Provable GPU Data-Races in Static Race Detection

Dennis Liew[1] Tiago Cogumbreiro[1] Julien Lange[2]
[1] University of Massachusetts Boston, Boston, USA [2] Royal Holloway, University of London, Egham, UK
zhenrong.liew001@umb.edu, tiago.cogumbreiro@umb.edu, julien.lange@rhul.ac.uk

We extend the theory behind the Faial tool-chain, which can soundly prove that CUDA programs (aka, kernels) are data-race free using specialized behavioral types called memory access protocols (MAPs). In this paper we extend the theory of MAPs to characterize kernels for which alarms can be identified as true alarms. We introduce a core calculus for CUDA, which we named BabyCUDA, and a behavioral type system for it. We show that if a BabyCUDA program can be assigned a MAP, then any alarm raised by Faial for this program is a true alarm.

1 Introduction

GPUs have become widespread in many computationally-demanding domains, thanks to their parallel processing capabilities. They are notably used in mission critical software (*e.g.*, self-driving cars); hence the correctness of GPU programs is essential. While several platforms (*e.g.*, CUDA) provide support for general purpose programming using GPUs, writing correct GPU programs (aka. kernels) is notoriously error prone. This is notably due to their unusual concurrency models and low-level programming language support. Bugs in low-level concurrent programming are often due to data-races, *i.e.*, when two or more threads access concurrently the same location in shared memory, and at least one access is a write; causing an unexpected source of nondeterminism. Such races can cause system failures that endanger both the safety and security of critical systems.

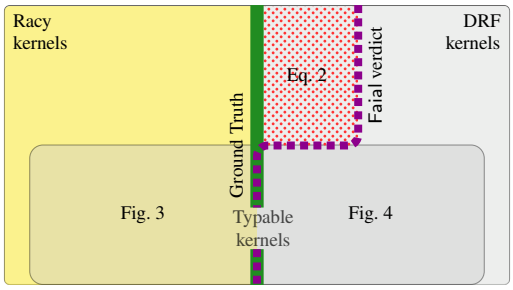

Figure 1: Completeness gap in Faial [2].

Over the last decade, several authors proposed static approaches to verify data-race freedom (DRF) of GPU kernels [1, 2, 9]. While these techniques guarantee the absence of bugs, they may report false alarms. Further, research shows that false alarms hamper the adoption of static analysis in industrial settings [7, 16]. The objective of this paper is to characterize a class of GPU kernels that can be verified *soundly* and *completely*.

Our approach is to extend the theory behind Faial [2], a static DRF analysis tool, so that alarms can be identified as true alarms, *i.e.*, provable data-races. Faial centers its verification around an abstraction called *memory access protocols* (MAP): a form of behavioral types that expresses where data is written to (resp. read from) shared arrays, but abstracts away the content of arrays. Faial leverages the fact that MAP does not represent the array's data to improve the performance of the DRF analysis. The goal

M. Carbone and R. Neykova (Eds.): 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software
EPTCS 356, 2022, pp. 36–45, doi:10.4204/EPTCS.356.4

$$
\begin{array}{llll}
\mathbb{N} \ni & i & ::= & 0 \mid 1 \mid \cdots \\
& n & ::= & x \mid i \mid n \star n \\
& o & ::= & \mathsf{wr} \mid \mathsf{rd} \\
\mathbb{B} \ni & c & ::= & \texttt{true} \mid \texttt{false} \mid n \diamond n \mid c \circ c \\
\mathcal{U} \ni & u & ::= & \mathsf{skip} \mid o[i] \mid u\,;u \mid \mathsf{if}\ c\ \{u\}\ \mathsf{else}\ \{u\} \mid \mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\} \\
\mathcal{S} \ni & p & ::= & \mathsf{sync} \mid p\,;p \mid \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\}
\end{array}
$$

Figure 2: Syntax of MAPs.

of this paper is to introduce a type system that identifies kernels for which the DRF analysis in [2] is sound and complete.

