be instructed where to insert instrumentation as well as what code to insert. For each of the instrumentation sites, SASSI will insert a CUDA ABI compliant function call to a user-defined instrumentation handler function, passing site-specific information as arguments to the handler. Therefore, users must instruct SASSI what information to pass to the instrumentation handler(s). In this paper, we use SASSI to inject instrumentation code before all SASS instructions and after the SASS instructions that modify memory locations. We extract and pass only memory information (e.g., addresses read and written) to the instrumentation handler of each site.

Unlike CPU instrumentation, GPU instrumentation must coordinate with the host CPU to both initialize instrumentation counters and to gather their values. We use the CUPTI library to initialize counters before kernels launch and to copy information off the device after kernels exit.

III. Detecting Memory Model Bugs

Bugaroo relies on SASSI [23] to instrument the original code statically. It instruments all memory and fence instructions. At the high level, when a thread tries to write to a shared or global data, it simulates a write buffer and stores the new value into that buffer. Bugaroo uses separate buffers for global and shared data. Bugaroo keeps buffering writes until it encounters a fence instruction. Bugaroo, then, flushes all values buffered so far (according to the semantic of the fence instruction). Eventually, if the program crashes, deadlocks, or produces incorrect results, Bugaroo reports a memory model bug. To pinpoint the bug, Bugaroo first determines whether the bug occurs due to the buffering of global or shared data. Then, Bugaroo does a binary search to determine the exact location of the bug.

A. Code Instrumentation

We need to insert instrumentation code both before and after some instructions. Fortunately, SASSI provides both options. More specifically, `sassi_before_handler` and `sassi_after_handler` functions can be used for this purpose respectively. Bugaroo uses `sassi_before_handler` to instrument any type of instructions (i.e., by specifying `all` flag during compilation) before it is executed and extract additional information related to memory accesses (i.e., by specifying `mem-info` flag). In addition, we use `sassi_after_handler` to instrument any memory related instruction (i.e., by specifying `memory` flag) after the instruction is executed. Inside the handlers, Bugaroo checks an instruction's opcode to determine its type. For NVIDIA GPUs (our experimental system), the type could be an access to shared or global memory, `__threadfence`, or `__syncthreads`.

B. Simulating Write Buffers

NVIDIA GPUs implement some form of RMO memory model [4]. RMO [24] is a weaker memory model that allows a later store to bypass an earlier store or load to a different location. Similarly, it also allows a later load to bypass an earlier load or store to a different location. If we consider a single thread with some number of memory access instructions, there can be exponential number of possible reorderings in RMO. Instead of enforcing each of them one by one, we randomly select some writes with probability p_w and buffer their updates. By changing the value of p_w, we can simulate different degrees of buffering. Bugaroo buffers the selected writes for as long as possible. The intuition is that by buffering those writes the longest, Bugaroo is likely to expose their worst case reordering scenarios.

In order to facilitate the buffering, we implement a per-thread write buffer. Bugaroo has separate buffers for shared and global data. Figure 2 provides the high level idea. When a thread, say T_1 issues a store operation to a shared or global variable, Bugaroo checks if it can select the store randomly with probability p_w. If Bugaroo selects the store, it buffers the new value in the corresponding write buffer. In other words, the variable still holds the old value. To implement this, Bugaroo records the old value of the variable immediately before it is written (i.e., inside `sassi_before_handler`). Immediately after the variable is written (i.e., inside `sassi_after_handler`), Bugaroo places the variable's new value into the thread's write buffer and restores the variable's old value back. The handler executes `__threadfence` to ensure that the old value is propagated to every thread in the device. During a brief period when the variable has its new value (i.e., right before the old value is restored back), it is possible that some other thread could observe the variable's new value. In order to prevent that, we can use a single global lock such that Bugaroo acquires the lock inside `sassi_before_handler`, and releases the lock inside `sassi_after_handler` right after restoring the old value back. This essentially serializes all handlers. However, we have not noticed any difference in results when Bugaroo uses the global lock versus when Bugaroo does not use the lock. Therefore, we have not used any global lock during our implementation of Bugaroo.

Bugaroo flushes the buffered values according to fence instructions i.e., `__syncthreads` and `__threadfence`. When the thread T_1 is about to execute any of those instructions (i.e., inside `sassi_before_handler`), Bugaroo writes the buffered values to the respective variables. Then, the thread executes the subsequent `__syncthreads` or `__threadfence` instruction and the updated values get propagated to other threads accordingly. In other words, `__threadfence` flushes the new values to shared and global memory to make sure ordering of writes are maintained over all the threads of the device whereas `__syncthreads` flushes the new values to shared and global memory so that they are visible to other threads of the same block (not throughout the device like `__threadfence`).

