Simulee: Detecting CUDA Synchronization Bugs via Memory-Access Modeling

Mingyuan Wu
Southern University of Science and Technology
Shenzhen, China
11849319@mail.sustech.edu.cn

Yicheng Ouyang
Southern University of Science and Technology
Shenzhen, China
11610313@mail.sustech.edu.cn

Husheng Zhou
University of Texas at Dallas
Dallas, USA
husheng.zhou@utdallas.edu

Lingming Zhang
University of Texas at Dallas
Dallas, USA
lingming.zhang@utdallas.edu

Cong Liu
University of Texas at Dallas
Dallas, USA
cong@utdallas.edu

Yuqun Zhang*
Southern University of Science and Technology
Shenzhen, China
zhangyq@sustech.edu.cn

ABSTRACT

While CUDA has become a mainstream parallel computing platform and programming model for general-purpose GPU computing, how to effectively and efficiently detect CUDA synchronization bugs remains a challenging open problem. In this paper, we propose the first lightweight CUDA synchronization bug detection framework, namely *Simulee*, to model CUDA program execution by interpreting the corresponding *LLVM* bytecode and collecting the memory-access information for automatically detecting general CUDA synchronization bugs. To evaluate the effectiveness and efficiency of *Simulee*, we construct a benchmark with 7 popular CUDA-related projects from *GitHub*, upon which we conduct an extensive set of experiments. The experimental results suggest that *Simulee* can detect 21 out of the 24 manually identified bugs in our preliminary study and also 24 previously unknown bugs among all projects, 10 of which have already been confirmed by the developers. Furthermore, *Simulee* significantly outperforms state-of-the-art approaches for CUDA synchronization bug detection.

ACM Reference Format:
Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang*. 2020. *Simulee*: Detecting CUDA Synchronization Bugs via Memory-Access Modeling. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380358

1 INTRODUCTION

CUDA [3] is a mainstream parallel computing platform and programming model that allows software developers to leverage general-purpose GPU (GPGPU) computing [4]. CUDA is advanced in simplifying I/O streams to memories and dividing computations into sub-computations since it parallelizes programs in terms of grids and blocks. In addition, CUDA enables more flexible cache management that speeds up the floating point computation of CPUs. CUDA is thus considered rather powerful for accelerating deep-neural-network-related applications where relevant matrix computations can be efficiently loaded.

Unlike traditional CPU multi-thread programs where the synchronization mechanism is built upon managing the shared resource that can be accessed by multiple threads, the typical synchronization mechanism of GPU programs is built upon synchronizing the instruction flows [1]. In particular, since GPU programs use barriers rather than locks for synchronization and enable simplified barrier-based happens-before relations, traditional lockset-based [13, 38] and happens-before-based bug detection approaches [18, 34] for CPUs become obsolete and expensive in detecting parallel-computing-related bugs for GPUs.

Recently, several approaches, e.g., [6, 9, 20, 30, 31, 37, 50, 51], have been proposed to detect synchronization bugs for CUDA kernel functions. In general, they can be categorized into two classes: (1) the compiler-based approaches that leverage compiler instrumentation with/without static analysis to identify race-free locations for detecting data-race bugs [6, 20, 37, 50, 51], and (2) the SMT-solver-based approaches that integrate static analysis and symbolic execution with SMT solver to detect data race and barrier-divergence bugs [9, 30, 31]. Although such approaches can detect CUDA synchronization bugs under their respectively defined environments, they have limited applicability and may rely on certain assumptions. In particular, the compiler-based approaches are limited due to relying on developer-provided test inputs and detecting data races only; while the SMT-solver-based approaches can be heavyweight due to triggering expensive static-analysis overhead and may also involve manually-provided test inputs.

In this paper, to address the aforementioned issues, we develop a systematic lightweight bug detection framework, namely *Simulee* [7],

Yuqun Zhang is the corresponding author.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380358

which automatically detects general synchronization bugs for CUDA kernel functions. In particular, *Simulee* generates a *Memory-Access Model* that maintains thread-wise memory-access information including thread id, visit order, and action. Accordingly, *Simulee* utilizes the *LLVM* bytecode of CUDA kernel functions to initialize the running environmental setups including arguments, dimensions, and global memory if necessary based on the *Memory-Access Model*. Moreover, *Simulee* applies Evolutionary Programming [21] to approach error-inducing inputs for exposing the dangerous program execution paths which may possibly induce synchronization bugs. Subsequently, *Simulee* collects the corresponding memory-access information from such paths. At last, by combining CUDA specifics, such collected memory-access information are analyzed to find whether they can potentially lead to synchronization bugs.

Compared with other CUDA synchronization bug detection approaches, *Simulee* can detect multiple bug types including data race, redundant barrier function, and barrier divergence fully automatically. Moreover, *Simulee* benefits from exploring the dangerous execution paths guided by the lightweight evolutionary search, without incurring large overhead (e.g., for constraint solving), such that it is more efficient than existing state-of-the-art approaches [9, 30, 31] that usually rely on heavyweight techniques or manual inputs.

To evaluate the effectiveness and efficiency of *Simulee*, we first collect 7 popular CUDA-related projects from *GitHub* (with a total of 17928 commits and 1.36 million LOC by Aug, 2019) as our benchmark suite. Then, as a preliminary study, we manually identified 24 real-world synchronization bugs from a subset of the studied real-world CUDA projects and the *GKLEE* [30] benchmark suite. We conduct a set of experiments to explore (1) how many of those manually identified bugs can be detected by *Simulee* for the preliminary analysis; (2) whether *Simulee* can detect previously-unknown bugs on those real-world CUDA projects; and (3) how *Simulee* compares with state-of-the-art approaches in CUDA bug detection. The experimental results suggest that *Simulee* can successfully detect 21 of the 24 manually identified synchronization bugs in our preliminary study. Furthermore, it even detects 24 previously unknown bugs from all the 7 real-world CUDA projects, 10 of which have already been confirmed by the corresponding developers. The experimental results also demonstrate that *Simulee* can be much more effective than state-of-the-art approaches, i.e., *GKLEE* [30], *GKLEE-SESA* [31], *GPUVerify* [9], and *RaceChecker* [6], in detecting synchronization bugs, e.g., *Simulee* can detect 2X more previously unknown bugs than any of the other studied state-of-the-art approaches. In addition, *Simulee* can significantly outperform the other studied approaches in terms of the runtime overhead.

In summary, our paper makes the following contributions:

- **Idea.** To the best of our knowledge, we propose the first idea of CUDA synchronization bug detection via evolutionary search guided by memory-access modeling.
- **Implementation.** We have implemented the proposed idea as a lightweight, fully automated, and general-purpose detection framework for CUDA synchronization bugs, namely *Simulee*, that can automatically detect a wide range of synchronization bugs in CUDA programs which are hard to be captured manually.

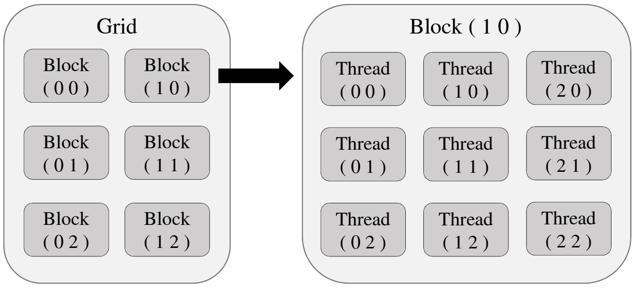

Figure 1: CUDA Hierarchy

- **Study.** We evaluate *Simulee* under multiple experimental setups. The results suggest that *Simulee* is able to detect most of the manually identified synchronization bugs in the benchmark. In addition, it even detects 24 previously-unknown bugs of the entire benchmark fully automatically and significantly outperforms state-of-the-art approaches.

2 BACKGROUND

In this section, we give an overview on CUDA, the CUDA parallel computing mechanism, and typical CUDA synchronization bugs.

CUDA Overview. CUDA is a parallel computing platform and programming model, which allows developers to use GPU hardware for general-purpose computing, e.g., autonoumos driving [48, 52, 53]. CUDA is composed of a runtime library and an extended version of C/C++. In particular, CUDA programs are executed on GPU cores, namely "device", while they also need to be allocated with resources on CPUs, namely "host", prior to execution. As a result, developers need to retrieve allocated resources such as global memory after CUDA program execution. To conclude, a complete CUDA program contains 3 runtime stages: (1) host resource preparation, (2) kernel function execution, and (3) host resource retrieve.