Contributions. Figure 1 illustrates the contribution of this paper in the context of static verification of GPU kernels. It is generally undecidable to determine whether a given kernel is racy or data-race-free (DRF). For a given kernel, Faial always gives a sound verdict (all bugs are caught) but may report spurious data-races (*i.e.*, the dotted area in Figure 1). Using a novel formal subset of CUDA, dubbed BabyCUDA, we characterize a subset of kernels, for which Faial's verdicts align with the ground truth. Our characterization takes the form of a syntax-directed typing system whereby when a BabyCUDA program is assigned a memory access protocol, then this protocol reflects and preserves the behavior of its kernel. We note that [2] studies formally the verification of MAPs, but it only informally presents the inference step from CUDA programs to MAPs. This paper addresses that concern via BabyCUDA, as our overarching goal is to mechanize the entire Faial tool-chain.

2 CUDA Programming and Memory Access Protocols

CUDA follows the Single-Instruction-Multiple-Threads (SIMT) programming model where multiple threads execute a copy of a GPU program (the kernel) in lockstep. Threads communicate over *shared arrays* that all threads can write to (resp. read from). CUDA includes a variable that uniquely identifies each thread to help, among other things, partition arrays among threads. Threads can await each other via barrier synchronization with `__syncthreads()`. A write to shared memory by one thread is only guaranteed to be visible by other threads after a barrier synchronization.

MAPs are behavioral types that codify how multiple threads interact over shared memory. MAPs act as an over-approximation of the behavior of CUDA program. In [2] we have shown that it is possible to verify whether or not MAPs are DRF using a transformation to Satisfiability Modulo Theories (SMT).

We review the syntax of MAPs in Figure 2. Hereafter, we use the following meta-variables for natural numbers ($\mathbb{N}$, nonnegative integers): i for thread identifiers, j for array indices, and k for array contents. An arithmetic expression n is either: a numeric variable x, a thread identifier variable i, or a binary operation on naturals yielding a natural. Boolean expression c is either: a boolean literal, an arithmetic comparison $\diamond$, or a propositional logic connective $\circ$. We note that CUDA denotes the

```
1 for (int x=0; x<M; x++) {
2     int y = A[x];
3     A[x] = y + 1;
4 }
```

```
1 for^U x ∈ 0..M {
2     let y = A[x] in
3     A[x] := y + 1
4 }
```

```
1 for^U x ∈ 0..M {
2     rd[x];
3     wr[x]
4 }
```

Figure 3: Racy example – CUDA (left), BabyCUDA (middle), MAP (right)

assignment and comparison operators as $=$ and $==$ respectively, whereas MAPs and BabyCUDA denotes the operators as $:=$ and $=$ respectively.

The language of protocols [2] distinguishes syntactically between protocol fragments that contain a barrier synchronization from those that do not. Faial infers synchronized fragments automatically, and leverages such distinction to greatly improve the performance of verifying DRF. Unsynchronized protocols contain no synchronization constructs. These include no-ops (skip), array accesses, unsynchronized sequencing, conditionals, and unsynchronized loops ($\mathsf{for}^{\mathsf{U}}$). Accessing an array is denoted as $o[j]$ where j is an index and o specifies whether the position is read (rd) or written (wr). Synchronized protocols $u \in \mathcal{U}$ on the other hand, must contain a synchronization construct. These include the barrier synchronization (sync), synchronized sequencing, and sync-loops ($\mathsf{for}^{\mathsf{S}}$). Hereafter, we focus on inferring protocols for the unsynchronized fragment. The treatment of the synchronized fragment follows similar (but simpler) lines.

Figure 3 (left) illustrates a program with a read and write access within a for-loop represented in CUDA. This kernel is racy because all threads concurrently execute the same CUDA program. In this example, all threads read and write to array `A` at the same index `x` at each loop iteration. In the protocol (right), only accesses (read/write and index) are retained. Hence the race is also visible, and can be verified using an off-the-shelf SMT solver.