C. Complete Algorithm

Algorithm 1 shows the SASSI handler functions of Bugaroo. At the high level, `sassi_before_handler` keeps recording old value of some randomly selected stores to global or

Algorithm 1 Main handler functions

```
1:  data structures:
2:      $store\_count[GLOBAL]$ is a per-thread counter for global stores
3:      $store\_buffer[GLOBAL]$ is a per-thread buffer for global stores
4:      $store\_count[SHARED]$ is a per-thread counter for shared stores
5:      $store\_buffer[SHARED]$ is a per-thread buffer for shared stores
6:
7:  procedure SASSI_BEFORE_HANDLER(...)
8:    asssumptions:
9:      $i$ is the instrumented instruction
10:     $tid$ is the thread id
11:     $random$ is a flag set to TRUE with $p_w$ probability
12:   begin:
13:     if $store\_count[GLOABAL][tid] \geq MAX\_STORE$ then
14:         $flush\_all\_stores(GLOBAL, tid)$
15:         execute __threadfence
16:     if _$store\_count[SHARED][tid] \geq MAX\_STORE$ then
17:         $flush\_all\_stores(SHARED, tid)$
18:         execute __threadfence
19:     if $i$ is a load from global or shared data then
20:         TYPE = type of data (i.e., GLOBAL or SHARED)
21:         for each store $st$ in $store\_buffer[TYPE][tid]$ do
22:             if  memory address of $i$ = $st.address$ then
23:                 load the value form $store\_buffer[TYPE][tid]$
24:                 terminate the loop
25:     else if $i$ is a store to global or shared data then
26:         TYPE = type of data
27:         for each store $st$ in $store\_buffer[TYPE][tid]$ do
28:             if memory address of $i$ = $st.address$ then
29:                 write $st.value$ form $store\_buffer[TYPE][tid]$ into $st.address$
30:                 execute __threadfence
31:                 $store\_count[TYPE][tid] \leftarrow store\_count[TYPE][tid] - 1$
32:                 terminate the loop
33:         set $random$ to TRUE with probability $p_w$
34:         if $random = TRUE$ then
35:             record address and old value of $i$
36:     else if $i$ is __threadfence then
37:         $flush\_all\_stores(GLOBAL, tid)$
38:         $flush\_all\_stores(SHARED, tid)$
39:     else if $i$ is __syncthreads then
40:         $flush\_all\_stores(SHARED, tid)$
41:     else if $i$ is EXIT from kernel then
42:         $flush\_all\_stores(GLOBAL, tid)$
43:         $flush\_all\_stores(SHARED, tid)$
44:
45: procedure SASSI_AFTER_HANDLER(...)
46:   asssumptions:
47:     $i$ is the instrumented instruction
48:     $tid$ is the thread id
49:   begin:
50:     if $i$ is a store to global or shared data and $random = TRUE$ then
51:         TYPE = type of data
52:         $restore\_old\_val(TYPE, i, tid)$
```

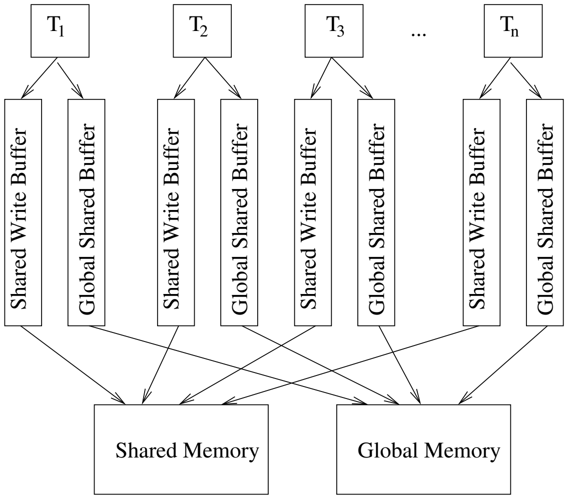


Fig. 2: Illustration of per-thread write buffer. T_1, T_2, ..., T_n are threads. Shared memory is shared within threads from the same block. Global memory is shared across all threads.

shared data (Line 35) and `sassi_after_handler` keeps retrieving the new value of each such store and restores the old value back (Line 50-52). If the thread has a read after write dependence to one of the buffered stores (Line 19-24), it reads the latest value from the store buffer. In other words, a thread will never read old or pre-write value after it writes. Additionally, if the thread has a write after write dependence to one of the buffered stores (Line 25-32), it flushes the older store. All other buffered stores still remain in the corresponding store buffer. If there is a `__threadfence` (Line 36) or `__syncthreads` (Line 39) or kernel exit (Line 41) or a maximum number of stores already buffered (Line 13, 16), the thread flushes stores from the proper buffer. If a thread needs to flush stores due to store buffer size limit or write after write dependence, it executes `__threadfence` (Line 12 and 22 respectively) to ensure that the old values are propagated to other threads.

Algorithm 2 shows the helper functions used in Algorithm 1. $flush\_all\_stores$ flushes all the buffered stores of a particular thread and a particular type and updates the per-thread $store\_count$. Flushing is done by writing the buffered value of a store to its address. $restore\_old\_value$ records the current value of a store and puts back the old value into the same address (Line 7-8). The thread executes `__threadfence` to ensure that the old value is propagated to all other threads (Line 9). The current value is stored in the proper $store\_buffer$.