CUDA Parallel Computing Mechanism. A CUDA kernel function refers to the part of CUDA programs that runs on the device side. Specifically, thread is the kernel function's basic execution unit. At the *physical* level, 32 threads are bundled as a thread warp wherein all the threads execute the same statement at any time except undergoing a branch divergence, while at the *logic* level, one or more threads are contained in a block, and one or more blocks are contained in a grid. The execution of kernel functions is initialized by setting runtime environments, e.g., dimensions of grids and blocks, passing relevant arguments, such that the computation can be divided into sub-computations and each sub-computation can be dispatched to different threads. Eventually, the results of sub-computations can be merged as the final result of the overall computation through applying algorithms such as reduction [35]. The hierarchy of the parallel computing mechanism of CUDA kernel functions is presented in Figure 1.

To synchronize threads, CUDA applies *barriers* at which all the threads in one block must wait before any can proceed. In CUDA kernel functions, the barrier function is "`__syncthreads()`" which synchronizes threads from the same block. When a thread reaches a barrier, it is expected to proceed to next statement if and only if all the threads from the same block have reached the same barrier. Otherwise, the program would be exposed to undefined behaviors.

```
1   tid = threadIdx.x;
2   ....
3   if (y > 0 && a < C)
4      f_val2reduce[tid] = f;
5   else
6     f_val2reduce[tid] = INFINITY;
+7  __syncthreads(); // fix by adding syncthreads
8   // get_block_min will write data to f_val2reduce
9   int ip = get_block_min(f_val2reduce, f_idx2reduce);
10  float up_value_p = f_val2reduce[ip];
....
```

Figure 2: An Example of Data Race

```
1   const unsigned tid = threadIdx.x;
2   s_median[tid] = FLT_MAX;
3   s_idx[tid] = 0;
-4  __syncthreads();

5   if (i < iterations) {
6       ...
7       s_idx[tid] = i;
8       s_median[tid] = m;
9   }
....
```

Figure 3: An Example of Redundant Barrier

CUDA Synchronization Bugs. There are three major synchronization bugs in CUDA kernel functions: data race, barrier divergence [14], and redundant barrier function [1]. Specifically, data race indicates that for accessing global or shared memory, CUDA cannot guarantee the visit order of "read&write" actions or "write&write" actions from two or more threads. For example, Figure 2 demonstrates the bug-fixing *Revision no.* "febf515a82" in the file "smo-kernel.cu" of the project "thundersvm" [42], one of the highly-rated *GitHub* projects. It can be observed from Figure 2 that the "if" statement writes to the memory of "f_val2reduce", while inside the device, the function "get_block_min" writes to the same memory. This "write&write" bug is fixed by adding "__syncthreads" which synchronizes actions among threads.

A barrier function is considered redundant when there is no data race after deleting it from source code. A redundant barrier function compromises the program performance in terms of time and memory usage. For instance, Figure 3 demonstrates bug-fixing *Revision no.* "31761d27f01" in file "kernel/homography.hpp" from project "arrayfire" [8]. It can be observed that the block is one-dimensional from Line 1, the value of "tid" is assigned only by "threadIdx.x". That indicates that the "tid"s are identical among different threads from the same block. As a result, "s_median[tid]" and "s_idx[tid]" can only be accessed by one thread, leading to a redundant barrier function in Line 4 because there is no race in "s_median" or "s_idx" after deleting it.

A barrier divergence takes place when some threads in a block complete their tasks and leave the barrier while the others have not reached the barrier yet. Figure 4 demonstrates the bug-fixing *Revision no.* "0ed6cccc5ff" in the file "nearest_neighbour.hpp" from the project "arrayfire" caused by barrier divergence. It can be indicated from Figure 4 that developers make sure all the threads in the same block reach the same barrier in every execution of the kernel function by moving the statement of "__syncthreads()" outside the given branch. Otherwise they will have to handle undefined behaviors.

```
....
1      s_dist[sid] = dist;
2      s_idx[sid]  = s_idx[sid + i];
-3     __syncthreads();
4    }
5    //fix by moving the barrier out.
+6   __syncthreads();
7  }
....
```

Figure 4: An Example of Barrier Divergence

3 FRAMEWORK OF SIMULEE

In this section, we introduce *Simulee*, a lightweight, automatic, and device-independent framework to detect real-world CUDA synchronization bugs. Typically, *Simulee* takes *LLVM* bytecode translated from CUDA kernel function programs as input. Then, it automatically generates the associated error-inducing test inputs, and yields *Memory-Access Model* to detect synchronization bugs. Specifically, *Simulee* is composed of two components—"Automatic Input Generation" and "Bug Detection via *Memory-Access Model*". "Automatic Input Generation" is initialized by inputting the *LLVM* bytecode of CUDA kernel function programs. Next, it slices the memory-access statements (e.g., read and write statements) and inputs them for Evolutionary Programming [21]. Subsequently, Evolutionary Programming helps generate error-inducing environmental setups by iteratively mutating and sorting dimensions/arguments and passes the acceptable ones to "Bug Detection via *Memory-Access Model*". At last, "Bug Detection via *Memory-Access Model*" traces real execution paths by using the error-inducing inputs and collects the memory-access information from the paths to detect whether there are synchronization bugs, as it were "simulating" runtime environment. The details can be found in Figure 5.

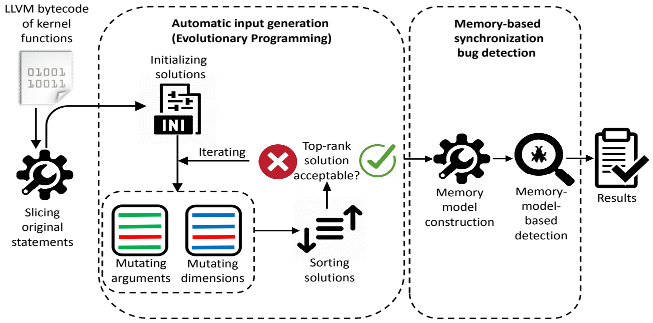
LLVM bytecode of kernel functions
01001 10011
Slicing original statements
Automatic input generation (Evolutionary Programming)
Initializing solutions
INI
Iterating
Top-rank solution acceptable?
Mutating arguments
Mutating dimensions
Sorting solutions
Memory-based synchronization bug detection
Memory model construction
Memory-model-based detection
Results

Figure 5: Framework of *Simulee*

3.1 Automatic Input Generation

Generating potentially error-inducing inputs is essentially equivalent to generating the inputs that can lead to the memory-access conflicts among threads to improve the possibility of CUDA synchronization bug occurrences. However, how to automatically generate such error-inducing inputs remains challenging. Intuitive solutions, e.g., random generation, coverage-oriented generation, can be limited in effectiveness and efficiency, because they are not specially designed for triggering memory-access conflicts. In this section, we introduce how *Simulee* automatically generates potentially error-inducing inputs for exposing the dangerous program execution paths which could lead to synchronization bugs in an effective and efficient manner.

3.1.1 Intuition. An effective and efficient automatic approach to generate potentially error-inducing inputs for triggering CUDA synchronization bugs implies generating as many memory-access conflicts as possible within a short time limit. Given the *i*th memory address and the kernel function inputs, i.e., grid and block dimensions and arguments, $f(i)$ is defined as the number of threads that access the *i*th memory address while *g(i)* is a function that returns 1 when the *i*th memory address is accessed by any thread and returns 0 otherwise. $[start, end]$ denotes the memory-access range. An intuitive target function $F(dimensions, arguments)$ can be presented in Equation 1 which denotes the ratio of the total number of the accessed memory addresses to the total number of the memory-access threads:

$$F(dimensions, arguments) = \frac{\sum_{i=start}^{end} g(i)}{\sum_{i=start}^{end} f(i)} \tag{1}$$

It can be derived that the max value of $F(dimensions, arguments)$ is 1 which denotes that there is no memory-access conflict between any thread pair. On the other hand, the smaller $F(dimensions, arguments)$ is, the higher chance the memory-access conflict takes place. Therefore, $F(dimensions, arguments)$ can be used for optimization to obtain error-inducing inputs that trigger CUDA synchronization bugs. Note that since $F(dimensions, arguments)$ is discrete, we choose Evolutionary Programming [44] as our optimization approach.

3.1.2 Algorithm. The framework of "Automatic Input Generation" is presented in Algorithm 1. First, *Simulee* randomly initializes *arguments* and *dimensions* to create and sort individual solutions for evolving (Lines 3 to 7). In each generation, each solution is mutated to generate two children, which are added to the whole population set (Lines 8 to 14). Next, the population winners survive for the subsequent iterations (Lines 15 to 16). The iterations can be terminated once it finds an acceptable solution. Otherwise, after completing the iterations, it returns the optimal solution.