Figure 4 (left) illustrates a minimal DRF kernel using conditionals. This kernel performs a write access on a single thread (with identifier $\mathsf{tid} = 0$) while all other threads perform a no-op. The kernel is DRF since no other array accesses occurs.

3 BabyCUDA: a Core Subset of CUDA

In this section we introduce the syntax and semantics of our calculus, BabyCUDA, that captures the concurrency mechanism and memory accesses of CUDA.

Syntax. We give the formal grammar of BabyCUDA in Figure 5. Our analysis assigns one behavioral type per array, so `A` denotes the only shared memory array accessible by all threads. We discuss the various constructs of the language with our two running examples.

Figure 3. BabyCUDA loops have an upper and lower bounds, and a loop stride of 1. In Line 1, the loop variable x starts at the lower bound of 0 and loops until $x = M - 1$. Besides loops, the only form of creating a local variable is by reading data from an array. In Line 2, we read from array `A` and store the result in local variable y. In Line 3, we mutate the array by incrementing the value of element x.

```
1 if (tid==0) {
2   A[0] = tid;
3 }
```

```
1 if (tid=0) {
2   A[0] := tid
3 } else { skip }
```

```
1 if (tid=0)
2  { wr[0] }
3 else
4  { skip }
```

Figure 4: DRF example – CUDA (left), BabyCUDA (middle), MAP (right).

Figure 4. This example introduces BabyCUDA conditionals and the skip. In Line 1, we show a conditional, which unlike C, must always include the else branch. Our calculus does not allow for mutation in the condition. So, any reads/writes that would appear in a condition, must be hoisted before the conditional. The skip in Line 2 denotes a no-op, as usual. Sequencing—not seen in the examples—is standard.

Semantics. We now discuss the semantics of BabyCUDA. We start by describing how we represent the runtime state of the array and formally define data-races. We then introduce our operational semantics.

Evaluating a program yields an *access history* (or, just history), which is defined as a collection of memory accesses. A memory access consists of the identifier of the thread, whether it was a read or a write, the index, and the data written if any. Histories organize accesses in terms of synchronization phases, *i.e.*, a history H is a list of phases. We define a list inductively, as usual, the empty list is $[\,]$, we construct a list with $P::H$, which prepends some element P to a list H. We also use the usual notation $[P_1,\ldots,P_n]$ to denote a list. Each phase groups the accesses per thread, and also distinguishes reads from writes. A phase, ranged over by P and Q, maps each thread identifier $i \in \mathcal{T}$ into a pair of reads and writes $A = (R,W)$. Pairs of reads-writes are ranged over by A, B, and C. Reads are a set of naturals, ranged over by R, and denote the indices read by a given thread. Writes map naturals into naturals, are ranged over by W, and denote the indices and respective data written by a given thread.

The following is an example of a history. Histories are ordered chronologically from right-to-left. We present the earliest phase first. Phase 0 states that thread 0 wrote number 9 to index 1, and thread 1 wrote number 10 to index 2. Phase 1 states that both threads wrote their thread identifier to index 0. Additionally, thread 1 read from index 0.

$$[\overbrace{\{0\colon(\emptyset,\{0:0\}),1\colon(\{0\},\{0:1\})\}}^{\text{phase 1}},\overbrace{\{0\colon(\emptyset,\{1:9\}),1\colon(\emptyset,\{2:10\})\}}^{\text{phase 0}}] \tag{1}$$

For instance, in Equation (1): phase 0 is *not* racy; phase 1 is racy (there is a write-write and a read-write data-races); the history is not safe, as phase 1 is racy.

Definition 1 (DRF program)**.** *P is* racy *if there exist identifiers $i_1,i_2 \in \mathcal{T}$ and index $j \in \mathbb{N}$ such that $i_1 \neq i_2$, $P(i_1) = (R_1,W_1)$, $j \in dom(W_1)$, $P(i_2) = (R_2,W_2)$, and $j \in R_2 \cup dom(W_2)$. P is DRF if P is not racy. H is DRF if every member of H is DRF. A program b is* DRF *if $[\emptyset],b\downarrow H$ and H is DRF.*

The semantics, given in Figure 5, is divided into two big-step operational semantics judgments First, the evaluation of a single thread. Second, the evaluation of a parallel program.