D. Root Cause Analysis

If Bugaroo causes a program to crash, deadlock or produce incorrect results, Bugaroo reports a memory model bug. When we observe crash, deadlock or incorrect results, we perform a root cause analysis to determine the location of the bug. We do the analysis in the following steps:

Algorithm 2 Helper functions

```
procedure RESTORE_OLD_VALUE($TYPE$, $i$, $tid$)
input parameters:
    store instruction $i$
    thread id $tid$
    type $TYPE$ of $i$
begin:
    read current value of $i.address$ and insert it into $store\_buffer[TYPE][tid]$
    write old recorded value to $i.address$
    execute __threadfence
    $store\_count[TYPE][tid] \leftarrow store\_count[TYPE][tid]+1$

procedure FLUSH_ALL_STORES($TYPE$, $tid$)
input parameters:
    thread id $tid$
    type $TYPE$ of stores
begin:
    for each store $st$ in $store\_buffer[TYPE][tid]$ do
        write $st.value$ into $st.address$
    $store\_count[TYPE][tid] \leftarrow 0$
```

1) *Which data?* We execute the program with Bugaroo one more time instrumenting only shared data. We ensure that the same seed is used for randomization so that $random$ goes through the same sequence of values. If the program still fails, the bug is related to shared data. Otherwise, the bug is related to global data.
2) *Which kernel?* If the program has multiple kernels, we instrument one kernel at a time. We can do this easily using SASSI by extracting the name of the kernels. If the program fails only when a specific kernel is instrumented, we can conclude that the bug lies within that kernel.
3) *Which location?* Once we determine the specific kernel and data type, we need to find out the exact location where `__threadfence` should be inserted. In order to do so, we put one `__threadfence` after each access of the specific type in the specific kernel. If we execute the program with Bugaroo, now it will not fail, i.e., the bug is resolved due to `__threadfence`s. To find out which `__threadfence` is necessary to resolve the bug, we apply binary search technique. We omit half of the `__threadfence`s. If the program does not fail, we can conclude that one of the remaining `__threadfence`s is necessary. On the other hand, if the program fails, we can assert that the omitted half contains the necessary `__threadfence`. We keep omitting half of the `__threadfence`s this way, until we finally find out the exact `__threadfence` that resolves the memory model bug. We report this `__threadfence` as part of the debugging information. If the kernel is missing multiple fences, we can find the location of only one fence in this way. We can repeat the whole process except this time we keep the fence that we just identify as required during the binary search process. If program fails during the search process, we will be able indentify the second fence. This way, we can find all missing fences.

IV. Experimental Results

The goal of this section is to (i) characterize the applications used, (ii) evaluate Bugaroo's ability to detect existing as well as injected memory model bugs in those applications, and (iii) show instrumentation overhead.

A. Experimental Setup

We used SASSI [23], a low level assembly language instrumentation tool for GPU to implement Bugaroo. All experiments were performed on an NVIDIA Tesla K80 GPU. It features 4992 NVIDIA CUDA cores with a dual-GPU design, 24 GB of GDDR5 memory, and display driver version 340.21. This GPU was connected to a machine with four Intel 2.30GHz Xeon E5-2686 v4 CPUs and 61 GB main memory. All experiments used CUDA 7 toolkit. Finally, we used 256 as the size of each write buffer.

B. Characterization of Applications

To evaluate the effectiveness of Bugaroo, we need applications that use fine-grained concurrency. We used two sets of GPU applications to evaluate Bugaroo. The first set has four GPU applications that are known to use fine-grained concurrency. These applications are taken from Sorensen et al. [22]. The second set has three GPU applications from Rodinia benchmark suite [11]. In order to determine whether an application fails or not, we add a function with each application. The function compares the application's results produced by GPU with those produced by CPU. This comparison is done at the end before the application exits. We adopt this approach because applications may exhibit nondeterminism. Therefore, it may not be sufficient to check repeated computations with identical results.

The evaluated applications are summarized in Table I. The table shows details about the application source code, the nature of communication, and the condition to check for failure. All applications in the first set are shown in the first four rows of the table. Except for ct-octree, the other three applications in the first set have the failure checking function. For ct-octree, we collected meta-data during the executions and used the data to check for failures. For applications in the second set (shown in the bottom three rows in the table), we obtained reference solutions from non-instrumented version of the applications. We checked for failures by comparing the computed values with the reference values.

We find that all application executions terminate within 4 seconds (natively), dominated by initialization of the CUDA framework. Kernel execution itself accounts for a small fraction of total time. To catch errors such as deadlocks and hangs, we set a timeout limit of 60 seconds per application execution. We ran each application 1000 times to collect the relevant data.

C. Instrumentation Details

1) Application Metadata: Table II summarizes applications' metadata collected by Bugaroo. For each application, we show the number of dynamic instructions, number of shared stores, and number of global stores executed (in instrumented

Program	Description	Communication	Failure Checking Condition
cbe_dot	Dot product routine given in the book CUDA by Example [21]	Global final reduction across blocks protected by a custom mutex	GPU result matches a CPU reference result
ct_octree	Octree partitioning routine by Cederman and Tsigas	Concurrent access to non-blocking queues	All original particles are in final octree
sdk_red	Reduction routine from the CUDA 7 SDK	Last block (via atomic counter) combines block-local results	GPU result matches a CPU reference result
cub_scan	Prefix scan from the CUB GPU library	Blocks communicate partial results using MP-style handshake	GPU result matches a CPU reference result
kmeans	Clustering algorithm used extensively in data-mining	Only block level synchronization with syncthreads	Relative distance between clusters matches results from reference implementation
streamcluster	Modified upon the streamcluster benchmark in the Parsec	No synchronization	Relative distance between clusters matches results from reference implementation
hotspot	Estimates processor temperature	Only block level synchronization with syncthreads	Instrumented result match results from reference implementation

TABLE I: Applications analyzed.

kernels) in Column 2, 4, and 5 respectively. Column 3 shows the total number of dynamic fence instructions executed in instrumented kernels along with the static count (i.e., unique fences) inside parenthesis. We have experimented with two versions of the cbe_dot. Original (i.e., cbe_dot_2_fence) version has fence inside lock and before unlock. Other version (cbe_dot_1_fence) has only one fence before the unlock. Except for kmeans and streamcluster, all the applications have both shared and global stores. Kmeans and streamcluster have only global stores. The values presented in Table II for streamcluster are for simsmall input.

