Automating CUDA Synchronization via Program Transformation

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

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

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

Shin Hwei Tan
Department of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China
tansh3@sustech.edu.cn

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

***Abstract*—While CUDA has been the most popular parallel computing platform and programming model for general-purpose GPU computing, CUDA synchronization undergoes significant challenges for GPU programmers due to its intricate parallel computing mechanism and coding practices. In this paper, we propose *AuCS*, the first general framework to automate synchronization for CUDA kernel functions. *AuCS* transforms the original LLVM-level CUDA program control flow graph in a semantic-preserving manner for exploring the possible barrier function locations. Accordingly, *AuCS* develops mechanisms to correctly place barrier functions for automating synchronization in multiple erroneous (challenging-to-be-detected) synchronization scenarios, including data race, barrier divergence, and redundant barrier functions. To evaluate the effectiveness and efficiency of *AuCS*, we conduct an extensive set of experiments and the results demonstrate that *AuCS* can automate 20 out of 24 erroneous synchronization scenarios.**

***Index Terms*—CUDA, program repair, synchronization automation, program transformation**

I. Introduction

CUDA [1] has recently become a dominating parallel computing platform and programming model for general-purpose GPU (GPGPU) computing [2], due to its advantages in (1) simplifying I/O streams to memories and dividing computations into sub-computations by parallelizing programs in terms of grids and blocks, and (2) enabling more flexible cache management that speeds up the floating point computation of CPUs. CUDA is thus considered rather powerful and widely adopted in deep-neural-network-related applications for efficiently processing relevant matrix computations.

Albeit its advantages in GPU computing, CUDA programming undergoes significant challenges for GPU programmers due to its specific parallel computing mechanism and coding practices [3] [4] [5] [6]. Since CUDA-based GPU programs enable synchronization which significantly differs from CPU programs by using barriers rather than locks [7] [8] and applying happens-before relations [9] [10], GPU programmers are expected to be competent domain experts for delivering correct program outputs with limited benefits from their knowledge of traditional CPU programs. However, the synchronization management skills of GPU programmers can be seriously challenged. In particular, since massive parallelism in CUDA-based GPU computing can be invoked by ballooning excessive thread interleavings, any two from thousands of threads accessing the same memory cell might trigger a data race and lead to incorrect computation results which are somewhat hard to be discovered by GPU programmers [4] [6]. Moreover, programmers' unawareness of using third-party programs/libraries of kernel functions can be another major reason to cause program execution failures. For instance, a data race can also be caused when programmers mistakenly delegate synchronization to the third-party programs or libraries which are not designed for such purpose [11]. Therefore, it is essential to assist GPU programmers by automating synchronization of CUDA programs for effectively developing GPU programs.

In this paper, we propose *AuCS* which, to the best of our knowledge, is the first general framework to automate LLVM-level synchronization for CUDA programs in multiple erroneous synchronization scenarios. To be specific, we automate CUDA synchronization in LLVM bitcode instead of source code because (1) integrated as part of compiler optimization [12], LLVM-level synchronization can be effective in concealing programming details from GPU programmers such that they can focus on delivering high-level program functionalities, and (2) automating source code level synchronization via patching can possibly deteriorate the source code with inferior readability and maintainability.

We first specify the erroneous CUDA synchronization scenarios: (1) the data race scenario that occurs when programmers fail to implement synchronization inside kernel functions; (2) the barrier divergence scenario that occurs when program-

* corresponding author

mers implement incorrect synchronization to cause barrier divergence inside kernel functions; and (3) the redundant barrier function scenario that occurs when programmers implement redundant synchronization inside kernel functions [13], [14]. Next, *AuCS* transforms the scenarios to be their corresponding automatic bug repair problems and solves them in their LLVM-bitcode level respectively. In particular, *AuCS* leverages a LLVM-bitcode tool that automatically detects CUDA synchronization bugs. Based on the detected CUDA bugs, *AuCS* applies a LLVM-level *program transformation rule* to transform the original LLVM-bitcode control flow graph (CFG) while preserving the original semantics. Our *program transformation* is able to expose the possible barrier function locations in the original program or create the potential barrier function location when no such location exists in the original program. As a result, *AuCS* develops a set of mechanisms to automate synchronization under multiple synchronization scenarios, i.e., (1) correctly placing barrier functions for eliminating data race and barrier divergence and (2) removing unnecessary barrier functions after detecting the synchronization problem. Eventually, *AuCS* can automatically enable correct synchronization in LLVM bitcode of CUDA kernel functions to alleviate the concerns and cost from GPU programmers on implementing correct synchronization in source code.

To evaluate the effectiveness and efficiency of *AuCS* on automating synchronization for CUDA programs, we conducted a set of experiments based on a real-world benchmark which consists of four *GitHub* projects with 24 erroneous synchronization scenarios. Our experimental results suggest that *AuCS* can effectively automate synchronization for CUDA kernel functions by fixing 13 data race bugs, 5 barrier divergence bugs, and 2 redundant barrier divergence bugs in their LLVM bitcode in relatively short time.

In summary, our paper makes the following contributions:

- To the best of our knowledge, we develop the first general framework, namely *AuCS*, that automates synchronization for CUDA kernel functions by correctly placing barrier functions in their corresponding LLVM bitcode.
- We introduce a set of program transformation rules that automatically generate synchronization for CUDA programs at the LLVM-bitcode level. Our transformation rules aim to preserve the semantics of the modified programs.
- We evaluate *AuCS* under multiple experimental setups. The results suggest that *AuCS* is able to automate synchronization under most of the erroneous synchronization scenarios in the studied projects under limited time cost.

The rest of the paper is organized as follows. Section II introduces the background of this paper including CUDA overview, parallel computing mechanism, synchronization bug types, and LLVM bitcode. Section III introduces a motivating example to illustrate the challenges on automating synchronization for CUDA programs. Section IV demonstrates *AuCS* including the proof for the semantic-preserving property of its *program transformation* and the corresponding mechanisms of automating synchronization under multiple scenarios. Section V presents the evaluation on the effectiveness and efficiency of *AuCS*. Sections VI to VIII present the related work, threats to validity, and conclusions of the paper, respectively.

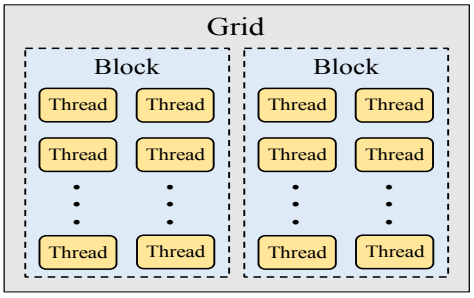


Fig. 1: CUDA Hierarchy

II. Background

In this section, we give an overview on CUDA, the CUDA parallel computing mechanism, typical CUDA synchronization bugs, and the LLVM-level CUDA Synchronization Bug Detection.

A. CUDA Overview and Parallel Computing Mechanism

CUDA provides a runtime library and an extended version of C/C++ for GPU programmers such that they can use GPU hardware for general-purpose computing. CUDA operates on a heterogeneous programming model where it involves both the CPU and GPU. In CUDA, the *host* refers to the CPU and its memory, while the *device* indicates the GPU and its memory [15].The device programs need to be allocated with resources from host programs prior to execution. Eventually, the allocated resources, e.g., global memory, need to be retrieved after CUDA program execution. A typical CUDA program contains three runtime stages: host resource preparation, kernel function execution, and host resource retrieval. In particular, a *kernel function* refers to the part of CUDA programs that is invoked during device execution and is the focus of this paper.