Single-threaded semantics. The semantics of a single thread is given by $A, b \downarrow_{H,i} B$, where A, b, H, and i are input parameters, and B is the result of evaluation. The read-write pair A represents the current phase. Parameter b states the program being evaluated. Parameter history H represents the phases preceding A, necessary to look up the contents of the array, and remains unchanged throughout evaluation. The identifier i states which thread is executing. Rules for control-flow (sequencing, conditionals, and looping) and for skip are standard. Next, we explain the rules for reading from and write to an array.

Rule READ acts as a standard let-binder. We evaluate the index expression n down to the index j, which we use as the first parameter of function *lastwrite*. The second parameter of *lastwrite* is the current history: we prepend the read-write set (R,W) of the current thread i to the rest of the history P, given as $\{i\colon (R,W)\}::H$. In the continuation b, we replace x by the last value written to index j, and we also update the read set to include this access j. The notation $b[x := n]$ stands for b in which all the free occurrences of x are replaced by n. The $\mathit{lastwrite}(j, H)$ function takes an index j and a history H, and returns the last value written to j. The function ranges over the phases of history H, from most recent to least recent. When history H is DRF, it can be shown that there is at most one writer per index, in every phase. We denote the domain of a map as $\mathit{dom}(W)$. Lastly, function *lastwrite* is partial, as it is undefined ($\perp$) when no value has been written to index j. Rule WRITE is simple, we evaluate the index i down to j and the payload m down to k and then update the write map. Map $W[j \mapsto k] = W'$ denotes adding the pair j and k to map W, such that $W'(i) = W(i)$ if $i \neq j$ otherwise $W'(i) = k$. Rule SEQ show phase continuity as it passes along the evaluation chain, with B being the intermediate phase and C the final result. Control-flow rules (*e.g.*, conditionals, loops, no-op) are standard. We omit the synchronized BabyCUDA fragment for presentation purposes.

Parallel semantics. The judgment for parallel execution is $H, b \downarrow I$, execute b, given an input history H, and produce an output history I. Rule PAR runs b in each thread $i \in \mathcal{T}$ independently, using the thread-semantics. As is usual in this form of parallelism, we substitute a special variable tid by the thread identifier, $b[\mathsf{tid} := i]$. In order to hide writes by other threads in the current phase P, each thread i only has access to its own read-writes, $P(i)$. All threads have access to the same remaining history H. Lastly, we "merge" the resulting read-writes of each thread to reconstruct the current phase. We note that our semantics ignores a form of memory movement caused by data-races due to evaluating threads independently. In CUDA, a thread *may* indeed observe a racy-write from another thread. However, our simplification still captures the occurrence of data-races, and as such, does not affect the correctness of DRF analysis.

4 A Behavioral Typing System for BabyCUDA

In this section, we introduce a type system that checks a BabyCUDA program against a behavioral type (MAP). We also state our main result that our DRF analysis is sound and complete for well-typed programs.

Figure 6 introduces our three typing judgments. The idea behind our type system is to disallow indexing arrays using data read from arrays — data-dependent array indexing. To this end, our type

Syntax

$$\begin{array}{rcl} \mathcal{B} \ni b & ::= & \mathtt{A}[n] := n \mid \mathsf{let}\ x = \mathtt{A}[n]\ \mathsf{in}\ b \\ & \mid & b\,;b \mid \mathsf{if}\ c\ \{b\}\ \mathsf{else}\ \{b\} \\ & \mid & \mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{b\} \mid \mathsf{skip} \\ H & ::= & [\,] \mid P::H \end{array} \qquad \begin{array}{rcll} \mathcal{T} & \subseteq & \mathbb{N} & \\ R & \in & \mathcal{R} & \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{N}) \\ W & \in & \mathcal{W} & \stackrel{\text{def}}{=} \mathbb{N} \mapsto \mathbb{N} \\ P & \in & \mathcal{T} & \mapsto (\mathcal{R} \times \mathcal{W}) \end{array}$$

