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]. 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-
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.

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.
Fig. 2: An Example of Data Race

Fig. 3: An Example of Barrier Divergence

Fig. 4: An Example of Redundant Barrier Function

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 remaining 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 the 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 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. “feb515a82” 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()”.

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 executed 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 bytecode is a low-level code representation in Static Single Assignment (SSA) form [19]. In particular, LLVM bytecode 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 to automatically generate the input for kernel functions that can trigger CUDA synchronization bugs. Next, by simulating kernel function execution with the bug-induced input, Simulee detects synchronization bugs and the associated locations in the original program. Moreover, there are other synchronization bug detection approaches for CUDA...
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 \texttt{num_elements} is set to 5, thread (0 0 0)(1 0 0) writes data to \texttt{input_array[1]} while thread (0 0 0)(2 0 0) reads data from \texttt{input_array[1]}, and thread (0 0 0)(3 0 0) writes data to \texttt{input_array[3]} while thread (0 0 0)(2 0 0) reads data from \texttt{input_array[3]}. Correspondingly in its LLVM bitcode, such race takes place between \texttt{Label 15} and \texttt{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., \texttt{Label 15} or \texttt{Label 21} in Figure 6. However, it would lead to barrier divergence. To illustrate, in Figure 6, by adding a barrier function in \texttt{Label 15}, the thread that executes \texttt{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 \texttt{Label 15} and \texttt{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 \textit{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 \textit{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 \textit{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 \texttt{Label 15} and \texttt{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:

- \textit{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 \textit{Label}. Different \textit{Labels} are connected by “\texttt{br}” instructions, as presented in Figure 8. In particular, \textit{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.

- \textit{Stable Label} is a \textit{Label} which does not contain any “\texttt{write}” instruction.

- \textit{Branch Graph} is a directed acyclic graph that represents a LLVM bitcode program without \textit{Loop Edges}. Its nodes and edges are the same as in a LLVM-bitcode CFG, except for \textit{Loop Edges}.

- \textit{Execution Path} refers to a single thread’s \textit{Label} sequence in a complete execution of a CUDA kernel function. Note that the intersection of two \textit{Execution Paths} is a set of \textit{Labels} belonging to both \textit{Execution Paths}.

- \textit{Loop Edge} refers to a transition relation between two \textit{Labels}. Suppose there is an \textit{Execution Path} $\rho = [\alpha, ..., \beta, \alpha]$.
Fig. 5: C++ version for GkleeTest Example

```cpp
void device_global(unsigned int *input_array, int num_elements) {
    int my_index = blockIdx.x + blockDim.x + threadIdx.x;
    if (my_index < num_elements) {
        if (my_index < 1) {
            input_array[my_index] = my_index;
        } else {
            input_array[my_index] = input_array[my_index+1];
        }
    }
}
```

Fig. 6: LLVM version for GkleeTest Example

```llvm
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>:32 ...
  br label %33 ; <label>:33 ...
  ret void
}
```

![Fig. 7: Topological Structure of LLVM](image)

where the first $\alpha$ is executed before than the first $\beta$. The transition from $\beta$ to $\alpha$ is defined as a Loop Edge.

- **Basic Block** is a Label that intersects all the possible Execution Paths 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 $\alpha$, the other Label named $\beta$. If there is no path from $\alpha$ to $\beta$ and no path from $\beta$ to $\alpha$ in their Branch Graph, then $\alpha$ and $\beta$ are defined to be Branch-Independent.
- **Semantic-Independent** Suppose that there are two different Execution Paths named $\rho_1$ and $\rho_2$, if $\rho_1 \cap \rho_2 = \rho_2$ and their difference set $\rho_1 - \rho_2$ does not contain any “write” instruction, then $\rho_1$ and $\rho_2$ 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, $\rho_1$ and $\rho_2$ 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 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 $P_i$” is executed, the Current Label is changed to “$P_i$” without any condition according to Rule 1. In Rule 2 and Rule 3 if Entry Condition “$C_i$” is true, Current Label is changed to “$P_i$”, otherwise “$P_j$”.

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 Labels 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 2. 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.
Algorithm 1 Transformation

**Input**: branch_graph, graph, conditions

**Output**: graph

```plaintext
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, each_label, graph)
9:     set_condition_edge(previous_node, !enter_condition, next_node, graph)
10: set_edge_label(each_label, next_node, graph)
11: previous_node ← next_node
12: return graph
```

Fig. 10: An Example of Program Transformation

Line 8 to line 13, the current Label generates a new predecessor Label with an Entry Condition-satisfaction edge pointing to it and a new successor Label with an 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 Labels before program transformation and the rectangle nodes in Figure 10(b) represent the generated Labels 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 belonging to cond[label].

```
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 in 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 in 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 in cond do
17:       final_cond ← empty_logic_expression
18:     for condition in cond[each_label] do
19:       final_cond ← final_cond ∧ condition
20:     cond[each_label] ← final_cond
21: return cond
```

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 code CFG $\varphi$ and its Branch Graph $\varphi$, if two Labels are Branch-Independent in $\varphi$, their Entry
Conditions cannot be both true.

