Sound and Partially-Complete Static Analysis of Data-Races in GPU Programs

DENNIS LIEW, University of Massachusetts Boston, USA
TIAGO COGUMBREIRO, University of Massachusetts Boston, USA
JULIEN LANGE, Royal Holloway, University of London, United Kingdom

GPUs are progressively being integrated into modern society, playing a pivotal role in Artificial Intelligence and High-Performance Computing. Programmers need a deep understanding of the GPU programming model to avoid subtle data-races in their codes. Static verification that is sound and incomplete can guarantee data-race freedom, but the alarms it raises may be spurious and need to be validated.

In this paper, we establish a True Positive Theorem for a static data-race detector for GPU programs, *i.e.*, a result that identifies a class of programs for which our technique only raises true alarms. Our work builds on the formalism of *memory access protocols*, that models the concurrency operations of CUDA programs. The crux of our approach is an *approximation analysis* that can correctly identify true alarms, and pinpoint the conditions that make an alarm imprecise. Our approximation analysis detects when the reported locations are reachable (control independence, or CI), and when the reported locations are precise (data independence, or DI), as well identify inexact values in an alarm. In addition to a True Positive result for programs that are CI and DI, we establish the root causes of spurious alarms depending on whether CI or DI are present.

We apply our theory to introduce FaialAA, the first sound and partially complete data-race detector. We evaluate FaialAA in three experiments. First, in a comparative study with the state-of-the-art tools, we show that FaialAA confirms more DRF programs than others while emitting 1.9× fewer potential alarms. Importantly, the approximation analysis of FaialAA detects 10 undocumented data-races. Second, in an experiment studying 6 commits of data-race fixes in open source projects OpenMM and Nvidia's MegaTron, FaialAA confirmed the buggy and fixed versions of 5 commits, while others were only able to confirm 2. Third, we show that 59.5% of 2,770 programs are CI and DI, quantifying when the approximation analysis of FaialAA is complete.

This paper is accompanied by the mechanized proofs of the theoretical results presented therein and a tool (FaialAA) implementing of our theory.

CCS Concepts: • **Theory of computation → Parallel computing models**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: data-race detection, GPU programming, static analysis, true positives

ACM Reference Format:
Dennis Liew, Tiago Cogumbreiro, and Julien Lange. 2024. Sound and Partially-Complete Static Analysis of Data-Races in GPU Programs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 357 (October 2024), 28 pages. https://doi.org/10.1145/3689797

Authors' Contact Information: Dennis Liew, University of Massachusetts Boston, Boston, USA, zhenrong.liew001@umb.edu; Tiago Cogumbreiro, University of Massachusetts Boston, Boston, USA, tiago.cogumbreiro@umb.edu; Julien Lange, Royal Holloway, University of London, Egham, United Kingdom, julien.lange@rhul.ac.uk.

ACM 2475-1421/2024/10-ART357
https://doi.org/10.1145/3689797

1 Introduction

Graphical Processing Units (GPUs) are crucial in a wide range of fields such as artificial intelligence and machine learning [58], molecular modeling [69], systems biology [21], and medical imaging [68]. The high-performance potential of massively parallel devices like GPUs comes with a high cost: they are notoriously difficult to program correctly. To fully utilize the level of parallelism and performance offered by a GPU, programmers need a deep understanding of the concurrent execution model of GPU devices. Failing to reason about correctness may lead to errors such as data-races which may produce unexpected behavior (nondeterminism), aborted computation, and memory corruption, which can be exploited by malicious agents [34, 73, 76].

A static *data-race detector* analyzes a program's source code to find data-races [13, 30, 74]. We say that a data-race analysis is *complete* if every reported data-race is possible in an execution of the program under analysis. There are currently no *complete* static data-race analysis for GPU programs. The main objective of this paper is to present a static data-race detector for GPU programs (*a.k.a.* kernels) and prove its completeness for a class of kernels by establishing a True Positive Theorem. Gorogiannis *et al.* [30] were the first to establish a True Positives theorem for multithreaded[1] programs. The challenge behind designing an effective data-race detector hinges on capturing a class of programs that is "large enough." Our analysis targets the class of kernels (*i.e.*, programs) where memory accesses are unaffected by input, called control-independent and data-independent (CIDI). In this paper, we show that CIDI kernels are common in practice (59.5% of 2,770 kernels), which highlights the effectiveness of our technique.

Static data-race freedom (DRF) detectors analyze a kernel's source code to guarantee that no data-races are possible. DRF detectors for kernels [9, 10, 15, 16, 46, 47] can be used as incomplete data-race detectors, since the reported data-races can be spurious. The drawback of this approach is that users must validate every reported alarm manually. Research shows that manual validation of alarms hampers the adoption of static analysis in industrial settings [35, 57, 67].

We introduce the first *sound and partially-complete* data-race analysis: for a subset of kernels our tool is *precise*, *i.e.*, any and all data-races are correctly detected, and every DRF kernel is correctly detected. Establishing partial-completeness of static analysis has been proposed in Abstract Interpretation [12, 26, 29, 62, 65], yet, not in the context of concurrent programs. There is a large body of work in categorizing alarms as true/false, and yet none has tackled the problem of data-races: model checking [5, 14, 19, 31, 55, 56, 59, 63, 70, 71], symbolic execution [3, 27, 28, 36, 49, 75], abstract interpretation [43, 62, 66], and even SMT solvers [25, 38]. In the context of data-race detectors, there has also been work on using a statistical model to rank false alarms, according to user feedback [64]. Our approach is to establish a result of partial-completeness, including a True Positives Theorem, by extending a sound DRF analysis called Memory Access Protocols (MAPs) [15, 16]. MAPs has several key benefits over the state of the art, such as avoiding the need for loop invariants (unlike [9, 11, 41]) and being able to handle loop bounds and array sizes symbolically (unlike [48, 51, 61]). We implement our analysis in a tool called FaialAA, combining a DRF detector with a data-race detector. Further, FaialAA leverages an approximation analysis to explain the degree of imprecision, by indicating which elements of an alarm are potentially inexact. This fine-grained feedback renders our approach more usable, a key requirement for developers adopting static analysis tools [57].

[1]Techniques used to analyze data-races in multithreaded programs are generally inapplicable or irrelevant to GPU programming, and vice-versa. In multithreaded programming, shared resources are protected with locks. The focus of data-race analysis is on lock usage, thread lifecycle, and pointer aliasing. In GPU programming, shared resources are accessible via arrays, and accesses are ordered by barrier synchronization. The focus is on the equality of array indexing expressions, and analyzing the concurrency of instructions.

Contributions. This paper presents the following contributions:

- (§3) We introduce a novel *approximation analysis* that identifies: when control-flow is unaffected by approximated variables (called CI for control-flow independent); and, when array indexing is unaffected by approximated variables (called DI for data independent). These results are formalized in Theorem 3.3.
- (§4) We specify the inference of MAPs and establish our soundness and partial-completeness results, see Theorem 4.3. We leverage our approximation analysis to establish the root causes of spurious alarms (for protocols that are CI, DI, both, or neither), see Corollary 4.4. We state a True Positives result in Theorem 4.5. The theory and results of this paper have been mechanized in Coq, see Section 4.4.
- (§5) We introduce FaialAA, as the first static analysis tool that combines a DRF detector with a data-race detector. Our tool expands upon Faial by introducing numerous improvements that make the analysis of CUDA kernels more precise, such as support for inter-procedural analysis, using typing information to represent integer bounds, detecting benign data-races, supporting array aliasing patterns, and improving the existing support for bitwise operations, C++ templates, and precise iteration spaces of loops.
- (§6) An extensive evaluation of our contributions. We compare FaialAA to the state of the art using a well-studied benchmark, and show that our tool outperforms others by detecting more DRF kernels (5 more) and raising fewer potential alarms (1.9× fewer). Importantly, our approximation analysis detected 10 racy kernels, 6 of which are *missed* by other tools. We introduce a novel dataset of 6 data-race fixing commits that originate from large open source projects, OpenMM [22] and Nvidia's Megatron-LM [42]. In an experiment to identify the bug in the pre-commit version and validate the post-commit version as DRF, FaialAA succeeds in 5 commits, while others only succeed in 2 commits. Finally, we found that the analysis of FaialAA is complete in 59.5% of 2,770 kernels, *i.e.*, detects all existing data-races or detects that the kernel is data-race free.

In the following section we give an overview our approach and contributions. Section 7 discusses related work and Section 8 concludes. The Coq mechanization of this theory and FaialAA are included in the supplemental material, and will be included in the artifact submission.

2 Overview

This section showcases the practical benefits of our contributions to the validation of four alarms of data-races. We demonstrate how our approximation analysis simplifies the validation process.

Our approximation analysis targets alarms from a data-race freedom (DRF) analysis implemented by Faial [15, 16]. A DRF analysis either proves that a program is free from data-races for any possible input, or raises alarms that identify potential data-races. A data-race is caused by unsynchronized accesses to a shared location issued by two distinct threads where at least one thread is writing. An alarm produced by FaialAA depicts (1) *(reachability)* the locations of the two access, *i.e.*, filename and line/column numbers, (2) *(where)* the array index, or shared variable, being accessed, and (3) *(valuation)* some program variables and their valuations, representing a partial state of both threads. An approximation report identifies whether (1), (2), and (3) are exact.

2.1 Understanding an Alarm With an Approximation Report

The data-race in Alarm 1 occurs because thread A writes to `sdata[16]` in Line 2, and concurrently thread B reads from `sdata[16]` in Line 3. Our approximation report indicates that Alarm 1 is exact, that the two accesses can happen concurrently (exact) and that both threads can target the reported index (exact). Thus, the user can conclude that Alarm 1 is a true alarm. To understand why, observe

Alarm 1. Data-race in kernel reduceMultiPass, location sdata[16].

Approximation report: Reachable accesses. Exact index. True alarm detected.

```
1 if (threadIdx.x < 32) {
2   sdata[threadIdx.x] = mySum // A
3     = mySum+sdata[threadIdx.x+16];// B
4 }
```

Variable	Thread	
	A	*B*
threadIdx.x	16	0

Alarm 2. Potential data-race in kernel TransitiveClosure/stage1/kernel, location P[0].

Approximation report: Potentially unreachable accesses (variable is_conn3 has been approximated). Exact index.

```
1 uint is_conn3 = P[0]; // A
2 if (is_conn3 != 0) {
3   P[threadIdx.x] = passnum * TILE_SIZE // B
4     + INDIRECTLY_CONNECTED;}
```

Variable	Thread	
	A	*B*
is_conn3	-1	-1
threadIdx.x	1	0

Alarm 3. Potential data-race in kernel collideD, location newVel[0].

Approximation report: Reachable accesses. Potentially inexact index (variable orig has been approximated).

```
1 uint index = blockIdx.x * blockDim.x + threadIdx.x;
2 uint orig = gridParticleIndex[index];
3 newVel[orig] =     // A, B
4   make_float4(vel + force, 0.0f);
```

Variable	Thread	
	A	*B*
blockIdx.x	0	0
blockDim.x	64	64
threadIdx.x	0	1
orig	0	0

that the variables that trigger the error are independent of the kernel's input: variable threadIdx.x is a unique thread identifier. Thus, the reported data-race happens in every execution of the kernel.

The potential data-race in Alarm 2 occurs on location P[0], because thread A reads in Line 1 while thread B concurrently writes in Line 3. Our approximation report indicates that the data-race *may* be spurious due to condition is_conn3 != 0 and also that is_conn3 has been approximated by the inference (by convention, we underline approximated variables in alarms). Technically, the warning appears because the approximation analysis identifies that possible values of variable is_conn3 are approximated. *To confirm the alarm, the user only needs to focus their attention on testing if thread B can reach Line 3.* Depending on the data from array P the condition is_conn3 != 0 may evaluate to true, which triggers the write in Line 3 to P[0], and therefore the data-race.

The potential data-race in Alarm 3 occurs in Line 3 while accessing newVel[0], when thread A and thread B read value 0 from array gridParticleIndex. The approximation report states that the location is spurious, since variable orig is approximate. *The user needs to confirm whether it is*

Alarm 4. Potential data-race in kernel `read-index`, location `M[0]`.

Approximation report: Reachable accesses. Potentially inexact index (variable `x` has been approximated).

```
M[threadIdx.x] = threadIdx.x;
int x = M[threadIdx.x];
M[x] = f(threadIdx.x); // A, B
```

Variable	**Thread** *A*	*B*
`threadIdx.x`	1	0
<u>`x`</u>	0	0

$p_A = \text{if tid} < 32\ \{\text{rd}[\text{tid} + 16]; \text{wr}[\text{tid}]\}$	CIDI	(Alarm 1)
$p_B = \text{rd}[0]; \text{var}\ x.\ \text{if}\ x \neq 0\ \{\text{wr}[\text{tid}]\}$	CDDI	(Alarm 2)
$p_C = \text{var}\ x.\ \text{wr}[x]$	CIDD	(Alarm 3)
$p_D = \text{wr}[\text{tid}]; \text{rd}[\text{tid}]; \text{var}\ x.\ \text{wr}[x]$	CIDD	(Alarm 4)

Fig. 1. Protocols inferred by FaialAA.

possible for threads A and B to have the same value of `orig`. Note that depending on what data is in array `gridParticleIndex`, a data-race may occur. In fact, the data-race can only be avoided if all indices read from `gridParticleIndex` are distinct. We are certain that the concurrent access will occur, but we are uncertain about the value of expression `orig` used to index the array `newVel`. We can say that in Alarm 2 our analysis is unsure about the *when* (reachability), whereas in Alarm 3 our analysis is unsure about the *where* (location).

Finally, we discuss a confirmed *spurious* alarm. The potential data-race in Alarm 4 occurs in Line 3 when threads A and B concurrently write to position 0, since both threads read a value 0 from array `M`. The approximation reports warns that the threads may not be accessing location `M[0]`, since variable `x` is over-approximated. Similarly to Alarm 3, *the user needs to confirm whether it would be possible for variable* `x` *to be 0 for both threads at once.* The alarm is spurious because for every thread program variable `x` must be equal to program variable `threadIdx.x`, however, thread A has `threadIdx.x=1` and `x=0`.