Thread is the basic execution unit in kernel functions. Specifically, in the *physical* level, a *warp* is a set of 32 threads, all of which are expected to execute the same instruction at any time, except when incurring branch divergence, while in the *logic* level, CUDA imposes a hierarchy where a *block* contains one or more threads, and a *grid* contains one or more blocks.

Kernel functions are executed by setting dimensions of grids and blocks. These functions divide computation into sub-computations and dispatch each sub-computation to different threads accordingly. Eventually, the results of sub-computations can be merged as the final result of the overall computation through applying algorithms such as reduction. Figure 1 shows the hierarchy of the parallel computing mechanism of CUDA kernel functions.

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

Fig. 2: An Example of Data Race

```
   ....
     s_dist[sid] = dist;
     s_idx[sid]  = s_idx[sid + i];
     // removing the barrier function next line
--   __syncthreads();
   }
   //fix by moving the barrier out.
++ __syncthreads();
 }
```

Fig. 3: An Example of Barrier Divergence

B. CPU Synchronization vs. CUDA Synchronization

Traditional CPU programs, e.g., Java programs, use a lock-based mechanism to synchronize different threads. In particular, instead of accessing memory with other threads as a group at the same time, a thread accesses a memory cell shared with other threads by acquiring a lock from the memory cell. If the lock is free, the thread obtains the lock, accesses the memory cell, and continues executing the remain statements while other threads have to enter pending state until the lock is released. Otherwise, the thread enters pending state.

Different from CPU synchronization, CUDA synchronization applies barriers to synchronize threads where all the threads in one block must wait before any can proceed. In particular, a barrier is represented as a barrier function `__syncthreads()` in CUDA kernel functions. When a thread reaches a barrier function, it is expected to proceed to next statement if and only if all the threads from the same block have reached the same barrier function.

C. CUDA Synchronization Bug Patterns

According to [16] [13], there are three major synchronization bug types in CUDA kernel functions: data race, barrier divergence, and redundant barrier function.

Data Race. Data race refers to that the visit order of "read&write" actions or "write&write" actions from two or more threads cannot be determined in CUDA programs. Figure 2 presents an example with bug-fixing *Revision no.* "`febf515a82`" in the file "`smo-kernel.cu`" of one highly-rated Github project "`thundersvm`" [17]. We can observe that the "`if`" statement writes to the memory of "`f_val2reduce`", meanwhile the function "`get_block_min`" writes to the same memory inside the device. This causes a "write&write" bug and could be fixed by inserting "`__syncthreads`".

```
1  int tid = threadIdx.x;
2  s_median[tid] = FLT_MAX;
3  s_idx[tid] = 0;
4 -- __syncthreads();
5
6  if (i < iterations) {...
7      s_idx[tid] = i;
8      s_median[tid] = m;
9  }....
```

Fig. 4: An Example of Redundant Barrier Function

Barrier Divergence. A barrier divergence occurs when more than one threads belonging to the same block complete their tasks and leave the barrier while some other threads in the same block have not reached the barrier yet. A sample barrier divergence can be found in the bug-fixing *Revision no.* "`0ed6cccc5ff`" in the file "`nearest_neighbour.hpp`" from the project "`arrayfire`" presented in Figure 3, where it can be observed that all the threads in the same block are ensured to reach the same barrier in every execution of the kernel function by moving the statement of "`__syncthreads()`" outside the given branch.

Redundant Barrier Function. A barrier function is defined to be redundant when no data race is triggered after deleting it. A redundant barrier function can result in the inferior program performance in terms of time and memory usage. For instance, a sample redundant barrier function can be found in the bug-fixing *Revision no.* "`31761d27f01`" in the file "`kernel/homography.hpp`" from the project "`arrayfire`" [18] presented in Figure 4. We can observe that the associated block is one-dimensional since from Line 1, the value of "`tid`" is assigned only from "`threadIdx.x`". Moreover, the "`tid`"s are identical among different threads from the same block. Therefore, only one thread is allowed to access "`s_median[tid]`" and "`s_idx[tid]`", leading to a redundant barrier function in Line 4 since no race can be triggered in "`s_median`" or "`s_idx`" after deleting the barrier function.

D. LLVM-level CUDA Synchronization Bug Detection

Low level virtual machine (LLVM) is a compiler framework for program analysis and transformation of source code, where LLVM bitcode is a low-level code representation in Static Single Assignment (SSA) form [19]. In particular, LLVM bitcode includes the following novel features: (1) language-independent type system, (2) type-conversion and low-level address arithmetic instructions, and (3) low-level exception handling instructions.

Simulee [13] is a LLVM-level CUDA synchronization bug detection tool. In particular, it first uses Evolutionary Programming [20] to automatically generate the input for kernel functions that can trigger CUDA synchronization bugs. Next, by simulating kernel function execution with the bug-inducted input, *Simulee* detects synchronization bugs and the associated locations in the original program. Moreover, there are other synchronization bug detection approaches for CUDA

programs, e.g., GPUVerify [21], CIVL [22] and ESBMC-GPU [23], which are designed for source-code-level other than LLVM-bitcode-level synchronization bug detection.

In this paper, we use *Simulee* to detect CUDA synchronization bugs for automating CUDA synchronization because (1) it can detect multiple bug types including data race, redundant barrier function, and barrier divergence automatically; and (2) it can simulate runtime CUDA programs without incurring much overhead for extra processing (e.g., searching), which makes it more efficient than the static/dynamic-analysis-based approaches that usually demand large search space [13].

III. Motivating example

In this section, we use a sample code snippet to illustrate why automating synchronization is beneficial and challenging for developing CUDA kernel functions. In particular, the sample code snippet is chosen from GkleeTest [24] and presented in Figure 5 while its corresponding LLVM bitcode is presented in Figure 6 and its control flow graph (CFG) is presented in Figure 7.

Assuming the grid dimension is [1, 1, 1] and the block dimension is [5, 1, 1], it can be derived that the code snippet in Figure 5 introduces a data race bug between lines 8 and 10 when executing the kernel function. Specifically, when `num_elements` is set to 5, thread (0 0 0)(1 0 0) writes data to `input_array[1]` while thread (0 0 0)(0 0 0) reads data from `input_array[1]`, and thread (0 0 0)(3 0 0) writes data to `input_array[3]` while thread (0 0 0)(2 0 0) read data from `input_array[3]`. Correspondingly in its LLVM bitcode, such race takes place between *Label* 15 and *Label* 21 in Figure 6.

Data race in CUDA programs can be fixed by adding barrier functions. For instance, in Figure 5, since the data race takes place in different branches, a barrier function should be added into one of the branches, e.g., either *Label* 15 or *Label* 21 in Figure 6. However, it would lead to barrier divergence. To illustrate, in Figure 6, by adding a barrier function in *Label* 15, the thread that executes *Label* 21 would never reach that barrier function, and vice versa.

To conclude, a complete automatic synchronization mechanism for CUDA kernel functions can be challenging because it should not only automatically detect and fix the existing synchronization bugs in the original CUDA kernel functions, i.e., data race, barrier divergence, and redundant barrier functions, but also avoid potential barrier divergence caused by adding barrier functions for fixing data race. Hence, we formulate the automatic synchronization problem for CUDA programs as a problem of *identifying the correct locations for placing barrier functions*. In this example, a barrier function is expected to be added in the *Basic Block* between *Label* 15 and *Label* 21, if there is any, to fix the data race without causing barrier divergence for automating synchronization for CUDA kernel functions. However, since no such *Basic Block* exists, fixing this data race remains challenging.

