Automatic Synchronization for GPU Kernels

Sourav Anand
UC San Diego

Nadia Polikarpova
UC San Diego

***Abstract*—We present a technique for automatic synthesis of efficient and provably correct synchronization in GPU kernels. Our technique relies on an off-the-shelf correctness oracle and achieves efficient synthesis by leveraging the *race location* information provided by the oracle in order to encode optimal synchronization synthesis as a MaxSAT problem. We have implemented our technique in a tool called AUTOSYNC that works on kernels written in CUDA and uses a static verifier GPUVERIFY as the correctness oracle. An evaluation on 18 realistic kernels from the GPUVERIFY benchmark suite shows that AUTOSYNC is able to synthesize optimal synchronization placements, and synthesis times are reasonable (20 seconds for our largest benchmark).**

I. INTRODUCTION

Recent years have seen increasing use of *graphics processing units* (GPUs) for speeding up general-purpose computations. GPU computations are highly parallel—with thousands of threads running concurrently—which creates ample opportunity for data races. To prevent races, programmers add *barrier synchronization* statements to their GPU code. Because synchronization incurs a performance penalty, a GPU programmer is faced with a challenging task of finding a placement of barrier statements that is both *correct* (eliminates all data races) and *optimal* (incurs the least overhead).

In response to this challenge, several verification techniques have been proposed [1], [2], [3], [4], [5], [6] for detecting data races in GPU code or proving their absence. These techniques would alert the programmer that a barrier statement is missing, but they neither suggest where to place the barrier, nor check whether the current placement is optimal. In this work we propose a computer-aided approach to GPU programming, where the programmer omits barrier statements from their code altogether, and our technique automatically synthesizes a correct and optimal barrier placement.

Barrier Synthesis. One approach to barrier synthesis is to search the space of all possible barrier placements, using an existing GPU verification tool [6], [4] as a black-box *correctness oracle*; among all correct placements, we can then select an optimal one according to some *cost model*. The benefit of such a black-box approach is that it automatically takes advantage of any current and future advances in GPU verification. Brute-force enumeration, however, would be prohibitively expensive for most programs, since the number of possible placements grows exponentially with the size of the program, and verifying each candidate placement is an expensive operation on its own.

In this paper we show how to leverage the *race location* provided by the oracle and the conservative *operational semantics* used in verification in order to avoid considering most invalid placements and thereby make the synthesis practical. Moreover, we demonstrate how to encode this information together with the cost model as a system of *soft Boolean constraints*, which allows our technique to delegate the bulk of the search to MaxSAT solvers. Our technique is *sound* and *complete* relative to the correctness oracle.

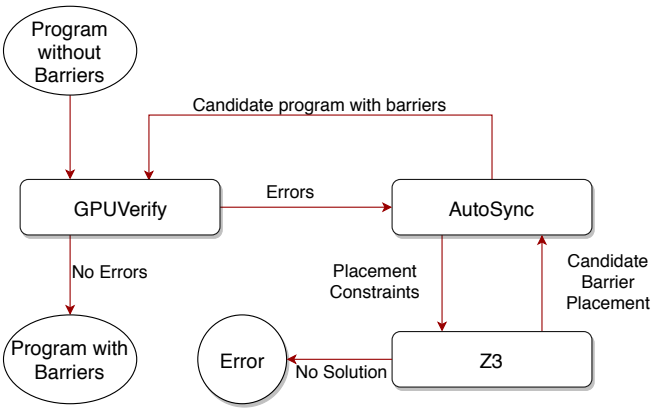

Fig. 1: The AUTOSYNC workflow

AUTOSYNC. We have implemented this constraint-based approach to barrier synthesis in a tool called AUTOSYNC (Fig. 1). The tool takes as input GPU programs—or *kernels*—written in the popular CUDA programming model, and uses the sound static verifier GPUVERIFY [6] as the correctness oracle. For constraint-based search, the tool relies on the νZ MaxSAT solver [7], which is part of Z3 [8].

Evaluation. We have evaluated AUTOSYNC on a series of small but challenging micro-benchmarks, as well as 18 realistic CUDA kernels from the GPUVERIFY benchmark suite. Our evaluation shows that in all these benchmarks, AUTOSYNC is able to recover a barrier placement that is at least as optimal as the one originally provided by the developer. Surprisingly, in 5 cases the automatically generated placement is *strictly better* than the original. Moreover, synthesis times are moderate and range from 1 to under 30 seconds. AUTOSYNC and all our benchmarks are available at www.souravanand.com/autosync.html.

II. MOTIVATING EXAMPLES

This section goes through a series of examples of data races in GPU programs, showcases the challenges of finding correct and optimal barrier placements, and provides the intuition for how AUTOSYNC addresses these challenges.

```
1  x = A[tid+1];
2  x = x+11;
3  A[tid] += x;
```

```
1  x = A[tid + 1];
2  x = x+11;
3  __syncthreads();
4  A[tid] += x;
```

Fig. 2: (left) A kernel with a race, and (right) the correctly synchronized version of the kernel.

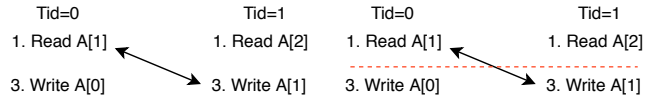

Fig. 3: Execution traces of kernels in Fig. 2 (left) and (right); the arrow depicts a data race; the dotted line depicts a barrier.

A. Straight-line Code

In the CUDA programming model, programmers describe a GPU computation as a *kernel*: a template to be executed by each GPU thread, implicitly parametrized by a unique thread id. For example, a simple kernel in Fig. 2 (left) instructs each thread to read from a shared array `A` at a distinct index, which depends on the thread's id **`tid`**, and then write into the array at the preceding index.

