A Formal Instruction-Level GPU Model for Scalable Verification*

Yue Xing, Bo-Yuan Huang, Aarti Gupta, Sharad Malik
Princeton University

ABSTRACT

GPUs have been widely used to accelerate big-data inference applications and scientific computing through their parallelized hardware resources and programming model. Their extreme parallelism increases the possibility of bugs such as data races and un-coalesced memory accesses, and thus verifying program correctness is critical. State-of-the-art GPU program verification efforts mainly focus on analyzing application-level programs, e.g., in C, and suffer from the following limitations: (1) high false-positive rate due to coarse-grained abstraction of synchronization primitives, (2) high complexity of reasoning about pointer arithmetic, and (3) keeping up with an evolving API for developing application-level programs.

In this paper, we address these limitations by modeling GPUs and reasoning about programs at the instruction level. We formally model the Nvidia GPU at the parallel execution thread (PTX) level using the recently proposed Instruction-Level Abstraction (ILA) model for accelerators. PTX is analogous to the Instruction-Set Architecture (ISA) of a general-purpose processor. Our formal ILA model of the GPU includes non-synchronization instructions as well as all synchronization primitives, enabling us to verify multi-threaded programs. We demonstrate the applicability of our ILA model in scalable GPU program verification of data-race checking. The evaluation shows that our checker outperforms state-of-the-art GPU data race checkers with fewer false-positives and improved scalability.

1 INTRODUCTION

Graphics processing units (GPUs) have become an essential element of computing platforms scaling from mobile devices, personal computers, to data centers. Through a highly parallel multi-threaded programming model and underlying hardware resources, GPUs can significantly accelerate a range of applications, from image processing, scientific computing, to large-scale inferencing. However, writing a correct (high-performance) parallel program for GPUs is notoriously tricky and error-prone. Careless program design may lead to bugs, e.g., data races [3, 7], un-coalesced memory accesses [1], bank conflicts [11], etc., due to subtle interleavings of threads in the presence of GPU synchronization mechanisms. This motivates the need for verification of GPU programs.

*This work was supported by the Application Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA

ICCAD '18, November 5–8, 2018, San Diego, CA, USA

ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240771

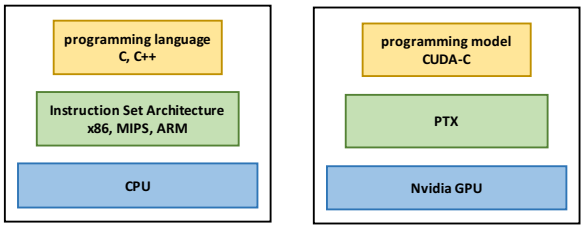

Figure 1: CPU/GPU Programming Stacks

Previous efforts have addressed this problem through static analysis [3, 11] and dynamic checking [19, 20] of code at the program level, e.g., for C programs. They use an abstraction of the application-programming interface (API) of the underlying GPU, such as the CUDA model shown in Figure 1 (right half). While program-level reasoning has achieved useful results, it still faces several challenges. First, the API model typically allows only a subset of synchronization operations. For example, an *acquire-release* pattern inferred by memory accesses with fences is typically not captured, leading to false-positives. Second, pointer analysis with complex data structures is difficult at the program level, often requiring a tradeoff between precision and scalability. Finally, the rapidly evolving expressiveness of the API requires the modeling effort to keep up.

In this paper, we propose modeling GPUs and reasoning about programs at the *instruction level* to address the above challenges. Specifically, we propose a formal model of the Nvidia GPU at the parallel thread execution (PTX) level. As shown in Figure 1, the PTX model for the GPU (shown on the right) is analogous to the Instruction-Set Architecture (ISA) for general-purpose processors (shown on the left). Compared to the program level (C/CUDA) analysis, modeling at the PTX level allows us to specify in detail the hardware thread execution model and synchronization mechanisms. It also naturally captures pointer arithmetic and thus reduces the overhead of software-level pointer analysis for complex data types. Moreover, as a relatively stable hardware specification, PTX is capable of serving more generations of GPUs and more versions of API library releases. Another advantage of PTX-level modeling is better expressiveness, since PTX is a superset of the CUDA programming model. Some PTX instructions (such as *add.cc* and *ld.ca*) cannot be expressed in CUDA, and are used by advanced programmers to achieve high-performance by inlining the bare instructions in programs. While PTX has all these advantages, thus far, it has not been formally specified or used in automated PTX program analysis.

We address this critical gap by formally modeling the GPU at the PTX level, based on recent work on Instruction-level Abstractions (ILA) for accelerators [9]. The ILA models the GPU as a set of instructions. To ease the effort in developing the ILA, we use semi-automatic techniques for template-based synthesis [16] to generate the ILA. Further, we use an actual GPU executing the

target program as an *oracle* (that generates outputs for given inputs) during synthesis, instead of a hardware simulator, as used in previous work [16].

We demonstrate the applicability of our ILA model in scalable GPU program verification by developing a data race checker, where race conditions are detected by using an SMT (Satisfiability Modulo Theory) solver [6]. We evaluate our checker on various benchmarks, including those from state-of-the-art GPU verification tools [5, 7, 8]. These cover different featured instructions in PTX and also several memory/fence synchronizations that have not been covered by previous tools. We also examine the scalability and practicality of our checker by running it on a range of applications from different domains. Experimental results show that our ILA-based checker outperforms state-of-the-art data race checkers with fewer false positives and improved scalability.