Semantics

$\boxed{lastwrite(j,H) = k}$ $\boxed{A, b \downarrow_{H,i} A}$ $\boxed{H, b \downarrow H}$

LASTWRITE-CURR
$$\frac{\exists i\colon P(i) = (R,W) \qquad W(j) = k}{lastwrite(j, P::H) = k}$$

LASTWRITE-PREV
$$\frac{\forall i\colon P(i) = (R,W) \implies j \notin dom(W)}{lastwrite(j, P::H) = lastwrite(j, H)}$$

LASTWRITE-UNDEF
$$\frac{}{lastwrite(j, [\,]) = \bot}$$

READ
$$\frac{n \downarrow j \qquad (R \cup \{j\}, W), b[x := lastwrite(j, \{i\colon (R,W)\}::H)] \downarrow_{H,i} B}{A, \mathsf{let}\ x = \mathtt{A}[n]\ \mathsf{in}\ b \downarrow_{H,i} B}$$

WRITE
$$\frac{n \downarrow j \qquad m \downarrow k}{(R,W), \mathtt{A}[n] := m \downarrow_{H,i} (R, W[j \mapsto k])}$$

SEQ
$$\frac{A, b_1 \downarrow_{H,i} B \quad B, b_2 \downarrow_{H,i} C}{A, b_1\,;b_2 \downarrow_{H,i} C}$$

IF-T
$$\frac{c \downarrow \mathtt{true} \quad A, b_1 \downarrow_{H,i} B}{A, \mathsf{if}\ c\ \{b_1\}\ \mathsf{else}\ \{b_2\} \downarrow_{H,i} B}$$

IF-F
$$\frac{c \downarrow \mathtt{false} \quad A, b_2 \downarrow_{H,i} B}{A, \mathsf{if}\ c\ \{b_1\}\ \mathsf{else}\ \{b_2\} \downarrow_{H,i} B}$$

FOR-1
$$\frac{(n \geq m) \downarrow \mathtt{true}}{A, \mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{b\} \downarrow_{H,i} A}$$

FOR-2
$$\frac{(n < m) \downarrow \mathtt{true} \qquad A, b[x := n] \downarrow_{H,i} B \qquad B, \mathsf{for}^{\mathsf{U}}\ x \in n+1..m\ \{b\} \downarrow_{H,i} C}{A, \mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{b\} \downarrow_{H,i} C}$$

SKIP
$$\frac{}{A, \mathsf{skip} \downarrow_{H,i} A}$$

PAR
$$\frac{Q = \bigcup \{i\colon A \mid P(i), b[\mathsf{tid} := i] \downarrow_{H,i} A \wedge i \in \mathcal{T}\}}{P::H, b \downarrow Q::H}$$

Figure 5: Syntax and semantics of BabyCUDA (unsynchronized fragment).