In summary, our approximation analysis can identify true data-races, as well as precisely identify which elements of an alarm need to be manually validated. This feedback helps developers triage alarms and sits in contrast to the current state-of-the-art that gives no such guarantees.

2.2 Approximation Analysis of Memory Access Protocols

In order to reason about the root cause of a spurious alarm, we must first understand the analysis that produces such alarms. The incompleteness of the MAP-based verification originates from transforming a CUDA kernel into a MAP, hereby known as *inference.* FaialAA infers one protocol per array used in a kernel, and each protocol is verified independently.

We show protocols for the running examples in Figure 1. We α-rename variables for presentation purposes. Program variable `threadIdx.x` appears as **`tid`**, and represents a *unique* thread identifier.

In Alarm 1, FaialAA infers protocol p_A for array `sdata`. The read access `sdata[threadIdx.x+16]` becomes $\text{rd}[\text{tid}+16]$ and the write access in `sdata[threadIdx.x]` is modeled by $\text{wr}[\text{tid}]$, *i.e.*, MAPs model which memory locations are accessed but not what is written to/read from arrays. Note the order of instructions in a protocol follows the evaluation order of C/C++. Thus, although the write of Line 2 appears before the read of Line 3, in C/C++ the right-hand side of an assignment (the read) executes *before* the left-hand side of an assignment (the write), so $\mathsf{rd}[\mathsf{tid}+16]$ precedes $\mathsf{wr}[\mathsf{tid}]$.

Our approximation analysis tests whether certain characteristics of the programs are abstracted when modeled by MAPs. Since p_A models all control-flow construct accurately, our analysis deems it *control-flow independent* (CI). Since p_A models precisely all indexing expressions of the original program, our analysis deems it *data-independent.* We denote both properties as CIDI. *Informally, a main result of our work is establishing that DRF analysis of CIDI protocols yields only true alarms (spurious alarms are impossible).* Our study of 2,544 kernels from public GitHub projects found that 58.2% of these kernels are CIDI, suggesting that our analysis is sound and complete for a vast majority of kernels.

In Alarm 2, `FaialAA` infers protocol p_B for array `P`. The inference turns instruction `P[0]` into $\mathsf{rd}[0]$. In MAPs, the symbolic variable binder $\mathsf{var}\ x.\ p$ introduces a new variable x bound in p (to some unknown value). Each thread can assign a different value to x. Declaration `uint is_conn3` becomes $\mathsf{var}\ x$. Th expression `passnum * TILE_SIZE + 0 + INDIRECTLY_CONNECTED` is not modeled as it does not affect the control-flow nor any indexing of the array. The report in Alarm 2 states that there is an over-approximated condition, because variable x occurs in a conditional. Thus, our analysis deems p_B *control-flow dependent* (CD). Since x does not appear in an index, then p_B is DI (all indexing expressions are modeled accurately). We denote both properties as CDDI. *Informally, a main result of our work is showing that spurious alarms of CDDI protocols can only be caused by unreachable accesses.*

In Alarm 3, `FaialAA` infers protocol p_C for array `newVel`. Since MAPs represent one protocol per array, the accesses of `gridParticleIndex[index]` is omitted here. The declaration of variable `orig` is represented by the binder for x. Variable `index` and the expression `make_float4(vel + force, 0.0f)` are not modeled here as they are unused in control flow and array indexing. The write access `newVel[orig]` is represented by $\mathsf{wr}[x]$. Our analysis deems p_C control-independent (CI) because no symbolic variable occurs in a branch or in a loop, and data-dependent (DD) because x appears in an index; this is denoted with CIDD. *Informally, a key result of this paper is showing that spurious alarms of CIDD protocols can only be caused by mismatched indices (of two reachable accesses).*

Finally, in Alarm 4, `FaialAA` infers protocol p_D for array `M`. After the write access and the read access, we have binder x that represents the declaration `int x`. The expression `f(threadIdx.x)` is modeled as it is unused in control flow and array indexing. Since x is not used in conditions or loops, protocol p_D is control independent (CI). The alarm warns about a spurious indexing location, because protocol p_D uses a symbolic variable x in an index, which makes the protocol data-dependent (DD).

3 Approximation Analysis of Memory Access Protocols

The MAPs we present include one novel extension wrt. their original presentation in [15, 16]: a symbolic variable binder to reason about points of abstraction. Additionally, for the sake of formalizing the results of the present paper (for which we need to reason about reachability), we instrument the semantics of MAPs so that they record the internal choices of a thread.

3.1 Syntax and Semantics

We give the syntax and semantics of MAPs in Figure 2.

Syntax. Let $\mathbb{N}$ be the set of natural numbers, *i.e.*, non-negative integers (natural numbers). Meta-variables i, j, k, and l range over natural numbers. We take the convention of using a distinct meta variable (natural number) in different context, *i.e.*, i for thread identifiers (also colored in blue), j for array indices, k for array contents, and l for loop bounds. An arithmetic expression, ranged over by

Syntax

$$
\begin{array}{rcl@{\qquad}rcl}
i, j, k, l & ::= & 0 \mid 1 \mid \cdots & o & ::= & \mathsf{wr} \mid \mathsf{rd} \\
e & ::= & j \mid x \mid e \star e & \alpha & ::= & i \colon o[k] \\
b & ::= & \top \mid \bot & P & ::= & \{\alpha_1, \ldots, \alpha_n\} \\
c & ::= & b \mid e \diamond e \mid c \circ c & t & ::= & \tau \mid \top \mid \bot \mid x{:=}i \mid t; t
\end{array}
$$

$$
p \;::=\; \mathsf{var}\ x.\ p \mid o[e] \mid \mathsf{skip} \mid p\ ;\ p \mid \mathsf{if}\ c\ \{p\}\ \mathsf{else}\ \{p\} \mid \mathsf{for}\ x \in e..e\ \{p\}
$$

Semantics $\boxed{p \downarrow_i P \triangleright t}$ $\boxed{p \downarrow P}$

$$
\text{P-VAR}\ \frac{p[x := k] \downarrow_i P \triangleright t}{\mathsf{var}\ x.\ p \downarrow_i P \triangleright t}
\qquad
\text{P-ACC}\ \frac{e \downarrow_i j}{o[e] \downarrow_i \{i \colon o[j]\} \triangleright \tau}
\qquad
\text{P-SKIP}\ \frac{}{\mathsf{skip} \downarrow_i \emptyset \triangleright \tau}
\qquad
\text{P-SEQ}\ \frac{p_1 \downarrow_i P_1 \triangleright t_1 \qquad p_2 \downarrow_i P_2 \triangleright t_2}{p_1\ ;\ p_2 \downarrow_i P_1 \cup P_2 \triangleright t_1; t_2}
$$

$$
\text{P-IF-T}\ \frac{c \downarrow_i \top \qquad p_t \downarrow_i P \triangleright t}{\mathsf{if}\ c\ \{p_t\}\ \mathsf{else}\ \{p_f\} \downarrow_i P \triangleright \top; t}
\qquad
\text{P-IF-F}\ \frac{c \downarrow_i \bot \qquad p_f \downarrow_i P \triangleright t}{\mathsf{if}\ c\ \{p_t\}\ \mathsf{else}\ \{p_f\} \downarrow_i P \triangleright \bot; t}
\qquad
\text{P-FOR-1}\ \frac{e_1 \geq e_2 \downarrow_i \top}{\mathsf{for}\ x \in e_1..e_2\ \{p\} \downarrow_i \emptyset \triangleright \tau}
$$

$$
\text{P-FOR-2}\ \frac{e_1 \downarrow_i l \qquad (l < e_2) \downarrow_i \top \qquad p[x := l] \downarrow_i P_1 \triangleright t_1 \qquad \mathsf{for}\ x \in l+1..e_2\ \{p\} \downarrow_i P_2 \triangleright t_2}{\mathsf{for}\ x \in e_1..e_2\ \{p\} \downarrow_i P_1 \cup P_2 \triangleright (x{:=}l; t_1); t_2}
$$

Fig. 2. Syntax and semantics of MAPs.

e, is a natural number i, a variable x (where $\mathcal{X}$ is the set of variables and $x \in \mathcal{X}$), or a closed binary operation on arithmetic expressions (*e.g.*, addition). A boolean expression c is a boolean literal b, an arithmetic comparison $e \diamond e$, or a logic connective $c \circ c$. A MAP p, or simply a *protocol* p, is as follows. A nondeterministic variable binder $\mathsf{var}\ x.\ p$ assigns variable x to some natural number in the scope of p. The placement of a variable declaration matters. For instance, the protocol:

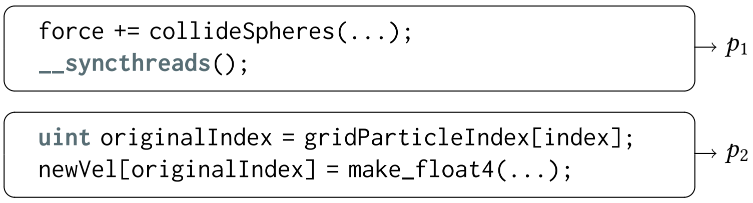
$$\mathsf{for}\ y \in 0..10\ \{\mathsf{var}\ x.\ \mathsf{wr}[x]\}$$

represents 10 writes to potentially different locations in the array. Whereas the following protocol represents 10 writes to the *same* location:

$$\mathsf{var}\ x.\ \mathsf{for}\ y \in 0..10\ \{\mathsf{wr}[x]\}$$

An array access $o[e]$ expresses how the array is being accessed with o (which is either reading rd or writing wr), and which index e is being accessed. The skip denotes a no-op. Sequencing and conditional are standard. When convenient, we abbreviate protocol $\mathsf{if}\ c\ \{p\}\ \mathsf{else}\ \{\mathsf{skip}\}$ to $\mathsf{if}\ c\ \{p\}$. A loop $\mathsf{for}\ x \in e_1..e_2\ \{p\}$ expresses a loop variable x with a new lexical scope of p (the loop body). A loop range $e_1..e_2$ denotes a lower bound e_1 (inclusive) and an upper bound e_2 (exclusive), *e.g.*, a loop range of 0..2 corresponds to iterations 0 and 1.

Observe that the syntax of MAPs does not include synchronization primitives (*e.g.*, __syncthreads). Cogumbreiro *et al.* [15, Theorem 1] established that the DRF analysis of a protocol with synchronization is *equivalent* to the DRF analysis of a set of synchronization-free protocols, so we limit our approach to synchronization-free protocols without loss of generality. For instance, in Figure 3 the code before synchronization, in the first box labelled by p_1, cannot run in parallel with the code after synchronization, in the second box labelled by p_2, so each synchronization-free protocol can be analyzed independently. The algorithm that extracts synchronization-free protocols from a protocol with synchronization is discussed in [15].



```
force += collideSpheres(...);
__syncthreads();
```

```
uint originalIndex = gridParticleIndex[index];
newVel[originalIndex] = make_float4(...);
```

Fig. 3. Illustration of analyzing code with barrier synchronization.

Assumptions. Thread-global variables are defined as variables with the same value for all threads. In our formalism, we encode thread-global variables as unknown number, *i.e.*, numeric parameters declared at the meta theoretical level. In the tool, we represent thread-global variables as a binder, much like we do for thread-local binders. Hereafter, we only consider *well-formed* protocols: protocols are closed and every binder (*i.e.*, for and var) introduces distinct variable names.

Semantics. We define a big-step semantics for protocols with judgment $p \downarrow_i P \triangleright t$ which represents the execution of a protocol p by a single thread i that yields a phase P (which holds the set of all concurrent accesses) and trace t (which records the internal choices of the protocol). Let $\mathcal{T} \subseteq \mathbb{N}$ be the set of all threads, defined as a meta-parameter of our theory. An access value $i\colon o[j]$ consists of a thread identifier $i \in \mathcal{T}$ that issues the access, the access mode o, and the index j of the array. For instance in Alarm 1, thread B with identifier 0 reads from index 16 of array sdata, thus we write $0\colon \mathsf{rd}[16]$. Let $\mathbb{A}$ be the set of all access values. The phase P is as a collection of access values representing all the read and write accesses in a protocol. For instance, for the kernel of Alarm 1, thread A accesses sdata twice, $\{16\colon \mathsf{rd}[32], 16\colon \mathsf{wr}[16]\}$. We postpone the discussion of traces to until they are used in Section 3.3.

Let us introduce the rules governing the semantics of MAPs. Rule P-VAR assigns some number k to symbolic variable x in $p[x \coloneqq k]$ to produce a phase P. Rule P-ACC states that executing an access $o[e]$ yields a singleton phase holding access value $i\colon o[j]$, where index j results from evaluating the indexing expression e. Protocol skip produces no accesses. Rule P-SEQ states that running a sequence of protocols $p_1; p_2$ consists of running each protocol independently and taking the union of both phases $P_1 \cup P_2$. The rules for the conditional are straightforward. There are two rules that govern loops. When the range is empty (P-FOR-1), where $e_1 \geq e_2 \downarrow_i \top$, we obtain no accesses $\emptyset$. Otherwise, (P-FOR-2), we run one iteration followed by the rest of the loop, yielding the union the respective phases $P_1 \cup P_2$. For the former, we assign the loop variable x to the lower bound l in the execution of the loop body $p[x \coloneqq l]$, which yields phase P_1. For the latter, we execute the loop by incrementing the lower bound $e_1 + 1$, which yields phase P_2.

Rule P-VAR renders the semantics of MAPs non-deterministic. Picking a different value k can yield different phases, *e.g.*, $\mathsf{var}\ x.\ \mathsf{wr}[x] \downarrow_0 \{0\colon \mathsf{wr}[k]\}$ holds for any k. The same is true for different traces, but we postpone a detailed discussion for Section 3.2.