IV. Approach

In this section, we propose *AuCS*, a general framework that automatically synchronizes CUDA kernel functions. Since automating synchronization for CUDA kernel functions is essentially locating barrier functions properly, how to locate barrier functions properly is the key process. In *AuCS*, we first propose a *program transformation rule* to transform LLVM-bitcode CUDA programs (LLVM_{cuda}) for identifying barrier function locations and provide proofs for ensuring its semantic-preserving property. Next, we demonstrate how *AuCS* leverages our transformed LLVM_{cuda} to automate synchronization for CUDA kernel functions.

A. Program Transformation

Based on the CFG concepts, it can be derived that a barrier function should only be placed in a proper basic block of the program for correct execution without incurring synchronization bugs. Specifically, correctly placing barrier function is equivalent to detecting whether there exists such basic block and, if not, whether it is possible to create such basic block. Moreover, it is essential to preserve the original semantics after such program simplification. For instance, in Figure 7, with a semantic-preserving program structure simplification approach, we can generate a basic block between *Label* 15 and *Label* 21 by changing the original CFG for placing a barrier function to fix the data race without changing the original program semantics.

In the following, we propose a semantic-preserving *program transformation* approach for correctly placing barrier functions. Specifically, we first list a set of definitions for constructing LLVM_{cuda} CFGs. Next, based on the definitions, we propose a set of *program transformation* rules. At last, we prove that such *program transformation* rules are semantic-preserving.

1) Definition:

- *Label* refers to LLVM-bitcode label which is a set with multiple statements of LLVM-bitcode programs corresponding to CUDA kernel functions. Each statement belongs to a *Label*. Different *Labels* are connected by "br" instructions, as presented in Figure 9. In particular, *Label* is the fundamental component for CFG which contains multiple LLVM instructions in LLVM bitcode such that the original CUDA program semantics can be maintained in LLVM bitcode.
- *Stable Label* is a *Label* which does not contain any "write" instruction.
- *Branch Graph* is a directed acyclic graph that represents a LLVM bitcode program without *Loop Edges*. Its nodes and edges are the same as in a LLVM-bitcode CFG, except for *Loop Edges*.
- *Execution Path* refers to a single thread's *Label* sequence in a complete execution of a CUDA kernel function. Note that the intersection of two *Execution Path*s is a set of *Labels* belonging to both *Execution Path*s.
- *Loop Edge* refers to a transition relation between two *Labels*. Suppose there is an *Execution Path* $\rho = [\alpha, ..., \beta, \alpha]$

```
__global__ void device_global(unsigned int *input_array,
                int num_elements) {
  int my_index = blockIdx.x * blockDim.x + threadIdx.x;
   // stop out of bounds access
  if (my_index < num_elements) {

    if (my_index%2 == 1) {
      input_array[my_index] = my_index;
     } else {
      input_array[my_index] = input_array[my_index+1];
    }
   }
}
```

Fig. 5: C++ version for GkleeTest Example

```
define void @_Z13device_globalPji(i32* %input_array,
                i32 %num_elements) {
  %1 = alloca i32*, align 8 ...
  br i1 %10, label %11, label %33
 ; <label>:11 ...
  br i1 %14, label %15, label %21
 ; <label>:15 ...
  br label %32
 ; <label>:21 ...
  br label %32
 ; <label>:32
  br label %33
 ; <label>:33
 ret void
}
```

Fig. 6: LLVM version for GkleeTest Example

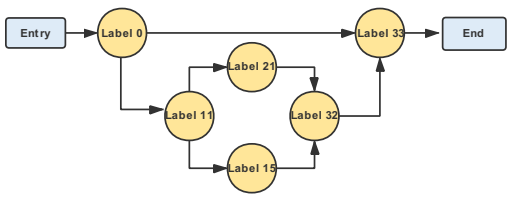


Fig. 7: Topological Structure of LLVM

where the first α is executed before than the first β. The transition from β to α is defined as a *Loop Edge*.

- *Basic Block* is a *Label* that intersects all the possible *Execution Path*s of a CUDA kernel function.
- *Program State* is a key-value dictionary structure that records all the states of a CUDA kernel function, where the keys refer to the variables' names and the values refer to their corresponding runtime values.
- *Entry Condition* is a boolean expression for a single *Label*. If an *Execution Path* satisfies *Entry Condition*, it would contain its corresponding *Label*. The value of *Entry Condition* is computed by one or more variables in *Program State*.
- *Branch-Independent* is a relation between two different *Labels*. Suppose that we have one *Label* named α, the other *Label* named β. If there is no path from α to β and no path from β to α in their *Branch Graph*, then α and β are defined to be *Branch-Independent*.
- *Semantic-Independent*
  Suppose that there are two different *Execution Path*s named ρ_1 and ρ_2, if $\rho_1 \cap \rho_2 = \rho_2$ and their difference set $\rho_1 - \rho_2$ does not contain any "write" instruction, then ρ_1 and ρ_2 are defined to be *Semantic-Independent*. For instance, assume that $\rho_1 = [a, b, c, d, e]$ and $\rho_2 = [a, c, e]$. It can be derived that $\rho_1 - \rho_2 = [b, d]$. Assuming that b and d do not contain any "write" instruction, ρ_1 and ρ_2 are *Semantic-Independent*.

We adopt the small-step operational semantics for LLVM$_{cuda}$ from GKLEE [5]. Figure 8 presents an excerpt of our modification on the LLVM$_{cuda}$ syntax which are *Current Label* and *Entry Condition*. Figure 9 shows the corresponding

$$
\begin{aligned}
\textit{Program State} &:= \Sigma(\textit{var} \mapsto \textit{value}) \\
\textit{Current Label} &:= \mathcal{P}(\textit{Label}) \\
\textit{Entry Condition} &:= \mathcal{C}
\end{aligned}
$$

Fig. 8: Syntax for modified LLVM$_{cuda}$

$$
\frac{\textit{stmt} = \textbf{br}\ \mathcal{P}_i}{(\Sigma, \mathcal{P}) \rightarrow_t (\Sigma, \mathcal{P}(\textit{Label}) \mapsto \mathcal{P}_i)} \quad (1)
$$

$$
\frac{\textit{stmt} = \textbf{br}\ \mathcal{C}_i\ \mathcal{P}_i\ \mathcal{P}_j \qquad \Sigma \vdash C_i}{(\Sigma, \mathcal{P}) \rightarrow_t (\Sigma, \mathcal{P}(\textit{Label}) \mapsto \mathcal{P}_i)} \quad (2)
$$

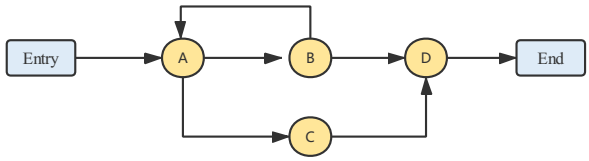
$$
\frac{\textit{stmt} = \textbf{br}\ \mathcal{C}_i\ \mathcal{P}_i\ \mathcal{P}_j \qquad \Sigma \vdash \neg C_i}{(\Sigma, \mathcal{P}) \rightarrow_t (\Sigma, \mathcal{P}(\textit{Label}) \mapsto \mathcal{P}_j)} \quad (3)
$$

Fig. 9: LLVM$_{cuda}$ Transition Rules For *Label*

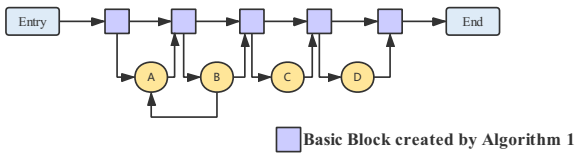
operational semantics for the LLVM$_{cuda}$ transition rules. In particular, *Current Label* refers to the *Label* executed by the current program counter. When statement "br $\mathcal{P}_i$" is executed, the *Current Label* is changed to "$\mathcal{P}_i$" without any condition according to Rule 1. In Rule 2 and Rule 3, if *Entry Condition* "$\mathcal{C}_i$" is true, *Current Label* is changed to "$\mathcal{P}_i$", otherwise "$\mathcal{P}_j$".