Overall this paper makes the following contributions:

- We propose an ILA-based methodology for instruction-level GPU modeling and scalable verification. Specifically, we develop a formal model for the Nvidia GPU at the PTX level, supporting both non-synchronization instructions and all synchronization primitives. To the best of our knowledge, this is the first instruction-level formal model for GPUs.
- We leverage template-based synthesis to generate the GPU model semi-automatically, using a GPU executing the target program as an oracle, instead of a simulator as in past work.
- We demonstrate the benefits of instruction-level PTX modeling in a novel data race checker for GPU programs, which improves upon previous efforts in showing fewer false positives and improved scalability.

The paper is organized as follows. In Section 2, we briefly discuss relevant background on the CUDA programming model, the PTX model, and the ILA. We then describe our instruction-level GPU model and the ILA synthesis framework in Section 3. Section 4 describes the modeling of synchronization primitives, followed by the data race checker in Section 5, evaluations in Section 6, and finally conclusions in Section 7.

2 BACKGROUND

2.1 CUDA Programming Model

CUDA is the application programming interface (API) provided by Nvidia for the development of parallel programs on their GPUs. It provides constructs to define threads, blocks, and kernels. As shown in Figure 2, the kernel is the basic unit of programming that describes the behavior of a single thread and is represented as a C function. All threads execute the same kernel function in parallel, where different thread IDs are assigned to each thread for data partitioning. In GPUs, threads are organized hierarchically where several threads form a block, and all blocks are grouped into a grid of blocks. In general, the programmer specifies the kernel function, the number of blocks, and the number of threads per block. The GPU hardware then manages the execution of the program and memory space. Each thread has its local memory, which is accessible only by that thread. Shared memory can only be used for communication among threads in the same block. There is also global memory for all threads running on the GPU.

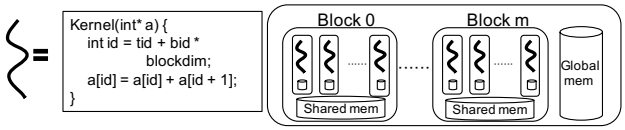

Figure 2: CUDA Programming Model

2.2 Parallel Thread Execution

Parallel thread execution (PTX) is the instruction-level specification that remains stable across multiple GPU versions [13]. Like the ISA of general-purpose processors, PTX specifies the set of instructions supported by the GPU hardware. Most of the PTX instructions are *non-synchronization* instructions, i.e., the operation only depends on the local memory in a thread. In addition, there are instructions that affect inter-thread behaviors, which we refer to as *synchronization primitives*. The operation of synchronization primitives depends on not only thread-local memory, but also on other threads. In PTX, each thread has special registers storing its thread ID and block ID to support the grid-block-thread hierarchy.

A *warp* is a sub-block of threads (typically 32 threads) where threads in the same warp share the same program counter and execute in lockstep. However, we do not assume this lockstep property in our model since the latest PTX [13] and CUDA 9 [12] specifications both disable this feature.

2.3 Instruction Level Abstraction

The Instruction-Level Abstraction (ILA) was recently proposed to provide a uniform abstraction for formally modeling processors and accelerators in heterogeneous computing platforms [9]. An ILA models the hardware behavior as a set of instructions at the architecture level, abstracting implementation-specific details. Here we introduce the ILA model by explaining its primary components. Essentially, an ILA model defines what the architectural (i.e., program-visible) state is, how instructions are fetched and decoded, and how each instruction updates the architectural state. Formally, an ILA model A is defined by a tuple:

$$
\begin{aligned}
\mathrm{A} \quad &= \quad \langle S, I, F, D, N \rangle, \text{ where} \\
& \quad S \quad \text{is a vector of state variables,} \\
& \quad I \quad \text{is a vector of initial values of the state variables,} \\
& \quad F \quad S \rightarrow bvec_w \text{ is a fetch function,} \\
& \quad D \quad = \{\delta_i : bvec_w \rightarrow \mathbb{B}\} \text{ is a set of decode functions,} \\
& \quad N \quad = \{N_i : S \rightarrow S\} \text{ are the next state functions.}
\end{aligned}
$$

For processors/GPUs, ILAs model the ISA/PTX level instructions. For accelerators, the instructions are commands at the accelerator interface. This formal model enables reasoning about programs comprising the instructions.

3 INSTRUCTION-LEVEL GPU MODEL

In GPUs, a large number of threads run in parallel where each thread has local state and communicates/synchronizes with others through shared state. Our GPU model is the composition of multiple **per-thread models**, in which we define the set of non-synchronization instructions, as shown in Figure 4. The synchronization primitives then define the interaction between threads.

```
kernel(.param .u64 p0) {
 // directive statements
 .reg .b32 %r<6>;
 .reg .b64 %rd<5>;

 mov %r1, %tid.x;
 // ...
 ld.global %r2, [%rd4+4];
 ld.global %r3, [%rd4];
 // ...
 st.global [%rd4], %r5;
 ret;
}
```

Figure 3: PTX Code

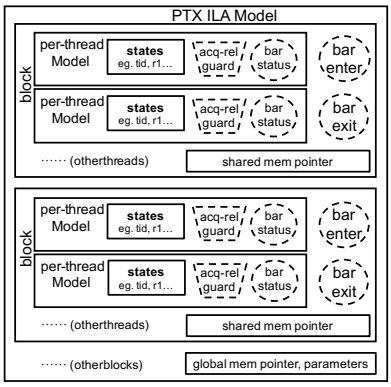

Figure 4: GPU Model Structure

In this section, we describe our modeling of non-synchronization instructions in the per-thread model. The synchronization primitives will be discussed separately in Section 4.