Explaining the alarms. An alarm results from a DRF analysis on MAPs, so we can now precisely show why each alarm exists. Below, for each alarm, we evaluate the inferred protocol for each thread

$$\boxed{X \vdash e} \quad \boxed{X \vdash c}$$

$$X \vdash i \qquad X \vdash \mathsf{tid} \qquad \frac{x \in X}{X \vdash x} \qquad \frac{X \vdash e_1 \quad X \vdash e_2}{X \vdash e_1 \star e_2} \qquad X \vdash b \qquad \frac{X \vdash e_1 \quad X \vdash e_2}{X \vdash e_1 \diamond e_2} \qquad \frac{X \vdash c_1 \quad X \vdash c_2}{X \vdash c_1 \circ c_2}$$

$$\boxed{X \vdash_\alpha p \quad \alpha \in \{\mathtt{CI}, \mathtt{DI}\}}$$

$$\textsc{i-skip}\ \frac{}{X \vdash_\alpha \mathsf{skip}} \qquad \textsc{i-seq}\ \frac{X \vdash_\alpha p_1 \quad X \vdash_\alpha p_2}{X \vdash_\alpha p_1 \ ; \ p_2} \qquad \textsc{i-var}\ \frac{X \vdash_\alpha p}{X \vdash_\alpha \mathsf{var}\ x.\ p}$$

$$\boxed{X \vdash_{\mathtt{CI}} p}$$

$$\textsc{ci-acc}\ \frac{}{X \vdash_{\mathtt{CI}} o[e]} \qquad \textsc{ci-if}\ \frac{X \vdash c \quad X \vdash_{\mathtt{CI}} p_1 \quad X \vdash_{\mathtt{CI}} p_2}{X \vdash_{\mathtt{CI}} \mathsf{if}\ c\ \{p_1\}\ \mathsf{else}\ \{p_2\}} \qquad \textsc{ci-for}\ \frac{X \vdash e_1 \quad X \vdash e_2 \quad X \cup \{x\} \vdash_{\mathtt{CI}} p}{X \vdash_{\mathtt{CI}} \mathsf{for}\ x \in e_1..e_2\ \{p\}}$$

$$\boxed{X \vdash_{\mathtt{DI}} p}$$

$$\textsc{di-acc}\ \frac{X \vdash e}{X \vdash_{\mathtt{DI}} o[e]} \qquad \textsc{di-if}\ \frac{X \vdash_{\mathtt{DI}} p_1 \quad X \vdash_{\mathtt{DI}} p_2}{X \vdash_{\mathtt{DI}} \mathsf{if}\ c\ \{p_1\}\ \mathsf{else}\ \{p_2\}} \qquad \textsc{di-for-d}\ \frac{X \vdash_{\mathtt{DI}} p}{X \vdash_{\mathtt{DI}} \mathsf{for}\ x \in e_1..e_2\ \{p\}} \qquad \textsc{di-for-i}\ \frac{X \vdash e_1 \quad X \vdash e_2 \quad X \cup \{x\} \vdash_{\mathtt{DI}} p}{X \vdash_{\mathtt{DI}} \mathsf{for}\ x \in e_1..e_2\ \{p\}}$$

Fig. 4. Judgments for Control-flow Independence and Data Independence.

and underline the offending access values. Recall that variable tid takes the value of `threadIdx.x`.

Thread A	Thread B	
$p_A \downarrow_{16} \{16\colon \mathsf{rd}[32], \underline{16\colon \mathsf{wr}[16]}\} \triangleright t_A$	$p_A \downarrow_{0} \{\underline{0\colon \mathsf{rd}[16]}, 0\colon \mathsf{wr}[0]\} \triangleright t_A$	(Alarm 1)
$p_B \downarrow_{1} \{\underline{1\colon \mathsf{rd}[0]}, 1\colon \mathsf{wr}[1]\} \triangleright t_B$	$p_B \downarrow_{0} \{0\colon \mathsf{rd}[0], \underline{0\colon \mathsf{wr}[0]}\} \triangleright t_B$	(Alarm 2)
$p_C \downarrow_{0} \{\underline{0\colon \mathsf{wr}[0]}\} \triangleright t_C$	$p_C \downarrow_{1} \{\underline{1\colon \mathsf{wr}[0]}\} \triangleright t_C$	(Alarm 3)
$p_D \downarrow_{1} \{1\colon \mathsf{wr}[1], 1\colon \mathsf{rd}[1], \underline{1\colon \mathsf{wr}[0]}\} \triangleright t_D$	$p_D \downarrow_{0} \{0\colon \mathsf{rd}[0], \underline{0\colon \mathsf{wr}[0]}\} \triangleright t_D$	(Alarm 4)

3.2 Approximation Analysis

In this section, we present our approximation analysis for MAPs. In Figure 4, we introduce judgments that track when symbolic variables (introduced with $\mathsf{var}\ x.\ p$) either do not occur in control-flow (in conditionals and in loop bounds), or do not occur in indexing accesses.

Let $X \subseteq \mathcal{X}$ range over a set of variables. Judgment $X \vdash e$ holds when the free variables of e are in X. Similarly, judgment $X \vdash c$ holds when the free variables of c are in X. Note that for the family of judgments given in Figure 4, one can always enlarge the set of variables being referred to, *e.g.*, if $X \vdash e$, then $X \cup X' \vdash e$.

Control-flow-independent (CI) protocols. Judgment $X \vdash_{\mathtt{CI}} p$ states that the control-flow in p is unaffected by symbolic variables, given a set of variables X that *can* affect the control-flow of p. We say that a protocol p is *control-flow independent* (CI) when $X \vdash_{\mathtt{CI}} p$ for some X. Rule i-skip states that skip is always control-flow independent. Rule i-seq states that $p_1; p_2$ is control-flow independent if p_1 and p_2 are both control-flow independent. Rule i-var dictates that $\mathsf{var}\ x.\ p$ is control-flow independent when the body p is control-flow independent; we note that the set of variables X that can affect control-flow remains the same. Rule ci-acc indicates that accesses are always control independent. The interesting rules are ci-if and ci-for. Since conditionals can affect control-flow of the program, we enforce that condition c only uses set X with $X \vdash c$, and that branches p_1 and p_2 are both control-flow independent. Rule CI-FOR enforces that the lower e_1 and upper e_2 bounds only use the variables in X. Since the bounds are unaffected by symbolic variable binders, then variable x can be used in the loop body p, augmenting the set of variables that can be used to be $X \cup \{x\}$. We write $X \nvdash_{\mathtt{CI}} p$ when there is no derivation for which $X \vdash_{\mathtt{CI}} p$ holds. Sequencing a control-flow dependent protocol, say p_B, with any other protocol yields a control-flow dependent protocol. Thus, $p_A; p_B$ is control-flow dependent.

We are now able to prove the control-flow independent results of Figure 1. We have that p_A is control-flow independent, since $\emptyset \vdash_{\mathtt{CI}} p_A$. Protocol p_B is control-flow dependent, since $\emptyset \nvdash_{\mathtt{CI}} p_B$. We have that p_C and p_D are both control-flow independent.

Data-independent (DI) protocols. Judgement $X \vdash_{\mathtt{CI}} p$ states that indexing expressions that appear in p are unaffected by variables defined with **var**, given that variables X *can* affect the control-flow of p. We say that a protocol p is *data independent* (DI) when $X \vdash_{\mathtt{DI}} p$ for some X. Rule DI-ACC states a memory access is data-independent if index e only uses the variables in X. Rule DI-IF allows the use of symbolic variable variables in conditions but ensures that branches p_1 and p_2 are data-independent. Two rules govern loops: the loop variable x available in the loop body p only when the loop bounds only use variables in X (Rule DI-FOR-I), otherwise the loop body p is restricted to X (Rule DI-FOR-D). We write $X \nvdash_{\mathtt{DI}} p$ when there is no derivation for which $X \vdash_{\mathtt{DI}} p$ holds.

We are now able to prove the data independent results of Figure 1. We have that p_A is data independent, since $\emptyset \vdash_{\mathtt{DI}} p_A$. We also have that p_B is data independent. Protocol p_C is data dependent, since $\emptyset \nvdash_{\mathtt{DI}} p_C$. Protocol p_D is also data dependent. Sequencing a data dependent protocol, say p_C, with any other protocol yields a data dependent protocol. Hence, $p_B; p_C$ is control-flow dependent and data-dependent.

3.3 Results

We now establish properties relating control-flow and data independence of protocols with their phases and traces.

We first define a trace t as the sequence of internal choices that a thread takes. We revisit the definition of traces given in Figure 2. A trace τ denotes taking no further branches. Trace $\top$ corresponds to taking the first branch of a conditional. Trace $\bot$ corresponds to taking the second branch of a conditional. Trace $x{:=}i$ indicates running one iteration of a loop where loop variable x is being assigned the value of i. Trace $t; t$ is the sequential composition of two traces.

Protocol p_A is control-flow independent and data independent, thus its execution yields a unique phase and a unique trace for each thread. The only derivation of protocol p_A for thread 0 is

$$p_A \downarrow_0 \{0\colon \mathsf{wr}[0], 0\colon \mathsf{rd}[16]\} \triangleright \top; \tau; \tau$$

Protocol p_B is control-flow dependent and data independent, thus there is a unique phase *per trace*. We have

$$\begin{array}{lll} p_B \downarrow_0 & \{0\colon \mathsf{rd}[0], 0\colon \mathsf{wr}[0]\} \triangleright \tau; \top; \tau & \text{when } x \neq 0 \text{ in Rule P-VAR} \\ p_B \downarrow_0 & \{0\colon \mathsf{rd}[0]\} \triangleright \tau; \bot; \tau & \text{otherwise} \end{array}$$

Protocol p_C is control-flow independent, thus there is a unique trace. Additionally, let us observe what happens to the phase. Recall that $p_C = \mathsf{var}\ x.\ \mathsf{wr}[x]$. When $x := 0$ in Rule P-VAR, we have $p_C \downarrow_0 \{0\colon \mathsf{wr}[0]\} \triangleright \tau$. Similarly, when $x := 1$ in Rule P-VAR, we have $p_C \downarrow_0 \{0\colon \mathsf{wr}[1]\} \triangleright \tau$. In fact, for any value picked the output phase only differs in the index.

To relate phases that have distinct access-value indices we introduce the following relations.

Definition 3.1 (Up-to relation). Let $\mathcal{R}$ be a binary relation. Given X and Y sets, we denote $X \equiv_{\mathcal{R}} Y$ if, and only if, for all $x \in X$, then $\exists y \in Y$ such that $(x, y) \in \mathcal{R}$, and for all $y \in Y$, then $\exists x \in X$ such that $(x, y) \in \mathcal{R}$.

Definition 3.2 (Equivalence modulo index). Let $\mathcal{I}$ be the largest binary relation such that $(i\colon o[k_1], i\colon o[k_2]) \in \mathcal{I}$ for all $i \in \mathcal{T}$ and $o \in \{\mathsf{wr}, \mathsf{rd}\}$.

For instance, we have that $(0\colon \mathsf{wr}[0], 0\colon \mathsf{wr}[1]) \in \mathcal{I}$. Additionally, $\{0\colon \mathsf{wr}[16], 0\colon \mathsf{rd}[32]\} \equiv_{\mathcal{I}} \{0\colon \mathsf{wr}[0], 0\colon \mathsf{rd}[1]\}$.

The following theorem makes precise our informal discussion. Result (1) states an expected result: two executions of the same protocol on a given thread i that make the same internal choices will produce phases that only differ in their indices. Result (2) states that control-flow independent protocols generate phases that may only differ wrt. the index of their accesses. Result (3) states that data independent protocols that make the same internal choices, produce the same phase. Result (4) states that control-flow and data independent protocols produce a unique phase and a unique trace.

Theorem 3.3 (Trace and phase determinism). *Let $i \in \mathcal{T}$, $p \downarrow_i P_1 \triangleright t_1$, and $p \downarrow_i P_2 \triangleright t_2$.*

(1) *If $t_1 = t_2$, then $P_1 \equiv_{\mathcal{I}} P_2$.*
(2) *(Trace determinism) If $\emptyset \vdash_{\mathtt{CI}} p$, then $P_1 \equiv_{\mathcal{I}} P_2$ and $t_1 = t_2$.*
(3) *(Phase determinism) If $\emptyset \vdash_{\mathtt{DI}} p$ and $t_1 = t_2$, then $P_1 = P_2$.*
(4) *(Trace and phase determinism) If $\emptyset \vdash_{\mathtt{CI}} p$ and $\emptyset \vdash_{\mathtt{DI}} p$, then $P_1 = P_2$ and $t_1 = t_2$.*

4 Jaminan: A Core Calculus for GPU Programming

The aim of this section is to reason precisely about real and spurious alarms. To this end, we introduce a small calculus (Jaminan) that captures the concurrency of GPU programming, then we define an algorithm that infers protocols from Jaminan programs. Our main results for this section are precise correspondences between Jaminan programs and their protocols. Next, we outline areas of focus and assumptions that informed the design of Jaminan.

A key feature of MAPs [15, 16] is distinguishing syntactically between a *synchronized* and an *unsynchronized* fragment of the protocol language. A synchronized fragment can contain unsynchronized fragments, but not vice-versa. Barrier synchronization can only appear in a synchronized fragment. Memory accesses can only appear in an unsynchronized fragment. Additionally, and as shown in [15, 16], a synchronized fragment can be divided into a collection of unsynchronized fragments that can be verified independently. Given this property of MAPs and that synchronization does not affect the approximation of expressions in MAPs, we only discuss the inference of the unsynchronized fragment (henceforth, protocols).

Any over-approximation produced by the inference can be expressed via a symbolic variable and therefore reasoned about by the approximation analysis. For instance, loops in this paper (in MAPs and in Jaminan) have a lower bound, an upper bound, and a loop stride of 1 (*i.e.*, incrementing the loop variable by 1 per iteration). Our implementation (FaialAA) supports loop strides of arbitrary sizes with addition, subtraction, division, and multiplication (includes bit-shift operators); unsupported loops are modelled using symbolic variables for loop bounds. As another example, FaialAA replaces unsupported numeric expressions by symbolic variables, *e.g.*, function calls that take data-types that are not integers.