Algorithm 1 Framework for Automatic Input Generation

```
    Input : population, generation
    Output: acceptable arguments and dimensions
 1: function EVOLUTION_ALGORITHM
 2:     populationLst ← list()
 3:     for i in population do
 4:         singleSolution ← InitialSolution()
 5:         singleScore ← fitness(singleSolution)
 6:         populationLst.append([singleSolution, singleScore])
 7:     sortByScore(populationLst)
 8:     for i in generation do
 9:         childLst ← list()
10:         for solution in populationLst do
11:             childrenSolutions ← mutation(solution)
12:             newScores ← fitness(childrenSolutions)
13:             childLst.append([childrenSolutions, newScores])
14:         populationLst.merge(childLst)
15:         sortByScore(populationLst)
16:         populationLst ← populationLst[:population]
17:         if populationLst[0] acceptable then
18:             return populationLst
19:     return populationLst
```

Initial Solutions. The initial dimensions and arguments are randomly generated and passed to fitness functions as initial solutions for future evolution. Note that the dimensions can be extracted from kernel functions. For instance, if a kernel function has "threadIdx.x" and "threadIdx.y", it means the block is two-dimensional.

Fitness Function. Equation 1 is chosen as the primary fitness function for Evolutionary Programming. Specifically, the output of $F(dimensions, arguments)$ is the fitness score for a solution of dimensions and arguments in Evolutionary Programming. However, it is difficult to derive an optimal solution of dimensions and arguments by only optimizing $F(dimensions, arguments)$. In particular, since $F(dimensions, arguments)$ is non-differentiable when the gradient does not exist, it is hard to find an optimal solution given the set of inferior solutions, e.g., all the solutions of $F(dimensions, arguments)$ are "1"s. To address such issues, we design a secondary fitness function such that they are sorted according to their possibility to be optimal: $R(start, end) = end - start$. In particular, it indicates that a smaller memory-access range leads to a higher possibility of memory-access conflict. As a result, we define fitness score of the primary fitness function as *primary score*, and the fitness score of the secondary fitness function as *secondary score*. During the population evaluation, the *primary score* is sorted first; if and only if the top-ranked *primary score* is 1, the *secondary score* is sorted to decide which solution is more likely to converge to the minimum of $F(dimensions, arguments)$.

Mutation. In *Simulee*, solutions are generated by mutation, where each solution generates two children in one generation. Specifically, *arguments* and *dimensions* are independent from each other during mutation with respective mutation strategies. The mutate strategy for *dimensions* is trivial: first, *Simulee* randomly generates an integer vector ranging from -1 to 1 according to the dimension size; next, the child's dimension is mutated by summing the parent's dimension and the generated integer vector.

The details of the mutation strategy for *arguments* is presented in Algorithm 2. Since the memory-access-relevant arguments are numbers, *Simulee* views them as floating numbers and converts them back to their actual types when executing $f(i)$. Accordingly, each generation generates two children: one adds a random number generated by standard Normal Distribution [5] ($N(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2 \big/ 2}$) to the arguments inherited from the parent solution, and the other adds a random number generated by standard Cauchy Distribution [2] ($C(x) = \frac{1}{\pi(1+x^2)}$) to the arguments inherited from the parent solution. We define the search step length of the arguments as

Algorithm 2 Mutating Arguments

```
Input : parent
Output: mutation children solutions
function ARGUMENT_MUTATION
    normalSolution ← copy(parent)
    cauchySolution ← copy(parent)
    for argument in parent do
        normalSolution[argument] ← parent[argument] + normal()
        cauchySolution[argument] ← parent[argument] + cauchy()
    return normalSolution, cauchySolution
```

the absolute value of the number generated from the two aforementioned distributions, with the expected values shown in Equations 2 and 3.

$$E_{normal}(x) = \int_0^\infty x \frac{1}{\sqrt{2\pi}} e^{-x^2 \big/ 2} dx = 0.399 \quad (2)$$

$$E_{cauchy}(x) = \int_0^\infty x \frac{1}{\pi(1+x^2)} dx = +\infty \quad (3)$$

We next explain why we apply the above two distributions. It can be observed from Equations 2 and 3 that, the step length generated from standard normal distribution is expected to be small. That indicates that if there is an optimal solution nearby, the generated child is likely to approach it. On the contrary, the step length generated from standard cauchy distribution is expected to be large. That indicates that if there is an inferior solution nearby, the generated child is likely to escape from it.

Acceptable Function. The acceptable function is used to terminate the whole process given an acceptable solution. In this paper, the acceptable solution is defined as that *primary score* is smaller than 0.3.

In summary, by applying Evolutionary Programming, *Simulee* is expected to deliver error-inducing grid and block dimensions and arguments that lead to memory-access conflicts and trigger CUDA synchronization bugs.

3.2 Memory-based Synchronization Bug Detection

With the auto-generated error-inducing inputs, the synchronization bug detection of *Simulee* is established on building a *Memory-Access Model* that depicts thread-wise memory-access instances. Based on the *Memory-Access Model*, *Simulee* develops a set of criteria to detect synchronization bugs including data race, redundant barrier functions, and barrier divergence.

3.2.1 Memory-Access Model. The *Memory-Access Model* accessed by the kernel functions is defined to be composed of a set of *Memory Units* where each *Memory Unit* corresponds to a memory address and is composed of a set of *Unit Tuples*. A *Unit Tuple* is defined as a three-dimensional vector space <*visit_order*, *thread_id*, *action*>, where *visit_order* represents the visit order to the associated memory address from different threads, *thread_id* represents the indices of such threads, and *action* refers to the read or write action from those threads.

An example of *Memory Unit* is demonstrated in Figure 6 with four *Unit Tuples* <0, (1 0 0), read>, <0, (2 0 0), read>, <0, (3 0 0), read>, and <1, (3 0 0), read> where threads (1,0,0), (2,0,0), and (3,0,0) read the same memory address in the same *visit_order* since none of them have reached any barrier function before they read. Assume all the threads reach a barrier function later and thread (3 0 0) reads, the *visit_order* is then incremented from 0 to 1 for thread (3 0 0) and the other threads afterwards.

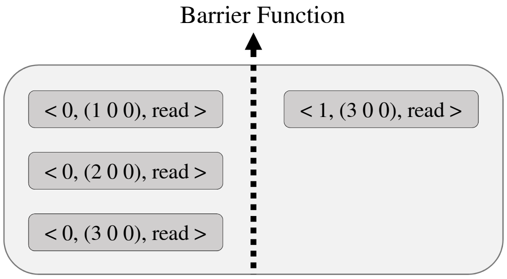

Figure 6: *Memory Unit* Example

3.2.2 Memory Accessing Model Construction. Since *Memory-Access Model* is only associated with barrier functions and memory-access statements, it is applicable to detect synchronization bugs by obtaining such statements and then extracting/analyzing the memory-access information instead of executing the complete CUDA programs on GPUs, i.e., modeling the execution of CUDA kernel function programs. This modeling process is initiated by inputting the auto-generated block and grid dimensions and arguments passed to the kernel functions. Next, it constructs the *Memory Unit* for each memory address by tracing back the execution path for each thread.

The overall *Memory-Access Model* construction is demonstrated in Algorithm 3. In particular, the algorithm is launched to initialize the block and grid dimensions as well as the global and shared memory for each thread (Lines 2 to 5). Next, for each block, the shared memory (Line 7) and the thread-wise *visit_order* for each global and shared memory address (Lines 8 to 9) are initialized. If there are still some unterminated threads, for all of them, their corresponding *Memory Unit*s are derived based on the collected parameters, e.g., `globalMem` and `visitOrderGlobal` (Lines 10 to 14). The construction of the thread-wise *Memory Unit*s for shared memory and global memory are completed if there is no running thread left (Lines 15 to 16).

Algorithm 4 illustrates the details of *Memory-Access Model* construction for a single thread. Specifically, given a running thread and the parameters passed by Algorithm 3 (Lines 2 to 4), Algorithm 4 is initialized by detecting whether the current statement is the end of file. If so, the thread would be terminated. If there is any thread halting afterwards, we can confirm there is a "barrier divergence" bug because that indicates at least a thread has not reached the barrier function where the other threads of the same block all have completed their tasks and left (Lines 5 to 9).