Fig. 3 (left) depicts an execution of this kernel by two threads with ids 0 and 1. This execution exhibits a read-write race: since the two threads are not synchronized, the read from `A[1]` by thread 0 is racing with the write to `A[1]` by thread 1. Eliminating this race requires adding a *barrier* statement `__syncthreads()` between the two racing instructions, as shown in Fig. 2 (right). A barrier requires all the threads to reach it before any thread can continue execution. When thread 1 encounters the barrier, it is forced to wait until thread 0 encounters the same barrier; hence the read from `A[1]` is now guaranteed to happen before the write to the same location.

Barrier synthesis. Given the kernel in Fig. 2 (left), AUTOSYNC first checks its correctness using GPUVERIFY (see Fig. 1), which reports a possible data race between lines 1 and 3. Based on this race location information, AUTOSYNC generates a *placement constraint*:

$$L_1 \vee L_2$$

Here each L_i is a propositional variable that indicates whether a barrier should be inserted after line i. Although any solution to this constraint would eliminate the race, setting more than one L_i to 1 is suboptimal, since every barrier incurs a performance overhead. To avoid suboptimal solutions, AUTOSYNC adds a *soft constraint* $\neg L_i$ for each line i in the program, which penalizes the solver for setting any L_i to 1. The resulting system of constraints is discharged by Z3's MaxSAT solver, producing the solution $\{L_2\}$[1]. The corresponding barrier placement, shown in Fig. 2 (right), is proven correct by GPUVERIFY, and the synthesis succeeds.

[1]We write a solution as a set of all variables set to 1.

```
1  for(i=0; i<n; i++){
2    x = A[tid+1];
3    x = x+i;
4    A[tid] += x;
5  }
```

```
1  for(i=0; i<n; i++) {
2    __syncthreads();
3    x = A[tid+1];
4    x = x+i;
5    __syncthreads();
6    A[tid] += x;
7  }
```

Fig. 4: (left) A kernel with a race inside a loop, and (right) the correctly synchronized version of the kernel.

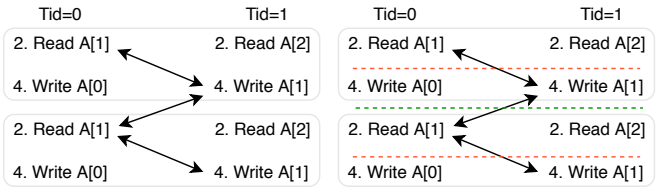

Fig. 5: Execution traces of kernels in Fig. 4 (left) and (right); each gray box corresponds to one loop iteration.

B. Loops

Given our first example, the reader might be wondering if all data races can be eliminated by simply inserting a barrier right before the second racing line. In the presence of loops, however, barrier placement becomes more challenging. Consider the kernel in Fig. 4 (left), which preforms a similar computation, but inside a loop. GPUVERIFY again reports a race between the two accesses to `A` (lines 2 and 4), however, adding a barrier between these two lines turns out to be *insufficient* to make the kernel race-free.

To see why, consider the execution trace of this kernel depicted in Fig. 5: a write to `A[1]` by thread 1 can race with a read executed by thread 0 either in the same loop iteration (*intra-iteration race*) or in a different one (*inter-iteration race*). Adding the "red" barrier between lines 2 and 4 forces the write to happen after the read in the same iteration, but imposes no order with the read from the next iteration. To synchronize this kernel correctly, a second barrier must be added *within* the loop body but *outside* the two racing lines ("green" barrier after line 1 in Fig. 4 and Fig. 5).

Barrier synthesis. Since the racing lines reported by GPUVERIFY are inside a loop, AUTOSYNC generates the following system of placement constraints:

$$P^0 \vee P^1$$
$$P^0 \Rightarrow (L_2 \vee L_3)$$
$$P^1 \Rightarrow (L_1 \vee L_4)$$

Intuitively, the verification error does not contain enough information to determine the type of the race—intra-iteration, inter-iteration, or both—hence we encode the possibility of each race type using a fresh propositional variable (P^0 for intra-iteration and P^1 for inter-iteration). As before, AUTOSYNC generates soft constraints $\neg L_i$ for all lines i, however, the

```
1  x = 0;
2  if(tid%2==0) {
3    x = A[tid+2];
4  }
5  if(tid%6==0) {
6    A[tid] += x;
7  }
```

```
1  x = 0;
2  if(tid%2==0) {
3    x = A[tid+2];
4  }
5  __syncthreads();
6  if(tid%6==0){
7    A[tid] += x;
8  }
```

Fig. 6: (left) A kernel with a race between two different basic blocks, and (right) the correctly synchronized version of kernel.

lines inside the loop are given a *higher weight*. This forces the solver to prefer placing barriers outside the loop, whereby minimizing performance overhead.

Given these constraints, Z3 might return a solution $\{P^0, L_3\}$, which violates only one soft constraint and corresponds to adding the "red" barrier after line 3 alone. An attempt to verify this solution reveals that the race is still present. Hence, in a second iteration of barrier synthesis, AUTOSYNC asks Z3 for a different assignment to the P variables, by adding a constraint $\neg(P^0 \wedge \neg P^1)$. The second solution, $\{P^1, L_1\}$ (the "green" barrier alone) does not solve the race either. In the third iteration, AUTOSYNC further adds a constraint $\neg(\neg P^0 \wedge P^1)$, which forces the solver to set both P^j to 1 and results in the final solution $\{P^0, P^1, L_1, L_3\}$.

Nested loops. In general, if both racing statements are inside a loop nest of depth d, we have to consider $d + 1$ possibilities: one intra-iteration and d inter-iteration races as different depths. If the race is inter-iteration, placing a barrier at depth d is always sufficient, but "shallower" barriers incur less overhead.