Codes	# of dyn. Inst	# of dyn. _threadfence	# of dyn. shared st	# of dyn. global st
cbe_dot_2_fence	1076216	64(2)	16352	32
cbe_dot_1_fence	1075828	32(1)	16352	32
ct_octree	31362438	41(1)	1372676	15623
sdk_red	999846	8192(1)	49600	130
cub_scan	2769764	10529(3)	83185	62435
kmeans	306772	0(0)	0	3600
streamcluster	2294132736	0(0)	0	5519623
hotspot	1841064	0(0)	29160	4096

TABLE II: Application Metadata

2) Characterizing Write Buffers: When a thread is about to flush stores of global or shared write buffer, we collect the number of stores in the corresponding buffer. For each application we present the collected data as a histogram of flush lengths over all threads.

Figure 3 shows the characterization of global stores for all seven applications, whereas, figure 4 shows the characterization of shared stores for five applications as streamcluster and kmeans do not have any store to shared memory. Since both cbe_dot_2_fence and cbe_dot_1_fence have same number of stores buffered and produce same histograms, we show a single histogram in both Figure 3(a) and Figure 4(a).

As it is demonstrated in the histograms of global and shared stores, most of the stores have a flush length of only 1. Overall, 69.71% of global stores (Figure 3) and 72.24% of shared stores (Figure 4) have a flush length of 1. Figure 3(e) demonstrates the flush length histogram for global stores in kmeans which has the highest flush length of 34. In a nutshell, a small write buffer (say, with 32 entries) will be enough for all the applications except for kmeans. Some of the stores in kmeans will find the store buffer to be full which will, then, force full flush of the buffered stores.

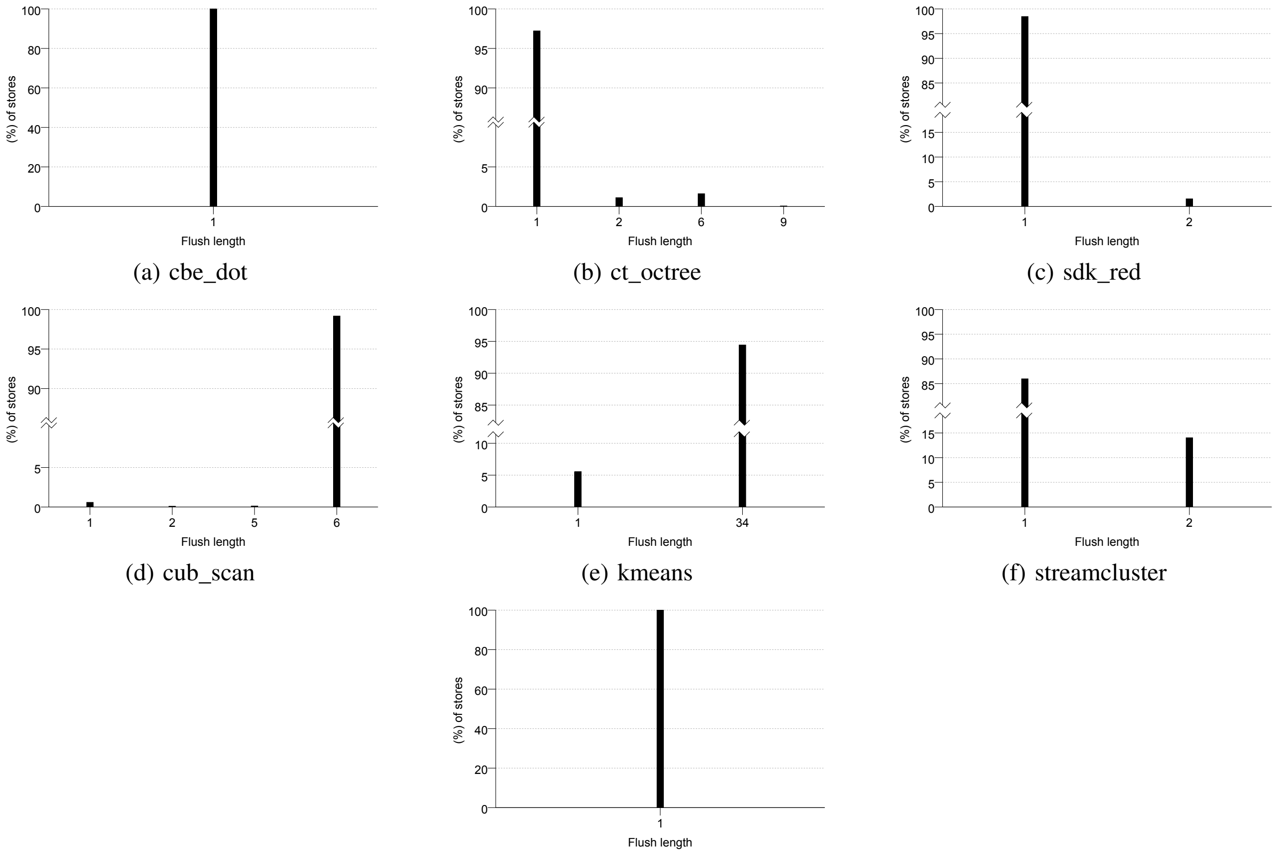
3) Runtime Overhead: Table III shows the average execution time (seconds) of native and instrumented versions of applications in column 2 and column 3 respectively. Column 4 shows the overhead (%) of Bugaroo. The median runtime overhead is 8.91%. The highest is 117.91% for streamcluster as Bugaroo has to instrumnet all 5519623 global stores (table II). On average, Bugaroo requires 15.39% less time to instrument cbe_dot_1_fence (2.546s) than cbe_dot_2_fence (3.009). We see the same trend in native runs as cbe_dot_1_fence (2.447s) is 11.34% faster than cbe_dot_2_fence (2.760s). This is because cbe_dot_1_fence executes __threadfence() 32 times less than cbe_dot_2_fence (since it does not have any fence inside lock). Possible performance gain by skipping __threadfence() inside lock will be much more for large codebases since the number of dynamic __threadfence() count will be much larger.