2) Program Transformation: LLVM$_{cuda}$ transformation is initialized by deriving the topological sorting of the *Branch Graph*. Accordingly, the original CFG is restructured by adding one *Basic Block* between two topologically-adjacent *Labels* with setting their edges based on Figure 9.

The details of the *program transformation* are presented in Algorithm 1, where `delete_edge_without_loop` deletes the edges of the given *Label* except *Loop Edges*, `set_condition_edge` creates a conditional edge between *Label*s according to Rule 2 and Rule 3 in Figure 9, and `set_edge` creates a non-conditional edge between *Labels* according to Rule 1 in Figure 9. Specifically, *program transformation* is initialized to obtain a topological ordering of "branch_graph" at line 2. Next, each *Label* is parsed as topological ordering at line 4. In line 5, the *Entry Condition* is resolved for the *Current Label* followed by deleting the original edges of each *Label* except *Loop Edges* at line 7. From



(a) Before Transformation



(b) After Transformation

Fig. 10: An Example of *Program Transformation*

Algorithm 1 Transformation

Input : branch_graph, graph, conditions
Output:graph

```
1: function TRANSFORMATION
2:     topology ← topological_sort(branch_graph)
3:     previous_node ← Label()
4:     for each_label in topology do
5:         enter_condition ← conditions[each_label]
6:         next_node ← Label()
7:         delete_edge_without_loop(each_label, graph)
8:         set_condition_edge(previous_node, enter_condition,
9:         each_label, graph)
10:        set_condition_edge(previous_node, !enter_condition,
11:        next_node, graph)
12:        set_edge(each_label, next_node, graph)
13:        previous_node ← next_node
14:    return graph
```

line 8 to line 13, the current *Label* generates a new predecessor *Label* with a *Entry Condition*-satisfaction edge pointing to it and a new successor *Label* with a *Entry Condition*-satisfaction edge pointed from it. In addition, the generated predecessor *Label* points to the generated successor *Label* with a *Entry Condition*-dissatisfaction edge. For instance, an example of *program transformation* is demonstrated in Figure 10 where Figure 10(a) and Figure 10(b) both refer to the identical CFG. The circle nodes in Figure 10(a) represent the original *Label*s before *program transformation* and the rectangle nodes in Figure 10(b) represent the generated *Label*s after *program transformation*.

Algorithm 1 is input with the function for extracting *Entry Condition* for each *Label* demonstrated in Algorithm 2 which is initialized by a *Branch Graph* and an empty dictionary in which *Label* is a key and a *Entry Condition* which is the corresponding value. The *Entry Condition* for each *Label* is generated according to Rule 4 and Rule 5. In Rule 4, A_i refers to the boolean-expressions set owned by i-th predecessors of `label`. In Rule 5, a_i refers to a single boolean expression

Algorithm 2 Construct *Label* Conditions

Input : branch_graph
Output: condition_dict

```
1: function CONSTRUCT_CONDITION
2:     topology ← topological_sort(branch_graph)
3:     cond ← dict()
4:     for each_label in topology do
5:         labels ← find_pre_label(each_label)
6:         cond_lst ← list()
7:         for label ∈ labels do
8:             if label transmits condition to each_label then
9:                 cond_lst.append(cond[label] ∪ condition)
10:            else
11:                cond_lst.append(cond[label])
12:        for each_cond ∈ cond_lst do
13:            if cond[each_label] is empty then
14:                cond[each_label] ← each_cond
15:            cond[each_label] ← cond[each_label] ∩ each_cond
16:    for each_label ∈ cond do
17:        final_cond ← empty_logic_expression
18:        for condition ∈ cond[each_label] do
19:            final_cond ← final_cond ∧ condition
20:        cond[each_label] ← final_cond
21:    return cond
```

belonging to `cond[label]`. Algorithm 2 is initialized with collecting boolean expressions for each *Label* according to Rule 4. Then the *Entry Condition* for each *Label* is generated by Rule 5. From line 4–15, the `cond[each_label]` of each *Label* is constructed based on Rule 4, and the *Entry Condition* for *Labels* is generated based on Rule 5 from line 16–20, where `find_pre_label` in line 5 is implemented to find all the predecessors of the given *Label*. Consider the example in Figure 7. Suppose that the "br" instruction of `Label 0` is based on a boolean expression γ_0, when γ_0 is `true`, `Label 0` transits to `Label 11`. Thus, the `cond[Label 11]` = $\{\gamma_0\}$. Suppose that the "br" instruction of `Label 11` is based on a boolean expression γ_1, `Label 11` transits to `Label 15` when γ_1 is `true`, otherwise transits to `Label 21`. As a result, the `cond[Label 15]` = $\{\gamma_0, \gamma_1\}$ and `cond[Label 21]` = $\{\gamma_0, \neg\gamma_1\}$. Therefore, using Rule 4, we obtain `cond[Label 32]` by `cond[Label 21]`$\cap$`cond[Label 15]` = $\{\gamma_0\}$. And the *Entry Condition* for `Label 32` is γ_0 based on Rule 5.

$$cond[label] = \bigcap_{i=1}^{m} A_i,\ m = predecessor_number \tag{4}$$

$$EnterCondition[label] = \bigwedge_{i=1}^{n} a_i,\ n = |cond[label]| \tag{5}$$

3) Semantic-Reserving Theorems: In this section, we propose and prove two theorems to validate that our *program transformation* is able to preserve the semantics of the original program where Theorem 1 is the basis of Theorem 2.

Theorem 1. Given a LLVM$_{cuda}$ CFG ϕ and its *Branch Graph* φ, if two *Labels* are *Branch-Independent* in φ, their *Entry*

Conditions cannot be both true.

Proof. The proof will be done by contradiction. Consider two *Labels* α and β from φ which are *Branch-Independent*, for the lowest common ancestor θ of α and β, there exists a boolean variable γ for θ. When γ is true, α is defined to be added into *Execution Path*, otherwise β is defined to be added into *Execution Path*. Suppose that α and β have the same *Entry Condition*. This indicates that there exists at least one descendant *Label* of θ which reassigns γ. Since LLVM static single assignment (SSA) form enforces that each variable is only assigned once in a single *Label*, this leads to a contradiction. Thus, if two *Labels* α and β are *Branch-Independent* in φ, their *Entry Conditions* cannot be both true. Therefore, Theorem 1 holds. □

Theorem 2. Given a LLVM bitcode CFG ϕ, for each *Execution Path* in ϕ, there exists a *Semantic-Independent Execution Path* contained in the generated CFG φ by *program transformation* given ϕ and its *Branch Graph* as input.