C. Barrier Divergence

Conditionals also complicate barrier placement. Consider the kernel in Fig. 6 (left), where lines 3 and 6 are racing. Placing a barrier right after line 3 or right before line 6 (i.e. inside a conditional) would make that barrier unreachable for some of the threads executing the kernel, leading to undefined behavior due to so called *barrier divergence* [1]. The only correct solution is to insert the barrier between the two conditionals, as shown in Fig. 6 (right).

Luckily, GPUVERIFY detects and reports if a candidate barrier placement might cause barrier divergence. In response to this error, AUTOSYNC adds a hard constraint that excludes all lines within the problematic **if**-block from consideration.

D. Multiple Races

When a kernel contains multiple data races, analyzing and eliminating each race independently might lead to a suboptimal barrier placement. Consider the kernel in Fig. 7 (left) with two pairs of racing lines: $\langle 1, 3\rangle$ and $\langle 2, 4\rangle$. Considering the two races independently might result in inserting two barrier

```
1  x=A[tid+1];
2  y=B[tid+1];
3  A[tid] = x+y;
4  B[tid] = x-y;
```

```
1  x=A[tid+1];
2  y=B[tid+1];
3  __syncthreads();
4  A[tid] = x+y;
5  B[tid] = x-y;
```

Fig. 7: (left) A kernel with two data races, and (right) the correctly synchronized version that requires just one barrier.

```
kernel  ::= blk
blk     ::= {(ℓ : stmt)*}
stmt    ::= local name | name := expr | barrier
          | name := rd(expr) | wr(expr, expr)
          | if expr blk | while expr blk
expr    ::= name | tid | n | expr op expr
```

Fig. 8: The syntax of KPL

statements, whereas in fact, a single barrier after line 2 eliminates both races. Such interactions between different races are difficult for programmers to reason about. AUTOSYNC, on the other hand, generates the optimal placement in the first iteration, since $\{L_2\}$ is the least-cost solution to the placement constraints $[L_1 \vee L_2, L_2 \vee L_3]$.

III. SYNTHESIS ALGORITHM

This section formalizes our synchronization synthesis algorithm for KPL (*Kernel Programming Language*), a core language we borrow from the work on GPUVERIFY [1].

A. Kernel Programming Language

Syntax. The syntax of KPL is presented in Fig. 8. Expressions `expr` are thread-local (do not access shared memory). Reading and writing from/to shared memory is accomplished via the statements **rd** and **wr**, respectively. A reserved variable **tid** gives the execution thread access to its unique id, which enables different threads to execute different behavior. Compared to the presentation in [1], we omit jump statements and **else** branches of conditionals (both can be desugared into our language in a standard way), and procedures, which—while not technically challenging—are currently not supported.

Each statement in a kernel is *labeled* with a unique label ℓ; $\mathrm{stmt}(\ell)$ denotes the statement with label ℓ. Labels of compound statements—**if** and **while**—double as labels of their enclosed blocks; the top-level block of the kernel has a reserved label $main$. A kernel's *label tree* is a tree whose nodes are statement labels, and a node's parent is the label of its enclosing block; in addition, we add a special *start node* ℓ_s as the leftmost child of each block[2]. We use $\mathrm{blks}(\ell)$ to denote the set of enclosing blocks of ℓ (its ancestors in the label tree); among those, $\mathrm{loops}(\ell)$ are the enclosing **while** blocks and $\mathrm{conds}(\ell)$ are the enclosing **if** blocks.

[2]Thus, to place a barrier at the beginning of the block, we place it after ℓ_s.

Sometimes we interpret these sets of blocks as sequences, ordered from the root downward. We define the *program text order* $\prec$ on labels as the *post-order* of the label tree. A *label interval* $[\ell_1, \ell_2)$ denotes the set of labels ℓ that lie between ℓ_1 and ℓ_2 in the program text ($\ell_1 \preceq \ell \prec \ell_2$) and share all enclosing blocks with at least one of the interval bounds ($\text{blks}(\ell) \subseteq (\text{blks}(\ell_1) \cup \text{blks}(\ell_2))$). For example, on Fig. 6 (left):[3] $\text{blks}(3) = \{main, 2{:}4\}$; $\text{blks}(5{:}7) = \{main\}$; $[3, 6) = \{3, 2{:}4, 5{:}7_s\}$, while $[3, 5{:}7) = \{3, 2{:}4\}$ ($5{:}7_s$ and 6 are excluded, since they do not share their enclosing block $5{:}7$ with any of the two bounds). Note that the set of all children of a block ℓ can be expressed as the interval $[\ell_s, \ell)$.

Semantics. Prior work [1] defined the semantics of KPL dubbed *synchronous, delayed visibility* (*SDV*). According to this semantics, all threads execute the kernel instructions *synchronously* (in lock step) but the effect of a **wr** statement may be *delayed*, i.e. not immediately visible to other threads. Importantly, the semantics of control structures models so-called *predicated execution*, illustrated informally in Fig. 9. Under predicated execution, the body of an **if** statement is always executed by all threads, but each statement in the body can be *enabled* or *disabled* for a given thread, depending on the value of a *predicate*—a thread-local Boolean variable initialized with the **if** guard; when a statement is disabled, it has no effect. Similarly, a **while** loop is executed the same number of times by all threads: it iterates as long as the loop guard holds for at least one thread. Due to synchronous predicated execution, at any point at run time all threads are always executing the same statement (which, however, might be disabled for some threads). More formally, we can define an execution *trace* as a sequence of instructions $\langle \ell_1, \vec{p_1} \rangle, \ldots, \langle \ell_n, \vec{p_n} \rangle$, where each ℓ_i is the label of the statement being executed and each $\vec{p_i} = [p_i^1, \ldots, p_i^T]$ is a Boolean vector of predicate values (here T is the total number of threads). A kernel's set of *feasible traces* can be derived from the SDV operational semantics.