Codes	Run time (s)		Overhead (%)
	Native	Instrumented	
cbe_dot_2_fence	2.760	3.009	9.02%
cbe_dot_1_fence	2.447	2.546	4.05%
ct_octree	2.792	3.117	11.64%
sdk_red	2.335	2.658	13.83%
cub_scan	2.761	3.024	9.53%
kmeans	2.765	2.907	5.14%
streamcluster	3.623	7.895	117.91%
hotspot	2.654	3.004	13.91%
MEAN	2.767	3.520	27.21%
MEDIAN	2.761	3.007	**8.91%**

TABLE III: Runtime Overhead

D. Findings and Comparison

To measure Bugaroo's bug detection ability, we ran each original application 1000 times with four different versions of Bugaroo with values of p_w = 0.25, 0.5, 0.75 and 1. First set of benchmark applications in Table I contain fence instructions which we removed to create no_fence variants. For cub_scan that has 3 different fence instructions, we created four variants

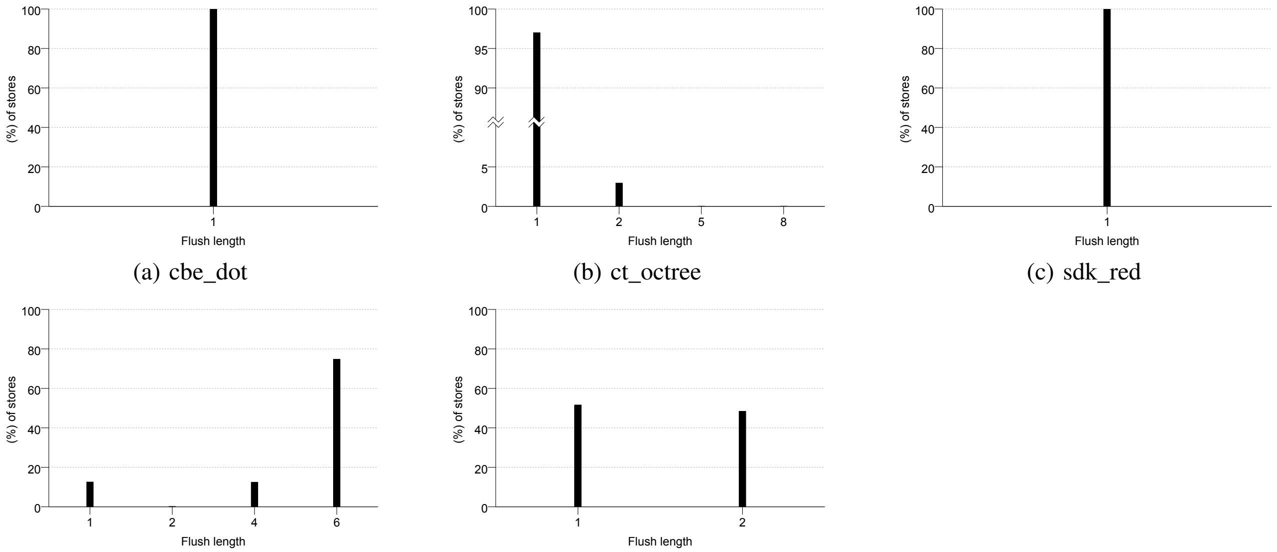


(a) cbe_dot (b) ct_octree (c) sdk_red

(d) cub_scan (e) kmeans (f) streamcluster

(g) hotspot

Fig. 3: Histogram of flush length for write buffers (global stores).