Proof. We prove that Theorem 2 holds by induction on the number of *Labels* ($numLabel$). For $numLabel$=1, this is the case where there is an *Execution Path* ρ_1 generated from ϕ with one *Label* whose initial *Entry Condition* is γ_1. Accordingly, an *Execution Path* ρ_2 with two *Labels* can be generated in CFG φ by passing γ_1 to CFG φ where one *Label* is a *Stable Label* created by *program transformation* and the other is the *Label* in ρ_1. As ρ_2 contains a *Stable Label* without "write" instruction and the only *Label* of ρ_1. Therefore, ρ_1 and ρ_2 satisfy the conditions of being *Semantic-Independent*.

Suppose that Theorem 2 holds for $numLabel$=n-1. For $numLabel$=n, this is the case where there exists an *Execution Path* ρ_1 generated from ϕ with n *Labels* and the *Program State* at the (n-1)-th *Label* is ϵ. As Theorem 2 holds when $numLabel$=n-1, an *Execution Path* ρ_2 is *Semantic-Independent* with ρ_1's n-1 previous *Labels*. Accordingly, ϵ is contained in ρ_2 and *Program State* is the same between ρ_1 and ρ_2 before ϵ. Therefore, if ϵ has a *Loop-Edge* jump, according to *program transformation*, such *Loop Edge* is reserved in φ with the identical successor *Label* in both ρ_1 and ρ_2 according to Rule 1. On the other hand, if ϵ does not have a *Loop-Edge* jump, the successor *Label* of ρ_2 can be determined either from the *Labels* it points to or the *Branch-Independent Labels*. Suppose that the successor *Label* in ρ_1 is ϵ_1, since the current *Program State* of ρ_2 is the same as ρ_1, ϵ_1's *Entry Condition* in ρ_2 is also satisfied. Thus, ϵ_1 can be one successor *Label* in ρ_2 according to Rule 2. As a result, it can be ensured that there are only *Basic Blocks* generated by *program transformation* between ϵ and ϵ_1 because if there exists any other *Label* between ϵ and ϵ_1 in φ, then it must be *Branch-Independent* with ϵ_1. According to theorem 1, their *Entry Conditions* cannot be both true and according to Rule 3, the *Basic Block* generated by *program transformation* is always selected when the other choice is *Branch-Independent*. The final state for this situation is presented at Figure 11. Therefore, all the elements of the difference set $\rho_2 - \rho_1$ are *Stable Labels*. Hence, ρ_2 and ρ_1 are *Semantic-Independent* for $numLabel$=n. Thus, Theorem 2 is true. □

$$\rho_1 = \left[\ \alpha,\quad \beta, \ldots, \varepsilon, \varepsilon_1\ \right]$$
$$\rho_2 = \left[\ \alpha,\quad \underbrace{\square, \ldots, \square}_{\text{Basic Block}}, \beta, \ldots, \varepsilon, \underbrace{\square, \ldots, \square}_{\text{Basic Block}}, \varepsilon_1\ \right]$$

Fig. 11: If ϵ does not have a Loop Jump to ϵ_1

To conclude, it can be derived that by applying the *program transformation rule*, we are able to transform the original complex CFG structure by generating *Basic Block* with *Stable Labels* while preserving the original semantics. Therefore, to properly locate barrier functions, the original CUDA kernel functions can be transformed to be to properly locate barrier functions in the generated *Stable Labels*.

B. Overall framework of AuCS

Figure 12 presents the overall framework of *AuCS*. *AuCS* is initialized by compiling CUDA kernel functions to LLVM bitcode and using *Simulee* to detect synchronization for such LLVM bitcode. In particular, a data race bug is reported as a pair of statements executed by different threads. Barrier divergence bugs and redundant barrier function bugs are reported with the locations of their associated buggy barrier functions.

Next, *AuCS* transforms the original program based on *program transformation rules*. In particular, *AuCS* "flattens" the original LLVM bitcode by adding extra *Stable Labels*. For data race bugs, *AuCS* provides a mechanism to find the appropriate *Stable Label* for placing barrier functions. For barrier divergence bugs, *AuCS* first removes the buggy barrier functions and then applies the mechanisms for handling data race for properly placing barrier functions. For redundant barrier functions, based on the *Memory Model* generated from *Simulee*, *AuCS* detects and removes all redundant barrier functions in given kernel function.

At last, *AuCS* automatically captures the erroneous synchronization in LLVM_{cuda} and fixes them.

1) Recognizing Synchronization Bugs: Based on the aforementioned definitions in Section IV-A1, CUDA synchronization bugs can be depicted as follows.

- Data race can occur in an intra-*Label* and inner-*Label* manner. Specifically, the statements which incur data race bugs can be grouped as inner-*Label* statements where such statements belong to the identical *Label*, and intra-*Label* statements where such statements belong to different *Labels*.
- Barrier divergence is only possible to occur when a barrier function is located in a non-*Basic Block Label*.

2) Automating Data Race Scenarios: Since it is possible to incur barrier divergence by adding barrier functions to fix data race as in Section III, *AuCS* attempts to fix data race without incurring barrier divergence for both inner-*Label* and intra-*Label* data-race-inducted statements.

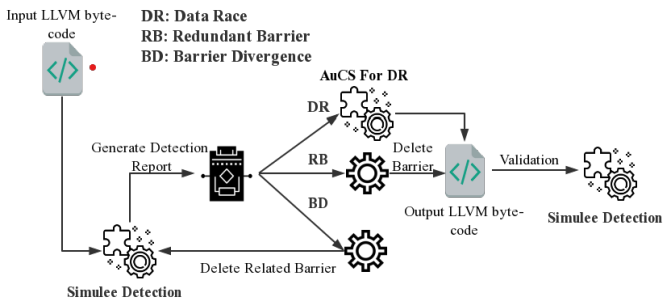


Fig. 12: Overview of *AuCS*

Algorithm 3 Auto Synchronization For Data Race

Input : label_1, label_2, branch_graph B, graph G
Output:graph

1: **function** AUTO_SYNC_RACE
2: topology ← topological_sort(B)
3: **if** label_2 $<_{topology}$ label_1 **then**
4: swap(label_1, label_2)
5: **if** label_1 is *Basic Block* **then**
6: add barrier at label_1
7: **return** graph
8: **if** label_2 is *Basic Block* **then**
9: add barrier at label_2
10: **return** graph
11: predecessor_1 ← find_basic_predecessor(label_1, B)
12: successor_1 ← find_basic_successor(label_1, B)
13: predecessor_2 ← find_basic_predecessor(label_2, B)
14: successor_2 ← find_basic_successor(label_2, B)
15: **if** successor_1 $\neq$ successor_2 **then**
16: add barrier at successor_1
17: **return** graph
18: sub_br ← extract_graph(B, predecessor_1, successor_1)
19: sub_graph ← extract_graph(G, predecessor_1, successor_1)
20: cond ← CONSTRUCT_CONDITION(sub_br)
21: sub_graph ← TRANSFORM(sub_br, sub_graph, cond)
22: replace_original_graph(G, sub_graph, successor1)
23: insert barrier at the new predecessor *Basic Block* of label_2
24: **return** graph

Intra-Label statements. For a data race incurred among intra-*Label* statements, *AuCS* first sorts the execution order of the two associated labels. Next, it identifies their respective predecessor and successor *Basic Block*s. *AuCS* would determine if it needs to apply *program transformation* based on whether the two *Labels* share the identical predecessor/successor *Basic Block*s. Lastly, *AuCS* adds barrier functions accordingly. We introduce the details of this mechanism in Algorithm 3, where `find_basic_predecessor` is used to find the predecessor *Basic Block*, `find_basic_successor` is used to find the successor *Basic Block* of the given *Label*, and `extract_graph` extracts the sub graph from the given original graph bounded by two given *Labels*.