Races and synchronization. Delayed visibility leads to a potential *data race* when two distinct threads access the same shared memory location, and at least one of the accesses is a write. Executing a **barrier** statement makes the effect of all previous writes visible to all threads, eliminating a potential data race with any following reads or writes. More formally, we say that a trace $\ldots, \langle \ell_i, \vec{p_i} \rangle, \ldots, \langle \ell_j, \vec{p_j} \rangle, \ldots$ *exhibits a race between* ℓ_i *and* ℓ_j, if $\text{stmt}(\ell_i)$ and $\text{stmt}(\ell_j)$ are potentially conflicting shared memory accesses and $\forall k \in (i, j) : \text{stmt}(\ell_k) \neq$ **barrier**. We say that a trace *exhibits barrier divergence at* ℓ if it contains a barrier instruction $\langle \ell, \vec{p} \rangle$ that is not uniformly enabled, i.e. if $\text{stmt}(\ell) =$ **barrier** and $\exists t, u : p^t \neq p^u$. A kernel is *correctly synchronized* if none of its feasible traces exhibit races or barrier divergence. We define the *kernel synchronization problem* as follows: given a KPL kernel without barriers, find a subset $\mathcal{L}$ of its labels, such that inserting a **barrier** statement as the right sibling of every $\ell \in \mathcal{L}$ yields a correctly synchronized kernel.

[3]Here each statement is labeled with its line number or line span.

```
                                  local p
if e {    local p   | while e {   p := e
  s1        p := e  |   s1        while ∃ t : t.p {
  s2        p => s1 |   s2          p => s1
}           p => s2 | }             p => s2
                                    p => p := e }
```

Fig. 9: Predicated form of conditionals (left) and loops (right)

B. Placement Constraints

Our approach to solving the kernel synchronization problem is to encode the set $\mathcal{L}$ as a solution to a system of Boolean *placement constraints* over the propositional variables L_ℓ, for each label ℓ in the kernel. Placement constraints are derived from race locations provided by the verification oracle. A *race location* is a pair of leaf labels $\langle \ell, \ell' \rangle$, such that $\ell \preceq \ell'$ and there exists a feasible trace tr that exhibits a race between ℓ and ℓ' *or* between ℓ' and ℓ. By definition, to eliminate the race in tr, it is sufficient to add a barrier instruction between the two racing instructions. Our key insight is that, thanks to SDV's synchronous predicated execution, this constraint on the barrier position *in the trace* translates into a constraint on its placement *in the program text*.

Consider a data race at location $\langle \ell, \ell' \rangle$. As we demonstrated in Sec. II, barrier placement depends on whether both racing statements are inside the same loop body. More precisely, we identify two types of data races: a *simple race* and a *loop race*.

Simple races arise when $\text{loops}(\ell) \cap \text{loops}(\ell') = \emptyset$. In this case, all occurrences of ℓ in any feasible trace tr precede all occurrences of ℓ', as illustrated in Fig. 3 (with $\ell = 1, \ell' = 3$). Hence, a simple race can always be fixed by placing a single barrier *anywhere* in the interval between the two racing statements, giving rise to the following placement constraint:

$$\bigvee \{L_i \mid i \in [\ell, \ell')\}$$

Loop races arise when both racing statements are inside a nest of loops of depth $d \geq 1$: $\text{loops}(\ell) \cap \text{loops}(\ell') = \{\ell^1, \ldots, \ell^d\}$. In this case, the occurrences of ℓ and ℓ' in tr are *interspersed*, as illustrated in Fig. 5 (with $\ell = 2, \ell' = 4$). Not all pairs of occurrences are necessarily conflicting, but the race location alone has insufficient information to discern which ones are. For barrier placement, we have $d + 1$ options that separate distinct subsets of conflicting instructions in tr.

The first option is to insert an *intra-iteration* barrier: a barrier inside the interval $[\ell, \ell')$. This barrier will separate every occurrence of ℓ in tr from the occurrence of ℓ' *within the same iteration* of the innermost loop ℓ^d, as illustrated by the red barrier in Fig. 5. Alternatively, we can insert an *inter-iteration* barrier: outside the two racing statements, but directly inside the body of one of their shared loops ℓ^j, $j \in [1, d]$. This barrier will separate every occurrence of ℓ from the occurrence of ℓ' in the previous iteration of ℓ^j, as illustrated by the green barrier in Fig. 5. A combination of an inter-iteration barrier at d and an intra-iteration barrier will separate every pair of

occurrences of ℓ and ℓ' in tr, and hence is guaranteed to fix the race, but this is also the solution with most run-time overhead. In the interest of optimality, our algorithm explores all non-redundant combinations of intra- and inter-iteration barriers.

To this end, for a loop race $\langle \ell, \ell' \rangle$, we introduce additional propositional variables that encode the choice of placement options: $P^0_{\ell,\ell'}$ for the intra-iteration barrier and $P^j_{\ell,\ell'}$ with $j \in [1, d]$ for each inter-iteration barrier. The system of placement constraints for a loop race then includes a guarded constraint for each placement option:

$$P^0_{\ell,\ell'} \Rightarrow \bigvee \{L_i \mid i \in [\ell, \ell')\}$$
$$P^j_{\ell,\ell'} \Rightarrow \bigvee \{L_i \mid i \in [\ell^j_s, \ell) \cup [\ell', \ell^j)\}$$

Since only some placement options actually fix the race, the synthesis engine iterates through all possible P-assignments, calling the oracle to validate the corresponding candidate solution. In each iteration, the placement constraints also contain the negation of each previously encountered invalid P-assignment, including the initial assignment $P_{\ell,\ell'} = \vec{0}$, which corresponds to the input program without barriers. Finally, to avoid exploring redundant placement combinations, we add a constraint $\neg(P^j_{\ell,\ell'} \wedge P^k_{\ell,\ell'})$ for all $j, k \geq 1, j \neq k$, since an inter-iteration barrier in an inner loop always subsumes one in an outer loop.