4.1 Syntax and Semantics

Syntax by example. Figure 5 introduces the syntax and semantics of Jaminan, the calculus from which we aim to infer a MAP. Let s range over the set of statements. A read $\mathsf{let}\ x = \mathtt{A}[e]\ \mathsf{in}\ s$ takes the value at index e and assigns it to variable x in the scope of statement s. A write $\mathtt{A}[e_1] := e_2$

Syntax

$$s \ ::= \ \mathsf{let}\ x = \mathsf{A}[e]\ \mathsf{in}\ s \ \mid \ \mathsf{A}[e] := e \ \mid \ s\ ;\ s \ \mid \ \mathsf{if}\ c\ \{s\}\ \mathsf{else}\ \{s\} \ \mid \ \mathsf{for}\ x \in e..e\ \{s\} \ \mid \ \mathsf{var}\ x.\ s \ \mid \ \mathsf{skip}$$

Semantics $\boxed{M, s \downarrow_i M, P \triangleright t}$

S-VAR
$$\frac{M, s[x := k] \downarrow_i M', P \triangleright t}{M, \mathsf{var}\ x.\ s \downarrow_i M', P \triangleright t}$$

S-SKIP
$$\frac{}{M, \mathsf{skip} \downarrow_i \emptyset, \emptyset \triangleright \tau}$$

S-READ
$$\frac{e \downarrow j \qquad \widehat{M}(j) = k \qquad M, s[x := k] \downarrow_i M', P \triangleright t}{M, \mathsf{let}\ x = \mathsf{A}[e]\ \mathsf{in}\ s \downarrow_i M', P \cup \{i\colon \mathsf{rd}[j]\} \triangleright \tau; t}$$

S-WRITE
$$\frac{e_1 \downarrow j \qquad e_2 \downarrow k}{M, \mathsf{A}[e_1] := e_2 \downarrow_i \{j \mapsto k\}, \{i\colon \mathsf{wr}[j]\} \triangleright \tau}$$

S-IF-T
$$\frac{c \downarrow \top \qquad M, s_1 \downarrow_i M', P \triangleright t}{M, \mathsf{if}\ c\ \{s_1\}\ \mathsf{else}\ \{s_2\} \downarrow_i M', P \triangleright \top; t}$$

S-IF-F
$$\frac{c \downarrow \bot \qquad M, s_2 \downarrow_i M', P \triangleright t}{M, \mathsf{if}\ c\ \{s_1\}\ \mathsf{else}\ \{s_2\} \downarrow_i M', P \triangleright \bot; t}$$

S-SEQ
$$\frac{M, s_1 \downarrow_i M', P \triangleright t_1 \qquad M \cup^{\bullet} M', s_2 \downarrow_i M'', P' \triangleright t_2}{M, s_1\ ;\ s_2 \downarrow_i M' \cup^{\bullet} M'', P \cup P' \triangleright t_1; t_2}$$

S-FOR-1
$$\frac{(e_1 \geq e_2) \downarrow \top}{M, \mathsf{for}\ x \in e_1..e_2\ \{s\} \downarrow_i \emptyset, \emptyset \triangleright \tau}$$

S-FOR-2
$$\frac{e_1 \downarrow_i l \qquad (l < e_2) \downarrow_i \top \qquad M, s[x := l] \downarrow_i M', P \triangleright t_1 \qquad M \cup^{\bullet} M', \mathsf{for}\ x \in l+1..e_2\ \{s\} \downarrow_i M'', P' \triangleright t_2}{M, \mathsf{for}\ x \in e_1..e_2\ \{s\} \downarrow_i M' \cup^{\bullet} M'', P \cup P' \triangleright (x{:=}l; t_1); t_2}$$

Fig. 5. Syntax and semantics of Jaminan.

assigns the value of e_2 into the array location indexed by e_1. Jaminan includes a symbolic variable binder to capture any form of abstraction produced by FaialAA.

We revisit our running examples written in Jaminan. In s_A, read statement $\mathsf{let}\ x = \mathsf{A}[\mathsf{tid} + 16]$ represents `sdata[threadIdx.x+16]`. In s_A, the write statement $\mathsf{A}[\mathsf{tid}] := x$ represents `sdata[threadIdx.x] = mySum + sdata[threadIdx.x+16]`. To ease presentation and without sacrificing generality, we opt not to include `mySum` in p_A. The remaining examples should be straightforward.

$$s_A = \mathsf{if}\ \mathsf{tid} < 32\ \{\mathsf{let}\ x = \mathsf{A}[\mathsf{tid} + 16]\ \mathsf{in}\ \mathsf{A}[\mathsf{tid}] := x\} \qquad \text{(Alarm 1)}$$
$$s_B = \mathsf{let}\ x = \mathsf{A}[0]\ \mathsf{in}\ \mathsf{if}\ x \neq 0\ \{\mathsf{var}\ y.\ \mathsf{A}[\mathsf{tid}] := y\} \qquad \text{(Alarm 2)}$$
$$s_C = \mathsf{var}\ x.\ \mathsf{var}\ y.\ \mathsf{A}[x] := y \qquad \text{(Alarm 3)}$$
$$s_D = \mathsf{A}[\mathsf{tid}] := \mathsf{tid}; \mathsf{let}\ x = \mathsf{A}[\mathsf{tid}]\ \mathsf{in}\ \mathsf{var}\ y.\ \mathsf{A}[x] := y \qquad \text{(Alarm 4)}$$

Semantics. We first define the finite maps we use to represent the state of an array. Let M, M', and M'' range over the set of finite maps from naturals into naturals. Let $\mathrm{dom}(M)$ denote the domain of M. Let $\emptyset$ denote the empty map *i.e.*, $j \notin \mathrm{dom}(\emptyset)$ for any j. Let $M = \{j \mapsto k\}$ denote the singleton map where $\mathrm{dom}(M) = \{j\}$ and $M(j) = k$. Next, we define *merging* maps, notation $M \cup^{\bullet} M'$, that gives precedence to the values of M'. Let $M \cup^{\bullet} M'$ be defined as a finite map M'' such that $M''(i) = M'(i)$ if $i \in \mathrm{dom}(M')$, otherwise $M''(i) = M(i)$. Finally, we define a lookup operator, notation $\widehat{M}(i)$, that is a total function $\widehat{M}(i) = M(i)$ when $i \in \mathrm{dom}(M)$ otherwise $\widehat{M}(i) = 0$.

We define a big-step operational semantics for protocols with judgement $M, s \downarrow_i P, M' \triangleright t$ that reads as: thread i executes statement s with memory M and results in an output memory M'

(capturing the new data written by i), a phase P (capturing the accesses of i), and a trace t (recording the internal choices of i). We overload the notation for substitution, *e.g.*, $s[x := k]$ means that all free occurrences of x in program s are replaced by k. Rule S-READ evaluates the index down to j, reads value k from memory M and replaces variable x by k in continuation s. The rule extends the resulting phase P that results from executing $s[x := k]$ with a read. The produced trace $(\tau; t)$ means that reading yields no internal choices followed by the choices of the continuation. Rule S-WRITE evaluates the index down to j, the payload down to k. A write statement outputs a singleton map assigning payload k to index j, and a singleton phase with the write-access value. The output trace τ states that no internal choices were taken. The rules for conditional S-IF-T and S-IF-F follow the similar principles as for MAPs. Rule S-SEQ evaluates s_1 to obtain some memory updates in M' and a phase P. Then, running s_2 takes $M \cup^\bullet M'$, so that the writes of s_1 are observed by s_2, yielding memory updates M'' and phase P'. The output of the sequence merges the memory updates and the phases of both statements. The remaining rules should be straightforward, as they are similar to those of MAPs. Similarly to MAPs, Jaminan's semantics is non-deterministic, notably because of the non-deterministic variable assignment.

Proving and disproving alarms. We are now able to *formally* confirm Alarm 1, Alarm 2, and Alarm 3 by evaluating statements s_A, s_B, and s_C, respectively. We give an initial empty memory $\emptyset$ for statements s_A and s_C. For s_B, let the initial memory $M = \{0 \mapsto 10\}$. In these examples, the resulting memory updates would record a single write. We underline the access values that appear in both resulting phases.

$$\begin{array}{ll}
\text{Thread A} & \text{Thread B} \\
\emptyset, s_A \downarrow_{16} \{16 \mapsto 0\}, \{16\colon \mathsf{rd}[32], \underline{16\colon \mathsf{wr}[16]}\} \triangleright t_A & \emptyset, s_A \downarrow_{0} \{0 \mapsto 0\}, \{\underline{0\colon \mathsf{rd}[16]}, 0\colon \mathsf{wr}[0]\} \triangleright t_A \\
M, s_B \downarrow_{1} \{1 \mapsto 2\}, \{\underline{1\colon \mathsf{rd}[0]}, 1\colon \mathsf{wr}[1]\} \triangleright t_B & M, s_B \downarrow_{0} \{0 \mapsto 1\}, \{0\colon \mathsf{rd}[0], \underline{0\colon \mathsf{wr}[0]}\} \triangleright t_B \\
\emptyset, s_C \downarrow_{0} \{0 \mapsto 1\}, \{\underline{0\colon \mathsf{wr}[0]}\} \triangleright t_C & \emptyset, s_C \downarrow_{1} \{0 \mapsto 2\}, \{\underline{1\colon \mathsf{wr}[0]}\} \triangleright t_C
\end{array}$$

Finally, we can show that Alarm 4 is spurious, because the statement is data-race free for any possible memory M. Given two reductions of s_D for some threads i and j where $i \neq j$, the access values of each thread index different locations (at i and j, respectively).

$$M, s_D \downarrow_i \{i \mapsto i\}, \{i\colon \mathsf{wr}[i], i\colon \mathsf{rd}[i]\} \triangleright t_D \qquad M, s_D \downarrow_j \{j \mapsto j\}, \{j\colon \mathsf{wr}[j], j\colon \mathsf{rd}[j], \} \triangleright t_D$$

4.2 Alarms Through the Lens of Approximation Analysis

In this section, we define, and reason about, a syntax-driven inference algorithm from a Jaminan statement to a MAP. Then, we establish the properties of soundness, and (partial) completeness.

We define our inference algorithm in Figure 6. Inferring a write statement yields a write-protocol and discards payload e_2. Inferring a read statement yields a read access followed by the introduction of a new symbolic variable. Next, we apply our inference algorithm to our running examples:

Protocol inference $\boxed{[\![s]\!] = u}$

$$[\![\mathsf{A}[e_1] := e_2]\!] = \mathsf{wr}[e_1] \qquad [\![\mathsf{let}\ x = \mathsf{A}[e]\ \mathsf{in}\ s]\!] = \mathsf{rd}[e]; \mathsf{var}\ x.\ [\![s]\!]$$

$$[\![s_1\ ;\ s_2]\!] = [\![s_1]\!]\ ;[\![s_2]\!] \qquad [\![\mathsf{if}\ c\ \{s_1\}\ \mathsf{else}\ \{s_2\}]\!] = \mathsf{if}\ c\ \{[\![s_1]\!]\}\ \mathsf{else}\ \{[\![s_2]\!]\}$$

$$[\![\mathsf{for}\ x \in e_1..e_2\ \{s\}]\!] = \mathsf{for}\ x \in e_1..e_2\ \{[\![s]\!]\} \qquad [\![\mathsf{skip}]\!] = \mathsf{skip} \qquad [\![\mathsf{var}\ x.\ s]\!] = \mathsf{var}\ x.\ [\![s]\!]$$

Sets of actions $\boxed{\textsc{p}\text{-}actions(\mathrm{p})}$ $\boxed{\textsc{j}\text{-}actions(\mathrm{s})}$ $\boxed{reachable\text{-}actions(\mathrm{s})}$ $\boxed{spurious\text{-}actions(\mathrm{s})}$

$$\frac{i \in \mathcal{T} \qquad p \downarrow_i P \triangleright t \qquad \alpha \in P}{(\alpha, t) \in \textsc{p}\text{-}actions(p)} \qquad \frac{i \in \mathcal{T} \qquad M, s \downarrow_i M', P \triangleright t \qquad \alpha \in P}{(\alpha, t) \in \textsc{j}\text{-}actions(s)}$$

$$\frac{M, s \downarrow_i M', P \triangleright t}{(i\colon o[k], t) \in reachable\text{-}actions(s)} \qquad \frac{\delta \in \textsc{p}\text{-}actions([\![s]\!]) \setminus \textsc{j}\text{-}actions(s)}{\delta \in spurious\text{-}actions(s)}$$

Fig. 6. Protocol inference rules and sets of actions.

$$\begin{aligned}
[\![s_A]\!] &= [\![\mathsf{if}\ \mathsf{tid} < 32\ \{\mathsf{let}\ x = \mathsf{A}[\mathsf{tid} + 16]\ \mathsf{in}\ \mathsf{A}[\mathsf{tid}] := x\}]\!] \\
&= \mathsf{if}\ \mathsf{tid} < 32\ \{\mathsf{rd}[\mathsf{tid} + 16]; \mathsf{var}\ x.\ \mathsf{wr}[\mathsf{tid}]\} \equiv p_A \\
[\![s_B]\!] &= [\![\mathsf{let}\ x = \mathsf{A}[0]\ \mathsf{in}\ \mathsf{if}\ x \neq 0\ \{\mathsf{var}\ y.\ \mathsf{A}[\mathsf{tid}] := y\}]\!] \\
&= \mathsf{rd}[0]; \mathsf{var}\ x.\ \mathsf{if}\ x \neq 0\ \{\mathsf{var}\ y.\ \mathsf{wr}[\mathsf{tid}]\} \equiv p_B \\
[\![s_C]\!] &= [\![\mathsf{var}\ x.\ \mathsf{var}\ y.\ \mathsf{A}[x] := y]\!] = \mathsf{var}\ x.\ \mathsf{var}\ y.\ \mathsf{wr}[x] \equiv p_C \\
[\![s_D]\!] &= [\![\mathsf{A}[\mathsf{tid}] := \mathsf{tid}; \mathsf{let}\ x = \mathsf{A}[\mathsf{tid}]\ \mathsf{in}\ \mathsf{var}\ y.\ \mathsf{A}[x] := y]\!] \\
&= \mathsf{wr}[\mathsf{tid}]; \mathsf{rd}[\mathsf{tid}]; \mathsf{var}\ x.\ \mathsf{var}\ y.\ \mathsf{wr}[x] \equiv p_D
\end{aligned}$$

For the sake of presentation, we removed unnecessary symbolic variable binders from the protocols in Figure 1 (*e.g.*, $\mathsf{var}\ x.\ \mathsf{wr}[\mathsf{tid}]$ becomes $\mathsf{wr}[\mathsf{tid}]$ in p_A). This has no effect on the protocols' semantics.

