Checking Data-Race Freedom of GPU Kernels, Compositionally

Tiago Cogumbreiro[1](✉), Julien Lange[2], Dennis Liew Zhen Rong[1], and Hannah Zicarelli[1]

[1] University of Massachusetts Boston, Boston, USA
`{tiago.cogumbreiro, zhenrong.liew001, hannah.zicarelli001}@umb.edu`
[2] Royal Holloway, University of London, Egham, UK
`julien.lange@rhul.ac.uk`

Abstract. GPUs offer parallelism as a commodity, but they are difficult to program correctly. Static analyzers that guarantee data-race freedom (DRF) are essential to help programmers establish the correctness of their programs (kernels). However, existing approaches produce too many false alarms and struggle to handle larger programs. To address these limitations we formalize a novel compositional analysis for DRF, based on access memory protocols. These protocols are behavioral types that codify the way threads interact over shared memory.
Our work includes fully mechanized proofs of our theoretical results, the first mechanized proofs in the field of DRF analysis for GPU kernels. Our theory is implemented in Faial, a tool that outperforms the state-of-the-art. Notably, it can correctly verify at least 1.42× more real-world kernels, and it exhibits a linear growth in 4 out of 5 experiments, while others grow exponentially in all 5 experiments.

Keywords: GPU · data-race · static analysis · behavioural types.

1 Introduction

GPUs are massively parallel devices that promise a great return on investment at a cost: they are notably difficult to program. In GPU programming, hundreds of lightweight threads share portions of arrays in parallel (without locks) — very different from the programming model of multithreaded programs written in C or Java with heavy-weight heterogeneous threads. Data-race freedom (DRF) analysis aims to guarantee that for all possible executions, every array cell being written by one thread cannot be concurrently accessed by another thread.

In the field of static analysis of DRF in GPU programs, there is a tension between efficiency and correctness (no missed data-races and no false alarms) that thus far is unresolved. Bug finding tools [26, 27, 33] favor correctness over efficiency: they provide correct results at small scales, by simulating the program execution. Such tools are incapable of handling certain parameters symbolically (*e.g.*, array size) and can easily exhaust users' resources (*e.g.*, loops with long iteration spaces or unknown bounds). Approaches based on Hoare logic [5, 7, 22]

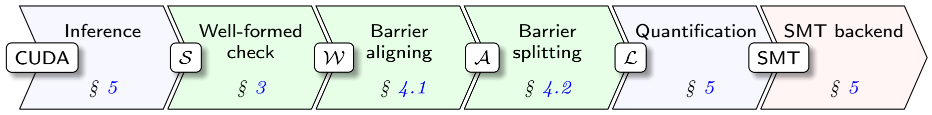


Fig. 1: Work-flow of the verification.

can cope with medium-sized programs, do not miss data-races, and do not require array size information; however, they suffer from a high-rate of false alarms and require code annotations written by concurrency experts. Finally, tools that can cope with larger programs and do not require array size information either miss data-races [24] or overwhelm the user with false alarms [37].

To appease this tension, we introduce a novel static DRF analysis that can handle larger programs and produce fewer false alarms than related work, without missing data-races. Additionally our analysis does not require code annotations or array size information. Our verification framework hinges on *access memory protocols*, a new family of behavioral types [1] that codify the way threads interact through shared memory. Our behavioral types also make evident two aspects of the analysis that can be made separate: concurrency analysis (*i.e.*, could these two expressions run in parallel?) and data-race conflict detection (*i.e.*, do these array indices match?).

Contributions and synopsis This paper includes the following contributions.