Divergence. Given a candidate solution with a barrier at ℓ, where $\mathrm{blks}(\ell) = \{main, \ell^1, \ldots, \ell^d\}$, the oracle reports barrier divergence at ℓ, if at least one of $\ell^1, \ldots, \ell^d$ has a thread-dependent guard (in which case the block might not be uniformly enabled for all threads). The synthesis engine responds by extending the placement constraints to disallow barriers inside the innermost block ℓ^d:

$$\bigwedge \{\neg L_i \mid i \in [\ell^d_s, \ell^d)\}$$

In the next iteration, the barrier will be placed outside of ℓ^d; iteration will continue as long as any of the remaining enclosing blocks have thread-dependent guards.

C. Cost Model

Our goal is to design a function $C\colon \mathcal{P}(\mathcal{L}) \to \mathbb{Q}^+$ such that the cost of a barrier placement $\mathcal{L}$ correlates with its overhead on the kernel execution time. Precise static analysis of execution time, however, is a hard problem; hence we opted for a simple cost model that approximates the number of barriers the kernel will encounter during its execution (we evaluate the adequacy of this model empirically in Sec. IV). More precisely, the cost of a placement is the sum of costs of all its barriers, and the cost of an individual barrier depends on the number of its enclosing loops and conditionals:

$$C(\mathcal{L}) = \sum_{\ell \in \mathcal{L}} C(\ell) \quad \text{where } C(\ell) = LC^{|\mathrm{loops}(\ell)|} \times IC^{|\mathrm{conds}(\ell)|}$$

Here, the constants $LC > 1$ and $0 < IC < 1$ conceptually represent the average number of times each loop is executed and the average proportion of times each conditional guard holds. In practice, the algorithm is not very sensitive to the precise values of these constants, since it rarely has to trade-off two solutions with different numbers of barriers. For example, in Fig. 4 (left) with $LC = 100$, $C(1{:}5) = 1$ and $C(2) = 100$.

Algorithm 1 The AUTOSYNC syntheis algorithm

```
1:  procedure SYNTHESIZE(kernel)
2:      S = NEWSOLVER
3:      for ℓ ∈ kernel do S.ASSERTSOFT(¬L_ℓ, C(ℓ))
4:      races = GETRACES(kernel)
5:      for ⟨ℓ, ℓ'⟩ ∈ races do
6:          {ℓ^1, ..., ℓ^d} = kernel.loops(ℓ) ∩ kernel.loops(ℓ')
7:          S.ASSERT(P^0_{ℓ,ℓ'} ⇒ ∨L_i | i ∈ [ℓ, ℓ'))
8:          for j ∈ [1, d] do
9:              S.ASSERT(P^j_{ℓ,ℓ'} ⇒ ∨L_i | i ∈ [ℓ^j_s, ℓ) ∪ [ℓ', ℓ^j))
10:             for k ∈ [1, d], k ≠ j do
11:                 S.ASSERT(¬(P^j_{ℓ,ℓ'} ∧ P^k_{ℓ,ℓ'}))
12:         S.ASSERT(∨P^j_{ℓ,ℓ'} | j ∈ [0, d])
13:     return REFINE(kernel, races, S)

14: procedure REFINE(kernel, races, S)
15:     kernel' = kernel
16:     while races ≠ ∅ do
17:         if S.CHECK = UNSAT then
18:             return "No solution"
19:         L = S.MODEL
20:         kernel' = INSERTBARRIERS(kernel, L)
21:         divs = GETDIVERGENCES(kernel')
22:         if divs ≠ ∅ then
23:             for ℓ ∈ divs do
24:                 l^d = last(kernel'.blks(ℓ))
25:                 S.ASSERT(∧¬L_i | i ∈ [ℓ^d_s, ℓ^d))
26:         else
27:             races = GETRACES(kernel')
28:             for ⟨ℓ, ℓ'⟩ ∈ races do
29:                 S.ASSERT(∨(P^j_{ℓ,ℓ'} ≠ L[P^j_{ℓ,ℓ'}]))
30:     return kernel'
```

D. Algorithm

Algorithm 1 describes the full barrier synthesis algorithm. The top-level procedure, SYNTHESIZE, takes as input a KPL $kernel$ and returns a correctly synchronized version of this kernel (or fails).

Initialization. We start by creating a fresh instance of a MaxSAT solver S and asserting soft constraints that penalize a barrier after any label ℓ proportionally to its cost (line 3). In lines 4–12, we query the oracle for the initial set of race locations $races$, and then generate initial placement constraints for each race. For a loop race at depth $d \geq 1$, we generate guarded placement constraints for an intra-iteration barrier (line 7) and all possible inter-iteration barriers (line 9); additionally, line 11 disallows redundant placements (multiple nested inter-iteration barriers), and line 12 forces the solver to place at least one barrier for the current loop race, since

the solution with an empty set of barriers is known to be incorrect. When $d = 0$, we are dealing with a simple race; in this case, lines 7 and 12 together generate an appropriate placement constraint.

Refinement loop. After asserting the initial constraints we invoke REFINE. This procedure alternates between asking the solver for a placement $\mathcal{L}$ that satisfies the current constraints and asking the oracle whether $\mathcal{L}$ is valid; if not, the constraints are refined to exclude $\mathcal{L}$ *and* equivalent invalid placements.

The refinement loop starts by asking the solver whether the current set of placement constraints is satisfiable (line 17). If not, the algorithm terminates with failure; otherwise the least-cost placement $\mathcal{L}$ is obtained as the model of the constraints (line 19). Next, INSERTBARRIERS builds a candidate solution $kernel'$ by inserting a **`barrier`** statement as the right sibling of every $\ell \in \mathcal{L}$ into $kernel$. We assume that INSERTBARRIERS leaves the labels of existing statements unmodified and assigns fresh labels to the barrier statements.