To reason about the soundness and completeness of the inference, we would like to show that the behaviors of the model either include (*i.e.*, soundness), or match (*i.e.*, completeness) the behaviors of the program under analysis. Here, the behavior corresponds to the set of accesses *per trace*. Let an *action* be defined as $\delta = (\alpha, t)$, representing an access value α occurring in some trace t.

Definition 4.1 (Equivalence modulo index for actions). We extend the equivalence modulo index (Definition 3.2) to actions in the natural way, *i.e.*, $\big((\alpha_1, t), (\alpha_2, t')\big) \in \mathcal{I}$ if, and only if $(\alpha_1, \alpha_2) \in \mathcal{I}$.

Next, we introduce the functions in Figure 6 that generate sets of actions from protocols and from statements. We say that an action (α, t) is in the set $\textsc{p}\text{-}actions(p)$ (for Protocol actions) when evaluating protocol p yields t and α is in the resulting phase. We say that protocol p *emits* action δ. For instance, let $(1\colon \mathsf{wr}[0], t_D) = \delta_D$ be an action we want to highlight in Alarm 4. We have that p_D emits δ_D and

$$\delta_D \in \textsc{p}\text{-}actions(p_D) = \bigcup_{i \in \mathcal{T}} \{(i\colon \mathsf{wr}[i], t_D), (i\colon \mathsf{rd}[i], t_D)\} \cup \{(i\colon \mathsf{rd}[j], t_D) \mid i \in \mathcal{T} \wedge j \in \mathbb{N}\}$$

Similarly, we say that action (α, t) is in the set J-*actions*(s) (for Jaminan actions) when evaluating statement s yields the internal choices t for some memory M and access value α is in the resulting phase. We say that statement s *emits* action δ. For instance, statement s_D emits δ_D and

$$\delta_D \in \text{J-}\mathit{actions}(s_D) = \bigcup_{i \in \mathcal{T}} \{(i\colon \mathsf{wr}[i], t_D), (i\colon \mathsf{rd}[i], t_D)\}$$

We say that an action (α, t) is *reachable* within statement s when thread i can evaluate statement s and produce the internal choices t, where $\alpha = i\colon o[k]$. Crucially, α does not need to be a member of the resulting phase P. For instance, action δ_D is reachable within s_D and we have

$$\delta_D \in \mathit{reachable\text{-}actions}(s_D) = \{(\alpha, t_D) \mid \alpha = i\colon o[k] \land i \in \mathcal{T}\}$$

Finally, we say that an action δ is in the set *spurious-actions*(s) (the set of of *spurious* actions) when protocol $[\![s]\!]$ emits δ (which is what the tool observes), but statement s does not emit δ (which is what the user observes). In such a case, we may say that action δ is spurious for s. For instance, action δ_D is spurious for s_D because

$$\delta_D \in \mathit{spurious\text{-}actions}(s_D) = \text{P-}\mathit{actions}(p_D) \setminus \text{J-}\mathit{actions}(s_D) = \{(i\colon \mathsf{rd}[j], t_D) \mid i \in \mathcal{T} \land j \in \mathbb{N} \land i \neq j\}$$

4.3 Results

We show that the inference is sound. That is, if a statement s emits an action δ, then the inferred protocol $[\![s]\!]$ also emits δ.

Theorem 4.2 (Soundness). *For all Jaminan program s, J-actions$(s) \subseteq$ P-actions$([\![s]\!])$.*

The next theorem relates the actions emitted by a Jaminan statement and its inferred protocol. Result (1) refines Theorem 4.2 and states that the actions emitted by statement s are the actions emitted by the inferred protocol that are feasible, up to the index (since they may be approximated). Result (2) states that when the inferred protocol $[\![s]\!]$ is control-independent, then the actions emitted by $[\![s]\!]$ are all feasible, but may have an approximate index. Result (3) states that when the inferred protocol $[\![s]\!]$ is data-independent, then the actions emitted by $[\![s]\!]$ have a precise index, yet may be unreachable within s. Result (4) states that when the inferred protocol $[\![s]\!]$ is control-independent *and* data-independent, then both the inferred protocol and the statement emit exactly the same actions.

Theorem 4.3 (Action-set correspondence). *The following propositions hold.*

(1) *J-actions$(s) \equiv_{\mathcal{I}}$ P-actions$([\![s]\!]) \cap$ reachable-actions(s).*
(2) *If $\emptyset \vdash_{\mathtt{CI}} [\![s]\!]$, then J-actions$(s) \equiv_{\mathcal{I}}$ P-actions$([\![s]\!])$.*
(3) *If $\emptyset \vdash_{\mathtt{DI}} [\![s]\!]$, then J-actions$(s) =$ P-actions$([\![s]\!]) \cap$ reachable-actions(s).*
(4) *If $\emptyset \vdash_{\mathtt{CI}} [\![s]\!]$ and $\emptyset \vdash_{\mathtt{DI}} [\![s]\!]$, then J-actions$(s) =$ P-actions$([\![s]\!])$.*

Finally, we establish the following properties that establish the root causes of a spurious action. Result (1) states that a spurious action δ is either unreachable or δ has an inaccurate index. Result (2) states when the inferred protocol is control-independent, then the index must be inaccurate. Result (3) states when the inferred protocol is data-independent, then the action is unreachable.

Corollary 4.4 (Possible root causes). *Let $\delta \in$ spurious-actions(s).*

(1) *$\delta \notin$ reachable-actions(s) or $\exists \delta' \in$ J-actions$(s)\colon (\delta, \delta') \in \mathcal{I}$.*
(2) *If $\emptyset \vdash_{\mathtt{CI}} [\![s]\!]$, then $\exists \delta' \in$ J-actions$(s)\colon (\delta, \delta') \in \mathcal{I}$.*
(3) *If $\emptyset \vdash_{\mathtt{DI}} [\![s]\!]$, then $\delta \notin$ reachable-actions(s).*

We are finally equipped to state a True Positives theorem which states that when a data-race occurs in the inferred protocol $[\![s]\!]$ of a CIDI program, then it must occur in the program s. A data-race is a pair of concurrent accesses where two distinct threads, say i and k, address the same location k and at least one writes to the array. Formally, we define a data-race between two actions δ_1 and δ_2 as

$$\frac{\delta_1 = (i\colon o_1[k], t) \qquad \delta_2 = (j\colon o_2[k], t) \qquad i \neq j \qquad \mathrm{wr} \in \{o_1, o_2\}}{\mathrm{datarace}(\delta_1, \delta_2)}$$

For any CIDI program s, the following theorem states that when an inferred protocol $[\![s]\!]$ emits a data-race of actions δ_1 and δ_2, then program s also emits actions δ_1 and δ_2.

Theorem 4.5 (True Positives). *Let* $\emptyset \vdash_{\mathtt{CI}} [\![s]\!]$, $\emptyset \vdash_{\mathtt{DI}} [\![s]\!]$, *and* $\mathrm{datarace}(\delta_1, \delta_2)$. *If* $\delta_1 \in$ *P-actions*$([\![s]\!])$ *and* $\delta_2 \in$ *P-actions*$([\![s]\!])$, *then* $\delta_1 \in$ *J-actions*(s) *and* $\delta_2 \in$ *J-actions*(s).

4.4 Mechanization of Results

This paper is accompanied by a Coq formalization of the theoretical contributions of this paper, which includes a proof of Theorems 3.3, 4.2, 4.3 and 4.5 and Corollary 4.4. Our additional Coq mechanization consists of 15,900 lines of code, 170 definitions, and 750 theorems. This proof extends a mechanization of MAPs presented in [15]. The technical differences of our work versus [15] include adding symbolic variable binders and traces to MAPs, the Jaminan language, the various judgments introduced in this paper, and the aforementioned results.

5 Implementation: FaialAA

This section introduces FaialAA that builds on Faial [15, 16] to implement the approximation analysis, along with other improvements. We refer the reader to [15, §6] for more details, where Cogumbreiro *et al.* cover the implementation details of various features supported by Faial, including: multi-dimensional thread identifiers, multi-dimensional arrays, kernel parameters, block-level synchronization, arrays in shared memory, arrays in global memory, data-races from threads of the same block, and data-races from threads of the same warp. Faial ignores memory fences. Faial relies on LLVM [44], particularly `libclang`, to parse the source code.

5.1 Improvements over Faial

FaialAA introduces numerous improvements over Faial, which make it more precise and able to handle more CUDA kernels. Here, we list the **new features** that have direct impact in the evaluation section:

- *Grid-level analysis.* Detects data-races from threads of different blocks.
- *Atomics.* Detects data-races using atomics and considers different scopes, *i.e.*, system, device, and block.
- *Inter-procedural analysis.* Our implementation expands the kernel definition on any call site.
- *Typing information.* The analysis now takes into account the types of program variables. This feature eliminates a source of false alarms that assumed a larger range of values than those admitted by a type, *e.g.*, an unsigned integer variable (non-negative) could be considered negative.
- *Benign data-races.* A benign data-race is defined as several concurrent writes of the same data to the same location without explicit synchronization. It is *not* considered a programming error. To achieve this, we extended MAPs to optionally capture the value being written.
- *Array aliasing.* An example of an array aliasing pattern consists of using pointer arithmetic to more easily index an array. For instance, in C, we declare an alias `s_ThreadBase[i]` that

$$(X_1, Y_1) + (X_2, Y_2) = (X_1 \cup X_2, Y_1 \cup Y_2) \qquad aa(X, \mathsf{var}\ x.\ p) = aa(X, p)$$

$$aa(X, o[e]) = (\emptyset, \mathit{fn}(e) \setminus X) \qquad aa(X, \mathsf{skip}) = (\emptyset, \emptyset) \qquad aa(X, p_1\ ;\ p_2) = aa(X, p_1) + aa(X, p_2)$$

$$aa(X, \mathsf{if}\ c\ \{p_1\}\ \mathsf{else}\ \{p_2\}) = (\mathit{fn}(c) \setminus X, \emptyset) + aa(X, p_1) + aa(X, p_2)$$

$$aa(X, \mathsf{for}\ x \in e_1..e_2\ \{p\}) = ((\mathit{fn}(e_1) \cup \mathit{fn}(e_2)) \setminus X, \emptyset) + aa(X \cup \{x \mid X \vdash e_1 \wedge X \vdash e_2\}, p)$$

Fig. 7. The set of approximating binders.

expands to `s_Hist[threadPos + i]` with the following code:
`uchar *s_ThreadBase = s_Hist + threadPos;`

Besides adding new features, we also **improved the following existing features of** `Faial`**:**

- *Bitwise operators.* Our implementation is an encoding of integers as bit-vectors, which are then handled by the Z3 [20] SMT solver. Faial support for bitwise operations was a *partial* encoding that relies on integer arithmetic theory of Z3, which cannot not cope with all bitwise expressions.
- *C++ templates.* Added support for handling arrays of a generic type and indices of a generic type, including support for C++ variable inference with `auto`.
- *Loops.* To make the analysis more precise, FaialAA infers the precise iteration space of a loop when possible. We added support for `while` loops that increment a loop variable at the end of the iteration, and `for`-loops that omit the initialization assignment.

5.2 Approximation Analysis

Our application of the approximation analysis adds the following elements to a data-race alarm:

(1) whether the indexing expressions of both concurrent accesses are DI;
(2) whether the enclosing contexts of both concurrent accesses are CI;
(3) which program variables are CI and DI in the valuation.

Given a protocol, to calculate (1) and (2) more precisely, we can discard any access that is unrelated to the valuation. Then, we can use the $X \vdash_{\mathtt{CI}} p$ and $X \vdash_{\mathtt{DI}} p$ predicates in Figure 4 to obtain (1) and (2). To calculate (3) we use function $aa(X, p)$ defined in Figure 7, which we introduce next.

Function $aa(X, p) = (Y, Z)$ keeps track of a set of exact binders in X and returns a set of control-dependent binders Y, and a set of data-dependent binders Z found within protocol p. The binders of the input protocol are assumed to be distinct so that variables can be compared by name alone without considering the scope. In the implementation, the top-level call of $aa(X, p)$ has X holding the thread-global variables, such as kernel parameters, and thread identifier variables, such as `threadIdx` and `blockIdx`.