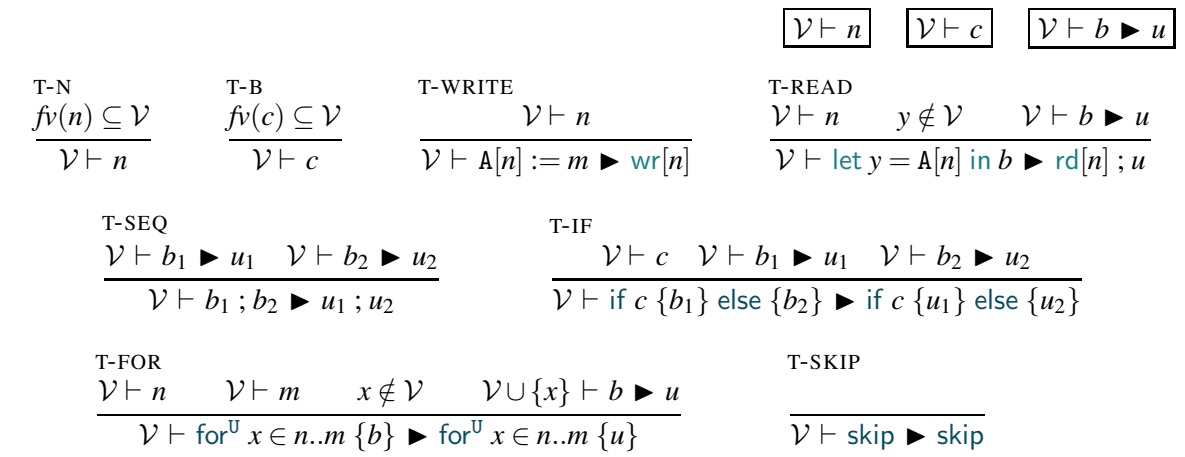

Figure 6: Behavioral type system for BabyCUDA (unsynchronized fragment).

system maintains the set of variables allowed in control flow and in indexing. Let $\mathcal{V}$ range over sets of variables. A well-typed numeric expression $\mathcal{V} \vdash n$ only mentions variables defined in $\mathcal{V}$. The $fv(\cdot)$ function returns the set of free variables of the argument, defined for numeric and boolean expression. Similarly, a well-typed boolean expression $\mathcal{V} \vdash c$ only mentions variables defined in $\mathcal{V}$. Program b has a type u (an unsynchronized protocol), under an environment $\mathcal{V}$. Rule T-WRITE constrains the use of the index expression n. The type of program $\mathtt{A}[n] := m$ is type $\mathsf{wr}[n]$, which does not mention the payload m. Rule T-READ also constrains the use of the index expression n. Since we do not extend the typing environment $\mathcal{V}$ with x, its continuation b cannot use x in to affect control-flow and indexing. To simplify the formalism, we require nested binders are all distinct from each other, hence $x \notin \mathcal{V}$. The type of $\mathsf{let}\ y = \mathtt{A}[n]\ \mathsf{in}\ b$ sequences a read with the type u of the continuation b, that is, $\mathsf{rd}[n]\,;u$. To ensure that nested binders are distinct, we have $x \notin \mathcal{V}$. The remaining rules yield a type that matches the shape of the program. Rules T-SEQ and T-IF are trivial: since there are no variables being declared, we simply propagate the typing environment. Rule T-FOR constrains the upper and lower bound of the loop and allows the use of variable x when typing the loop body b. Rule T-SKIP states that skip is always well-typed.

Examples. Figure 3 (middle) is well-typed under context $\{M, \mathsf{tid}\}$.

$$\{M, \mathsf{tid}\} \vdash \mathsf{for}^{\mathsf{U}}\ x \in 0..M\ \{\mathsf{let}\ y = \mathtt{A}[x]\ \mathsf{in}\ \mathtt{A}[x] := y + 1\} \blacktriangleright \mathsf{for}^{\mathsf{U}}\ x \in 0..M\ \{\mathsf{rd}[x]; \mathsf{wr}[x]\}$$

Figure 4 (middle) is well-typed under context $\{\mathsf{tid}\}$.

$$\{\mathsf{tid}\} \vdash \mathsf{if}\ \mathsf{tid} = 0\ \{\mathtt{A}[0] := \mathsf{tid}\}\ \mathsf{else}\ \{\mathsf{skip}\} \blacktriangleright \mathsf{if}\ \mathsf{tid} = 0\ \{\mathsf{wr}[0]\}\ \mathsf{else}\ \{\mathsf{skip}\}$$

The following is an ill-typed program that falls within the dotted area in Figure 1. The kernel triggers a data-dependent false alarm in Faial, caused by variable x that was read from an array being used to index the array in $\mathtt{A}[x] := 9$. In our experiments [2], we did not find any data-dependent false alarms when dealing with real-world kernels.