If the current statement calls barrier function and all the other threads have reached the same barrier function, the *visit_order* for both global and shared memory would be incremented if they have been visited before (Lines 10 to 13), since it indicates that (1) all the threads in one block have visited the current memory address and (2) the subsequent visits in the same block would be made rigorously later than the previous visits. On the other hand, if the

Algorithm 3 *Memory-Access Model* construction

```
Input : gridDim, blockDim, arguments
Output: Memory-Access Model
function CONSTRUCT_MEMORY_MODEL
    BLOCKS ← generateFromDimension(gridDim)
    THREADS ← generateFromDimension(blockDim)
    globalMem ← [MemoryUnit() for i in range(globalSize)]
    sharedMemLst ← list()
    for blk in BLOCKS do
        sharedMem ← [MemoryUnit() for i in range(sharedSize)]
        visitOrderGlobal ← [0 for i in range(globalSize)]
        visitOrderShared ← [0 for i in range(sharedSize)]
        while hasUnterminatedThread() do
            for t in THREADS do
                env ← Environment(arguments)
                PROCESS_THREAD(t, globalMem, sharedMem,
                visitOrderGlobal, visitOrderShared, env)
        sharedMemLst.append(sharedMem)
    return globalMem, sharedMemLst
```

current statement does not call barrier function, the corresponding *visit_order* and the *action* of the associated thread is recorded to construct the *Memory-Access Model* (Lines 14 to 21).

3.2.3 *Bug Detection via Memory-Access Model Mechanism.*

The design of *Memory-Access Model* can be used in *Simulee* to detect CUDA synchronization bugs, i.e., data race, redundant barrier function, and barrier divergence.

Data Race. In general parallel computing programs, a possible data race takes place when multiple threads access the identical memory address in the same visit order and at least one of them writes. Specifically in CUDA kernel functions, besides the generic circumstances, a data race also takes place when (1) the threads are from different thread warps, or (2) the threads from the same thread warp underwent branch divergence, or (3) the threads from the same thread warp without undergoing branch divergence write to the same memory address by the same statement. By combining the data race detection criteria above and the design of *Memory-Access Model*, *Simulee* can detect data race in CUDA kernel functions as described in Theorem 3.1.

Theorem 3.1. *Given two Unit Tuples ψ_i and ψ_j from the identical Memory Unit, a data race between them takes place if the conditions below are met:*

- $\psi_i[visit\_order] = \psi_j[visit\_order]$
- $\psi_i[thread\_id] \mathrel{!=} \psi_j[thread\_id]$
- $\psi_i[action]$ = 'write' or $\psi_j[action]$ = 'write'

when the threads of ψ_i and ψ_j are (1) from different thread warps or (2) executing the "write" action on the same statements in the same thread warp or (3) underwent branch divergence before the current "write" action.

Redundant Barrier Function. A redundant barrier function indicates that no data race can be detected by removing that barrier function. In CUDA kernel functions, the *visit_order* is incremented for one *Unit Tuple* when at least one thread reaches a barrier function. In other words, two *Unit Tuple*s with adjacent *visit_order* in one *Memory Unit* indicates the presence of a barrier function, shown in Figure 6. Therefore, to detect whether a barrier function is redundant or not, it is essential to collect all the associated *Unit Tuple*s and analyze whether they together would lead to data race. The barrier function is defined to be redundant if no data race can be detected among such *Unit Tuple*s.

The details of how to detect data race and redundant barrier function based on *Memory-Access Model* are presented in Algorithm 5. For each *Memory Unit*, to detect data race, *Simulee* first groups the *Unit Tuple*s with the same *visit_order*. For all the *Unit Tuple*s in one group, *Simulee* checks whether any *Unit Tuple* has data race with others according to Theorem 3.1 (Lines 4 to 16). To detect redundant barrier function of one *Memory Unit*, *Simulee* extracts its *visit_order* and groups all the *Unit Tuple*s with adjacent *visit_order* to find out whether any data race can take place (Lines 18 to 22). If there is no data race, *Simulee* identifies the associated barrier function and increments its recorder by 1 (Lines 23 to 24). At last, it checks whether the total recorder number matches the total number of the changing *visit_order* caused by that barrier function which can be obtained after constructing the *Memory-Access Model*. This barrier function is redundant if the two numbers are equivalent (Lines 25 to 28).

Barrier Divergence. As mentioned in Section 3.2.2, barrier divergence can be detected during constructing *Memory-Access Model* when there is any halting thread after the current execution is terminated, because it indicates that there is at least one thread which has not reached the barrier function while the others have already left.

To conclude, *Simulee* first applies Evolutionary Programming to generate error-inducing grid and block dimensions and arguments. Next, *Simulee* inputs such dimensions and arguments to construct *Memory-Access Model* that delivers thread-wise memory-access information. Eventually, such information, along with the CUDA synchronization bug detection mechanism, are used to detect whether there exists any CUDA synchronization bug.

4 EVALUATION

In this section, we conduct an extensive experimental study to evaluate the effectiveness and efficiency of *Simulee* in detecting synchronization bugs of CUDA kernel functions. In particular, we first perform a preliminary study on 24 manually identified CUDA synchronization bugs to explore the efficacy of *Simulee*. Next, we explore the capability of *Simulee* in detecting previously-unknown bugs from all the real-world CUDA projects in our benchmark suite. Furthermore, we also compare *Simulee* with multiple existing state-of-the-art approaches to explore whether *Simulee* can outperform them.

4.1 Benchmark Construction

To conduct a preliminary study for evaluating *Simulee*, it is essential to establish a set of CUDA synchronization bugs as the ground truth. To this end, we first consider an existing benchmark, i.e., the *GKLEE* dataset [22], and select 4 synchronization bugs that can represent all the basic synchronization bug patterns in the dataset.

Furthermore, we augment the bug dataset with more real-world CUDA bugs. In this paper, we aim to collect important and influential real-world CUDA benchmark projects for our evaluation by defining a set of policies for selecting open-source CUDA

Algorithm 4 Thread Processor

```
Input : thread, globalMem, sharedMem, visitOrderGlobal,
        visitOrderShared, env
Output:None or BARRIER_DIVERGENCE
function PROCESS_THREAD
    if shouldHalt() or isFinished() then
        return
    curStmt ← env.getNextInstruction()
    if curStmt.isEOF() then
        thread.finish()
        if hasHaltThreads() then
            return BARRIER_DIVERGENCE
        return
    if curStmt.isSyncthreads() then
        if all threads reach same barrier then
            updateCurrentVisitOrder(visitOrderShared)
            updateCurrentVisitOrder(visitOrderGlobal)
    else
        isGlobal, memIndex ← simulateExecute(curStmt, env)
        if isGlobal then
            index ← visitOrderGlobal[memIndex]
            updateMemoryModel(globalMem, memIndex, index)
        else
            index ← visitOrderShared[memIndex]
            updateMemoryModel(sharedMem, memIndex, index)
    return
```

Table 1: Subject Statistics

Projects	Star Number	Commit Number	LoC
kaldi	6860	8681	364K
arrayfire	2791	5314	381K
thundersvm	1014	827	343K
cuda-cnn	368	135	12K
cudaSift	123	247	24K
cudpp	268	302	58K
gunrock	550	2422	178K

projects in *GitHub*. Specifically, we initialize our CUDA project collection by searching the keyword "CUDA" and collect more than 12,000 projects from *GitHub* in the first place. Next, we sort these projects in terms of the star number and commit number. We randomly select 7 projects with large star/commit numbers. As a result, we collect "`kaldi`" [29], "`arrayfire`" [8], "`thundersvm`" [42], "`cuda-cnn`" [54], "`cudaSift`" [11], "`cudpp`" [15] and "`gunrock`" [24] for augmenting our benchmark suite as listed in Table 1.