On line 21 we query the oracle for the set $divs$ of barrier divergence locations in the candidate solution. If the barrier at label ℓ is diverging, we can safely exclude all statements in ℓ's innermost enclosing block from consideration (line 25).

In the absence of divergence, we query the oracle for the remaining set of $races$ (line 27). Note that each of these races $\langle \ell, \ell' \rangle$ must be a loop race for which the solver chose an invalid assignment to $P^j_{\ell,\ell'}$. In response, on line 29, we add a constraint that disables the current P-assignment, which prompts the solver to look for the next best combination of placement options in the next iteration.

The procedure terminates either when it finds a valid placement ($races = \emptyset$) or when the current constraints are unsatisfiable (line 18). The latter can happen for two reasons: (1) a race is of the form $\langle \ell, \ell \rangle$—a **`wr`** statement racing with itself—so the disjunction in line 7 is empty, or (2) a race is inside a block with a thread-dependent guard, so the divergence constraint in line 25 is inconsistent with the other placement constraints for this race. Such races cannot be eliminated by inserting barriers, and hence are out of scope.

E. Guarantees

Soundness. A synchronization synthesis algorithm is *sound* if every solution it returns is correctly synchronized. Since Algorithm 1 relies on the oracle to validate candidate placements, we obtain the soundness guarantee for free as long as the oracle is sound (which is true for GPUVERIFY).

Completeness. A synchronization synthesis algorithm is *complete* if it returns a valid placement as long as one exists. Algorithm 1 is *complete relative to the oracle*: it will discover a placement as long as there is one that the oracle can verify.

Proof. Consider a feasible trace tr that exhibits a race between its i-th and j-th instructions. If this race can be eliminated by barrier placement, it must be that $\exists k \in [i, j) : tr[k] = \langle \ell_k, \vec{1} \rangle$, i.e. a uniformly enabled instruction occurs between i and j, so the barrier can be inserted after ℓ_k. Let us define the set F_{ℓ_i,ℓ_j} of *feasible labels* as follows:

$$F_{\ell_i,\ell_j} = L^d \cap \begin{cases} [\ell_i, \ell_j) & \text{if } \ell_i \prec \ell_j \\ [\ell^d_s, \ell_j) \cup [\ell_i, \ell^d) & \text{if } \ell_j \prec \ell_i \end{cases}$$

where L^d is the set of children of $\mathrm{last}(\mathrm{blks}(\ell_i) \cap \mathrm{blks}(\ell_j))$. In other words, feasible labels are labels in the smallest enclosing block of the two racing instructions, which occur between i and j in the trace. Note that we can safely pick any label from F_{ℓ_i,ℓ_j} as the race solution ℓ_k, because (a) F_{ℓ_i,ℓ_j} is nonempty according to trace semantics, and (b) its labels are the least nested in $[i, j)$, hence they must be uniformly enabled if any $[i, j)$ instructions are.

We can now show that if every trace has a verifiable solution ℓ_k, then the constraints generated by Algorithm 1 never become inconsistent. We build a (non-optimal) model $\mathcal{L}$ of the placement constraints as follows: for every $\langle \ell_1, \ell_2 \rangle \in races$

$$\mathcal{L}[P^j_{\ell_1,\ell_2}] \iff j = 0 \vee j = d$$
$$\mathcal{L}[L_\ell] \iff \ell \in F_{\ell_1,\ell_2} \vee \ell \in F_{\ell_2,\ell_1}$$

This model obviously satisfies line 12; it satisfies lines 7 and 9 by definition of $F_{\ell,\ell'}$; it satisfies line 25 because labels in $F_{\ell,\ell'}$ cannot be divergent if a valid placement exists; finally, because we include at least one feasible label for both orderings of labels in every race, $\mathcal{L}$ is guaranteed to eliminate all races, hence no further constraints will be added in line 29. □

As explained in [1], the SDV semantics is *conservative*; in particular, it rejects some barrier placements that could be considered valid if we made more assumptions about the concrete GPU platform. Our synthesis algorithm benefits from SDV in two ways: on the one hand, soundness wrt. SDV guarantees that the resulting kernel will execute correctly on any GPU platform; on the other hand, SDV's synchronous predicated execution helps us prune the search space while maintaining relative completeness.

Termination Procedure REFINE terminates because every iteration eliminates at least one assignment to the propositional variables, and the number of variables is defined by the size of the original kernel. Moreover, the number of iterations is upper-bounded by $2 \times \sum_{\ell \in leaves} \mathrm{depth}(\ell)$.

Optimality. A synchronization synthesis algorithm is *optimal* (relative to a given cost metric) if it always finds the solution with the lowest cost among all valid solutions. Since Algorithm 1 relies on a MaxSAT solver to perform the search, and thanks to the soft constraints in line 3, it always finds the placement $\mathcal{L}$ with the minimal cost $C(\mathcal{L})$ among all models of the placement constraints. Not all valid placements, however, satisfy the constraints. Consider the following snippet:

```
1 for(i=0;i<20;i++){
2   if (i<10 && tid%2==0) {
3     x = rd(tid+i+1) }
4   if (i<10) {
5     x = x + 1 }
6   if (i<10 && tid%6==0) {
7     wr(tid+i,x) }
8 }
```

Here, the optimal placement—inside the middle **if**-statement—will not be discovered because as discussed above, $5 \notin [3, 7)$. The reason for the exclusion is that without analyzing the **if**-guards we cannot be sure that 5 occurs in every trace between each occurrence of 3 and 7. Hence, we do not provide a theoretical guarantee of optimality, but we have not encountered such examples in practice.

IV. Implementation and Evaluation

We have implemented the technique from Sec. III in a prototype tool called AUTOSYNC. The implementation comprises 650 lines of Python code, and uses GPUVERIFY (revision 1937) and Z3 (version 4.6.0).