(1) In §3, we formalize the syntax, semantics, and well-formedness conditions for access memory protocols, which are behavioral types for GPU programs. This behavioral abstraction results in a simpler yet more expressive theory than previous works, *e.g.*, it does not require user-provided loop invariants.
(2) In §4, we show that our DRF analysis of access memory protocols can be soundly and completely reduced to the satisfiability of an SMT formula, see Theorems 1 and 3. Our theory and results on access memory protocols are fully mechanized in Coq. To the best of our knowledge, this is the first mechanized proof of correctness of a DRF analysis for GPU programs.
(3) We show that our DRF analysis of access memory protocols is compositional when protocols satisfy a structural property, see Theorem 2. Additionally, we show how to transform protocols when they do not meet this property.
(4) In §5 we present Faial, which infers access memory protocols from CUDA kernels and implements our theory. Our experiments show that Faial is more precise and scales better than existing tools.
(5) In §6, we present a thorough experimental evaluation of Faial against related work [5, 24, 26, 27], the largest comparative study of GPU verification (5 tools in 260 kernels, 3 tools compared in 487 kernels). Faial verified 218 out of 227 real-world kernels (at least 1.42× more than other tools) and correctly verified more handcrafted tests than other tools (4 out of 5). In a synthetic benchmark suite (250 kernels), Faial is the only tool to exhibit linear growth in 4 out of 5 experiments, while others grow exponentially in all 5 experiments.

Listing 2.1: Examples of racy kernels, l.h.s. is from [34] and r.h.s. simplifies l.h.s. for clarity, with one-dimensional array and thread identifier, and 1-stride loops.

```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<TILE_DIM; i+=BLOCK_ROWS)
   { tile [ tid .y+i ][ tid .x] = idata [index_in+i*width]; }
  __syncthreads();
  for (int j = 0; j<TILE_DIM; j+=BLOCK_ROWS)
   { odata[index_out+j*height] = tile [ tid .x][ tid .y+j ];}}
```

```
for (int r = 0; r < N; r++) {
  for (int i = 0; i<M; i++)
    { tile [ tid ] = ...;}
  __syncthreads();
  for (int j = 0; j<M; j++)
    {... = tile [ tid +j ];}}
```

Our paper is accompanied by an implementation (Faial), an evaluation framework (inc. datasets), and proof scripts (in Coq) for each theorem. All of these are available in our artifact [9].

2 Overview

This section gives an overview of our approach by examining a data-race we found in published work [17] and [34]. We discuss the challenges that such examples pose to the state-of-the-art of DRF analysis. Then we introduce a verification framework based on *access memory protocols*: behavioral types [1] that codify the way threads interact via shared memory. Figure 1 gives an overview of the verification pipeline. We start from CUDA kernels, from which we infer access memory protocols. Protocols are then checked for well-formedness and transformed in three steps into formulas that are verified by an SMT solver.

2.1 Challenges of GPU Programming

GPU Programming Model The key component of GPU programming is the *kernel* program, or just kernel, that runs according to the Single-Instruction-Multiple-Thread (SIMT) execution model, where multiple threads run a single instruction concurrently. A kernel is parameterized by a special variable that holds a thread identifier, henceforth named tid. In parallel, each member of a group of threads runs an instantiated copy of the kernel by supplying its identifier as an argument. Threads communicate via shared memory (arrays) and mediate communication via barrier synchronization (an execution point where all threads must wait for each other before advancing further). Writes are only guaranteed to be visible to other threads after a barrier synchronization.

GPU programming platforms usually group threads hierarchically in multiple levels, across which no inter-groups synchronization is possible. In both the literature [6, 24] and this work, the focus is on intra-group communication.

Challenges We motivate the difficulty of analyzing data-races by studying a programming error found in the wild, reported in Listing 2.1 (left). This excerpt comes from a tutorial [34] on optimizing numeric algorithms for GPUs. The code listing transposes a matrix N-times with an outer loop indexed by variable r.

Remarkably, the tutorial [34] does not inform the readers that Listing 2.1 contains a subtle *data-race*: one transpose-operation starts (the writes to tile in line 3) without awaiting the termination of the previous transpose-operation (the reads from tile in line 6), thus corrupting the data over time and possibly skewing the timing of the optimization to appear faster than it should be. We found a related data-race in [17], which reuses code from [34].

Our tool, Faial, successfully identifies the program state that triggers the data-race in Listing 2.1: when r=1 and N=2. However, state-of-the-art tools struggle to accurately analyze Listing 2.1, as evaluated in Section 6 (Claim 1: Test 1). Symbolic execution tools, such as [26,27], timeout for N>1, and, in general, cannot handle symbolic (unknown) bounds. GPUVerify [6], a tool based on Hoare logic, reports a false alarm: a spurious data-race when r=0 and N=1. PUG [24] incorrectly identifies the example as DRF, as its analysis appears to ignore data-races originating from different iterations of a loop.