More specifically, we randomly select projects "arrayfire", "kaldi", and "thundersvm" from our real-world CUDA benchmark suite to retrieve their historical synchronization bugs. Note that we do not consider all the projects from our benchmark suite since the manual bug retrieval process can be quite time-consuming. The synchronization bugs for those selected projects are identified based on their commit messages and "`git diff`" results. The specific operations are listed as follows. Following prior study on other types of bugs [49], we first filter the commits and only keep the commits with the messages that contain at least one keyword in the set {"fix", "error", "sync"} to retain the commits that have higher chances to contain synchronization bugs. However, the commit messages only with these keywords might not be relevant with CUDA bugs. Therefore, next, among the filtered commit messages, we further filter them according to whether they have at least a keyword in the set {"`__global__`", "`__device__`"} or match at least one regular expression in the set {"`cuda\w+\s*[(]`", "`[ˆ<]<<<[ˆ<]`"} with its parent node's "`git diff`" results. To illustrate, "`__global__`" is the modifier of kernel functions and "`__device__`" is the modifier of the device functions that can be called by kernel functions. "`cuda\w+\s*[(]`" is designed in accordance with the information that the resource is prepared/released in host side before/after executing kernel functions. For instance, "`cudaMalloc((void **) &host, sizeof(int) *100)`" allocates a global 400-byte memory for kernel functions before execution; "`cudaFree(&host)`" releases the allocated memory for kernel functions after execution. "`[ˆ<]<<<[ˆ<]`" is designed in accordance with the scenario that sets up the environment for kernel functions, e.g., "`function<<<grid_size, block_size>>>(arguments)`". All these regular expressions together deliver the complete life cycle of executing kernel functions such that all the bugs of the whole life cycle can be covered. We further manually review all the remaining commits after the above two rounds of filtering to remove any potential false positive. Due to the tedious and time-consuming manual inspection, all the selected CUDA projects are analyzed within the most recent 1000 commits or all of them if there are less than 1000 commits. As a result, we collected a total of 20 real-world

Algorithm 5 Bug Detection via *Memory-Access Model*

```
Input : memoryModel, changingVisitOrderNumber
Output:DATA_RACE, REDUNDANT_BARRIERS
function EXAMINE_MEMORY_MODEL
    DATA_RACE ← False
    REDUNDANT_BARRIERS = dict()
    for memoryUnit in memoryModel do
        for visitOrder in memoryUnit do
            tuples ← getTuplesByOrder(visitOrder)
            for thread performing write in tuples do
                otherTs ← getDifferentThreads(thread, tuples)
                for t in otherTs do
                    if inSameWarp(t, thread) then
                        if usingSameStmt(t, thread) then
                            DATA_RACE ← True
                        if hasBranchDivergence(t, thread) then
                            DATA_RACE ← True
                    else
                        DATA_RACE ← True
        barrierDict = dict()
        for visitOrder in memoryUnit do
            nextOrder ← visitOrder + 1
            current ← getTuplesByOrder(visitOrder)
            target ← getTuplesByOrder(nextOrder)
            if canMergeWithoutRace(target, current) then
                barrier ← getSplitBarrier(nextOrder, memoryUnit)
                barrierDict[barrier] ++
    for barrier in barrierDict do
        REDUNDANT_BARRIERS[barrier] ←
        isRedundant(barrierDict[barrier],
        changingVisitOrderNumber[barrier])
    return DATA_RACE, REDUNDANT_BARRIERS
```

CUDA bugs. By combining with the 4 synchronization bugs from the *GKLEE* benchmark suite, we obtain a total of 24 CUDA bugs as the ground truth for our preliminary study.

4.2 Environment Setups

We performed our evaluation on a desktop machine, with Intel(R) Xeon(R) CPU E5-4610 and 320 GB memory. The operating system is Ubuntu 16.04. For the Evolutionary Programming settings of "Automatic Input Generation" in *Simulee*, the population and generation are both set to be 10 by default. Note that the *Simulee* webpage [7] includes more experimental results under different settings for the Evolutionary Programming component.

We select state-of-the-art CUDA synchronization bug detection approaches, i.e., *GPUVerify* [9], *GKLEE* [30], *GKLEE-SESA* [31], and *RaceChecker* [6] for the performance comparison with *Simulee*. More specifically, *RaceChecker* is the NVIDIA's official tool and represents state-of-the-art compiler-based approach, while the rest approaches represent state-of-the-art SMT-solver-based approaches. Note that the timeout for all studied techniques are uniformly set to be 5 hours.

4.3 Result Analysis

4.3.1 *Preliminary Study on Known Bugs.*

We first present the experimental results for detecting the 24 manually identified synchronization bugs in our preliminary study in Table 2. In the table, DR, BD, and RB respectively denote data race, barrier divergence, and redundant barrier function. "✓", "✗", and "F" respectively denote the successful, failed, and false-positive bug-detection attempts. **TO** denotes that the associated bug detection attempt incurs timeout, while **N/A** denotes that the buggy kernel function is out of the scope of the bug-detection capability of the corresponding techniques. Note that each row represents one kernel function, which could include multiple bugs (indicated by Column "Bug Num"). For such cases, we present all the reported bugs for each kernel function, e.g., "✓FFF" denotes that the corresponding technique reports four bugs in total for the kernel function, 3 of which are false positives.

From the table, we can observe that, in terms of the overall effectiveness, *Simulee* can detect 21 (87.5%) of the the 24 manually identified synchronization bugs within seconds, e.g., *Simulee* at most costs 10.73 seconds (when applied on kernel function `JacobiSVD`). We further analyze the 3 cases for which *Simulee* fails to detect bugs. *Simulee* fails to detect the data race in kernel function `hamming_matcher` because this bug can only be exposed under occasional branch coverage, which is out of the scope of the input generation component of *Simulee* that focuses on the memory-access conflict potentials. *Simulee* fails to detect bugs for kernel function `_softmax_reduce` and `_div_rows_vec` because they are largely reimplemented to reduce the usage of unnecessary barrier functions while the current version of *Simulee* is not designed for such challenging bugs requiring large code refactoring.

We next analyze the comparison results between *Simulee* and state-of-the-art *GPUVerify*, *GKLEE*, *GKLEE-SESA* and *RaceChecker* from the table in details. Note that *GPUVerify* requires user-provided dimension settings and the other techniques require the overall user-provided environmental settings. We provide them all with the ideal settings that can trigger the most possible bugs for fair comparison with *Simulee*.

GPUVerify. *GPUVerify* is designed to detect data-race and barrier-divergence bugs via integrating static analysis with SMT solvers. From the table, we can observe that *GPUVerify* can successfully detect 17 out of the 24 synchronization bugs.

Meanwhile, *GPUVerify* also reports 6 false positives. We next manually check all such false positives and observe that *GPUVerify* tends to report false positives due to two reasons. First, the false positives are triggered by the bottleneck of the static analysis optimization. For instance, in kernel functions `deadlock_0` and `computeDescriptor`, *GPUVerify* reports false positives due to nonexistent execution paths. Second, its adopted SMT solvers are used to detect data-race bugs caused by thread-wise access conflicts; however, the thread warp mechanism adopted in CUDA kernel functions can resolve some of such bugs, e.g., the inter-instruction "read&write" race. Shown in Figure 7, `warpSize` is equal to 32 in Revision `5cc9731af4f` of function `_trace_mat_mat_trans` from project `kaldi`. Suppose there are two threads `(0 0 0)` and `(1 0 0)`, and the `tid` of thread `(0 0 0)` is 0 while the `tid` of thread `(1 0 0)` is 1. Moreover, when the loop terminates, `shift` is set to 1. Meanwhile, for thread `(0 0 0)`, statement `ssum[0] += ssum[0 + 1]` is executed; for thread `(1 0 0)`, statement `ssum[1] += ssum[1 + 1]` is executed. In traditional CPU programs, since thread `(0 0 0)` is reading data from thread `ssum[1]` while thread `(1 0 0)` is attempting to write data to `ssum[1]`, it can incur a "read&write" data race. However, in CUDA kernel functions, since thread `(0 0 0)` and thread `(1 0 0)` are located in the same thread warp without branch divergence, they essentially are executing the same instruction at the same time. Since the statement `ssum[tid] += ssum[tid + shift]` can be compiled to the following two instructions `%1 = load ssum[tid+shift]; store %1 ssum[tid]`, the "read" action is executed strictly prior to the "write" action. As a result, it turns to be a false-positive data race in CUDA kernel functions. We issued this case to its corresponding developers who further verified our finding as follows:

> *"You are correct that ssum[0] += ssum[0 + 1] and ssum[1] += ssum[1 + 1] are executed at the same time. But since we are now in a warp, all the 32 threads (tid=0..31) are synchronized. So reading data from ssum[1] and ssum[2] always happens before writing data to ssum[0] and ssum[1] for thread tid=0 and tid=1."*— kaldi

***GKLEE* and *GKLEE-SESA*.** *GKLEE* and *GKLEE-SESA* are designed to detect synchronization bugs via integrating concolic execution with SMT solvers. By launching the environmental setups, they collect "read" and "write" statements into different sets and use SMT solvers to detect synchronization bugs accordingly. From our preliminary study, *GKLEE* and *GKLEE-SESA* can detect 16 and 4 manually identified bugs, respectively, without any false positive. Interestingly, *GKLEE-SESA* incurs more timeouts and detects less bugs than *GKLEE*. The reason is that *GKLEE-SESA* leverages more static analysis techniques to reduce its dependency on the initial user-provided environmental setups while such techniques are rather time-consuming. Moreover, similar to *GPUVerify*, because they are both SMT-solver-based approaches, they are also likely to report false positives on data race (further confirmed by our later study