$$\{\mathsf{tid}\} \not\vdash \mathtt{A}[\mathsf{tid}] := \mathsf{tid}\ ;\ \mathsf{let}\ x = \mathtt{A}[\mathsf{tid}]\ \mathsf{in}\ \mathtt{A}[x] := 9 \blacktriangleright \mathsf{wr}[\mathsf{tid}]\ ;\ \mathsf{rd}[\mathsf{tid}]\ ;\ \mathsf{wr}[x] \tag{2}$$

Let Λ range over a phase of MAP, defined as a set of access values $\alpha ::= i\colon o[j]$, where $o \in \{\mathsf{rd}, \mathsf{wr}\}$. Further, we define $\alpha \hat{\in} P$ if $\alpha = i\colon o[j]$, $P(i) = (R, W)$, and either $o = \mathsf{rd} \wedge j \in R$, or $o = \mathsf{wr} \wedge j \in dom(W)$.

Theorem 1 (Correctness)**.** *Let $H, b\downarrow H'$ and $u\downarrow\Lambda$. If $\{\mathsf{tid}\} \vdash b \blacktriangleright u$, H is DRF, then H' is DRF if, and only if, P is DRF.*

Proof sketch. We prove a more general result from which this proof follows trivially. Every access value in the source phase is in the target phase, and vice-versa: if $H, b\downarrow P :: H'$, $u\downarrow\Lambda$, $\{\mathsf{tid}\} \vdash b \blacktriangleright u$, and H is DRF, then $\alpha \hat{\in} P$ if, and only if, $\alpha \in \Lambda$. The proof follows by induction on the derivation of $H, b\downarrow P :: H'$.

□

Discussion. The main result establishes that well-typed programs are analyzed soundly and completely. The type system characterizes a subset of BabyCUDA for which all data-race are provably correct. This theorem extends easily to the synchronized fragment.

5 Related Work

Completeness in static analysis. Gorogiannis *et al.* [5] introduce the first DRF static analysis for multithreaded programs that is sound and complete, for a subset of all programs. We note, however, analyzing multithreading (where shared resources are protected with locks) is generally inapplicable (or irrelevant) to GPU programming, and vice-versa. The focus of DRF analysis for multithreading is on lock usage, thread lifecycle, and pointer aliasing, not on array access patterns. Giacobazzi *et al.* [4] develop a deductive system to prove completeness of program analyses over an abstract domain. Ranzato [15] uses completeness to better understand the approximation of some intra-procedural analysis (*e.g.*, signedness, constant propagation) applied to model checking.

DRF analysis for GPUs. Static analysis include [1,2,9]. We note that Faial exhibited the lowest rate of false alarms in [2]. Symbol execution approaches can prove DRF but are unable to scale to larger kernels [10,11,14]. Dynamic data-race detection, *e.g.*, [3,6,8,12,13,18,19], is sound and complete for all programs, but only for a single input and for a single arbitrary thread schedule. Finally, bug finding tools, such as [17] sample thread schedules to identify data-races, but are unable to prove DRF.

6 Conclusion & Future Work

We tackle the problem of formally characterizing true data-races in the context of static analysis for GPU kernels. This paper introduces a core language (BabyCUDA) and a behavioral type system for BabyCUDA. Our main result is that our type system guarantees a sound and complete DRF analysis.

Future work. Following the suggestions of [7, 16], we want to improve upon the confidence of data-race reports, by identifying which accesses can be analyzed soundly *and* completely. To this end, we will extend Faial to annotate accesses according to the output of the analysis introduced in this paper. Additionally, we will continue our work in the Coq mechanization of our main result.

References

[1] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson & John Wickerson (2015): *The Design and Implementation of a Verification Technique for GPU Kernels*. Transactions on Programming Languages and Systems 37(3), pp. 1–49, doi:10.1145/2743017.

[2] Tiago Cogumbreiro, Julien Lange, Dennis Liew & Hannah Zicarelli (2021): *Checking Data-Race Freedom of GPU Kernels, Compositionally*. In: Proceedings of CAV, Springer, pp. 403–426, doi:10.1007/978-3-030-81685-8_19.