2.2 Memory Access Protocols by Example

We now investigate the data-race in Listing 2.1 with an access memory protocol. For presentation purposes, we focus our discussion on Listing 2.1 (r.h.s.), that simplifies the l.h.s. whilst retaining the root cause of its data-race, which stems from the interaction between both loops. We discuss how we support multi-dimensional arrays, multi-dimensional thread identifiers, and arbitrary loop strides in Section 5. In our Coq formalism the notion of "accesses" (and their dimensions) is a parameter of the theory, thus orthogonal to the theory presented here.

Consider the execution of the end of the first iteration (r=0) and the beginning of the second (r=1) iteration of the outer-loop. In this case, the execution of the j-loop when r=0 is not synchronized with the execution of the i-loop when r=1 as there is no call to `__syncthreads()` in between.

The access memory protocol in Listing 2.2 captures this *partial* execution from the viewpoint of array tile. By design access memory protocols over approximate kernels by abstracting away *what* data is being written to/read from an array, to focus on *where* data is being written. The protocol models the two problematic loops of Listing 2.1, *i.e.*, the j-loop when r=0 and the i-loop when r=1. The first loop reads (rd[tid+j]) from the array, while the second writes (wr[tid]) to it. Evaluation of a protocol follows the SIMT model: each thread evaluates wr[tid] by instantiating tid with their unique identifier (hereafter, an integer), *e.g.*, thread 0 yields wr[0] and thread 1 yields wr[1].

Analysis of Unsynchronized Protocols We say that a protocol is DRF when all concurrent accesses are pair-wise DRF, *i.e.*, when issued by different threads on the same index, then neither access is a write. For instance the respective sets of concurrent accesses of threads 0 and 1 in Listing 2.2 is given below

$$\overset{\mathsf{tid}=0}{\{\mathsf{rd}[j] \mid 0 \le j < M\} \cup \{\mathsf{wr}[0]\}} \quad \textit{DRF with?} \quad \overset{\mathsf{tid}=1}{\{\mathsf{rd}[1+j] \mid 0 \le j < M\} \cup \{\mathsf{wr}[1]\}}$$

Listing 2.2: Minimal representative example of an access memory protocol highlighting the data-race in Listing 2.1.

```
// r = 0
for^U j in 0..M    // for (int j = 0; j<M; j++)
  {rd[tid+j]};     //   _ = tile[tid+i];
// r = 1
for^U i in 0..M    // for (int i = 0; i<M; i++)
  {wr[tid]}        //   tile[tid] = _;
```

When M>1, thread 0 (l.h.s) accesses rd[1] and thread 1 (r.h.s) accesses wr[1]. Thus, there is a data-race on index 1 of the array.

A fundamental challenge of static DRF verification is how to handle loops. Symbolic execution approaches that unroll loops, *e.g.*, [26, 27], cannot handle large nor symbolic iteration spaces. Static approaches that use Hoare logic, *e.g.*, [5, 7, 22], require user-provided loop invariants. Another approach is to reduce loops to verifying the satisfiability of a corresponding universally quantified formula, *e.g.*, [25, 30]. This has the advantage of being fast and not requiring invariants. However, its previous application to GPU programming, *i.e.*, PUG, is unsound due to the interaction between barrier synchronizations and loops, *e.g.*, PUG misses the data-race in Listing 2.1. We give more details in Section 6.

Our approach A key contribution of our work is to identify conditions that allow a kernel to be reduced to a first-order logic formula, by precisely characterizing the effect of barrier synchronization in loops. To this end, the language of access memory protocols distinguishes syntactically between protocol fragments that synchronize from those that do not. For instance, the protocol in Listing 2.2 is identified as *unsynchronized*, as it does not include any synchronization.