3.1 GPU Architectural State

The main difference between PTX programs in GPUs and assembly programs in general-purpose processors is that state variables for registers in PTX are program-specific and not architecture-specific. For example, Figure 3 shows a PTX program that starts with *directive statements* that specify the required registers in a thread. Similar to declaring variables in a C program, these registers map to hardware resources and are considered *program-visible architectural state* in our GPU model.

For each PTX program, we automatically extract the architectural state from these directive statements. Table 1 shows the three types of architectural states associated with a PTX program: *general purpose registers*, *special registers*, and *parameter registers*. *Special registers* are pre-defined in the PTX specification. For example, *%tid* and *%bid* are used to distinguish threads and blocks, and are modeled with an additional constraint during verification – for any two threads, either *%tid* or *%bid* is different. *Parameter registers* store input parameters or parameter pointers, which are read-only and shared by all threads. Note that pre-defined pointers to shared/-global memory are similar to pointers to input parameter pointers, so they fall in the same category.

3.2 Non-Synchronization Instructions

Modeling non-synchronization instructions in a per-thread model is similar to modeling general-purpose processors, as has been done for the RISC-V base instruction set in prior work [9]. The difference is that here we include the thread ID and block ID in the decode function (guards for instructions) to differentiate between threads. We modeled both the arithmetic and control-flow instructions (e.g., add and predicated branch), which together constitute the non-synchronization instruction set. For complex instructions like floating point operations, we modeled them as uninterpreted functions during verification, since these are not directly or indirectly used as synchronization variables.

3.3 GPU Memory Model

Shared memory in GPUs has two primary uses: exchanging intermediate data and synchronizing between threads. A precise memory model should capture the detailed consistency model and present all possible values a load instruction can receive for different execution traces. However, an explicit model of memory, with support for indirect memory addresses, would not be scalable for reasoning. Therefore, we use an over-approximate memory model for non-synchronization instructions. Specifically, we model the load instruction as a nondeterministic function that returns an arbitrary value. Note that the over-approximation is applied only to non-synchronization instructions (for exchanging intermediate data) but not to synchronization primitives.

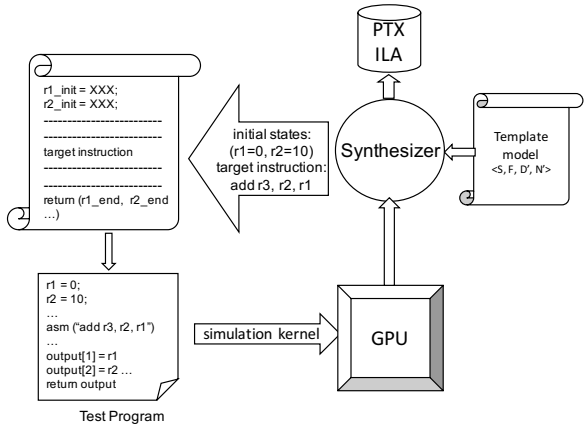

Figure 5: ILA Synthesis with Real GPU as the Oracle

3.4 ILA Synthesis for a GPU

Handcrafting an ILA model for a GPU that contains a large number of instructions is time-consuming and error-prone. Here we leverage the counter-example guided inductive synthesis (CEGIS) technique to help construct the ILA model [15, 16]. Given a partially defined model (i.e., a template), the CEGIS engine can synthesize the complete model by querying an *oracle* that generates the GPU state (oracle output) for a given instruction and state (oracle input). This oracle is usually a formal specification or a reference simulator of the design. However, although several GPU simulators have been proposed, many of them get out-of-date as new versions of GPUs are released [2, 17]. Thus, in this work, unlike previous work which used a simulator as an oracle, we use an actual GPU running the target program as the oracle. As shown in Figure 5, our synthesis tool converts every query from the CEGIS engine into a *test program*. When executed on the GPU, the test program first initializes the architectural state, executes the target instruction, and then writes back the updated state after the instruction completes. We implemented our synthesis tool on top of an existing CEGIS engine [16], along with an Nvidia Tesla K20 GPU; and successfully synthesized all non-synchronization instructions.