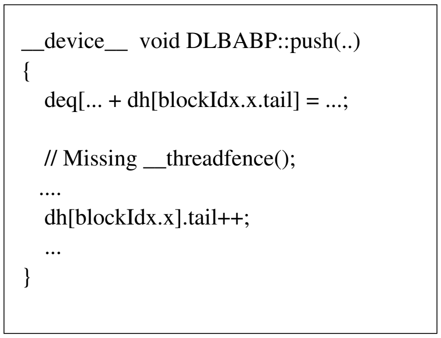
(a) cbe_dot (b) ct_octree (c) sdk_red

(d) cub_scan (e) hotspot

Fig. 4: Histogram of flush length for write buffers (shared stores).

by removing all three fences or one at a time. Like Sorensen et al. [22], this allows us to test if the provided fences are (a) experimentally needed to prevent memory model bugs and (b) sufficient to prevent memory model bugs. Findings from these experiments are demonstrated application wise. We compare each finding with that of Sorensen et al. [22] to determine whether the finding is a new one or not. As kmeans, streamcluster, and hotspot applications do not have any fence instruction and the original versions run without any failure, we discard them from further discussion.

1) *cbe_dot*
   After Alglave et al. [4] reported missing fences, NVIDIA updated cbe_dot by placing `__threadfence` inside lock and unlock. We refer this variant of cbe_dot as cbe_dot_original_2_fence. Then we removed the `__threadfence` inside lock (Figure 1) and created another variant - cbe_dot_1_fence. Finally, we removed both of the `__threadfence` and created a variant - cbe_dot_no_fence. After 1000 runs of cbe_dot_original_2_fence and cbe_dot_1_fence, we did not observe any failure with any of the four versions of Bugaroo. This proves our argument presented in section I-A empirically that `__threadfence` inside lock is unnecessary. We refer cbe_dot_original_2_fence as overfenced. However, failure probability of cbe_dot_no_fence variant is 1 for each of four versions of Bugaroo (although, % error increases with higher value of p_w) and is denoted as underfenced. So, only cbe_dot_1_fence shows the correct behavior with the minimum number of `__threadfence` and we refer to it as the optimal variant.
2) *ct_octree*
   All four versions of Bugaroo detects memory model bug with failure probability 1 for original ct_octree application. Original version has no fence and we denote it as ct_octree_original_no_fence. After performing root cause analysis as described in Section III-D, we were able to detect missing `__threadfence` inside *push* method that enqueues a task to the shared array. As shown in Figure 5, the `__threadfence` should be placed in the *push* method between accesses to shared array *dh* and *deq*. The bug will appear whenever these two accesses are reordered (or buffered). This bug is also detected by Sorensen et al. [22]. We created a variant denoted as ct_octree_1_fence placing this fence. This variant shows no failure.
3) *sdk_red*
   In sdk_red application, the reduction kernel reduces an arbitrary sized array in a single kernel invocation. It does so by keeping track of how many blocks have finished. After each thread block completes the reduction of its own block of data, it *takes a ticket* by atomically incrementing a global counter. If the ticket value is equal to the number of thread blocks, then the block holding the ticket knows that it is the last block to finish. This last block is responsible for summing the results of all the other blocks. In order for this to work, a `__threadfence` is required to make sure that before a block takes a ticket, all of its memory transactions have completed. In other words, `__threadfence` ensure that the results of all outstanding memory transactions within the calling thread are visible to all other threads. This is why, when we created sdk_red_no_fence variant by removing the `__threadfence`, it fails with empirical probability of 1 for each of the four versions of Bugaroo (and like cbe_dot_no_fence, % error increases with higher value of p_w).



```
__device__  void DLBABP::push(..)
{
  deq[... + dh[blockIdx.x.tail] = ...;

  // Missing __threadfence();
 ....
  dh[blockIdx.x].tail++;
  ...
}
```

Code from lbabp.h

Fig. 5: Fence instruction details of ct_octree

4) *cub_scan*
   Application cub_scan has 3 different fence instructions. As shown in Figure 6a, first `__threadfence` (f1_fence) is placed in *Sync* method which implements a software global barrier among thread blocks within a CUDA grid. Second `__threadfence` (f2_fence) is placed in *SetInclusive* method between the updates of *tile inclusive value* and *tile status* (Figure 6b). Third and final `__threadfence` (f3_fence) is placed in *SetPartial* method between the updates of *tile partial value* and *tile status* (Figure 6b).
   We created 5 different variants from all fence active variant, cub_scan_original_all_3_fence to cub_scan_no_fence where all 3 fences are removed. Column 2 of Table IV shows all different variants of four applications. For example, f1_f2_fence for cub_scan means that fence f1 and f2 are active and f3 is removed. After running Bugaroo for all 5 variants of cub_scan we conclude that all three fences are necessary as removing any of them will cause failure. This is a *new* finding as Sorensen et al. [22] suggested having only 2 fence is enough to produce a correct result. As presented in column 3 to 6 of Table IV, probability of failure varies for different variants. For example, for cub_scan_f2_f3_fence where we skip the `__threadfence` (f1_fence) inside function Sync, failure probability is very low (0.001) compared to the failure probability of 0.857 for cub_scan_f1_f3_fence. This is because there is a `__syncthread` immediately after the (skipped) f1_fence in orginal code (Figure 6 (a)) which will execute and synchronize stores