[3] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky & Joseph Devietti (2017): *BARRACUDA: Binary-level Analysis of Runtime RAces in CUDA Programs*. In: Proceedings of PLDI, ACM, pp. 126–140, doi:10.1145/3062341.3062342.

[4] Roberto Giacobazzi, Francesco Logozzo & Francesco Ranzato (2015): *Analyzing Program Analyses*. In: Proceedings of POPL, ACM, p. 261–273, doi:10.1145/2676726.2676987.

[5] Nikos Gorogiannis, Peter W. O'Hearn & Ilya Sergey (2019): *A True Positives Theorem for a Static Race Detector*. Proceedings of the ACM on Programming Languages 3(POPL), doi:10.1145/3290370.

[6] Anup Holey, Vineeth Mekkat & Antonia Zhai (2013): *HAccRG: Hardware-Accelerated Data Race Detection in GPUs*. In: Proceedings of ICPP, pp. 60–69, doi:10.1109/ICPP.2013.15.

[7] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill & Robert Bowdidge (2013): *Why Don't Software Developers Use Static Analysis Tools to Find Bugs?* In: Proceedings of ICSE, IEEE, p. 672–681, doi:10.1109/ICSE.2013.6606613.

[8] Aditya K. Kamath, Alvin A. George & Arkaprava Basu (2020): *ScoRD: A Scoped Race Detector for GPUs*. In: Proceedings of ISCA, IEEE, pp. 1036–1049, doi:10.1109/ISCA45697.2020.00088.

[9] Guodong Li & Ganesh Gopalakrishnan (2010): *Scalable SMT-based verification of GPU kernel functions*. In: Proceedings of FSE, ACM, pp. 187–196, doi:10.1145/1882291.1882320.

[10] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh & Sreeranga P. Rajan (2012): *GKLEE: Concolic Verification and Test Generation for GPUs*. In: Proceedings of PPoPP, 47, ACM, pp. 215–224, doi:10.1145/2370036.2145844.

[11] Peng Li, Guodong Li & Ganesh Gopalakrishnan (2014): *Practical Symbolic Race Checking of GPU Programs*. In: Proceedings of SC, IEEE, pp. 179–190, doi:10.1109/SC.2014.20.

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

[13] Yuanfeng Peng, Vinod Grover & Joseph Devietti (2018): *CURD: A Dynamic CUDA Race Detector*. In: Proceedings of PLDI, ACM, pp. 390–403, doi:10.1145/3192366.3192368.

[14] Phillipe Pereira, Higo Albuquerque, Hendrio Marques, Isabela Silva, Celso Carvalho, Lucas Cordeiro, Vanessa Santos & Ricardo Ferreira (2016): *Verifying CUDA Programs Using SMT-Based Context-Bounded Model Checking*. In: Proceedings of SAC, ACM, pp. 1648–1653, doi:10.1145/2851613.2851830.

[15] Francesco Ranzato (2013): *Complete Abstractions Everywhere*. In: Proceedings of VMCAI, 7737, Springer, p. 15–26, doi:10.1007/978-3-642-35873-9_3.

[16] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon & Ciera Jaspan (2018): *Lessons from Building Static Analysis Tools at Google*. Communications of the ACM 61 Issue 4, pp. 58–66, doi:10.1145/3188720.

[17] Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu & Yuqun Zhang (2020): *Simulee: Detecting CUDA Synchronization Bugs via Memory-Access Modeling*. In: Proceedings of ICSE, ACM, pp. 937–948, doi:10.1145/3377811.3380358.

[18] Mai Zheng, Vignesh T. Ravi, Feng Qin & Gagan Agrawal (2011): *GRace: A Low-overhead Mechanism for Detecting Data Races in GPU Programs*. In: Proceedings of PPoPP, ACM, pp. 135–146, doi:10.1145/1941553.1941574.

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