**Proof.** The proof will be done by contradiction. Consider two Labels $\alpha$ and $\beta$ from $\varphi$ which are Branch-Independent, for the lowest common ancestor $\theta$ of $\alpha$ and $\beta$, there exists a boolean variable $\gamma$ for $\theta$. When $\gamma$ is true, $\alpha$ is defined to be added into Execution Path, otherwise $\beta$ is defined to be added into Execution Path. Suppose that $\alpha$ and $\beta$ have the same Entry Condition. This indicates that there exists at least one descendant Label of $\theta$ which reassigns $\gamma$. 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 $\alpha$ and $\beta$ are Branch-Independent in $\varphi$, their Entry Conditions cannot be both true. Therefore, Theorem 1 holds. $\square$

**Theorem 2.** Given a LLVM bitcode CFG $\varphi$, for each Execution Path in $\varphi$, there exists a Semantic-Independent Execution Path contained in the generated CFG $\varphi$ by program transformation given $\varphi$ 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 $\rho_1$ generated from $\varphi$ with one Label whose initial Entry Condition is $\gamma_1$. Accordingly, an Execution Path $\rho_2$ with two Labels can be generated in CFG $\varphi$ by passing $\gamma_1$ to CFG $\varphi$ where one Label is a Stable Label created by program transformation and the other is the Label in $\rho_1$. As $\rho_2$ contains a Stable Label without "write" instruction and the only Label of $\rho_1$. Therefore, $\rho_1$ and $\rho_2$ 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 $\rho_1$ generated from $\varphi$ with $n$ Labels and the Program State at the (n-1)-th Label is $\epsilon$. As Theorem 2 holds when $numLabel=n-1$, an Execution Path $\rho_2$ is Semantic-Independent with $\rho_1$’s n-1 previous Labels. Accordingly, $\epsilon$ is contained in $\rho_2$ and Program State is the same between $\rho_1$ and $\rho_2$ before $\epsilon$. Therefore, if $\epsilon$ has a Loop-Edge jump, according to program transformation, such Loop Edge is reserved in $\varphi$ with the identical successor Label in both $\rho_1$ and $\rho_2$ according to Rule 1. On the other hand, if $\epsilon$ does not have a Loop-Edge jump, the successor Label of $\rho_2$ can be determined either from the Labels it points to or the Branch-Independent Labels. Suppose that the successor Label in $\rho_1$ is $\epsilon_1$, since the current Program State of $\rho_2$ is the same as $\rho_1$, $\epsilon_1$’s Entry Condition in $\rho_2$ is also satisfied. Thus, $\epsilon_1$ can be one successor Label in $\rho_2$ according to Rule 2. As a result, it can be ensured that there are only Basic Blocks generated by program transformation between $\epsilon$ and $\epsilon_1$ because if there exists any other Label between $\epsilon$ and $\epsilon_1$ in $\varphi$, then it must be Branch-Independent with $\epsilon_1$. 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, $\rho_2$ and $\rho_1$ are Semantic-Independent for $numLabel=n$. Thus, Theorem 2 is true. $\square$

$$\rho_1 = [\alpha, \beta, \ldots, \epsilon, \epsilon_1]$$

$$\rho_2 = [\alpha, \Box, \ldots, \Box, \beta, \ldots, \epsilon, \Box, \ldots, \Box, \epsilon_1]$$

Fig. 11: If $\epsilon$ does not have a Loop Jump to $\epsilon_1$

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 code 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-induced statements.
**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 \( \leftarrow \) topological_sort(\( B \))
3: if label_2 \( \leftarrow \) 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 \( \leftarrow \) find_basic_predecessor(label_1, \( B \))
12: successor_1 \( \leftarrow \) find_basic_successor(label_1, \( B \))
13: predecessor_2 \( \leftarrow \) find_basic_predecessor(label_2, \( B \))
14: successor_2 \( \leftarrow \) 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 \( \leftarrow \) extract_graph(\( B \), predecessor_1, successor_1)
19: sub_graph \( \leftarrow \) extract_graph(\( G \), predecessor_1, successor_1)
20: cond \( \leftarrow \) CONSTRUCT_CONDITION(sub_br)
21: sub_graph \( \leftarrow \) TRANSFORM(sub_br, sub_graph, cond)
22: replace_original_graph(\( G \), sub_graph, successor_1)
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 Blocks. AuCS would determine if it needs to apply program transformation based on whether the two Labels share the identical predecessor/successor Basic Blocks. 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 [V-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 Blocks. 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 Blocks 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 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) 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-
II. 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, 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 is a dynamic detection tool that uses test inputs generated by Evolutionary Programming. On the other hand, Gklee [9] traces execution flow of threads and collects write statement 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 is an instrumented compiler to detect races by using diffs between memory snapshots. CURD is a compiler-based race detector 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 match, 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 the race location exists in the buggy GPU programs but we consider all synchronization scenarios, including data races, barrier divergence and redundant 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 Blocks to avoid barrier divergence. Based on the detection reported by Simulée, AuCS creates Basic Blocks 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 to 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. KQTD201612251435531), 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**