4 SYNCHRONIZATION PRIMITIVES

Synchronization primitives in PTX guarantee the *ordering* between instructions of different threads, and we model this using the happens-before relation [10]. There are two kinds of synchronization, namely *strong* and *relaxed*, supported in the PTX model. Strong synchronization (e.g., barrier) requires one specific instruction to happen before another. Relaxed synchronization, on the other hand, requires only

Table 1: PTX Architectural State

Nvidia PTX Spec.	ILA Model	Feature
general reg	state (bv)	Program specific general-purpose registers (with bit-length 64/32/16/8)
special reg	state (bv)	Specialized read-only register for distinguishing threads, e.g., %tid and %ntid
.param, .global, .shared pointers	state (bv)	Kernel input/output pointer; program defined shared/global memory pointer; read-only; same value for all threads (.shared pointer is for intra-block threads)

that the two instructions are at least ordered, but may not specify the ordering, which is determined at runtime. In this section, we discuss our modeling of the synchronization primitives using the ILA model. We first explain a counter-based model for the barrier instruction, and then present modeling of the memory-inferred synchronizations.

4.1 Modeling Barrier Synchronization

Barrier synchronization provides block-width synchronization among threads in the same block, where all threads run the same kernel function. As shown in Figure 6 (where instructions go top to down, and time goes from left to right), a barrier ensures that no thread can proceed further unless all threads have reached the barrier instruction, i.e., the common synchronization point. This property ensures that all instructions before the barrier *happen-before* the instructions after the barrier.

To model the barrier synchronization, we propose a counter-based model that uses two auxiliary state variables, the *bar_enter* counter, and the *bar_exit* counter. As indicated by its name, *bar_enter* is used to count how many threads have reached the barrier. As long as *bar_enter* is not equal to the total number of threads in the block, no thread can proceed. Likewise, *bar_exit* counts the number of threads that have left the barrier, i.e., have continued down the program. It is used to reinitialize the barrier for reuse, after the last thread leaves the barrier.

In addition to these two counters, each thread has a control state *status* that can have three possible values: *init*, *wait*, and *exit*. Based on the value of *status*, the behavior of each thread when fetching a barrier instruction is defined as follows:

(1) If the thread first reaches the barrier (i.e., *status* is *init*), it increments *bar_enter* by 1 and changes *status* to *wait*. The program counter remains unchanged.
(2) If the thread is waiting for other threads (i.e., *status* is *wait*), it changes *status* to *exit* if *bar_enter* is equal to the total number of threads. Otherwise, all states remain unchanged.
(3) If *status* is *exit*, meaning all other threads have reached the barrier, it increments *bar_exit* by 1, resets *status* to *init*, and updates the program counter to the next instruction.

The last thread reaching/leaving the barrier is in charge of resetting *bar_exit*/*bar_enter* to 0 for the next use, respectively. Barrier modeling states, i.e., *bar_enter*, *bar_exit*, and *status*, are shown as dotted circles in Figure 4.

To model PTX programs with multiple barriers, we extend our model by adding copies of counter-pairs and assigning them unique barrier IDs. Note that, in practice, only barrier 0 will be used (although the PTX specification allows up to 16 barriers [13]), because only one barrier can be compiled in the latest CUDA release [12].

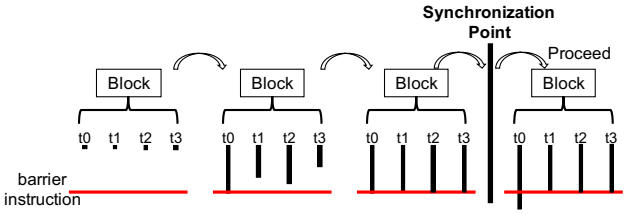

Figure 6: Barrier Instruction Illustration

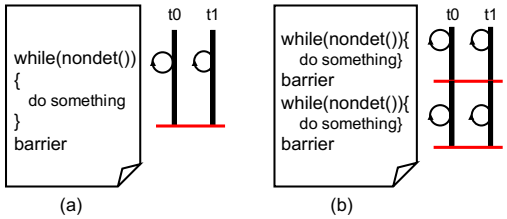

Figure 7: Generalized Programs with Barriers

We checked the correctness of our barrier model with two generalized programs shown in Figure 7. In program (a), the kernel executes some non-synchronization instructions for an arbitrary number of times before reaching the barrier. Program (b), based on program (a), is a cascaded use of multiple barrier synchronizations. Note that the number of iterations of the loops may differ from thread to thread. We first check a *barrier validity* property, which specifies that "if any thread exits the barrier, all other threads should have already reached the barrier." In cases with multiple barriers, we also check *barrier reusability*, where the validity of the last barrier should be guaranteed. The two properties are encoded as safety properties and we checked them using bounded model checking (BMC [4]).

4.2 Memory-Inferred Synchronization

In general-purpose processors, shared memory has been used to implement various synchronization primitives such as locks, mutex, and semaphores. Similarly, GPUs utilize shared memory for lock-based and ticket-based synchronization in practice [5, 12, 18]. Figure 8 shows an example of inferring synchronization ordering using shared memory. Consider two threads, `t0` and `t1`, executing two loads and two stores, respectively. Under sequential consistency (SC), if `i0` loads a value 1 to register `r1`, we can infer that `i1` and `i2` are ordered. Since SC respects program order and has a total order on memory accesses, we can conclude that `i1` and `i2` cannot happen concurrently.