Specifically, Algorithm 3 is initialized by inputting two *Labels* "label_1", "label_2", *Branch Graph* "branch_graph", and CFG "graph". In lines 2–4, assuming that "label_1" is ensured to happen before "label_2", if "label_1" or "label_2" is a *Basic Block*, we can add barrier functions directly without incurring barrier divergence in lines 5–10 according to Section IV-B1. On the contrary, the predecessor and successor *Basic Block* for "label_1" and "label_2" can be found in lines 11–14. Specifically in lines 15–17, if the predecessor and successor *Basic Block* are not identical, we can add barrier function at the nearest *Basic Block* after "label_1" to synchronize the program. Otherwise, if the predecessor and successor *Basic Block* are identical, *AuCS* extracts the associated sub CFG and sub *Branch Graph* in lines 18–19 according to the given predecessor and successor *Basic Block*, and constructs *Entry Conditions* for each *Label* in the extracted sub CFG in line 20. Next, it applies *program transformation* to the sub CFG in order to create a *Basic Block* between "label_1" and "label_2" in lines 21–23. As a result, adding a barrier function in the *Stable Label* generated by *program transformation* can automate synchronization of CUDA kernel functions by eliminating data race bugs.

Inner-Label statements. When the data-race-induced statements are in the same *Label*, *AuCS* splits the original *Label* at the first statement into two *Labels* and transfer the original inner-*Label* data race to intra-*Label* data race which can be fixed by applying Algorithm 3.

Note that so far *AuCS* is not designed for the synchronization scenario where different threads are executed under different iterations for the same loop. Please refer to more details discussed in Section V.

3) Automating Barrier Divergence Scenarios: Automatically fixing barrier divergence bugs is expected to be intricate because barrier divergence is highly involved with data race. Specifically, an intuitive solution is to simply remove the barrier functions in which the barrier divergence takes place. However, since a barrier divergence bug indicates possible data race bugs among different non-*Basic Block Labels*, it is possible that deleting the barrier function might introduce a new data race bug into the program. On the other hand, manually fixing data race might lead to a barrier divergence bug while the data race bug takes place in a non-*Basic Block*. For instance, revision d88e6a3540f of "arrayfire" [25] tried to fix data race but incurred additional barrier divergence, which was fixed in 0d0d7d1285a [26].

AuCS, on the other side, enables an effective solution for fixing barrier divergence by transforming it to automatically fixing data race. In particular, *AuCS* first deletes all the barrier functions that cause barrier divergence reported by *Simulee*. Next, *Simulee* is called again to check whether there is any data race. If not, it indicates that the barrier divergence is already fixed; otherwise we can apply the approach IV-B2 to fix the generated data race bugs.

4) Automating Redundant Barrier Function Scenarios: To fix the redundant barrier function bugs, *AuCS* applies *Simulee* to effectively detect the locations of the unnecessary barrier functions and remove them.

Overall, by applying *Simulee* and the *program transformation* for "flattening" the original program CFG, *AuCS* can effectively detect the synchronization bugs, identify/create the

proper locations for adding barrier functions to fix various erroneous synchronization scenarios. Therefore, *AuCS* can automate synchronization for LLVM bitcode of CUDA kernel functions such that the developers could save time and effort in fixing all erroneous synchronization scenarios.

C. Validation

Algorithm 3 applies *program transformation* to the erroneous synchronization scenarios, and inserts barrier functions in discovered/created *Basic Block*s. According to Theorem 2, the original program semantics remain unchanged.

We invoke *Simulee* to validate the LLVM bitcode generated by *AuCS* to check if the synchronization bug has been fixed. It is essential to consider whether adding barrier function to CUDA programs can inject barrier divergence bugs; in fact, since barrier functions are always inserted in *Basic Block*s that are executed by all the threads according to the definition of *Basic Block*, no new synchronization bug can be introduced into the original LLVM_{cuda} within the scope of this paper.

V. Experimental Evaluation

In this section, we conduct a set of experiments to evaluate the effectiveness and efficiency of *AuCS*. We select all the erroneous synchronization scenarios including data race, barrier divergence, and redundant barrier function from Gklee benchmark [24] and three real-world popular CUDA projects from *Simulee* dataset [27]: arrayfire (5143 stars, 8419 commits and 364K LoC), kaldi (2499 stars, 5171 commits and 381K LoC), thundersvm (818 stars, 790 commits and 343K LoC). Such studied projects were systematically selected in prior work [13], [27].

A. Experimental Setup

We performed our evaluation[1] on a desktop machine, with Intel(R) Xeon(R) CPU E5-4610 and 320 GB memory. The operating system is Ubuntu 16.04. We use the default values for all the parameters for running *Simulee*.

B. Result Analysis

Table I shows the experimental results, where the first three columns are used to identify the specific erroneous synchronization scenarios. The next two columns present the bug types and the automatic synchronization results performed by *AuCS*. The following column shows whether it is feasible for AutoSync [14] to automate synchronization. We split the execution time into two parts in the last two columns: the detection cost and automatic synchronization cost. For each automatic erroneous synchronization scenario, we use both *Simulee* and manual analysis to confirm whether the relevant synchronization bug is fixed or not. In particular, the manual analysis is used to observe whether the generated LLVM bitcode is semantically equivalent with the corresponding patch committed by developers.

[1]please refer to https://github.com/aucs2019/AuCS for *AuCS* details.

```
1  __shared__ float shrdMem[512];
2  float* desc = shrdMem;
3  ...
4  if (f < total_feat) {
5      const int histlen = 16;
6      const int hist_off = (tid_x % histsz) * desc_len;
7      ...
8      for (int i = tid_x; i < histlen*histsz; i += bsz_x)
9          desc[tid_y*histlen+i] = 0.f;
10     __syncthreads(); // barrier divergence
11         ...
12     for (int l = tid_x; l < desc_len*2; l += bsz_x)
13         desc[l     ] += desc[l+2*desc_len];
14     __syncthreads(); // barrier divergence
15     for (int l = tid_x; l < desc_len; l += bsz_x)
16         desc[l] += desc[l+desc_len];
17         ...
18 }...
```

Fig. 13: Simplified revision ee4d0bd77d7 for arrayfire

1) Effectiveness: First we apply *AuCS* to a total of 24 erroneous synchronization scenarios from the selected dataset. *AuCS* can successfully automate synchronization for 20 of them from Table I. Specifically, *AuCS* can successfully resolve difficult erroneous synchronization scenarios. e.g., the motivating example in section III.

Fixing data race bugs manually may introduce new barrier divergence bugs. For instance, the revision d88e6a3540f of arrayfire attempted to fix a data race bug but incurred an additional barrier divergence bug, which was fixed in a subsequent revision 0d0d7d1285a. In contrast, our experiment shows that *AuCS* is able to successfully synchronize LLVM_{cuda} in one step fully automatically without causing barrier divergence.