A. Research Questions

Our empirical evaluation aims to answer the following research questions:

1 Is AUTOSYNC effective at synthesizing correct barrier placements?
2 Are the placements synthesized by AUTOSYNC optimal? Does our cost model faithfully estimate execution time?
3 Is AUTOSYNC efficient?

B. Experiment Setup

Benchmark selection. We evaluated AUTOSYNC on the kernels from NVIDIA GPU Computing SDK v5.0 which is used by GPUVERIFY. We have selected 18 benchmarks from this benchmark suite, which (1) contained a barrier in the original program (2) were verifiable by GPUVERIFY within the timeout of five minutes, and (3) did not contain procedures, which are currently not supported by AUTOSYNC.

For each benchmark, we compare the synthesized solution with a *baseline version*, which is correctly annotated with barrier statements by the developer, and can be verified by GPUVERIFY. In addition to the benchmark suite, we designed eight micro-benchmarks that exercise various challenging scenarios for barrier synthesis.

Running AUTOSYNC. For each benchmark, we first remove all barrier statements from the baseline version, pass the resulting program to AUTOSYNC, and check whether the barrier synthesis succeeded. If so, we manually compare the generated output with the baseline in terms of the number of barriers and their cumulative cost according to our cost model.

We also developed a *naive version* of barrier synthesis, which uses brute-force enumerative search. The naive version first inserts a barrier after each statement in the input kernel, then removes all barriers that lead to divergence, and finally, iterates over the remaining barriers, removing each barrier unless that causes a data race. The naive method is guaranteed to correctly synchronize the kernel, but it requires many more calls to the oracle, and serves as a baseline in our evaluation of the AUTOSYNC's synthesis times.

All experiments were conducted on a machine with Intel i7-4700MQ CPU @ 2.40GHz and 8 GB RAM. Each timing presented in the results is the median of three runs. The cost model is evaluated on a p2.xlarge instance of AWS which runs the GPU kernels on NVIDIA K80 GPU.

Benchmark	LoC	N-V	AS-V	N-B	AS-B	N-Time	AS-Time
1-1-If.cu	11	6	4	1	1	7.1	3.6
1-1-loop:inter.cu	10	10	2	1	1	11.5	1.1
1-1-loop:intra.cu	7	6	3	1	1	9.5	3.1
1-1-main.cu	5	6	2	1	1	6.6	1.1
2-1-main.cu	6	7	2	1	1	8.0	1.1
2-1-loop-d2-intra.cu	9	8	5	1	1	12.8	6.4
2-2-loop-d2-both.cu	18	15	5	2	2	25.2	6.6
2-2-loop:both.cu	7	7	4	2	2	8.9	4.0

Loc: Lines of Code, N-*: Naive Method, AS-*: AutoSync, -V: Number of calls to GPUVERIFY, -B: Number of Barriers Inserted, -Time: Synthesis Time (sec)

TABLE I: Evaluation on Micro Benchmarks

C. Results

Micro-benchmarks. Tab. I present the evaluation result on the micro-benchmarks written by us. Benchmark names follow the convention "n-m-description.cu", where n is the number of data races present and m is the minimum number of barriers required to correctly synchronize the kernel. We make the following observations about the results:

- All micro-benchmarks are correctly synchronized by both the naive method and AUTOSYNC.
- The number of barriers inserted in the synthesized kernel is the same as the expected minimum number of barriers.
- AUTOSYNC is significantly more efficient than the naive method, because it performs fewer expensive calls to GPUVERIFY.
- AUTOSYNC's synthesis time increases linearly with the number of calls to GPUVERIFY, which in practice is proportional to the maximum nesting depth of loop races present in the kernel. On the other hand, the synthesis time of the naive method increases almost linearly with the size of the input program.

Original Benchmarks. The results of evaluating AUTOSYNC on the 18 original benchmarks from NVIDIA SDK are presented in Tab. II.

- Most of the barriers synthesized by AUTOSYNC were placed at the same or equal-cost position as compared to the baseline version of the kernel.
- For five benchmarks AUTOSYNC was able to generate a *more optimal placement* (with fewer barriers) than the baseline version. After a closer inspection, the reason was that AUTOSYNC treats barrier placement as a global optimization problem instead of handling each barrier independently (as the programmer likely would).
- In practice, AUTOSYNC requires very few iterations of the refinement loop (2–4), since the race locations are not nested very deeply; consequently the synthesis time does not necessarily grow with the size of the program.

Cost Model. We performed an experiment to evaluate the adequacy of the cost model we proposed Sec. III-C. We wrote multiple programs which were a combination of loops and conditionals and added barriers at different locations. We then measured the time taken by the kernels containing the barrier at different cost positions and generated the run time vs cost

Benchmark	LoC	V	O-B	AS-B	Time
convolutionColumnsKernel.cu	55	2	1	1	13.1
convolutionRowsKernel.cu	57	2	1	1	8.4
d_transpose.cu	26	2	1	1	1.1
imageDenoising_nlm2_kernel.cu	88	2	1	1	1.9
matrixMul.cu	72	4	2	2	15.5
mergeHistogram256Kernel.cu	25	3	1	1	3.2
mergeHistogram64Kernel.cu	26	3	1	1	3.3
reduce0.cu	26	3	2	1	3.3
reduce1.cu	28	4	2	2	6.1
reduce2.cu	30	4	2	1	5.6
reduce3.cu	32	3	2	1	3.2
reduce5.cu	94	3	4	3	10.1
reduce6.cu	104	4	4	3	15.8
sobol.cu	93	3	1	1	19.9
sum0.cu	25	3	2	2	4.7
sum1.cu	22	3	2	2	4.6
uniformUpdate.cu	17	3	1	1	2.6
uniform_add.cu	17	3	1	1	2.4

Loc: Lines of Code, V: Number of calls to GPUVERIFY, O-B: Number of Barriers in the original benchmark, AS-B: Number of Barriers in the synthesized program, Time: Synthesis Time (sec)