As shown in the previous example, we show how to infer instruction ordering under a sequential consistency memory model. However, GPUs usually have a weak memory consistency model to achieve better performance. Further, even in SC, the ordering

init: [x] = 0; [y] = 0;	
t0	t1
i0: ld [x], r1;	i2: st [z], 10;
i1: ld [y], r2;	i3: st [x], 1;

Figure 8: Inferring Ordering from Shared Memory

may not always be inferrable simply by observing the shared memory. For instance, consider the execution traces (`i0`; `i1`; `i2`; `i3`) and (`i0`; `i2`; `i1`; `i3`). The synchronization between `i1` and `i2` cannot be determined by register `r1` alone.

To address the above issues, modern GPU compilers and experienced programmers use extra instructions surrounding the pair of load/store instructions to infer the synchronization ordering. These instructions form *patterns* that are used for inferring memory-based synchronization. For example, fences can be used to restrict memory access reorderings, and loops to ensure synchronization for all traces. Next, we explain how we model these patterns as *acquire-release* pairs, where there is a happen-before ordering from a release to an acquire.

init: [x] = 0; [y] = 0;	
t0	t1
i0: ld [x], r1;	i5: st [z], 10;
i1: set.ne p, r1, 1;	
i2: @p bra i0;	
i3: fence;	i6: fence;
i4: ld [y], r2;	i7: st [x], 1;

Figure 9: Memory-Inferred Synchronization

4.2.1 Patterns for Memory-Inferred Synchronization. A pattern is composed of two building blocks: (1) a fence that restricts memory access reordering, and (2) a code structure ensuring synchronization for all traces, e.g., a loop.

Figure 9 shows a synchronizing pattern based on the example in Figure 8. The goal is to synchronize the ordering between instructions `i4` and `i5`. In thread `t0`, the instruction `i0` is guarded by a loop (the code structure) so that `t0` can exit the loop only when `i0` loads value 1 to the register. Meanwhile, fences are inserted between the two loads (`i0` and `i4`) and between the two stores (`i5` and `i7`). It ensures that the two memory accesses in the same thread follow the program order. Note that location `[x]` serves as a lock here, which provides the synchronization between threads.

We list various code structures used in synchronization, such as ticket-based and lock-based mechanisms, in Figure 10. Note that *ld*, *st* and *fence* here denote syntactical representatives for actual PTX instructions. The PTX ISA has a fine-grained set of *ld*, *st*, and *fence* instructions with different strengths of reordering in the consistency model distinguished by suffixes, e.g., *sc* and *relaxed*. However, only certain combinations of *fence/st* and *ld/fence* are strong enough to enforce an order. In Figure 11, we list all the valid combinations of fine-grained instructions that can be used in the synchronization patterns shown in Figure 10.

4.2.2 Modeling Acquire-Release Pairs. As per the above discussion for memory-inferred synchronization, each pattern is essentially an instruction sequence that manipulates the shared memory at location `M` (possibly in a loop or with a fence to achieve synchronization). Based on its semantics, we categorize the patterns into two types. The first type has the semantics of storing a particular

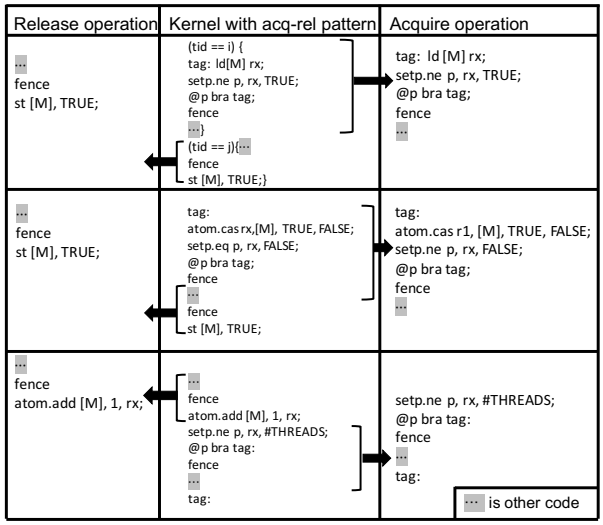

Figure 10: Abstracted Patterns for Synchronization

ld [M] ; fence	fence ; st [M]
ld.relaxed[M]; fence.acq_rel;	fence.acq_rel; st.relaxed[M];
ld.relaxed[M]; ld.acquire[M];	st.released[M]; st.relaxed[M];
ld.acquire[M];	st.release[M];
ld.weak[M]; fence.sc;	fence.sc; st.weak[M];

Figure 11: Valid ld/st + fence Combinations for Synchronization Patterns

value to location `M` and is called a *release* operation on `M`. The other type is to load the value from location `M` and is called an *acquire* operation on `M`. We say an instruction is *guarded* by `M` if it appears between an acquire `M` and a release `M`, or between the program entry point and a release `M`, or between an acquire `M` and the program exit point. A release-acquire pair in the program is used to ensure that *two instructions from different threads are synchronized if they are both guarded by the same location* `M`.

We model memory-inferred synchronization patterns as either an acquire operation or a release operation, as shown in Figure 10. For each instance of an acquire operation in the program, we add a guarding state *guard* to the model. An acquire operation updates *guard* to its guarding address (location `M`). A release operation checks and resets *guard* if it matches its guarding address. When analyzing synchronization of instructions in different threads, guarding states are checked to see if they store the same value.