In Section 4, we show that the DRF analysis of unsynchronized protocols can be precisely reduced to a first-order logic formula, where universally quantified formulae represent loops, thus obviating the need to unroll them explicitly. For instance, we reduce the verification of Listing 2.2 to asking whether for all M, t_1, and t_2, where $t_1 \neq t_2$ are thread identifiers, the following holds:

$$\forall j_1, i_1, j_2, i_2 \colon 0 \le j_1 < M \land 0 \le i_1 < M \land 0 \le j_2 < M \land 0 \le i_2 < M \implies \\ \{\mathsf{rd}[t_1 + j_1]\} \cup \{\mathsf{wr}[t_1]\} \quad \textit{DRF with?} \quad \{\mathsf{rd}[t_2 + j_2]\} \cup \{\mathsf{wr}[t_2]\}$$

This formula is *unprovable* since $\mathsf{rd}[t_1 + j_1]$ races with $\mathsf{wr}[t_2]$ when, *e.g.*, $t_1 = 0$, $t_2 = 1$, $j_1 = 1$, and $M > 1$. Hence, our technique flags Listing 2.2 as racy.

Analysis of Synchronized Protocols The protocol in Listing 2.3 (left) models *all* the interactions over the shared array tile from Listing 2.1. This protocol consists of one outer loop r that contains two inner loops separated by a barrier synchronization (sync). The first inner loop writes (wr[tid]) to the array, while the second reads (rd[tid + j]) from the array.

This protocol illustrates how our language syntactically differentiates between protocols fragments that synchronize from those that do not. Concretely, Listing 2.3: Access memory protocol (left) of array tile from Listing 2.1 and its aligned version (right).

```
1 for^S r in 0..N {
2   for^U i in 0..M { wr[tid] }
3   sync;
4   for^U j in 0..M { rd[tid + j] }
5 }
```

aligns to

```
1 for^U i in 0..M { wr[tid] }
2 sync;
3 for^S r in 1..N {
4   for^U j in 0..M { rd[tid + j] }
5   for^U i in 0..M { wr[tid] }
6   sync; }
7 for^U j in 0..M { rd[tid + j] }
```

our language precludes an unsynchronized loop ($\mathsf{for}^{\mathtt{U}}\ x \in n..m\ \{u\}$) from calling sync anywhere in u, and it requires that a synchronized loop ($\mathsf{for}^{\mathtt{S}}\ x \in n..m\ \{p\}$) includes at least one occurrence of sync. The superscript U (resp. S) stands for *s*ynchronized (resp. *u*nsynchronized). This distinction can be inferred automatically and yields a compositional analysis, as we explain below.

The behavior of synchronized loops is difficult to analyse because they may contain data-races that span more than one iteration. For instance an instruction of iteration r in Listing 2.3 may race with an instruction of iteration r+1.

Our approach In this work we show that the DRF analysis of synchronized protocols can safely be reduced to a first-order logic formula when such loops are *aligned*, *i.e.*, when there is at least one synchronization exactly before the loop and one at the end of its body. In Section 4.1 we show how to transform an arbitrary access memory protocol into an aligned protocol using a syntax-driven transformation technique called *barrier aligning*. Intuitively, barrier aligning normalizes loops so that they do not "leak" accesses between iterations. The right-hand side of Listing 2.3 shows the result of applying *barrier aligning* on the protocol from Listing 2.3 (left). Observe that the fragment before the aligned loop (line 1) corresponds to the unsynchronized part of the original loop (before sync). The original loop itself is rearranged so that the part succeeding sync is moved to the beginning of the aligned loop (lines 3–6). The fragment following the aligned loop (line 7) corresponds to the unsynchronized loop that appears after the sync in the original protocol.

In Section 4.1 we show that aligned protocols enable *compositional* verification: protocol fragments between two barriers can be analyzed independently. This compositional analysis is possible because (i) there is no causality between instructions, except through sync and (ii) aligned protocols syntactically delimit the causality induced by sync. For instance, the aligned protocol in Listing 2.3 can be reduced to analyzing the following three protocol fragments independently:

$$\mathsf{for}^{\mathtt{U}}\ i \in 0..M\ \{\mathsf{wr}[\mathsf{tid}]\} \qquad \mathsf{for}^{\mathtt{U}}\ j \in 0..M\ \{\mathsf{rd}[\mathsf{tid}+j]\}$$
$$\mathsf{for}^{\mathtt{S}}\ r \in 1..N\ \{\mathsf{for}^{\mathtt{U}}\ j \in 0..M\ \{\mathsf{rd}[\mathsf{tid}+j]\}; \mathsf{for}^{\mathtt{U}}\ i \in 0..M\ \{\mathsf{wr}[\mathsf{tid}]\}; \mathsf{sync}\}$$