Codes	Version	Failure Probability (P)				Comment	New Findings
		Buffer 100% st	Buffer 75% st	Buffer 50% st	Buffer 25% st		
cbe_dot	original_2_fence	0	0	0	0	Overfenced	Yes
	1_fence	0	0	0	0	Optimal	
	no_fence	1	1	1	1	Underfenced	
ct_octree	original_no_fence	1	1	1	1	Underfenced	No
	1_fence	0	0	0	0	Optimal	
sdk_red	original_1_fence	0	0	0	0	Optimal	No
	no_fence	1	1	1	1	Underfenced	
cub_scan	original_all_3_fence	0	0	0	0	Optimal	Yes
	f1_f2_fence	0.002	0	0	0	Underfenced	
	f1_f3_fence	0.857	0.714	0.29	0	Underfenced	
	f2_f3_fence	0.001	0	0	0	Underfenced	
	no_fence	0.861	0.51	0.285	0	Underfenced	

TABLE IV: Summary of findings

within blocks. This makes the memory model bug in cub_scan_f2_f3_fence very rare. Overall, we refer cub_scan_original_all_3_fence as the optimal version and the rest as underfenced.

```
__device__ __forceinline__ void Sync(...) const
{
   ...
   __threadfence(); // fence f1

  ....
}
```

(a) Code from grid_barrier.cuh

```
__device__ __forceinline__ void SetInclusive() const
{
   // Update tile inclusive value
   ThreadStore<..>(... tile_inclusive);

   __threadfence(); // fence f2

   // Update tile status
   ThreadStore<..>(d_tile_status ...);

}

__device__ forceinline__ void SetPartial(...)

{
   //Update tile partial value
   ThreadStore<..>(... tile_partial);

   __threadfence(); // fence f3

   //Update tile status

   ThreadStore<..>(d_tile_status ...);
}
```

(b) Code from single_pass_scan_operators.cuh

Fig. 6: Fence instruction details of cub_scan

Table IV summarizes all findings for above described applications and their different variants for corresponding values of p_w. We compare the findings against Sorensen et al. [22] in the last column.

E. Memory Overhead

Memory overhead of instrumented code comes from per thread write buffers (global and shared) and the corresponding counters. For a GPU with 65535 threads, this causes an overhead of 0.45 GB for each of shared and global write buffer. Here, we assume that each buffer has 256 entries. Thus, the total memory overhead is slightly less than 1 GB.

V. Limitations

We identified two limitations of Bugaroo. *First*, Bugaroo keeps buffering some randomly selected stores for as long as possible. Although this approach is likely to expose many scenarios of reordering, not all reordering scenarios will be exposed. Thus, Bugaroo might miss some memory model bugs. However, if we run more experiments with different probabilities, we may be able reduce such misses. *Second*, in our current implementation, root cause analysis is done by executing the application with Bugaroo few more times with a slightly changed (due to less instrumentation and more fences) scenario. Since this might create a different interleaving, we may need to verify the root cause by doing the same analysis few more times.

VI. Related Work

Despite the importance of memory model bugs, there are very few proposals to detect these bugs in GPUs. The most relevant one was proposed by Sorensen et al. [22] which stresses the memory to expose memory model bugs. Compared to Sorensen et al. [22], Bugaroo has a better bug deetection ability and it uncovered two new findings in two applications. In addition to that, Bugaroo performs a better root cause analysis to determine the location of the bug. Litmus test is one of the most popular techniques to detect weak behaviors on CPUs. *TSOTOOL* [15] ran tests on systems with the TSO memory model (e.g., x86 CPUs). Litmus test has recently been applied to GPUs with the tool *GPU LITMUS* [4]. Unlike these tools, Bugaroo is built on SASSI [23], a low level assembly language instrumentation tool. Another tool *SASSIFI* [16] is also built on SASSI [23]. *SASSIFI* is an error injection tool. It studies the soft error resilience of massively parallel applications running on NVIDIA GPUs. Alglave et al. [5] survey static methods for inserting fences to restore sequential consistency in CPU applications, evaluating each method based on the number of fences inserted and the associated runtime overhead. They propose a new method based on linear programming.

Current GPU program analysis tools focus on data-race freedom, barrier properties and memory safety. The CUDA-MEMCHECK [2] tool, provided with the CUDA SDK, dynamically checks for illegal memory accesses and data-races, but does not account for weak memory effects.

Several methods exist to analyze and detect memory model bugs in CPUs. The *JUMBLE* [13] tool creates an execution environment which intentionally provides stale values (simulating weak behaviors) attempting to crash applications. It classify race conditions as destructive or benign on systems with relaxed memory models. *Orion* [17] is an active testing technique that can detect, expose, and classify any arbitrary Sequential Consistency (SC) violations in any program. *Orion* works by first, finding potential SC violation cycles by focusing on racing accesses. Then it exposes each SC violation cycle by enforcing the exact scheduling order. Burnim et al. [10] proposed another active testing technique, called *Relaxer*. It first finds potential SC violations and then, exposes them by buffering some stores while speeding up or stalling certain other thread. The *CDSCHECKER* tool [19] buffers loads and stores and is configured to simulate the C++11 memory model. All these tools detects memory model bugs in CPUs whereas, Bugaroo exposes memory model bugs in GPUs.