5 DATA RACE CHECKING FOR GPUS

We demonstrate the use of our formal instruction-level GPU model in a verification application for data race checking. We formalize the detection of a race as a safety property over the architectural states in the GPU model and use bounded model checking [4] to find violations. The overall verification flow is the same as for standard property verification, and we describe details of our prototype implementation in this section.

Table 2: Race Condition Formulation

For any two different memory operations (mop_1 and mop_2)	
Formula	Explanation
$p_1 \coloneqq ((tid_1 \neq tid_2) \vert (bid1 \neq bid2))$	Different threads
$p_2 \coloneqq (addr_1 = addr_2)$	Same address
$p_3 \coloneqq ((type_1 = w) \vert (type_2 = w))$	At least one write
$p_4 \coloneqq (\neg(atom_1 \wedge atom2))$	Not both atomic
$p_5 \coloneqq ((bid_1 = bid_2) \rightarrow (bar(mop_1) = bar(mop_2)))$	No barrier synchronization
$p_6 \coloneqq ((acq(mop_1) \cap acq(mop_2)) = \varnothing)$	No memory inferred synchronization

5.1 Problem Definition

A data race is defined as *two threads accessing the same shared memory location concurrently, where at least one of the accesses is a write operation*, i.e., the two memory accesses are not synchronized, either by a barrier or by a memory-inferred *acquire-release* pair. The result of a race is considered undefined at the program or instruction level, thus making it impossible to reason about correctness of the rest of the program. Note that inferring synchronization patterns is crucial in order to reduce false positives. Note also that atomic memory operations are excluded from race checking, because two atomic operations are regarded as being mutually exclusive.

5.2 Formalization of Race Condition

Our main approach for detecting a data race is to focus on memory operations and represent them symbolically. We define a memory operation as a tuple: *mop*(*tid*, *bid*, *addr*, *type*, *atom*). The *tid* and *bid* identify the thread that executes the operation, *addr* is the memory address, *type* indicates whether it is a *ld* or *st*, and *atom* specifies whether it is an atomic operation. Each *ld*, *st* instruction in each thread corresponds to such a memory operation. In addition, we define two auxiliary functions to describe the synchronization for each memory operation: *bar()* is a function from *mop* to integer, indicating how many barrier synchronizations have been finished before the *mop*. *mop*s with different *bar()* values are separated by at least one barrier, and can therefore never occur concurrently. *Acq()* is a function from a *mop* to a set of variables that denote the memory-inferred synchronizations that have been acquired but not yet released.

Our formulation of a race condition is shown in Table 2. Here, the first three formulas (p_1, p_2, p_3) capture two conflicting accesses from different threads, where at least one access is a write. The fourth formula (p_4) excludes atomic operations. The remaining two formulas (p_5, p_6) correspond to no barrier or no acquire-release, respectively, between the two *mop*s.

The race condition, shown below, says that there exist two mops such that the conjunction of all formulas from Table 2 is satisfiable:

$$\exists mop_1, mop_2 \text{ s.t. } (mop_1 \neq mop_2) \wedge p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \quad (1)$$

5.3 Implementation of Race Checker

We use two sets of symbolic variables to represent any pair of *mop*s, and the race condition is directly encoded using these variables. As mentioned in 3.1, *tid* and *bid* are states in each thread model. To capture the other elements in the race condition, we add a list of auxiliary state variables to a per-thread model: *aux_address*, *aux_atom*, *aux_type* and *aux_acq*. Correspondingly, they are used to record a *mop* for that thread. The memory scope (global/shared) is also encoded as an extra bit in the *aux_address*. In particular, *aux_acq* is used to capture acquire-release synchronization. In practical programs, the number of *acquire-release* memory operations is statically analyzable as in 4.2, and is usually no more than 2. Therefore, the number of auxiliary state variables is also statically determined. For all programs we have seen, the *acquire-release* patterns are not nested; thus, only one set of auxiliary state variables is enough to capture memory-inferred synchronization.

Searching over all mops. When a *mop* is fetched and decoded in the ILA model, the information for that *mop* can be recorded in these auxiliary state variables. In other words, these auxiliary state variables can capture the required information about the *most recently fetched mop*. To represent any possible *mop* in a thread, we *nondeterministically* choose whether to update these at a particular *mop*. Thus, based on the nondeterministic selection, the auxiliary state variables will represent the information for *any* one of the *mop*s that have executed. As discussed later, when an SMT solver is used to check the race condition, all nondeterministic choices, i.e., all *mops*, are searched implicitly. We have also implemented a two-thread reduction to improve the scalability, similar to [3, 11].

Handling barriers. As shown in *p5*, a pair of *mops* can possibly race only if they are mapped to the same value by *bar()*. Hence, we only need to check *mops* pairs which meet that condition. To achieve this, we implement the checker by disabling all recorded *mops* when threads are synchronized at a barrier. Since our model of the barrier instruction always has a synchronization point such that all threads wait at that point before any of them can proceed, this checker implementation guarantees that all checked *mops* pairs are not separated by any barrier, which matches *p5*. Note that, barrier only synchronizes for threads within a same block, for those in different blocks, the barrier does not affect, and all *mops* will be checked.