Table 2: Detection Results of the Identified Bugs

Project	Revision	Kernel Function	Bug Type	Bug Num	Simulee		GPUVerify		GKLEE		GKLEE-SEAS		RaceChecker	
					Effect	Time(s)	Effect	Time(s)	Effect	Time(s)	Effect	Time(s)	Effect	Time(s)
arrayfire	a7a297ba814	scan_nonfinal_kernel	DR	1	✓	1.01	✓	3.01	✓	1.6	✗	TO	✓	0.78
	a7a297ba814	scan_dim_nonfinal_kernel	DR	1	✓	1.17	✓	3.4	✓	0.89	✗	TO	✓	1.17
	0c5a38182b7	hamming_matcher	DR	1	✗	6.56	✓	23.53	✓	2.97	✗	TO	✓	0.37
	0c5a38182b7	hamming_matcher_unroll	DR	1	✓	4.01	✓	24.93	✓	2.38	✗	TO	✓	0.57
	d7abcf2358e	JacobiSVD	DR	2	✓✓	10.73	✓✓ F F F	80.5	✗✗	TO	✗✗	TO	✓✓	1.21
	c59116e3ec3	warp_reduce	DR	1	✓	1.34	✓	2.34	✓	0.69	✗	TO	✗	N/A
	a515b112076	scan_dim_kernel	DR	1	✓	2.06	✓	1.06	✓	0.34	✗	TO	✓	0.28
	1050816e422	hamming_matcher	DR	1	✓	1.38	✓	120.85	✓	27.58	✗	TO	✓	0.47
	dfbfca5fb77	select_matches	BD	1	✓	0.61	✓	5.16	✗	0.52	✗	TO	✗	N/A
	0e0c726d7d0	hamming_matcher_unroll	BD	1	✓	0.43	✗	TO	✓	0.65	✗	TO	✗	N/A
	ee4d0bd77d7	computeDescriptor	BD	1	✓	2.26	✓ F F	7.78	✓	1.84	✗	TO	✗	N/A
	0d0d7d1285a	warp_reduce	BD	1	✓	1.42	✓	2.34	✓	0.69	✗	TO	✗	N/A
	31761d27f01	computeMedian	RB	1	✓	0.47	✗	N/A	✗	N/A	✗	N/A	✗	N/A
	faefa30c3a0	harris_response	RB	1	✓	0.66	✗	N/A	✗	N/A	✗	N/A	✗	N/A
GkleeTests	10eb6373d53	device_global	DR	1	✓	0.53	✓	2.23	✓	1.56	✓	0.91	✗	N/A
	10eb6373d53	colonel	DR	1	✓	1.11	✓	2.12	✓	0.71	✓	0.65	✗	N/A
	10eb6373d53	dl@deadlock_0	BD	1	✓	1.61	✓ F	2.39	✓	0.61	✓	0.23	✗	N/A
	10eb6373d53	dl@deadlock_2	BD	1	✓	1.94	✓	2.1	✓	1.12	✓	0.41	✗	N/A
kaldi	bc13196e7fe	_add_diag_mat_mat	BD	1	✓	9.38	✓	3.41	✗	181.57	✗	TO	✗	N/A
	42352b63e62	_softmax_reduce	RB	1	✗	N/A	✗	N/A	✗	N/A	✗	TO	✗	N/A
	bb589475b10	_div_rows_vec	RB	1	✗	N/A	✗	N/A	✗	N/A	✗	TO	✗	N/A
thundersvm	febf515a826	nu_smo_solve_kernel	DR	2	✓✓	1.23	✗✗	TO	✓✓	1.93	✗✗	TO	✓✓	1.24
Total Detection Result					21 ✓, 3 ✗, 0 F		17 ✓, 7 ✗, 6 F		16 ✓, 8 ✗, 0 F		4 ✓, 20 ✗, 0 F		10 ✓, 14 ✗, 0 F	

Table 3: Detection Results of the Previously Unknown Bugs

Project	Revision	Kernel Function	Bug Type	Bug Num	Simulee		GPUVerify		GKLEE		GKLEE-SEAS		RaceChecker	
					Effect	Time(s)	Effect	Time(s)	Effect	Time(s)	Effect	Time(s)	Effect	Time(s)
cuda-cnn	c843bb2861e	g_getCost_3	RB	1	✓	1.39	✗	N/A	✗	N/A	✗	N/A	✗	N/A
cudaSift	a2e57327ddc	FindMaxCorr	RB	1	✓ F F F	0.93	✗	N/A	✗	N/A	✗	N/A	✗	N/A
	a2e57327ddc	FindMaxCorr1	RB	1	✓	1.02	✗	N/A	✗	N/A	✗	N/A	✗	N/A
	a2e57327ddc	FindMaxCorr2	RB	1	✓	0.97	✗	N/A	✗	N/A	✗	N/A	✗	N/A
	a2e57327ddc	FindMaxCorr3	RB	1	✓	1.05	✗	N/A	✗	N/A	✗	N/A	✗	N/A
cudpp	9dc7357ee81	sparseMatrixVectorFetchAndMultiply	RB	1	✓	0.7	✗	N/A	✗	N/A	✗	N/A	✗	N/A
	9dc7357ee81	sparseMatrixVectorSetFlags	RB	1	✓	0.61	✗	N/A	✗	N/A	✗	N/A	✗	N/A
	9dc7357ee81	yGather	RB	1	✓	1.13	✗	N/A	✗	N/A	✗	N/A	✗	N/A
gunrock	248a12107ef	Join	BD	1	✓	1.21	✗	TO	✗	1.84	✗	TO	✗	N/A
kaldi	5cc9731af4f	_add_diag_vec_mat	DR	2	✓✓	1.23	✓✓	1.17	✓✓	18.7	✗✗	TO	✗✗	N/A
	5cc9731af4f	_copy_low_upp	DR	1	✓	2.7	✓ F	3.21	✓ F	0.572	✓ F	0.649	✗	N/A
	5cc9731af4f	_copy_upp_low	DR	1	✓	0.79	✓ F	3.11	✓ F	0.598	✓ F	0.712	✗	N/A
	5cc9731af4f	_splice	DR	1	✓	0.73	✓	1.31	✓	23.61	✗	N/A	✗	N/A
	5cc9731af4f	_copy_from_tp	DR	3	✓✓✓	0.96	✓✓✓	1.99	✓✓✓	26.3	✗✗✗	TO	✗✗✗	N/A
	5cc9731af4f	_copy_from_mat	DR	1	✓	0.69	✓	2.11	✓	20.59	✗	TO	✗	N/A
	5cc9731af4f	_sum_reduce	DR / RB	2 / 1	✓✓ / ✓	0.14	✓✓ / ✗	2.86 / N/A	✓✓ / ✗	1.06 / N/A	✗✗ / ✗	TO / N/A	✗✗ / ✗	0.34 / N/A
thundersvm	05de37f83b6	c_smo_solve_kernel	DR	3	✓✓✓	3.11	✗✗✗	TO	✗✗✗	6.43	✗✗✗	TO	✓✓✓	1.38
Total Detection Result					24 ✓, 0 ✗, 3 F		11 ✓, 13 ✗, 2 F		11 ✓, 13 ✗, 2 F		2 ✓, 22 ✗, 2 F		3 ✓, 21 ✗, 0 F	

on detecting new bugs) and fail to detect any redundant-barrier-function bugs because their detecting mechanism is not designed for such bugs. We next discuss the failure cases for the better *GKLEE* because *GKLEE-SESA* timed out on most cases: *GKLEE* fails to detect barrier-divergence bugs in kernel functions `select_matches` and `_add_diag_mat_mat` because *GKLEE* models the kernel functions over two parametric threads and tends to ignore important execution paths for detecting barrier-divergence bugs.

RaceChecker. *RaceChecker* is a device-dependent tool designed only for detecting CUDA data-race bugs. Specifically, it can only detect the data-race bugs incurred in shared memory. In particular, *RaceChecker* can detect all the 10 data-race bugs that are relevant to shared memory, but fails to detect other synchronization bugs including the data-race bugs incurred in global memory from the manually identified bugs for our preliminary study. Note that since *RaceChecker* does not involve SMT solver, it does not report any false-positive data-race bug as the other SMT-solver-based approaches.