The first two protocols are handled like Listing 2.2 because they are unsynchronized. Representing a synchronized loop as a formula becomes possible when the protocol is *aligned*: both threads must share the same value for r at each iteration. Hence, we reduce the verification to asking whether for all N, M, t_1, and t_2 where $t_1 \neq t_2$ and the following holds:

$$\forall r, j_1, i_1, j_2, i_2 : 1 \leq r < N \wedge 0 \leq j_1 < M \wedge 0 \leq i_1 < M \wedge 0 \leq j_2 < M \wedge 0 \leq i_2 < M$$
$$\implies \{\mathsf{rd}[t_1 + j_1]\} \cup \{\mathsf{wr}[t_1]\} \quad \textit{DRF with?} \quad \{\mathsf{rd}[t_2 + j_2]\} \cup \{\mathsf{wr}[t_2]\}$$

Our technique identifies Listing 2.3 as racy since this formula is *unprovable*, *i.e.*, $\mathsf{rd}[t_1{+}j_1]$ races with $\mathsf{wr}[t_2]$ when $r = 1$, $t_1 = 0$, $t_2 = 1$, $j_1 = 1$, $N > 1$ and $M > 1$.

3 Access Memory Protocols

An access memory protocol describes the interaction between a group of threads and a single shared-memory location. A protocol records *where* in memory accesses take place, but abstracts away from *what* data is read from/written to memory. The language of protocols distinguishes between an unsynchronized protocol fragment $u \in \mathcal{U}$, that disallows synchronization, from a synchronized fragment $p \in \mathcal{S}$ that must include a synchronization. The syntax and semantics of access memory protocols is given in Figure 2. Our operational semantics is inspired by the synchronous, delayed semantics (SDV) from Betts et al. [6], where threads execute independently and communicate upon reaching a barrier.

Hereafter, i, j, k are metavariables over non-negative integers picked from the set $\mathbb{N}$. An arithmetic expression n is either: an integer variable x, an integer i, or a binary operation on integers that yields an integer. A boolean expression b is either a boolean literal, an arithmetic comparison $\diamond$, or a propositional logic connective $\circ$. We write $n \downarrow i$ when expression n evaluates to integer i, where evaluation is defined in the natural way. We overload the notation for Boolean expressions, *e.g.*, $b \downarrow \texttt{true}$ means that expression b evaluates to `true`.

Unsynchronized fragment A protocol $u \in \mathcal{U}$ either does nothing (skip), accesses a shared memory location $o[i]$ (reads from/writes to index i), performs sequential composition, or loops. Figure 2 gives the semantics of unsynchronized protocols, which is parameterized by a set of thread identifiers $\mathcal{T} \subseteq \mathbb{N}$, where $|\mathcal{T}| \geq 2$.

Evaluation of an unsynchronized protocol u by a thread identifier i, written $u \downarrow_i P$, yields a *phase*, *i.e.*, a set $P \in \mathcal{P}$ of *access values* $\alpha \in \mathbb{A}$. Each access value, or just access, notation $i{:}o[j]$, consists of its issuing thread identifier i, an access mode o (read/write), and an index j. Protocol skip produces no accesses. A memory access $o[n]$ evaluates the index and creates a singleton phase. Sequencing and looping are standard. Loop ranges include the lower bound and exclude the upper bound. Similarly to SDV, Rule $\mathcal{U}$-par executes a copy of the unsynchronized code u for each thread $i \in \mathcal{T}$ by replacing the special variable tid by the thread identifier, $u[\mathsf{tid} \coloneqq i]$, which results in the union of the accesses of all threads. To simplify the presentation we omit the unsynchronized conditionals, however they are included in our Coq formalism and are fully supported by Faial, see Section 5.

Synchronized fragment A protocol $p \in \mathcal{S}$ may perform barrier synchronization sync, run unsynchronized code u, perform sequential composition, and loop.