Bounded model checking (BMC). Finally, we use BMC [4] to check whether the race condition can be satisfied. BMC unrolls the GPU model up to a bounded length, and each unrolling goes through the fetch, decode, and state updates corresponding to the PTX instructions. The race condition formula is checked at each synchronization barrier point and at the end of the program. For programs with loops, we first unwind loops up to a given iteration count (we use 5 by default). After unrolling the model, we use the SMT solver Z3 [6] on the resulting formula. A satisfying solution corresponds to an error trace starting from the initial states, followed by a sequence of state transitions until a data race is detected.

6 EXPERIMENTAL EVALUATION

We evaluated our data race checker based on the GPU PTX-level model.[1] All experiments were done on a 2.50 GHz Intel Ivybridge processor, 50GB of RAM, running Springdale Linux 7.4. Our PTX programs were compiled using *nvcc* from CUDA 9 Toolkit, version 9.0.176. We used z3 v4.5.1 as the SMT solver.

[1]The GPU PTX-level model and the data race checker are publicly available on https://github.com/yuex1994/ICCAD2018_submission .

Table 3: Results for Small Test Examples [7]

kernel	# of Variants	Ave. LoC	Ave. Check Time(s)
\atomic	6	6	0.14
\base ld/st	10	16	0.2
\barrier	6	29	2.9
\branch	6	22	0.3
\sync block	5	23	0.45
\intra warp	4	13	0.6
\lock	20	21	1.02

6.1 Results

We requested the Barracuda test suites from authors of the paper [7]. There are 57 small race/race-free CUDA-C (some with inline PTX assembly) kernels in total, including various featured bugs such as races in different memory spaces, races inside a branch, missed barrier, memory-inferred synchronization, as shown in Table 3. Our checker correctly identifies all buggy programs and passes all correct kernels. One difference between our checker and Barracuda is that we do not assume warp-level lockstep as discussed in Section.2.2. Therefore, intra-warp races are also detected by our checker.

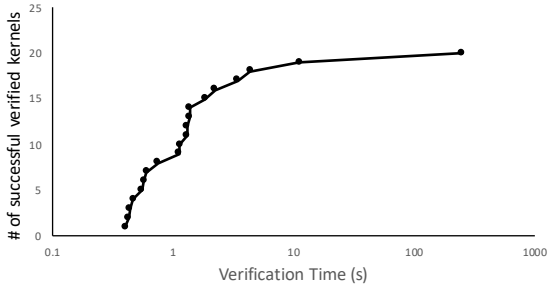

Figure 12: Cumulative Time (sec) for # of Successfully Verified Kernels from Nvidia SDK

We validated the usefulness of our checker on several other benchmarks: 20 medium-sized kernels from CUDA SDK program demos [14]; *bfs, backprop, gaussian, hotspot, hybridsort, nn, kmeans, particlefilter,* and *pathfinder* from Rodinia suite [5], and *hashtable* from GPU-TM [8]. Some applications include more than one kernel program and we checked each of them individually.

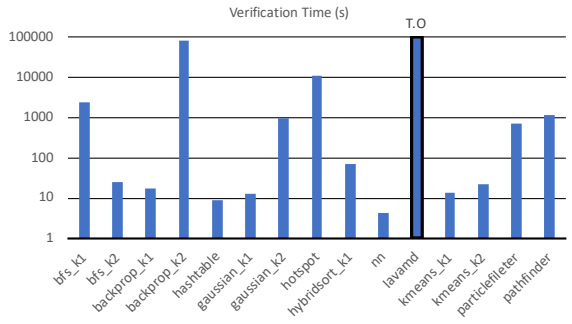

Figure 13: Verification Time (sec) for Practical Applications

The scalability of medium-sized kernels is shown in Figure 12. The verification time is typically less than 10 seconds when the size of the program does not exceed 50 instructions, as is the case for these kernels. Figure 13 shows the verification time for each kernel from Rodinia and GPU-TM, and the corresponding results are in Table 4. Our checker is capable of handling kernels scaling up to hundreds of instructions, and it correctly detects potential bugs for several kernels (detailed discussion in Section 6.2).

6.2 Discussion

We provide details on some interesting benchmark examples and how our PTX-level checker handles them.

hashtable. Our checker demonstrates the following advantages: (1) At the program level, this kernel defines complex data structures – a hash table, hash buckets, hash bucket entries, with pointer arithmetic to select an entry from a hash bucket. Pointer arithmetic is challenging for program level analysis but is captured naturally through program state updates in our PTX ILA model. (2) Memory-inferred synchronization supported by our model is used in this program. With no fences, there is a bug in this program that implements a lock to let threads update each bucket mutually exclusively. However, due to lack of fences, other memory accesses can be reordered even with the lock. In our checker, the lack of fences leads to lack of a *acquire-release* pattern, which we identify as a bug. We corrected the bug by adding valid fences, after which this modified kernel is correctly verified in 3 seconds.

Note that this example cannot be handled by previous race analysis tools such as GPUVerify [3], PUG [11] which check races at the program level. First, they handle arrays by extracting the offset expressions and only consider programs with unified types (e.g., *int*). Further, as the fence (*__threadFence()* API) is beyond the scope of their tools, they do not consider memory-inferred synchronizations, thereby leading to false positives.