In the following example, we show a CIDD protocol, where a symbolic variable x appears in a write to an array. We highlight x with an underline and colored in pink.

$$\begin{aligned}
&aa(\{z\}, \mathsf{var}\ \underline{x}.\ \mathsf{for}\ y \in 0..z\ \{\mathsf{wr}[\underline{x}]\}) \\
=&aa(\{z\}, \mathsf{for}\ y \in 0..z\ \{\mathsf{wr}[\underline{x}]\}) \\
=&(\emptyset, \emptyset) + aa(\{z, y\}, \mathsf{wr}[\underline{x}]) \\
=&(\emptyset, \{\underline{x}\})
\end{aligned}$$

Next, we have a CDDI protocol, where a symbolic variable x appears in the control flow, but is not used to index an array. We highlight x with an underline and colored in pink.

$$\begin{aligned}
&aa(\{z\}, \mathsf{var}\ \underline{x}.\ \mathsf{for}\ y \in 0..\underline{x}\ \{\mathsf{wr}[z]\})\\
=&aa(\{z\}, \mathsf{for}\ y \in 0..\underline{x}\ \{\mathsf{wr}[z]\})\\
=&(\{\underline{x}\}, \emptyset) + aa(\{z\}, \mathsf{wr}[z])\\
=&(\{\underline{x}\}, \emptyset) + (\emptyset, \emptyset)\\
=&(\{\underline{x}\}, \emptyset)
\end{aligned}$$

6 Evaluation

Our evaluation tries to answer 3 main questions, each answered in a subsection.

- **§6.1: How does** FaialAA **compare to the state of the art?** We compare our tool against the state of the art by targeting a well-known dataset. We find that FaialAA outperforms other tools and is able to detect 10 undocumented racy kernels, including 6 that are only found by our tool.
- **§6.2: Can we use** FaialAA **to confirm real defects and fixes?** We analyze the buggy and fixed versions of 6 kernels found in commit messages of large open source projects and show that FaialAA is able to confirm both versions of 5, while others can do the same for only 2 kernels.
- **§6.3: How common are CIDI kernels?** We develop a dataset of 2,770 kernels from GitHub repositories and benchmarks, and find that 59.5% of them are CIDI.

Experimental setup. All experiments were executed in Ubuntu 24.04, on a AMD Ryzen 7 3700X 3.6GHz (16 cores) with 16GB of RAM. For GPUVerify, we used the version included in the artifact of [6], with the command line option `--only-intra-group`. For Faial, we used the version included in the artifact of [16], with the command line option `--parse-gv-args`.

6.1 Static Analysis of GPU Kernels

Our primary goal with this experiment is to run the state-of-the-art of static analysis of GPU kernels against a well-studied group of benchmarks. To this end, we reproduce an experiment carried out by Bardsley *et al.* [6].

Data selection. The *CAV'14* dataset is a well-studied benchmark suite of GPU kernels [6]. The dataset consists of 226 CUDA kernels from 4 benchmark suites: NVIDIA GPU Computing SDK v2.0 (8 kernels), NVIDIA GPU Computing SDK v5.0 (165 kernels), Microsoft C++ AMP Sample Projects (20 kernels), gpgpu-sim benchmarks [4] (33 kernels). The following synchronization mechanisms appear in the dataset: 90 files use `__syncthreads()`, and 2 files use atomics. Both FaialAA and GPUVerify support these synchronization mechanisms. Faial ignores atomics. Kernels are annotated with verification-specific conditions that enable the verification, in the form of kernel pre-conditions and loop invariants. In total, this dataset includes 208 loop-invariant conditions. The DRF analysis of Faial and FaialAA eschews the need for loop invariants, as first discussed in [16]. This experiment is the first to measure the dramatic difference between a MAPs-based approach and a Hoare-logic based approach of GPUVerify (208 more conditions). Finally, we note that the CAV'14 dataset is supposed to be DRF according to its authors (our emphasis):

> Our default assumption is that **these benchmarks are free from defects**, thus our aim is verification. However, in the process of applying GPUVerify we have identified, reported and fixed several data race bugs. [...] We know that some of these failures are

Table 1. Each column tallies a total number of the following. Column "DRF:" DRF kernels. Column "P-Racy:" kernels with at least one alarm and no true alarms. Column "T-Racy:" kernels with at least one true alarm. Column "Coverage:" percentage of analyzable kernels over all kernels, *i.e.*, DRF, P-Racy, or T-Racy. Column "Timeout:" kernels where the analysis timed out (up to 300 seconds). Column "Crash:" kernels where the analysis failed to produce an answer (*e.g.*, parsing error). Column "True:" true alarms produced by the analysis. Column "Potential:" potential alarms produced by the analysis.

	KERNELS						**ALARMS**	
	DRF	P-Racy	T-Racy	Coverage	Timeout	Crash	True	Potential
GPUVerify	194	17	n/a	93.4%	9	6	n/a	50
Faial	207	11	n/a	96.5%	0	8	n/a	21
FaialAA	212	4	10	100.0%	0	0	18	11

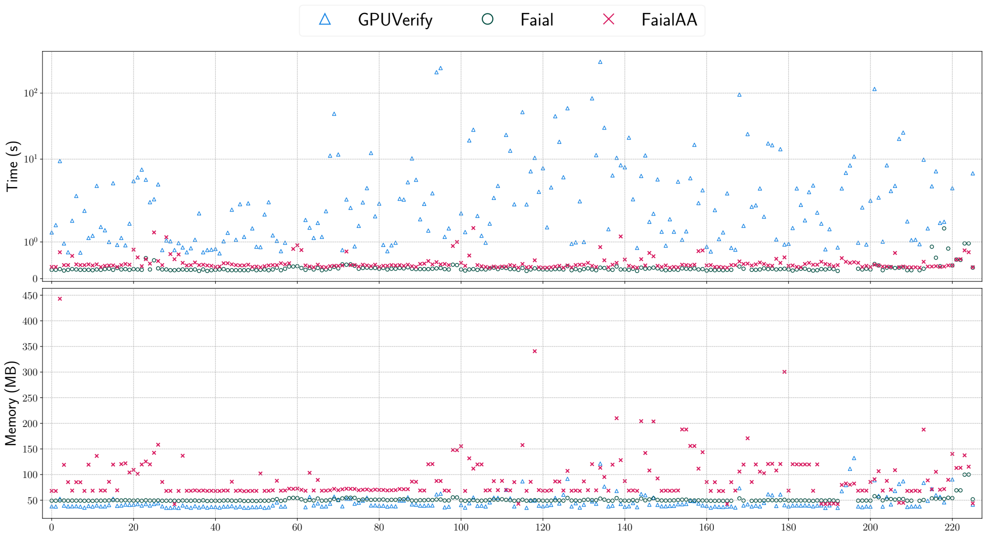


Fig. 8. Runtime (top) and memory usage (bottom) of analyzable kernels. Timeouts and crashes are omitted. The X-axis is the kernel identifier. The Y-axis of the top graph (time) is in logarithmic scale. The Y-axis of the bottom graph (memory) is in linear scale.

(and expect most to be) false positives that demand improved invariant inference, but **some may correspond to further bugs that we have not yet identified**.

Thus, the expectation of this experiment is to have tools report a higher value of DRF, and, ideally, 0 timeouts and 0 crashes (unsupported kernels).

Table 1 tallies the results of the experiment for each tool (rows). In the first section (columns 2–6), we summate the result of the analysis per kernel. In the second section (columns 7–8), we summate the result of the analysis of each alarm. Recall that an alarm corresponds to a report of data-race, hence the same file may yield multiple alarms. Since the analysis of GPUVerify and Faial are *incomplete*, thus any alarms issued by these tools are potential. FaialAA may issue either true or potential alarms, following its approximation analysis. Regarding the first section, the possible status of a kernel summarizes all alarms issued, as given in Table 1. A T-Racy kernel has at least one true alarm (only available in FaialAA). A P-Racy kernel has at least one potential alarm and 0

Table 2. Column Kernel: unique identifier. Column AA: approximation analysis of the kernel. Column LoC: total lines of code. Columns GPUVerify and Faial: ✔ when the analysis reports at least one alarm, ✖ when the analysis reports the kernel DRF, t/o when the analysis times out (up to 300 seconds). Kernels annotated with ^{P} require inter-procedural analysis and with ^{B} require support for bitwise operators.

Kernel	AA	LoC	GPUVerify	Faial
laplace3d^F	CIDI	65	✔	✖
md5^P	CDDD	23	t/o	✖
md5_overlap^P	CDDD	22	t/o	✖
reduceMultiPass^P	CIDI	9	✖	✖
scalarProd	CIDI	83	✔	✖
scalarProdIntraGroup	CIDI	62	✖	✖
sha1^P	CDDD	24	t/o	✖
sha1_overlap^P	CDDD	22	t/o	✖
spPostprocess2D^F	CIDI	52	✔	✖
spPreprocess2D^F	CIDI	56	✔	✖

true alarms. Figure 8 shows the time and memory usage of each tool. Faial and FaialAA take about the same time to analyze kernels, yet FaialAA uses more memory, we suspect that difference stems from our tool being able to analyze more kernels in more detail. The time and memory usage of GPUVerify and Faial are comparable to the evaluation of [15, 16], GPUVerify has better memory usage, and Faial is faster. The average runtime for FaialAA is 0.4 s ± 0.2 s, GPUVerify 9.6 s ± 32.2 s, and Faial 0.3 s ± 0.1 s. As for memory, the average usage for FaialAA is 92.7 MB ± 45.0 MB, GPUVerify 45.7 MB ± 14.9 MB, and Faial 50.9 MB ± 5.3 MB.

We list the number of kernels where multiple tools agree on the same report. All three tools report DRF on 174 kernels (77.0%). There are 4 DRF kernels identified by both GPUVerify and Faial, 21 by Faial and FaialAA, and 14 by FaialAA and GPUVerify. Since Faial and FaialAA share the largest similarity in their source code, it is unsurprising that the two tools have the largest shared reports. There were no racy reports confirmed by all three tools.

Discussion. FaialAA is the tool that can confirm most kernels as DRF, followed by Faial, and then GPUVerify. Further, FaialAA **discovered 10 undocumented racy kernels**. We highlight the importance of being able to categorize alarms as true: GPUVerify identifies as racy 17 kernels. Yet, since there can be multiple alarms per kernel, a user would need to manually validate 50 potential alarms. In contrast, since FaialAA can confirm 18 alarms as true, users only need to validate 11 potential alarms (4.5× fewer). This experiment also shows the improvements of FaialAA over Faial: our tool can analyze 8 more kernels than Faial, which result from unsupported files. Finally, observe that out of 10 true-racy kernels, there are 4 that are either control-dependent or data-dependent, yet their alarms were confirmed true. This is possible because our implementation can restrict the approximation analysis just to the accesses that are involved on the data race and when both accesses are true, the alarm is true *c.f.*, Section 5.2.

Unsound reports. Table 2 lists the racy kernels confirmed by FaialAA, so that we can report on how GPUVerify and Faial fared. The expectation is that each tool reports all 10 kernels as racy. Any DRF (✖) report is unsound and may be caused by unsupported C++ features or bugs in the analysis of the given tool. GPUVerify was able to confirm 4 kernels, was unsound in 2 kernels, and timed out in 4. Faial was able to confirm 0 kernel but was unsound in 10 kernels. We note that neither Faial nor GPUVerify were able to confirm any of the 9 kernels that require inter-procedural analysis, identified with ^{P}. If we subtract the unsound DRF reports, then we get that GPUVerify categorizes 188 kernels as DRF (down from 194) and Faial categorizes 197 kernels as DRF (down from 207).

True alarms. We wanted to better understand the root cause of these 10 racy kernels. We discovered that 7 out of the 10 kernels (those which are annotated with ^{F}) had incorrect or missing pre-conditions. In other words, FaialAA is able to classify as DRF these 7 kernels with a different set of pre-conditions. The original pre-conditions are written by the authors of the CAV'14 dataset. Some of these pre-conditions were retrieved by using runtime information [7], although there is no information on which pre-conditions. To correct the pre-conditions of these 7 kernels, we sampled the parameters of the kernels until FaialAA reported the kernel as DRF. Finally, the kernel `reduceMultiPass` is a known data-race [23] that can be fixed with added synchronization. The remaining kernels, `md5`, `md5_overlap`, `sha1`, and `sha1_overlap`, originate from the gpgpu-sim benchmark; we were unable to fix the defects.

FaialAA confirms **more kernels** as DRF than other tools, with at least **1.9× fewer potential alarms**, and without any analysis errors and timeouts. The approximation analysis of FaialAA is able to find **10 undocumented racy kernels**, including 6 that are missed by GPUVerify and by Faial.

6.2 Confirming Defects in Open Source Projects

This experiment seeks to understand how the state of the art is equipped to confirm known data-races and their fixes. The intent behind our data selection was to curate a set of actual defects in large-scale open source projects, preferably, in code bases that are more up-to-date than those used in the available benchmarks of static analysis. We study 6 kernels found by searching for data races in the commit messages of GitHub repositories.

Data selection. To find buggy kernels, we used GitHub's search to query the terms "fix race" in commit messages. We made our best effort to identify all CUDA kernels that were affected by a data-race, by sieving through multiple results and were able to find 6 buggy kernels. Table 3 lists the kernels we found as well as information regarding the commit where we found the fix. The commits came from 2 projects. The OpenMM [22] project is a high performance molecular dynamics library used for molecular simulation. The Megatron-LM project [42] is a library of training transformer models designed for Large Language Models (LLM), being developed by Nvidia.

Discussion. The experiment is to have each tool analyze a *commit*, which consists of analyzing two versions of a kernel, a buggy version with a data-race and a fixed version that is DRF. For a given commit, we expect each tool to raise an alarm for the buggy kernel and categorize the fixed kernel as DRF. Table 3 lists the results of the experiment. FaialAA is able to correctly categorize both versions of 6 kernels. No tool was able to correctly analyze kernel `sortBucket`, that iteratively issues a bitonic sort [8] on parts of an array. The kernel is challenging for FaialAA because it is `CDDD`: (`CD`) at each iteration a range of elements is being sorted either using shared or global memory and such choice uses data read from an array; (`DD`) the range of elements being sorted is read from another array, so the locations being written are unknown to the analysis. All tools could categorize the commit of kernels `bucketPos` and `compRange`, although only FaialAA can guarantee the data-races as true alarms. The missed data-races are due to unsupported C++ features by GPUVerify and Faial. We found that GPUVerify was unsound in 2 kernels, since GPUVerify incorrectly identifies as bug-free (DRF) the racy versions of kernels `layerNorm` and `gradInput`.