Syntax

$$\begin{array}{rcl}
\mathbb{N} \ni i & ::= & 0 \mid 1 \mid \cdots \\
n & ::= & x \mid i \mid n \star n \\
\mathcal{B} \ni b & ::= & \texttt{true} \mid \texttt{false} \mid n \diamond n \mid b \circ b \\
\mathcal{U} \ni u & ::= & \mathsf{skip} \mid o[n] \mid u\,;u \mid \mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\} \\
\mathcal{S} \ni p & ::= & \mathsf{sync} \mid u \mid p\,;p \mid \mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\}
\end{array}
\qquad
\begin{array}{rcl}
o & ::= & \mathsf{wr} \mid \mathsf{rd} \\
\mathbb{A} \ni \alpha & ::= & i{:}o[i] \\
\mathcal{P} \ni P & ::= & \{\alpha_1, \ldots, \alpha_n\} \\
H & ::= & [] \mid P :: H
\end{array}$$

Big-step semantics for $\mathcal{U}$ $\boxed{u \downarrow_i P}$ $\boxed{u \downarrow_{\mathcal{T}} S}$

$$\mathcal{U}\text{-SKIP}\ \frac{}{\mathsf{skip} \downarrow_i \emptyset}
\qquad
\mathcal{U}\text{-ACC}\ \frac{n \downarrow j}{o[n] \downarrow_i \{i{:}o[j]\}}
\qquad
\mathcal{U}\text{-SEQ}\ \frac{u_1 \downarrow_i P_1 \qquad u_2 \downarrow_i P_2}{u_1\,;u_2 \downarrow_i P_1 \cup P_2}
\qquad
\mathcal{U}\text{-FOR-1}\ \frac{(n \geq m) \downarrow \texttt{true}}{\mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\} \downarrow_i \emptyset}$$

$$\mathcal{U}\text{-FOR-2}\ \frac{(n < m) \downarrow \texttt{true} \qquad u[x := n] \downarrow_i P_1 \qquad \mathsf{for}^{\mathsf{U}}\ x \in n+1..m\ \{u\} \downarrow_i P_2}{\mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\} \downarrow_i P_1 \cup P_2}$$

$$\mathcal{U}\text{-PAR}\ \frac{S = \bigcup \{u[\mathsf{tid} := i] \downarrow_i P_i \mid i \in \mathcal{T}\}}{u \downarrow_{\mathcal{T}} S}$$

History concatenation and merging $\boxed{H \cdot H}$ $\boxed{H \odot H}$

$$[P_1 \ldots P_n] \cdot [P_{n+1} \ldots P_{n+k}] = [P_1 \ldots P_{n+k}] \qquad (H \cdot [P]) \odot ([P'] \cdot H') = H \cdot [P \cup P'] \cdot H'$$

Big-step semantics for $\mathcal{S}$ $\boxed{p \downarrow H}$

$$\mathcal{S}\text{-SYNC}\ \frac{}{\mathsf{sync} \downarrow [\emptyset, \emptyset]}
\qquad
\mathcal{S}\text{-PAR}\ \frac{u \downarrow_{\mathcal{T}} P}{u \downarrow [P]}
\qquad
\mathcal{S}\text{-SEQ}\ \frac{p \downarrow H \qquad q \downarrow H'}{p\,;q \downarrow H \odot H'}
\qquad
\mathcal{S}\text{-FOR-1}\ \frac{(n \geq m) \downarrow \texttt{true}}{\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\} \downarrow [\emptyset]}$$

$$\mathcal{S}\text{-FOR-2}\ \frac{(n < m) \downarrow \texttt{true} \qquad p[x := n] \downarrow H \qquad \mathsf{for}^{\mathsf{S}}\ x \in n+1..m\ \{p\} \downarrow H'}{\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\} \downarrow H \odot H'}$$

Structurally well-formed protocols $\boxed{\mathit{swf}(p)}$

$$\mathit{swf}(u\,;\mathsf{sync})
\qquad
\frac{\mathit{swf}(p) \qquad \mathit{swf}(q)}{\mathit{swf}(p\,;q)}
\qquad
\frac{\mathit{swf}(p) \qquad \mathsf{tid} \notin \mathit{fv}(n) \cup \mathit{fv}(m)}{\mathit{swf}(u_1\,;\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;u_2\})}$$

Data-race, safe phase, and safe history $\boxed{\alpha \,\#\, \beta}$ $\boxed{\mathit{safe}(P)}$ $\boxed{\mathit{safe}(H)}$

$$\frac{\mathsf{wr} \in \{o, o'\} \qquad i \neq j}{i{:}o[k] \,\#\, j{:}o'[k]}
\qquad
\frac{\forall \alpha, \beta \in P\colon \neg(\alpha \,\#\, \beta)}{\mathit{safe}(P)}
\qquad
\frac{\forall P \in H\colon \mathit{safe}(P)}{\mathit{safe}(H)}$$

Fig. 2: Syntax, semantics, and properties of access memory protocols.