Barracuda [7], the state-of-the-art dynamic checker does reason about execution traces with *acquire-release* semantics. They correctly identify the synchronization when memory operations and fences are involved. However, as a dynamic checker, they are limited to checking only one trace at a time, unlike our checker which symbolically covers all traces.

bfs. Our checker identifies two bugs in kernel *bfs*: The first *bfs* kernel uses a shared Boolean flag to compute the disjunction of each thread's result. The flag is implemented as follows: at the beginning of the kernel, it is initialized to be *false*; each thread will either update the flag to *true* or do nothing based on that thread's result. That is, more than one thread can set the flag concurrently. This implementation relies on the assumption that "if multiple threads write the same value to a memory location concurrently, that value will be correctly stored." However, Nvidia GPUs do not guarantee this property and we correctly identified it as a bug. Further, if we relax the checker by ignoring this kind of *store*, we can verify the correctness of the kernel in 10 seconds. (ii) In *bfs*, the second kernel has some races at the kernel level. Our checker identifies that, under some specific inputs, different threads will both load from and store to a common address. This kernel-level race can be avoided under a carefully designed environment that blocks the invalid inputs, as the host program provided in the benchmark does. This can be easily added as an environmental constraint during our verification.

Table 4: Verification Results for Practical Applications

kernel	LoC	Bug Found
bfs_k1	61	Race
bfs_k2	27	Race
backprop_k1	87	False Positive
backprop_k2	75	Correct
hashtable	38	Race
gaussian_k1	33	Correct
gaussian_k2	55	Correct
hotspot	225	Correct
hybridsort_k1	151	Race
nn	29	Correct
lavamd	513	Timeout:100000s
kmeans_k1	33	Correct
kmeans_k2	143	Correct
particlefilter	51	Correct
pathfinder	110	Correct

hybridsort. *hybridsort* has a similar issue that setting specific inputs can avoid certain races. Since we model memory using non-deterministic functions, indirect memory accesses can cause races. For example, memory accesses in $a[b[tid]]$ will cause races in array a because array b can return any value, and different threads' $b[tid]$ can be the same. In the benchmark *hybridsort*, they can have the same indirect memory accesses, which has the vulnerability for a race. When we checked this, we found that the input array b is guaranteed to get different values loaded for different indices, i.e., $b[tid]$ are different under different tid. So, the programmer allows such races in the kernel, since they are avoided by the code outside of the kernel.

backprop. Precise modeling of floating point arithmetic is out of the scope of this paper. In kernel 1 of the *backprop* benchmark, some memory indexes are first calculated as float values then rounded. We substitute those calculation with integer arithmetic, and the kernel was then verified to be correct.

lavamd. This is the only benchmark where our checker could not complete within the time limit. This kernel contains more than 150 load/store instructions in each thread. Even though the checker involves only two threads, this still leads to more than 20000 clauses in the checking formula for each pair of memory accesses! Further improvement can be made by constraining the search space by providing invariants. For example, in the future, we plan to prune redundant checking by exploiting symmetry in the two threads.

7 CONCLUSIONS

This paper presents a formal model of the Nvidia GPU at the PTX level. Our model covers both non-synchronization instructions and synchronization primitives in the PTX specification. This model is built using the recently proposed Instruction-Level Abstraction (ILA) and synthesized using semi-automatic techniques for template-based program synthesis.

We show the usefulness of our formal model by developing a checker for data races in PTX code, where we use bounded model checking on our model and an SMT solver to detect race conditions. Experimental results show that our checker outperforms state-of-the-art data race checkers with fewer false positives and better scalability for programs with pointers and complex data structures.

REFERENCES

[1] Rajeev Alur, Joseph Devietti, Omar S Navarro Leija, and Nimit Singhania. 2017. GPUDrano: Detecting Uncoalesced Accesses in GPU Programs. In *International Conference on Computer Aided Verification*. Springer, 507–525.

[2] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 163–174.

[3] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 113–132.

[4] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. 2003. Bounded model checking. *Advances in computers* 58, 11 (2003), 117–148.

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 44–54.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[7] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. BARRACUDA: Binary-level Analysis of Runtime RAces in CUDA programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 126–140.

[8] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 407–420.

[9] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *arXiv preprint arXiv:1801.01114* (2018).

[10] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

[11] Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 187–196.

[12] Nvidia. 2017. CUDA C Programming Guide 9.1. (2017). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[13] Nvidia. 2017. CUDA Parallel Thread Execution ISA 6.1. (2017). https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[14] Nvidia. 2017. CUDA Toolkit 9.1. (2017). https://developer.nvidia.com/cuda-downloads

[15] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 41. ACM, 404–415.

[16] Pramod Subramanyan, Bo-Yuan Huang, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2017. Template-based Parameterized Synthesis of Uniform Instruction-Level Abstractions for SoC Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).

[17] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: a simulation framework for CPU-GPU computing. In *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*. IEEE, 335–344.

[18] Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.

[19] Mai Zheng, Vignesh T Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: a low-overhead mechanism for detecting data races in GPU programs. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 135–146.

[20] Mai Zheng, Vignesh T Ravi, Feng Qin, and Gagan Agrawal. 2014. Gmrace: Detecting data races in gpu programs via a low-overhead scheme. *IEEE Transactions on Parallel and Distributed Systems* 25, 1 (2014), 104–115.