There are some software based techniques that either search exhaustively or use constraint solver to detect weaker memory model bugs in CPUs [7], [25], [14]. There are some dynamic approaches based on data race detection [12], [10]. Exhaustive and constraint solver based approaches work well for small programs and kernels. However, they cannot handle large applications with many shared memory accesses as well as GPU applications. There are some runtime monitoring algorithms such as Sober [8] and [9] to detect memory model related bugs. They rely on a model checker in order to find all violations of sequential consistency. They check SC executions to find SC violations in close enough relaxed executions. However, they are only applicable to CPU applications. Several proposals [8], [6] including Relaxer have used operational definitions for TSO, PSO and other relaxed memory models. However, Bugaroo does not need any such definitions to expose memory model bugs.

VII. Conclusion

Memory model bugs are crucial in the context of GPU applications. We proposed Bugaroo to expose memory model bugs in any arbitrary GPU program. It works by statically instrumenting the code to buffer some shared and global data for as long as possible. Any program failure after such buffering indicates the presence of subtle memory model bugs in the program. Bugaroo, then, provides detailed debugging information regarding the failure. Bugaroo is the *first* proposal to expose memory model bugs of GPU programs by simulating memory buffers. We evaluated it using 7 programs and uncovered 2 new findings in 2 applications. Bugaroo opens up new ways to uncover memoy model bugs in many-core applications.

VIII. Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We also thank the members of PALab for the discussions and comments. This work is supported by NSF under Grant No. 1652655.

References

[1] "CUDA C programming guide, version 7," .http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[2] "Nvidia. CUDA-memcheck," https://developer.nvidia.com/CUDA-MEMCHECK, 2015.

[3] S. V. Adve and H.-J. Boehm, "Memory models: A case for rethinking parallel languages and hardware," *COMMUNICATIONS OF THE ACM*, vol. 53, no. 8, pp. 90–101, 2010.

[4] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "Gpu concurrency: Weak behaviours and programming assumptions," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15, 2015, pp. 577–591.

[5] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, "Don&rsquo;t sit on the fence: A static analysis approach to automatic fence insertion," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 2, pp. 6:1–6:38, May 2017. [Online]. Available: http://doi.acm.org/10.1145/2994593

[6] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi, "On the verification problem for weak memory models," *SIGPLAN Not.*, vol. 45, no. 1, Jan. 2010.

[7] S. Burckhardt, R. Alur, and M. M. K. Martin, "Checkfence: checking consistency of concurrent data types on relaxed memory models," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 12–21. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250737

[8] S. Burckhardt and M. Musuvathi, "Effective program verification for relaxed memory models," in *CAV*, Jul 2008.

[9] J. Burnim, K. Sen, and C. Stergiou, "Sound and complete monitoring of sequential consistency for relaxed memory models," in *Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems: part of the joint European conferences on theory and practice of software*, ser. TACAS'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 11–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987389.1987393

[10] ——, "Testing concurrent programs on relaxed memory models," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011.

[11] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, ser. IISWC '10, 2010, pp. 1–11.

[12] Y. Duan, X. Feng, L. Wang, C. Zhang, and P.-C. Yew, "Detecting and eliminating potential violations of sequential consistency for concurrent c/c++ programs," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 25–34. [Online]. Available: http://dx.doi.org/10.1109/CGO.2009.29

[13] C. Flanagan and S. N. Freund, "Adversarial memory for detecting destructive races," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 244–254. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806625

[14] G. Gopalakrishnan, Y. Yang, and H. Sivaraj, “Qb or not qb: An efficient execution verification tool for memory orderings,” in *In Computer-Aided Verification (CAV)*, 2004.

[15] S. Hangal, D. Vahia, C. Manovit, J. Y. J. Lu, and S. Narayanan, “Tsotool: a program for verifying memory systems using the memory consistency model,” in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, June 2004, pp. 114–123.

[16] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, “Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation,” in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 249–258.

[17] M. M. Islam and A. Muzahid, “Detecting, exposing, and classifying sequential consistency violations,” in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Oct 2016, pp. 241–252.

[18] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Transactions on Computer*, vol. 28, no. 9, pp. 690–691, Sep. 1979. [Online]. Available: http://dx.doi.org/10.1109/TC.1979.1675439

[19] B. Norris and B. Demsky, “Cdschecker: Checking concurrent data structures written with c/c++ atomics,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, ser. OOPSLA ’13. New York, NY, USA: ACM, 2013, pp. 131–150. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509514

[20] NVIDIA, “NVIDIA Developer Forum,” https://devtalk.nvidia.com.

[21] J. Sanders and E. Kandrot, *CUDA by Example*. Addison-Wesley, 2011.

[22] T. Sorensen and A. F. Donaldson, “Exposing errors related to weak memory in gpu applications,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’16, 2016, pp. 100–113.

[23] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O’Connor, and S. W. Keckler, “Flexible software profiling of gpu architectures,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 185–197.

[24] D. Weaver and T. Germond, *The SPARC Architecture Manual Version 9*. Prentice Hall, Englewood Cliffs, N.J., 1994.

[25] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind, “Nemos: a framework for axiomatic and executable specifications of memory consistency models,” in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004.