Table 3. Column Kernel: unique identifier. Column LoC: total lines of code. Column Year: the date of the commit. Column Project: the repository holding the commit. Column AA: the approximation analysis of the kernel. Column R for the buggy, racy version of the kernel. Column D for the fixed, DRF version of the kernel. `t/o` when the analysis times out (up to 300 seconds). `n/a` when the analysis aborts. T-R when the analysis reports at least one true alarm. P-R when the analysis reports only potential alarms. D when the analysis reports the kernel DRF. ✖ when the analysis reports an unexpected result.

					GPUVerify		Faial		FaialAA	
Kernel	LoC	Year	Project	AA	R	D	R	D	R	D
`bucketPos`	18	2018	OpenMM	`CIDI`	P-R	D	P-R	D	T-R	D
`compRange`	36	2013	OpenMM	`CIDI`	P-R	D	P-R	D	T-R	D
`reduceVal`	61	2018	OpenMM	`CIDD`	P-R	✖	`n/a`	`n/a`	T-R	D
`sortBucket`	69	2018	OpenMM	`CDDD`	`t/o`	`t/o`	`n/a`	`n/a`	P-R	✖
`gradInput`	112	2021	Megatron-LM	`CDDI`	✖	D	`n/a`	`n/a`	T-R	D
`layerNorm`	147	2021	Megatron-LM	`CIDI`	✖	D	`n/a`	`n/a`	T-R	D

Table 4. FaialAA's approximation analysis on CAV'14, GH'22, and both datasets combined (CAV'14+GH'22). Columns `CIDI`, `CIDD`, `CDDI`,and `CDDD` tally the percentage per approximation analysis outcome of the kernel. Column "Total" tallies the total amount of kernels analyzed.

Dataset	`CIDI`	`CIDD`	`CDDI`	`CDDD`	Total
CAV'14	**74.8%**	8.0%	9.3%	8.0%	226
GH'22	**58.2%**	8.9%	8.4%	24.5%	2,544
Total	**59.5%**	8.8%	8.4%	23.2%	2,770

Our understanding is that GPUVerify ignores accesses that target an array of a generic type, in a type-parametric kernel, which occurs in the data-races of `layerNorm` and `gradInput`. Faial lacks support for C++ templates and inter-procedural analysis, so it is unable to analyze 4 kernels.

FaialAA gives the most **sound and complete** results, confirming 5 out of 6 commits.

6.3 Occurrence of CIDI Kernels in the Wild

Our goal is to understand how often do CIDI kernels appear in the wild, since for such kernels FaialAA is complete. To this end, developed a new dataset of kernels downloaded from GitHub repositories. Our experiment is to run the approximation analysis and report what each kernel is.

Data selection. We introduce a novel dataset, called GH'22, that consists of 2,544 kernels retrieved from GitHub repositories, in July of 2022. We select 2696 files from 261 repositories using GitHub's Search API, queried for the language selected as CUDA, using the default sorting parameters ("best match"). While we removed duplicate files, the same kernel could appear in different repositories. Some files under analysis have missing dependencies, *e.g.*, C headers from a third-party library. In such cases our tool is still able to infer a partial kernel, but any use of missing data and functions is elided. We also include the CAV'14 dataset from Section 6.1 in this experiment.

Discussion. Table 4 shows a breakdown of the dataset into the four possible approximation outcomes, along with the total number of kernels. The analysis of FaialAA is complete for 74.8% on the CAV'14 dataset, and 58.2% on the GH'22 dataset, totaling 59.5% of kernels. In our experience, kernels identified as CD and DD often arise from unsupported source code features: *e.g.*, loop variables not being summarized precisely, or unsupported array aliasing patterns. While there are several reasons that could explain the decrease of CIDI and increase of CDDD in GH'22 when compared against CAV'14, we believe that an important source of CDDD is for kernels that refer to third-party code that is absent from the dataset, which can introduce extraneous sources of approximation.

We found that 59.5% of the 2,770 kernels is CIDI.

7 Related Work

Proving alarms. We build on the work in [52], which introduces a MAPs-inference function and analysis that is sound and complete, and extend this approach in multiple ways, both theoretically and practically. On the theory front, we present a novel notion of *partial completeness* and establish Theorem 3.3 to state phase-determinism (caused by DI) and trace-determinism (caused by CI). We establish a root cause analysis in the presence of CI and DI (Corollary 4.4) and a True Positives result (Theorem 4.5). On the practical front, our work includes an implementation and a comprehensive evaluation of the implementation.

The work by Leung *et al.* [45] also explores a notion akin to CIDI, called *access invariance*, in the context of race detection. Their approach is to develop a partially-sound and complete *dynamic* race detector, whereas ours is a sound and partially-complete static race detector. When a kernel enjoys access invariance, their dynamic detection algorithm is able to prove DRF, which is not possible with other dynamic race detection tools. Besides exploring a different application of a similar insight (CIDI vs access invariance), the theoretical developments in both works are considerably different. Since [45] is designed for dynamic analysis, the semantics being explored is simpler (no control flow, such as loops or branching) and there are no results on partial exactness (Corollary 4.4).

The work in [30] introduces RacerDX and also establishes true positives result (data-races) for a subset of Java multithreaded programs. Both FaialAA and RacerDX are partially complete, however only FaialAA is sound (and therefore able to detect DRF kernels). Besides [30] covering different domains, the two analysis are vastly different, *e.g.*, their work builds on under-approximation, whereas ours builds on over-approximation. Moreover, [30] has no support for partial exactness.

The field of Abstract Interpretation includes works on the completeness of sound analysis for some classes of programs. Rival [66] combines a forward and backward analysis to identify true alarms of critical embedded software. Popeea and Chin [62] introduce completeness in the context of numerical analysis. Ranzato [65] formalize a notion of completeness. Giacobazzi et al. [29] introduce a proof system to show that a given analysis is complete.

Absence of data-races. The state-of-the-art of sound static analysis of data-races includes Faial [15, 16], GPUVerify [9], and PUG [46]. All three tools are capable of detecting when a kernel is DRF, but only report potential data races. FaialAA is the first to detect DRF and true data races. An evaluation of various DRF verification tools conducted in [16] showed that Faial scales better than GPUVerify and PUG in terms of verification time and memory usage. Further, PUG was unable to analyze 62.6% of the kernels. GPUVerify's analysis uses Hoare logic, which require user-provided loop invariants to enable the analysis. In Section 6.1, we show that GPUVerify needed 208 loop invariants in a dataset of 226 kernels. Because FaialAA relies on MAPs, it does not require loop invariants, which greatly simplifies its usage.

Existence of data-races. Data-race finder tools often use dynamic race detection, symbolic execution, or model checking. Dynamic data-race detection techniques [23, 32, 33, 37, 50, 60, 77, 78] monitor the execution of a kernel, and require input data. These techniques impose high memory and time verification overheads, so their usage is limited by the available resources. Symbolic execution and model checking techniques do not require input data, but do not scale well to larger kernels [17, 18, 48, 61, 72]. We believe dynamic approaches are a potentially effective complimentary technique to detect data-races in CDDD kernels.

GPU verification. There is a large body of work in the verification of GPU kernels: static detection of uncoalesced accesses [1, 2], static cost of uncoalesced and bank-conflicts [54], mechanized semantics of CUDA assembly [24], and deductive reasoning of CUDA kernels [11, 39–41].

8 Conclusion

We proposed a novel static analysis technique to detect true data-race in GPU kernels. Our technique relies on analyzing the inference outcome (memory access protocols [15, 16]) and assigning two dimensions of preciseness (CI and DI) to the protocols. Our core calculus Jaminan states the correspondence between protocols and programs according to CI and DI, focusing on the inference of array-operations in CUDA programs. We establish a True Positive Theorem for data-race detection in GPU kernels; a result that identifies a specific class of programs where our analysis reports only true alarms, thus reducing the effort of validating alarms. Given that the state of the art can only flag potential data races, users must manually validate every alarm generated by these tools. Additionally, we establish theoretical results to identify the root cause of spurious alarms due to array accesses either being unreachable or containing imprecise indices. We provide mechanized proofs of our theoretical results in Coq.

Our theory was implemented in FaialAA, as the first sound and partially complete race detector, that can be used to confirm DRF kernels and detect true data-races. Indeed, our evaluation shows that FaialAA emits 1.9× fewer potential alarms compared to Faial and GPUVerify, due to the True Positives Theorem. FaialAA detected 10 undocumented, true data-races in a well-studied benchmark suite of 226 kernels, and correctly confirms 5 pairs of both racy and fixed DRF versions of kernels from open source projects OpenMM and Nvidia's Megatron-LM. FaialAA builds upon Faial, incorporating multiple features designed to analyze a larger number of kernels. These include support for inter-procedural analysis, array aliasing, and the identification of integer bounds based on type. Finally, we found that 59.5% of 2,770 kernels are CIDI.

Data Availability Statement

An artifact with all the tools and datasets presented in Section 6, and mechanized proofs of the theoretical results mentioned in Section 4.4 are publicly available on Zenodo [53]. FaialAA is part of an ongoing open source project available at `https://gitlab.com/umb-svl/faial/`.

Acknowledgments

We thank the anonymous reviewers for numerous helpful remarks. This material is based upon work supported by the National Science Foundation under Grant No. 2204986. We thank Francis Alcos, Gregory Blike, Ayden Diel, Samyak Gangwal, Austin Guiney, Ramsey Harrison, Paul Maynard, Udaya Sathiyamoorth, and Hannah Zicarelli for their contributions to Faial and FaialAA.

References

[1] Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. 2017. GPUDrano: Detecting Uncoalesced Accesses in GPU Programs. In *Proceedings of CAV (LNCS, Vol. 10426)*. Springer, Berlin, Heidelberg, 507–525. https://doi.org/10.1007/978-3-319-63387-9_25

[2] Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. 2022. Static detection of uncoalesced accesses in GPU programs. *Formal Methods in System Design* 60 (2022), 1–32. https://doi.org/10.1007/s10703-021-00362-8

[3] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. 2015. Using Targeted Symbolic Execution for Reducing False-Positives in Dataflow Analysis. In *Proceedings of SOAP*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2771284.2771285

[4] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of ISPASS*. IEEE, Piscataway, NJ, USA, 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[5] Thomas Ball and Sriram K. Rajamani. 2001. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of SPIN*. Springer, Berlin, Heidelberg, 103–122. https://doi.org/10.1007/3-540-45139-0_7

[6] Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. 2014. Engineering a Static Verification Tool for GPU Kernels. In *Proceedings of CAV*, Vol. 8559. Springer, Berlin, Heidelberg, 226–242. https://doi.org/10.1007/978-3-319-08867-9_15

[7] Ethel Bardsley, Alastair F. Donaldson, and John Wickerson. 2014. KernelInterceptor: Automating GPU Kernel Verification by Intercepting Kernels and their Parameters. In *Proceedings of IWOCL*. ACM, New York, NY, USA, Article 7, 5 pages. https://doi.org/10.1145/2664666.2664673

[8] Ken E. Batcher. 1968. Sorting networks and their applications. In *Proceedings of AFIPS*. ACM, New York, NY, USA, 307–314. https://doi.org/10.1145/1468075.1468121

[9] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. 2015. The Design and Implementation of a Verification Technique for GPU Kernels. *Transactions on Programming Languages and Systems* 37, 3, Article 10 (2015), 49 pages. https://doi.org/10.1145/2743017

[10] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a Verifier for GPU Kernels. In *Proceedings of OOPSLA*. ACM, New York, NY, USA, 113–132. https://doi.org/10.1145/2384616.2384625

[11] Stefan Blom, Marieke Huisman, and Matej Mihelčić. 2014. Specification and Verification of GPGPU Programs. *Science of Computer Programming* 95, P3 (2014), 376–388. https://doi.org/10.1016/j.scico.2014.03.013

[12] Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. 2022. Partial (In)Completeness in abstract interpretation: limiting the imprecision in program analysis. *Proceedings of the ACM Programming Languages* 6, POPL, Article 59 (2022), 31 pages. https://doi.org/10.1145/3498721

[13] Prasanth Chatarasi, Jun Shirako, Martin Kong, and Vivek Sarkar. 2017. An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection. In *Proceedings of LCPC'16*. Springer, Berlin, Heidelberg, 106–120. https://doi.org/10.1007/978-3-319-52709-3_10

[14] Bharti Chimdyalwar, Priyanka Darke, Anooj Chavda, Sagar Vaghani, and Avriti Chauhan. 2015. Eliminating Static Analysis False Positives Using Loop Abstraction and Bounded Model Checking. In *Proceedings FM*. Springer, Cham, 573–576. https://doi.org/10.1007/978-3-319-19249-9_35

[15] Tiago Cogumbreiro, Julien Lange, Dennis Liew Zhen Rong, and Hannah Zicarelli. 2023. Memory Access Protocols: Certified Data-Race Freedom for GPU Kernels. *FMSD* (2023), 38 pages. https://doi.org/10.1007/s10703-023-00415-0

[16] Tiago Cogumbreiro, Julien Lange, Dennis Liew Zhen Rong, and Hannah Zicarelli. 2021. Checking Data-Race Freedom of GPU Kernels, Compositionally. In *Proceedings of CAV (LNCS, Vol. 12759)*. ACM, New York, NY, USA, 403–426. https://doi.org/10.1007/978-3-030-81685-8_19

[17] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2014. Symbolic Crosschecking of Data-Parallel Floating-Point Code. *IEEE Transactions on Software Engineering* 40, 7 (2014), 710–737. https://doi.org/10.1109/TSE.2013.2297120

[18] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. 2012. Symbolic Testing of OpenCL Code. In *Proceedings of HVC*. Springer, Berlin, Heidelberg, 203–218. https://doi.org/10.1007/978-3-642-34188-5_18

[19] Priyanka Darke, Bharti Chimdyalwar, Avriti Chauhan, and R. Venkatesh. 2017. Efficient Safety Proofs for Industry-Scale Code Using Abstractions and Bounded Model Checking. In *Proceedings of ICST*. IEEE, Piscataway, NJ, USA, 468–475. https://doi.org/10.1109/ICST.2017.53

[20] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of TACAS*. Springer, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[21] Lorenzo Dematté and Davide Prandi. 2010. GPU computing for systems biology. *Briefings in Bioinformatics* 11, 3 (03 2010), 323–333. https://doi.org/10.1093/bib/bbq006

[22] Peter Eastman, Jason Swails, John D. Chodera, Robert T. McGibbon, Yutong Zhao, Kyle A. Beauchamp, Lee-Ping Wang, Andrew C. Simmonett, Matthew P. Harrigan, Chaya D. Stern, Rafal P. Wiewiora, Bernard R. Brooks, and Vijay S. Pande. 2017. OpenMM 7: Rapid development of high performance algorithms for molecular dynamics. *Computational Biology* 13, 7 (07 2017), 1–17. https://doi.org/10.1371/journal.pcbi.1005659

[23] 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 PLDI*. ACM, New York, NY, USA, 126–140. https://doi.org/10.1145/3062341.3062342

[24] Benjamin Ferrell, Jun Duan, and Kevin W. Hamlen. 2019. CUDA au Coq: A Framework for Machine-validating GPU Assembly Programs. In *Proceedings of DATE*. IEEE, Piscataway, NJ, USA, 474–479. https://doi.org/10.23919/DATE.2019.