4.3.2 Further Study on Previously Unknown Bugs. After our preliminary study, we further apply *Simulee* and all the compared techniques to detect previously unknown synchronization bugs for all the 7 projects with the results demonstrated in Table 3, which follows the same format as Table 2. From the table, we can observe that *Simulee* detects 24 bugs and reports 3 false positives in total, where all the false positives are redundant-barrier-function bugs in kernel function `FindMaxCorr`. Note that *Simulee* reports such false positives because it is possible that "Automatic Input Generation" may miss some potential error-inducing inputs that can trigger synchronization bugs such that a barrier function can be reported as a redundant barrier function. Meanwhile, even the most effective existing technique, i.e., *GPUVerify* can only detect 11 previously unknown bugs with 2 false positives.

We have also reported all the detected bugs to the corresponding developers and show their feedback statistics in Table 4. To date, they have confirmed 10 bugs in total (TT), including 1 data-race bugs (DR), 1 barrier-divergence bug (BD), and 8 redundant-barrier-function bugs (RB). To be specific, the developers of cudpp and CudaSift responded as follows:

> *"I think you're right... There are considerably faster ways to do matrix multiply calls..."* — cudpp

Table 4: Developer Feedback Statistics

Projects	Detected				Confirmed				Under Discussion				Nonresponse			
	TT	DR	RB	BD	TT	DR	RB	BD	TT	DR	RB	BD	TT	DR	RB	BD
kaldi	12	11	1	0	2	1	1	0	6	6	0	0	4	4	0	0
thundersvm	3	3	0	0	0	0	0	0	3	3	0	0	0	0	0	0
CudaSift	4	0	4	0	4	0	4	0	0	0	0	0	0	0	0	0
CUDA-CNN	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0
cudpp	3	0	3	0	3	0	3	0	0	0	0	0	0	0	0	0
gunrock	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0
Total	24	14	9	1	10	1	8	1	9	9	0	0	5	4	1	0

```
const int32_cuda tid = threadIdx.y * blockDim.x
                                   + threadIdx.x;
...
if (tid < warpSize) {
#   pragma unroll
  for (int shift = warpSize; shift > 0; shift >>= 1) {
    ssum[tid] += ssum[tid + shift];
  }
...
}
```

Figure 7: An Example of False Race

> *"Yes, there is a bit of cleaning up to do there. Sometimes when I detect oddities in the output, I add an unnecessary synchronization just in case. In fact those things should be all run on the same thread, since it cannot be parallelized anyway. Thank you for pointing it out."* — CudaSift

Since barrier divergence is an undefined behavior, it may not hang on every situation. The developers of gunrock responded as follows and further fixed the bug in a later commit [25]:

> *"I do see what @Stefanlyy / @eagleShanf mean for the divergence issue, and surprise the code didn't hang."*

In addition, 9 data-race bugs are still being actively discussed by developers. We label such bugs "under discussion" as stated in Table 4.

In summary, it can be observed from Section 4.3.1 and Section 4.3.2 that *Simulee* can correctly detect most of the synchronization bugs while the other approaches are all limited in their detection scopes, failing to detect certain bugs that they are designed for, or triggering additional false positives. We now summarize the reasons why *Simulee* can outperform the other approaches: *Simulee* applies lightweight evolutionary test generation (guided by effective memory-access modeling) and dynamic runtime monitoring in tandem for powerful CUDA synchronization bug detection. On the other hand, the other approaches are usually bounded by heavyweight techniques, such as constraint solvers, which prevent the techniques from exploring all the possible cases.

5 THREATS TO VALIDITY

The threats to external validity mainly lie in the subjects and faults used in our benchmark. Though the projects of our benchmark suite may not represent the overall project distributions, they shall be selected in order to possibly maximize the overall features of the CUDA projects. In this way, our benchmark is derived based on real-world programs from GitHub, i.e., we select seven popular CUDA-related projects with a total of 17928 commits and 1.36 million LOC.

The threats to internal validity mainly lie in the potential bugs in our implementation due to the complicated mechanism of *Simulee*. To reduce the threats, three graduate students, closely mentored by three SE/Systems supervisors, have been carefully working for over one year. We manually reviewed all our implementation code and also included corresponding tests for verifying our implementation. In addition, the effectiveness of "Automatic Input Generation" can impact on the performance synchronization bug detection. It is possible that the "Automatic Input Generation" component may miss certain inputs that can trigger synchronization bugs such that *Simulee* would miss detecting the bug or report a false positive. To reduce this threat, we set a large number of suitable parameters for the evolutionary algorithm adopted in "Automatic Input Generation" to reduce the probability of missing error-inducing inputs.

To threats to construct validity mainly lie in the metrics used in this work. To reduce the threats, we measure the number of both previously known and the identified bugs detected by the studied techniques as well as their false-positive rate and corresponding time cost.

6 RELATED WORK

As our work investigates the automatic bug detection techniques for CUDA programs, the related work includes the following two parts: empirical studies on CUDA programs and techniques of CUDA bug detection. Moreover, since *Simulee* essentially is a search-based bug-detection technique, we also discuss such relevant work.

Empirical studies for CUDA programs There are several existing work that study bugs and other features on CUDA programs. For instance, Yang et al. [43] delivered the empirical study on the features of the performance bugs on CUDA programs, Burtscher et al. [10] studied the control-flow irregularity and memory-access irregularity and found that both irregularities are mutually dependent and exist in most of kernels. Che et al.[12] examined the effectiveness of CUDA to express with different sets of performance characteristics. Some researchers are keen on the comparisons between CUDA and OpenCL. For instance, Demidov et al. [17] compared some C++ programs running on top of CUDA and OpenCL and found that they work equally well for problems of large size. Du et al. [19], on the other side, studied the discrepancies in the OpenCL and CUDA compilers' optimization that affect the associated GPU computing performance. In our previous work, we also conducted empirical studies to explore CUDA program features. For instance, we investigated the features and the distribution of multiple CUDA program bug types based on a collected GitHub dataset [41]. Moreover, we developed an approach that can automatically repair CUDA synchronization bugs via program transformation and validated its performance via an experimental study based on real-world benchmarks [40].

CUDA bug detection Unlike traditional program bugs which can be deterministically tested [39, 46] and debugged [23, 32, 47], CUDA synchronization bugs, especially data race and barrier divergence can often result in undefined behaviors. To detect such bugs, there are typically two types of approaches—compiler-based approaches and static analysis (SMT solver)-based approaches. In particular, compiler-based approaches, e.g., [37][20][6], usually link the detectors to the applications in the compiling stage and detect the bugs in the runtime process of GPU programs. They are limited by not being "fully automatic" because developers have to manually provide test cases. Moreover, given inferior inputs, the synchronization bugs might not be triggered and detected because such bugs could only occur under limited conditions. They can also be expensive since such runtime detection demands compiling process and GPU computing environment. To the best of our knowledge, these approaches fail to detect barrier divergence and redundant barrier function because of their detection mechanisms. On the other hand, various automatic synchronization bug detection approaches, e.g., [30][9], depend on static analysis and SMT solver [16] which could lead to poor runtime performance when handling complicated GPU programs. Besides, such approaches tend to report false positives or false negatives because it lacks runtime information. Although developers do not have to provide whole test inputs for them, they still need to provide heuristic settings in order to avoid path explosions, e.g., dimension settings for *GPUVerify*, main functions and initial environments of kernel functions for *GKLEE*.

Compared with these approaches, *Simulee* can automate the detection process to achieve superior detection performance by enabling the "Automatic Input Generation" component and the "Memory-based Synchronization Bug Detection" component.

Search-based Software Engineering. The optimization approaches such as Evolutionary Programming are widely used to solve software engineering problem by modeling them into optimization problems [26]. Yu et al. [45] proposed a metric, *PSet* constraint to detect CPU-based synchronization bugs. Harman et al. [27] applied evolving pareto front approximation to refactor software systems. Hierons et al. [28] used Many-Objective Evolutionary Optimisation to optimize the process of software product selection. McMinn et al. [33] evolved coverage criteria to improve the performance of bug detection. Ouni et al. [36] modeled the process of refactoring into a multiple-objective search-based problem for generating refactor patches.