TABLE II: Evaluation of original Benchmarks

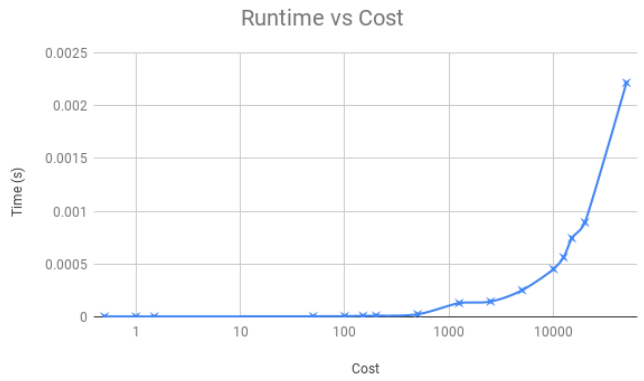

Fig. 10: The run-time overhead (sec) of placing barriers at different costs. The cost of barrier is computed as the cost model discussed above where LC=100 and IC=0.5 and every loop performs 100 iterations.

graph (Fig. 10). We can clearly see from the graph that the run time increases rapidly with the cost of the barrier placement. This graph suggests that the cost model described in Sec. III-C correlates well with the actual run-time overhead of barrier placement.

D. Threats to Validity

Out of the 18 benchmarks in our evaluation, 8 contained some user-provided invariants which were essential for GPUVERIFY to successfully verify the kernel. We believe it is fair to use these annotated programs because our tool is agnostic to the choice of oracle, and we hope that as invariant inference improves, our tool will become fully automatic. In addition, all our benchmarks are obtained by removing barriers from correctly synchronized kernels; hence a valid barrier placement always exists. In general, synchronization is not limited to placing barriers and might require more substantial changes to the code; such changes are out of the scope for our technique.

V. Related Work

Synchronization synthesis for various concurrency models is a rich and active area of research. Prior work focused on traditional shared memory concurrency [9], [10], [11], [12], [13], [14] and network programs [15]. To our knowledge, AUTOSYNC is the first tool to perform synchronization synthesis for GPUs. GPUs are an interesting new domain for this line of work, because of the subtleties of the concurrency model, such as barrier divergence. Our technique shares similarities with [13], which also uses MaxSAT to find an optimal synchronization placement.

An important difference between AUTOSYNC and prior work in this area, is that we use an off-the-shelf verifier as a correctness oracle and define the minimal interface between the search engine and the oracle—data race locations—that still supports efficient synthesis. This design decision gives us soundness for free and allows AUTOSYNC to automatically leverage any future advances in GPU verification technology.

Code generation. A complementary approach to automatic synchronization is to compile a high-level language into GPU code [16], [17]. This approach works well when the high-level language matches the task at hand, but falls short if the programmer needs to hand-optimize the low-level GPU code.

Race detection for GPU kernels is also an extremely active research area [1], [6], [2], [3], [4], [5]. As mentioned in the introduction, these techniques can detect a missing barrier, but do not help the programmer find an optimal placement for the barrier. In this paper we show how to leverage these verification techniques as correctness oracles for synchronization synthesis. Even though our implementation uses GPUVERIFY [6], it can be adapted to work with any sound verification engine that uses predicated execution semantics and reports race locations and divergent barriers.

VI. Conclusions

We have presented a technique for automatically inserting barrier synchronization in GPU kernels. Our main contribution is two-fold. First, we show how to reuse an existing verifier as a correctness oracle and still achieve efficient synthesis by leveraging error information from failed verification attempts. Second, we show how to combine this error information with information about program structure to encode the search for an optimal barrier placement as a MaxSAT problem.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback, as well as Jeroen Ketema and Alastair Donaldson for their help with the GPUVERIFY benchmarks.

REFERENCES

[1] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: A verifier for gpu kernels," in *OOPSLA*, 2012.

[2] G. Li and G. Gopalakrishnan, "Scalable smt-based verification of gpu kernel functions," in *FSE*, 2010.

[3] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "Gklee: Concolic verification and test generation for gpus," in *PPoPP*, 2012.

[4] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner, "Verifying gpu kernels by test amplification," in *PLDI*, 2012.

[5] S. Blom, M. Huisman, and M. Mihelčić, "Specification and verification of gpgpu programs," *Sci. Comput. Program.*, vol. 95, no. P3, Dec. 2014.

[6] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson, "The design and implementation of a verification technique for gpu kernels," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 3, pp. 10:1–10:49, May 2015.

[7] N. Bjørner, A. Phan, and L. Fleckenstein, "νz - an optimizing SMT solver," in *TACAS*, 2015.

[8] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.

[9] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Program*, 1981.

[10] M. T. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization," in *POPL*, 2010.

[11] P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach, "Efficient synthesis for concurrency by semantics-preserving transformations," in *CAV*, 2013.

[12] P. Cerny, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach, "Regression-free synthesis for concurrency," in *CAV*, 2014.

[13] P. Cerný, E. M. Clarke, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, R. Samanta, and T. Tarrach, "From non-preemptive to preemptive scheduling using synchronization synthesis," in *CAV*, 2015.

[14] A. Gupta, T. A. Henzinger, A. Radhakrishna, R. Samanta, and T. Tarrach, "Succinct representation of concurrent trace sets," in *POPL*, 2015.

[15] J. McClurg, H. Hojjat, and P. Cerný, "Synchronization synthesis for network programs," in *CAV*, 2017.

[16] J. Guo, J. Thiyagalingam, and S.-B. Scholz, "Breaking the gpu programming barrier with the auto-parallelising sac compiler," in *DAMP*, 2011.

[17] "TensorFlow documentation," https://www.tensorflow.org/programmers_guide/using_gpu, 2018.