8715160
[25] Mikhail R. Gadelha, Enrico Steffinlongo, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. 2019. SMT-Based Refutation of Spurious Bug Reports in the Clang Static Analyzer. In *Proceedings of ICSE-Companion*. IEEE, Piscataway, NJ, USA, 11–14. https://doi.org/10.1109/ICSE-Companion.2019.00026
[26] Pierre Ganty, Francesco Ranzato, and Pedro Valero. 2021. Complete Abstractions for Checking Language Inclusion. *ACM Transactions on Computational Logic* 22, 4, Article 22 (sep 2021), 40 pages. https://doi.org/10.1145/3462673
[27] A. Yu. Gerasimov. 2018. Directed Dynamic Symbolic Execution for Static Analysis Warnings Confirmation. *Programming and Computer Software* 44, 5 (2018), 316–323. https://doi.org/10.1134/S036176881805002X
[28] A. Yu. Gerasimov, L. V. Kruglov, M. K. Ermakov, and S. P. Vartanov. 2018. An Approach to Reachability Determination for Static Analysis Defects with the Help of Dynamic Symbolic Execution. *Programming Computer Software* 44, 6 (2018), 467–475. https://doi.org/10.1134/S0361768818060051
[29] Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. 2015. Analyzing Program Analyses. In *Proceedings of POPL*. ACM, New York, NY, USA, 261—273. https://doi.org/10.1145/2676726.2676987
[30] Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. 2019. A true positives theorem for a static race detector. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. https://doi.org/10.1145/3290370
[31] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy Abstraction. In *Proceedings of POPL*. ACM, New York, NY, USA, 58–70. https://doi.org/10.1145/503272.503279
[32] Anup Holey, Vineeth Mekkat, and Antonia Zhai. 2013. HAccRG: Hardware-Accelerated Data Race Detection in GPUs. In *Proceedings of ICPP*. IEEE, Piscataway, NJ, USA, 60–69. https://doi.org/10.1109/ICPP.2013.15
[33] John Jacobson, Martin Burtscher, and Ganesh Gopalakrishnan. 2024. HiRace: Accurate and Fast Data Race Checking for GPU Programs. In *Proceedings of SC*. IEEE, Piscataway, NJ, USA, 12 pages.
[34] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2019. Exploiting Bank Conflict-Based Side-Channel Timing Leakage of GPUs. *ACM Transactions on Architecture and Code Optimization* 16, 4, Article 42 (2019), 24 pages. https://doi.org/10.1145/3361870
[35] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of ICSE*. IEEE, Piscataway, NJ, USA, 672–681. https://doi.org/10.1109/ICSE.2013.6606613
[36] Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. 2021. Validating Static Warnings via Testing Code Fragments. In *Proceedings of ISSTA*. ACM, New York, NY, USA, 540–552. https://doi.org/10.1145/3460319.3464832
[37] Aditya K. Kamath, Alvin A. George, and Arkaprava Basu. 2020. ScoRD: A Scoped Race Detector for GPUs. In *Proceedings of ISCA*. IEEE, Piscataway, NJ, USA, 1036–1049. https://doi.org/10.1109/ISCA45697.2020.00088
[38] Youil Kim, Jooyong Lee, Hwansoo Han, and Kwang-Moo Choe. 2010. Filtering False Alarms of Buffer Overflow Analysis Using SMT Solvers. *Information and Software Technology* 52, 2 (2010), 210–219. https://doi.org/10.1016/j.infsof.2009.10.004
[39] Kensuke Kojima and Atsushi Igarashi. 2013. A Hoare Logic for SIMT Programs. In *Proceedings of APLAS*, Vol. 8301. Springer, Berlin, Heidelberg, 58–73. https://doi.org/10.1007/978-3-319-03542-0_5
[40] Kensuke Kojima and Atsushi Igarashi. 2017. A Hoare Logic for GPU Kernels. *Transactions on Computational Logic* 18, 1, Article 3 (2017), 43 pages. https://doi.org/10.1145/3001834
[41] Kensuke Kojima, Akifumi Imanishi, and Atsushi Igarashi. 2018. Automated Verification of Functional Correctness of Race-Free GPU Programs. *Journal of Automated Reasoning* 60, 3 (2018), 279–298. https://doi.org/10.1007/s10817-017-9428-2
[42] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. In *Proceedings of MLSys*, Vol. 5. Curran, Red Hook, NY, USA, 341–353. https://proceedings.mlsys.org/paper_files/paper/2023/file/80083951326cf5b35e5100260d64ed81-Paper-mlsys2023.pdf
[43] Shrawan Kumar, Bharti Chimdyalwar, and Ulka Shrotri. 2013. Precise Range Analysis on Large Industry Code. In *Proceedings of ESEC/FSE*. Association for Computing Machinery, New York, NY, USA, 675–678. https://doi.org/10.1145/2491411.2494569
[44] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO*. IEEE, Piscataway, NJ, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665
[45] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU kernels by test amplification. In *Proceedings of PLDI*. ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/2254064.2254110
[46] Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of FSE*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/1882291.1882320
[47] Guodong Li and Ganesh Gopalakrishnan. 2012. Parameterized Verification of GPU Kernel Programs. In *Proceedings of IPDPSW*. IEEE, Piscataway, NJ, USA, 2450–2459. https://doi.org/10.1109/IPDPSW.2012.302

[48] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012. GKLEE: Concolic Verification and Test Generation for GPUs. In *Proceedings of PPoPP*, Vol. 47. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/2370036.2145844

[49] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, and Heejo Lee. 2013. Software Vulnerability Detection Using Backward Trace Analysis and Symbolic Execution. In *Proceedings of ARES*. IEEE, Piscataway, NJ, USA, 446–454. https://doi.org/10.1109/ARES.2013.59

[50] Pengcheng Li, Xiaoyu Hu, Dong Chen, Jacob Brock, Hao Luo, Eddy Z. Zhang, and Chen Ding. 2017. LD: Low-Overhead GPU Race Detection Without Access Monitoring. *Transactions on Architecture and Code Optimization* 14, 1, Article 9 (2017), 25 pages. https://doi.org/10.1145/3046678

[51] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2014. Practical Symbolic Race Checking of GPU Programs. In *Proceedings of SC*. IEEE, Piscataway, NJ, USA, 179–190. https://doi.org/10.1109/SC.2014.20

[52] Dennis Liew, Tiago Cogumbreiro, and Julien Lange. 2022. Provable GPU Data-Races in Static Race Detection. In *Proceedings of PLACES (EPTCS, Vol. 356)*. OPA, Waterloo, Australia, 36–45. https://doi.org/10.4204/EPTCS.356.4

[53] Dennis Liew, Tiago Cogumbreiro, and Julien Lange. 2024. *Sound and partially-complete static analysis of data-races in GPU programs (Artifact)*. https://doi.org/10.5281/zenodo.12666682

[54] Stefan K. Muller and Jan Hoffmann. 2021. Modeling and Analyzing Evaluation Cost of CUDA Kernels. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 25 (2021), 31 pages. https://doi.org/10.1145/3434306

[55] Tukaram Muske and Uday P. Khedker. 2015. Efficient elimination of false positives using static analysis. In *Proceedings of ISSRE*. IEEE, Piscataway, NJ, USA, 270–280. https://doi.org/10.1109/ISSRE.2015.7381820

[56] Tukaram Muske and Alexander Serebrenik. 2020. Techniques for Efficient Automated Elimination of False Positives. In *Proceedings of SCAM*. IEEE, Piscataway, NJ, USA, 259–263. https://doi.org/10.1109/SCAM51674.2020.00035

[57] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. 2019. Explaining Static Analysis — A Perspective. In *Proceedings of ASEW*. IEEE, Piscataway, NJ, USA, 29–32. https://doi.org/10.1109/ASEW.2019.00023

[58] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. 2019. Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review* 52, 1 (2019), 77–124. https://doi.org/10.1007/s10462-018-09679-z

[59] Thu-Trang Nguyen, Toshiaki Aoki, Takashi Tomita, and Iori Yamada. 2019. Multiple Program Analysis Techniques Enable Precise Check for SEI CERT C Coding Standard. In *Proceedings of APSEC*. IEEE, Piscataway, NJ, USA, 70–77. https://doi.org/10.1109/APSEC48747.2019.00019

[60] Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: A Dynamic CUDA Race Detector. In *Proceedings of PLDI*. ACM, New York, NY, USA, 390–403. https://doi.org/10.1145/3192366.3192368

[61] Phillipe Pereira, Higo Albuquerque, Hendrio Marques, Isabela Silva, Celso Carvalho, Lucas Cordeiro, Vanessa Santos, and Ricardo Ferreira. 2016. Verifying CUDA Programs Using SMT-Based Context-Bounded Model Checking. In *Proceedings of SAC*. ACM, New York, NY, USA, 1648–1653. https://doi.org/10.1145/2851613.2851830

[62] Corneliu Popeea and Wei-Ngan Chin. 2013. Dual analysis for proving safety and finding bugs. *Science of Computer Programming* 78, 4 (2013), 390–411. https://doi.org/10.1016/j.scico.2012.07.004

[63] H. Post, C. Sinz, A. Kaiser, and T. Gorges. 2008. Reducing False Positives by Combining Abstract Interpretation and Bounded Model Checking. In *Proceedings of ASE*. IEEE, Piscataway, NJ, USA, 188–197. https://doi.org/10.1109/ASE.2008.29

[64] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of PLDI*. ACM, New York, NY, USA, 722–735. https://doi.org/10.1145/3192366.3192417

[65] Francesco Ranzato. 2013. Complete Abstractions Everywhere. In *Proceedings of VMCAI*. Springer, Berlin, Heidelberg, 15–26. https://doi.org/10.1007/978-3-642-35873-9_3

[66] Xavier Rival. 2005. Understanding the Origin of Alarms in ASTRÉE. In *Proceedings of SAS*. Springer, Berlin, Heidelberg, 303–319. https://doi.org/10.1007/11547662_21

[67] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66. https://doi.org/10.1145/3188720

[68] Lin Shi, Wen Liu, Heye Zhang, Yongming Xie, and Defeng Wang. 2012. A survey of GPU-based medical image computing techniques. *Quantitative imaging in medicine and surgery* 2 (2012), 188–206. Issue 3. https://doi.org/10.3978/j.issn.2223-4292.2012.08.02

[69] John E. Stone, David J. Hardy, Ivan S. Ufimtsev, and Klaus Schulten. 2010. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling* 29, 2 (2010), 116–125. https://doi.org/10.1016/j.jmgm.2010.06.010

[70] Manuel Valdiviezo, Cristina Cifuentes, and Padmanabhan Krishnan. 2014. A Method for Scalable and Precise Bug Finding Using Program Analysis and Model Checking. In *Proceedings of APLAS*. Springer, Cham, 196–215. https://doi.org/10.1007/978-3-319-12736-1_11

[71] Lei Wang, Qiang Zhang, and PengChao Zhao. 2008. Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking. In *Proceedings of SCAM*. IEEE, Piscataway, NJ, USA, 165–173. https://doi.org/10.1109/

SCAM.2008.24

[72] Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2020. Simulee: Detecting CUDA Synchronization Bugs via Memory-Access Modeling. In *Proceedings of ICSE*. ACM, New York, NY, USA, 937–948. https://doi.org/10.1145/3377811.3380358

[73] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. 2012. Concurrency Attacks. In *Proceedings of HotPar*. USENIX, Berkeley, CA, USA, 15–15. https://www.usenix.org/conference/hotpar12/concurrency-attacks

[74] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. 2013. Array dataflow analysis for polyhedral X10 programs. In *Proceedings of PPoPP* (Shenzhen, China). ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/2442516.2442520

[75] Bin Zhang, Chao Feng, Bo Wu, and Chaojing Tang. 2016. Detecting integer overflow in Windows binary executables based on symbolic execution. In *Proceedings of SNPD*. IEEE, Piscataway, NJ, USA, 385–390. https://doi.org/10.1109/SNPD.2016.7515929

[76] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. 2018. OWL: Understanding and Detecting Concurrency Attacks. In *Proceedings of DSN*. IEEE, Piscataway, NJ, USA, 219–230. https://doi.org/10.1109/DSN.2018.00033

[77] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: A Low-overhead Mechanism for Detecting Data Races in GPU Programs. In *Proceedings of PPoPP*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1941553.1941574

[78] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2014. GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. *Transactions on Parallel and Distributed Systems* 25, 1 (2014), 104–115. https://doi.org/10.1109/TPDS.2013.44

Received 2024-04-05; accepted 2024-08-18