7 CONCLUSIONS

In this paper we develop a fully automated approach, namely *Simulee*, that can successfully detect CUDA synchronization bugs efficiently based on accurate memory-access modeling. More specifically, *Simulee* consists of two different components: the "Automatic Input Generation" component that applies Evolutionary Computation for automatically generating bug-inducing test inputs, and the "Bug Detection via *Memory-Access Model*" component that builds an accurate memory model for deriving the underlying CUDA synchronization bugs. To evaluate the efficacy of *Simulee*, we construct a benchmark from real-world CUDA-related projects. Our evaluation results suggest that *Simulee* can detect most of the manually identified synchronization bugs out of the studied projects, and successfully detect 24 previously unknown bugs which have never been reported/detected before. In addition, *Simulee* can achieve better effectiveness and efficiency than multiple state-of-the-art approaches.

8 ACKNOWLEDGEMENT

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Shenzhen Peacock Plan (Grant No. KQTD2016112514355531), and Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. JCYJ20170817110848086). This work is also partially supported by National Science Foundation under Grant Nos. CCF-1763906 and CCF-1942430, as well as Ant Financial Services Group.

REFERENCES

[1] 2014. *Professional CUDA C Programming* (1st ed.). Wrox Press Ltd., Birmingham, UK, UK.

[2] 2019. Cauchy distribution. https://en.wikipedia.org/wiki/Cauchy_distribution.

[3] 2019. CUDA program introduction. https://en.wikipedia.org/wiki/CUDAr.

[4] 2019. GPGPU introduction. https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units.

[5] 2019. Normal distribution. https://en.wikipedia.org/wiki/Normal_distribution.

[6] 2019. Racecheck Tool. https://docs.nvidia.com/cuda/cuda-memcheck/index.html#racecheck-tool.

[7] 2019. The Simulee project. https://github.com/Lebronmydx/Simulee.

[8] arrayfire. 2019. ArrayFire. https://github.com/arrayfire/arrayfire.

[9] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: A Verifier for GPU Kernels. *SIGPLAN Not.* 47, 10 (Oct. 2012), 113–132. https://doi.org/10.1145/2398857.2384625

[10] M. Burtscher, R. Nasre, and K. Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. 141–151. https://doi.org/10.1109/IISWC.2012.6402918

[11] Celebrandil. [n.d.]. SIFT features with CUDA. https://github.com/Celebrandil/CudaSift.

[12] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. 2008. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel and Distrib. Comput.* 68, 10 (2008), 1370 – 1380. https://doi.org/10.1016/j.jpdc.2008.05.014 General-Purpose Processing using Graphics Processing Units.

[13] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. *SIGPLAN Not.* 37, 5 (May 2002), 258–269. https://doi.org/10.1145/543552.512560

[14] Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. 2013. Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 270–289.

[15] cudpp. [n.d.]. cudpp. https://github.com/cudpp/cudpp.

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. (2008), 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[17] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling. 2013. Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries. *SIAM Journal on Scientific Computing* 35, 5 (2013), C453–C472. https://doi.org/10.1137/120903683 arXiv:https://doi.org/10.1137/120903683

[18] A. Dinning and E. Schonberg. 1990. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming (PPOPP '90)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/99163.99165

[19] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.* 38, 8 (2012), 391 – 407. https://doi.org/10.1016/j.parco.2011.10.002 APPLICATION ACCELERATORS IN HPC.

[20] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. BARRACUDA: Binary-level Analysis of Runtime RAces in CUDA Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 126–140. https://doi.org/10.1145/3062341.3062342

[21] Lawrence J. Fogel. 1999. *Intelligence Through Simulated Evolution: Forty Years of Evolutionary Programming.* John Wiley & Sons, Inc., New York, NY, USA.

[22] Geof23. 2019. GkleeTests. https://github.com/Geof23/GkleeTests.

[23] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30.

[24] gunrock. [n.d.]. Gunrock. https://github.com/gunrock/gunrock.

[25] gunrock. [n.d.]. Gunrock Commit. https://github.com/gunrock/gunrock/commit/16294438272cdae8bee4d2db0f4ff65e28bf4331.

[26] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based Software Engineering: Trends, Techniques and Applications. *ACM Comput. Surv.* 45, 1, Article 11 (Dec. 2012), 61 pages. https://doi.org/10.1145/2379776.2379787

[27] Mark Harman and Laurence Tratt. 2007. Pareto Optimal Search Based Refactoring at the Design Level. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07)*. ACM, New York, NY, USA, 1106–1113. https://doi.org/10.1145/1276958.1277176

[28] Robert M. Hierons, Miqing Li, Xiaohui Liu, Sergio Segura, and Wei Zheng. 2016. SIP: Optimal Product Selection from Feature Models Using Many-Objective

Evolutionary Optimization. *ACM Trans. Softw. Eng. Methodol.* 25, 2, Article 17 (April 2016), 39 pages. https://doi.org/10.1145/2897760

[29] kaldi asr. [n.d.]. Kaldi. https://github.com/kaldi-asr/kaldi.

[30] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic Verification and Test Generation for GPUs. *SIGPLAN Not.* 47, 8 (Feb. 2012), 215–224. https://doi.org/10.1145/2370036.2145844

[31] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2014. Practical Symbolic Race Checking of GPU Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 179–190. https://doi.org/10.1109/SC.2014.20

[32] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. 169–180. https://doi.org/10.1145/3293882.3330574

[33] Phil McMinn, Mark Harman, Gordon Fraser, and Gregory M. Kapfhammer. 2016. Automated Search for Good Coverage Criteria: Moving from Code Coverage to Fault Coverage Through Search-based Software Engineering. In *Proceedings of the 9th International Workshop on Search-Based Software Testing (SBST '16)*. ACM, New York, NY, USA, 43–44. https://doi.org/10.1145/2897010.2897013

[34] Robert H. B. Netzer and Barton P. Miller. 1991. Improving the Accuracy of Data Race Detection. *SIGPLAN Not.* 26, 7 (April 1991), 133–144. https://doi.org/10.1145/109626.109640

[35] Nvidia. 2019. Optimizing Parallel Reduction in CUDA. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[36] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study. *ACM Trans. Softw. Eng. Methodol.* 25, 3, Article 23 (June 2016), 53 pages. https://doi.org/10.1145/2932631

[37] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: A Dynamic CUDA Race Detector. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 390–403. https://doi.org/10.1145/3192366.3192368

[38] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[39] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-aware static regression test selection. *PACMPL* 3, OOPSLA (2019), 187:1–187:29. https://doi.org/10.1145/3360613

[40] M. Wu, L. Zhang, C. Liu, S. H. Tan, and Y. Zhang. 2019. Automating CUDA Synchronization via Program Transformation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 748–759. https://doi.org/10.1109/ASE.2019.00075

[41] Mingyuan Wu, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2019. Characterizing and Detecting CUDA Program Bugs. *CoRR* abs/1905.01833 (2019). arXiv:1905.01833 http://arxiv.org/abs/1905.01833

[42] Xtra-Computing. [n.d.]. THUNDERSVM. https://github.com/Xtra-Computing/thundersvm.

[43] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. 2012. Fixing Performance Bugs: An Empirical Study of Open-Source GPGPU Programs. In *2012 41st International Conference on Parallel Processing*. 329–339. https://doi.org/10.1109/ICPP.2012.30

[44] Xin Yao, Yong Liu, and Guangming Lin. 1999. Evolutionary Programming Made Faster. *Trans. Evol. Comp* 3, 2 (July 1999), 82–102. https://doi.org/10.1109/4235.771163

[45] Jie Yu and Satish Narayanasamy. 2009. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. Association for Computing Machinery, New York, NY, USA, 325–336. https://doi.org/10.1145/1555754.1555796

[46] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. 2010. Test generation via dynamic symbolic execution for mutation testing. In *2010 IEEE International Conference on Software Maintenance*. 1–10.

[47] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid. 2019. An Empirical Study of Boosting Spectrum-based Fault Localization via PageRank. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2911283

[48] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 132–142. https://doi.org/10.1145/3238147.3238187

[49] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

[50] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: A low-overhead mechanism for detecting data races in GPU programs. In *PPoPP*.

[51] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. 2014. GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. *IEEE Transactions on Parallel and Distributed Systems* 25, 1 (Jan 2014), 104–115. https://doi.org/10.1109/TPDS.2013.44

[52] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2020. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020, Seoul, Korea, May 23 - 29, 2020*.

[53] Husheng Zhou, Wei Li, Yuankun Zhu, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2018. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. *CoRR* abs/1812.10812 (2018). arXiv:1812.10812 http://arxiv.org/abs/1812.10812

[54] zhxfl. 2019. CUDA-CNN. https://github.com/zhxfl/CUDA-CNN.