Additionally, *AuCS* can also automate more erroneous synchronization scenarios compared to AutoSync which uses a cost model to select an optimal placement for the barrier function. Such approach cannot fix either the motivating example in Section III or the erroneous synchronization scenarios when there is no *Basic Block* between buggy statements which frequently occur in real-world projects. For example, the revision ee4d0bd77d7 of arrayfire is presented in Figure 13. The barrier function should be inserted at lines 11 and 15 to synchronize data. Meanwhile, because the statements between line 6 and line 17 do not belong to any *Basic Block* and they are inside an if block, there should not be any barrier function inside this block. Otherwise a barrier divergence bug would be introduced. Since AutoSync cannot restructure code, it cannot fix this bug by inserting barrier functions into the original code. In addition, AutoSync cannot fix a read-write single-statement data race such as `a[i] += a[i + 1]`, because a barrier function cannot be inserted within a single statement. However, it can be resolved by *AuCS* because *AuCS* can translate such statement into two independent instructions in LLVM_{cuda} and further resolve it. Specifically, the sixth column of Table I shows if a synchronization scenario is beyond the search space of AutoSync. It can be observed that 11 of 24 erroneous synchronization scenarios are beyond the search space of AutoSync.

We can observe that there are four erroneous synchro-

TABLE I: Evaluation Results

Project	Revision	Kernel Function	Bug Type	AuCS (This work)	AutoSync [14] Feasibility	Detection Time Cost(s)	AuCS Time Cost (s)
GkleeTests	10eb6373d53	device_global	data race	✓	✗	1.35	0.003
GkleeTests	10eb6373d53	colonel	data race	✗	✗	1.27	N/A
GkleeTests	10eb6373d53	dl@deadlock_0	barrier divergence	✓	✗	3.96	0.025
GkleeTests	10eb6373d53	dl@deadlock_2	barrier divergence	✓	✓	3.65	0.001
arrayfire	0a8371a876b	computeEvalHomography	data race	✓	✓	14.49	0.013
arrayfire	a7a297ba814	scan_nonfinal_kernel	data race	✓	✓	4.236	0.021
arrayfire	a7a297ba814	scan_dim_nonfinal_kernel	data race	✓	✓	1.773	0.023
arrayfire	0c5a38182b7	hamming_matcher	data race	✓	✓	12.90	0.023
arrayfire	0c5a38182b7	hamming_matcher_unroll	data race	✓	✓	9.934	0.020
arrayfire	d7abcf2358e	JacobiSVD	data race	✓	✓	17.11	0.029
arrayfire	c59116e3ec3	warp_reduce	data race	✓	✓	4.215	0.014
arrayfire	a515b112076	scan_dim_kernel	data race	✓	✓	1.97	0.036
arrayfire	d88e6a3540f	warp_reduce	data race	✓	✓	4.81	0.029
arrayfire	1050816e422	hamming_matcher	data race	✓	✗	15.89	0.305
arrayfire	1050816e422	hamming_matcher_unroll	data race	✓	✗	28.24	0.368
arrayfire	0ed6cccc5ff	select_matches	barrier divergence	✓	✓	3.036	0.017
arrayfire	dfbfca5fb77	select_matches	barrier divergence	✗	✗	4.56	N/A
arrayfire	0e0c726d7d0	hamming_matcher_unroll	barrier divergence	✗	✗	12.45	N/A
arrayfire	ee4d0bd77d7	computeDescriptor	barrier divergence	✓	✗	23.31	0.055
arrayfire	0d0d7d1285a	warp_reduce	barrier divergence	✓	✓	3.92	0.029
arrayfire	31761d27f01	computeMedian	redundant barrier function	✓	✗	5.22	0.017
arrayfire	faefa30c3a0	harris_response	redundant barrier function	✓	✗	2.21	0.028
kaldi	bc13196e7fe	_add_diag_mat_mat	barrier divergence	✗	✗	8.53	N/A
thundersvm	febf515a826	nu_smo_solve_kernel	data race	✓	✓	6.55	0.071

nization scenarios that *AuCS* cannot resolve. In particular, for the revision 10eb6373d53 from GkleeTests, the data race in the kernel function "colonel" is a single-instruction race which is a write-write data race that incurs at the same LLVM-bitcode instruction. Such scenario cannot be resolved by inserting barrier function. The revision bc13196e7fe of kaldi, dfbfca5fb77 and 0e0c726d7d0 of arrayfire are another set of synchronization scenarios that cannot be resolved by *AuCS*. In those scenarios, different threads execute the same loop with different number of iterations. Meanwhile, a barrier function should be put inside the loop to synchronize data because of data race. Under such settings, even if the barrier function is located in the *Basic Block* inside the loop, when some threads complete fewer iterations than others, they would leave the loop while other threads have not completed the loop. Therefore, that incurs barrier divergence. Nevertheless, AutoSync cannot fix any of those synchronization bugs either.

2) Efficiency: We can observe from Table I that the detection part from *Simulee* is the most time-consuming part. Taking into account the detection time, the average time cost for the whole process is 8.21s. Meanwhile, the average time cost for automatic synchronization is 0.056s, which occupies only 0.686% of the total time cost (calculated as the time cost of *AuCS* devided by detection-inclusive total time cost). The max time cost for automatic synchronization is 0.368s. Overall, our results show that *AuCS* can generate suitable LLVM-bitcode patches for real-world projects rapidly.

VI. THREATS TO VALIDITY

In terms of external threat to validity, the effectiveness of *AuCS* has only been evaluated in the 24 erroneous synchronization scenarios and may not be able to generalize to other datasets. Nevertheless, we mitigate this threat by taking erroneous synchronization scenarios from two sources: GkleeTests [24] and *Simulee*'s dataset [27].

In addition, we identify two main limitations of *AuCS*. Firstly, that *AuCS* depends on the detection tool for CUDA synchronization bugs. In our work, we use *Simulee* to detect CUDA synchronization bugs and pass the result to *AuCS*. Nevertheless, our experiment results demonstrate that *AuCS* can automatically synchronize target kernel function effectively when the detection part is robust and reliable. Secondly, *AuCS* cannot handle the situation when different threads execute the same loop in different iterations and data synchronization is needed inside the loop. Although this situation rarely occurs in the data set we explored, we leave the synchronization of such scenarios as future work.

VII. RELATED WORK

CUDA synchronization bug detection. While the approaches regarding traditional software bug detections have been largely studied [8], [10], [28], [29], there are quite limited studies on CUDA synchronization bug detection. Several techniques exist for verifying the correctness of synchronization for multi-threaded CPU programs [30], [31]. In this work, we choose *Simulee* as our detection part to detect synchronization bugs since it has been shown to represent the state-of-the-art in terms of performance and detection ability. *Simulee* [13] is a dynamic detection tool that uses test inputs generated by Evolutionary Programming. On the other hand, Gklee [5] traces execution flow of threads and collects write statement set and read statement set, then determines whether there is any synchronization bugs by applying SMT solver. LDetector [32] instrumented compiler to detect races by using diffs between memory snapshots. CURD [3] is a compiler-based race detector like [33] which uses LLVM to instrument memory accesses and barriers in a real running process.

Synchronization bug repair. Automated debugging techniques have been proposed to localize [34]–[40] and fix [41]–[53] different types of bugs. In the context of synchronization bugs, there are many automated program repair approaches

for traditional multi-thread CPU programs. CFix [54] inserts synchronization operations into buggy code to make a correct patch, and selects best patch among the candidates to achieve a better performance. PFix [55] fixes synchronization bugs by inferring locking policies from memory access pattern. Program synthesis is another research area that is closely related to automated program repair. Program synthesis techniques have been successfully applied in the context of program repair to automatically synthesize expression/statements for repairing buggy programs [56]–[60]. In the context of program synthesis, AutoSync [14] is the most closely related work to our paper. Similar to our work, AutoSync fixes synchronization problems for GPU kernels by inserting barrier functions. There are several key differences of our work compared to AutoSync: (1) AutoSync relies on GPUVerify for determining the race location and providing a black-box correctness oracle, whereas we use *program transformation* for finding the correct location for placing the barrier functions; (2) AutoSync assumes that the race location exists in the buggy GPU programs but we show in Section III that creation of new blocks is needed for fixing more complex synchronization bugs; and (3) our approach is more general than AutoSync as we consider all synchronization scenarios, including data races, barrier divergence and redundant barrier function.

Compiler optimization. Compilers have applied different methods to transform the structure of original code without changing its semantics. Bacon et al. [12] introduced a bunch of transformation methods to restructure the original program in order to achieve a better performance. As our *program transformation* acts on the LLVM bitcode level and our results show that *AuCS* could automate synchronization for CUDA programs rapidly, a potential future work would be to integrate the workflow of *AuCS* as part of compiler optimization.

VIII. Conclusions

In this paper, we propose an automatic synchronization tool named *AuCS* for CUDA program in order to save developers from designing error-prone and complicated synchronization mechanism. Fixing synchronization bugs for CUDA programs is challenging because the barrier functions should be located at *Basic Block*s to avoid barrier divergence. Based on the detection reported by *Simulee*, *AuCS* creates *Basic Block*s among the buggy statements via program transformation without changing the original semantics for barrier functions to synchronize the data flow. *AuCS* can automatically synchronize 20 of 24 synchronization scenarios from the three real-world CUDA projects and Gklee benchmark.

IX. Acknowledgement

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169 and No. 61902170), Shenzhen Peacock Plan (Grant No. KQTD2016112514355531), and Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. ZDSYS201703031748284 and No. JCYJ20170817110848086). This work is also partially supported by National Science Foundation under Grant No. CCF-1763906 and Amazon. The authors also thank Yicheng Ouyang for the help with editing the paper.

References

[1] "Cuda program introduction," https://en.wikipedia.org/wiki/CUDAr, 2019.

[2] "Gpgpu introduction," https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units, 2019.

[3] Y. Peng, V. Grover, and J. Devietti, "Curd: A dynamic cuda race detector," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 390–403. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192368

[4] A. Eizenberg, Y. Peng, T. Pigli, W. Mansky, and J. Devietti, "Barracuda: Binary-level analysis of runtime races in cuda programs," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 126–140. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062342

[5] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "Gklee: Concolic verification and test generation for gpus," *SIGPLAN Not.*, vol. 47, no. 8, pp. 215–224, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2370036.2145844

[6] P. Li, X. Hu, D. Chen, J. Brock, H. Luo, E. Z. Zhang, and C. Ding, "Ld: Low-overhead gpu race detection without access monitoring," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 1, pp. 9:1–9:25, Mar. 2017. [Online]. Available: http://doi.acm.org/10.1145/3046678

[7] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," *SIGPLAN Not.*, vol. 37, no. 5, pp. 258–269, May 2002. [Online]. Available: http://doi.acm.org/10.1145/543552.512560

[8] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997. [Online]. Available: http://doi.acm.org/10.1145/265924.265927

[9] A. Dinning and E. Schonberg, "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*, ser. PPOPP '90. New York, NY, USA: ACM, 1990, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/99163.99165

[10] R. H. B. Netzer and B. P. Miller, "Improving the accuracy of data race detection," *SIGPLAN Not.*, vol. 26, no. 7, pp. 133–144, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/109626.109640

[11] "The simulee project," https://github.com/Lebronmydx/Simulee, 2019.

[12] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345–420, 1994.

[13] M. Wu, H. Zhou, L. Zhang, C. Liu, and Y. Zhang, "Characterizing and detecting cuda program bugs," *arXiv,1905.01833*, 2019.

[14] S. Anand and N. Polikarpova, "Automatic synchronization for gpu kernels," in *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018, pp. 1–9.

[15] "Introduction cuda c," https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/, 2019.

[16] P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer, "Interleaving and lock-step semantics for analysis and verification of gpu kernels," in *Programming Languages and Systems*, M. Felleisen and P. Gardner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 270–289.

[17] Xtra-Computing, "Thundersvm," https://github.com/Xtra-Computing/thundersvm.

[18] arrayfire, "Arrayfire," https://github.com/arrayfire/arrayfire, 2019.

[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[20] X. Yao, Y. Liu, and G. Lin, "Evolutionary programming made faster," *IEEE Transactions on Evolutionary computation*, vol. 3, no. 2, pp. 82–102, 1999.

[21] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: A verifier for gpu kernels," *SIGPLAN Not.*, vol. 47, no. 10, pp. 113–132, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2398857.2384625

[22] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel, "Civl: formal verification of parallel programs," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 830–835.

[23] F. R. Monteiro, E. H. d. S. Alves, I. S. Silva, H. I. Ismail, L. C. Cordeiro, and E. B. de Lima Filho, "Esbmc-gpu a context-bounded model checking tool to verify cuda programs," *Science of Computer Programming*, vol. 152, pp. 63–69, 2018.

[24] Geof23, "Gkleetests," https://github.com/Geof23/GkleeTests.

[25] arrayfire, "Fix race condition in reduce_first_kernel." https://github.com/arrayfire/arrayfire/commit/d88e6a3540f89d3289df0ee8f42c3bda0682597c, 2019.

[26] arrayfire, "Remove the need for volatile memory by always using __syncthreads()," https://github.com/masashi-y/arrayfire/commit/0d0d7d1285aa4a2c8cce5a5f7b377fe4b4965f60, 2019.

[27] "The simulee project," https://github.com/Lebronmydx/Simulee/tree/master/raw_data_report_script, 2019.

[28] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting atomicity violations via access interleaving invariants," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168864

[29] L. Chew and D. Lie, "Kivati: Fast detection and prevention of atomicity violations," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 307–320. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755945

[30] P. d. C. Gomes, D. Gurov, M. Huisman, and C. Artho, "Specification and verification of synchronization with condition variables," *Science of Computer Programming*, vol. 163, pp. 174–189, 2018.

[31] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 331–340.

[32] P. Li, C. Ding, and T. Soyata, "Ldetector: A low overhead race detector for gpu programs," 2014.

[33] "Racecheck tool," https://docs.nvidia.com/cuda/cuda-memcheck/index.html#racecheck-tool, 2019.

[34] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 169–180.

[35] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, "An empirical study of boosting spectrum-based fault localization via pagerank," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[36] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 92, 2017.

[37] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[38] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 609–620.

[39] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *OOPSLA*, vol. 48, no. 10, 2013, pp. 765–784.

[40] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 273–283.

[41] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 19–30.

[42] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang, "History-driven build failure fixing: how far are we?" in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 43–54.

[43] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 187–198. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180243

[44] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2017.45

[45] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE' 2013*. IEEE Press, 2013, pp. 802–811.

[46] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 727–738.

[47] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 471–482. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818813

[48] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 1–12.

[49] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 180–182.

[50] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury, "A correlation study between automated program repair and test-suite metrics," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2948–2979, 2018.

[51] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 740–751.

[52] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S. H. Tan, and D. Marinov, "Reassert: a tool for repairing broken unit tests," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1010–1012.

[53] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *ASE*, 2019, to appear.

[54] G. Jin, W. Zhang, and D. Deng, "Automated concurrency-bug fixing," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 221–236.

[55] H. Lin, Z. Wang, S. Liu, J. Sun, D. Zhang, and G. Wei, "Pfix: fixing concurrency bugs based on memory access patterns," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 589–600.

[56] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[57] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 691–701.

[58] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 448–458.

[59] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 4, p. 15, 2018.

[60] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 593–604.