Figure 2 gives the semantics of a protocol, notation $p \downarrow H$. Evaluation of a protocol p yields a *history* (ranged over by H), *i.e.*, a list of phases (P) that records how memory was accessed. We use $::$ as list constructor and $\cdot$ for the usual list concatenation operator. Histories are merged using the special $\odot$-operator.

A barrier synchronization creates two empty phases, corresponding to phases before and after synchronization. Running an unsynchronized protocol yields a single phase containing all accesses performed by that protocol. Sequencing two synchronized protocols p with q merges the last phase of the former with the first phase of the latter, as these two phases run concurrently. The base case of a synchronized loop produces a singleton history containing the empty phase. Running one iteration of a synchronized loop sequences the history of the first iteration with the rest of the loop, by merging the two histories.

Next, we introduce the notion of well-formed protocols, a restriction of structurally well-formed protocols, see $\mathit{swf}(p)$ in Figure 2. We discuss how well-formedness is enforced in Section 5. We write $\mathit{fv}(p)$ (resp. $\mathit{fv}(n)$) for the free variables of p (resp. n).

Definition 1 (Well-formed protocol, $p \in \mathcal{W}$). *We say that a protocol is well-formed, notation $p \in \mathcal{W}$, when $\mathit{swf}(p)$, $\mathit{fv}(p) \subseteq \{\mathsf{tid}\}$, and every synchronized loop executes at least one iteration.*

DRF is formalized at the bottom of Figure 2. Two accesses are in a data-race (or racy) when there exist two different threads that access the same index k, and one of these accesses is a write. Our definition does not distinguish between harmful and *benign* data races, a data-race in which both threads write the same value. Phase P is *safe* iff each pair of accesses it contains is not racy. History P is *safe* when all of its phases are safe. We say that p is DRF iff $p \downarrow H$ and $\mathit{safe}(H)$.

4 DRF-Preserving Transformations of Protocols

This section presents the main steps of the DRF analysis summarized in Figure 1: barrier aligning (Section 4.1) and splitting (Section 4.2).

This section also includes our key theoretical results. We establish that these steps preserve and reflect data-races (*i.e.*, any and all data-races are found), see Theorem 1 and Theorem 3. We make precise the notion of compositionality that makes our approach scalable in Theorem 2.

4.1 Aligning Protocols

The first transformation step normalizes protocols by aligning synchronized loops, which in turn enables a form of compositional verification. The goal of the transformation is to produce protocols which belong to $\mathcal{A}$, see top of Figure 3.

Barrier aligning (or just aligning) is performed by function *align*, given in the bottom half of Figure 3. The function returns a pair whose first element is an aligned and synchronized protocol, and whose second element is an unsynchronized protocol. Intuitively, the pair represents a sequence which we delimitate

Aligned protocols $\boxed{p \in \mathcal{A}}$

$$u\,;\mathsf{sync} \in \mathcal{A} \qquad \frac{p \in \mathcal{A} \qquad q \in \mathcal{A}}{p\,;q \in \mathcal{A}} \qquad \frac{p \in \mathcal{A} \qquad q \in \mathcal{A}}{p\,;\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{q\} \in \mathcal{A}}$$

Sequencing aligned protocols $\boxed{⨟ \colon \mathcal{U} \to \mathcal{A} \to \mathcal{A}}$ $\boxed{⨟ \colon (\mathcal{A} \times \mathcal{U}) \to (\mathcal{A} \times \mathcal{U}) \to \mathcal{A} \times \mathcal{U}}$

$$u ⨟ (u'\,;\mathsf{sync}) = (u\,;u')\,;\mathsf{sync} \qquad u ⨟ (p\,;q) = (u ⨟ p)\,;q \qquad (p,\,u) ⨟ (q,\,u') = (p\,;(u ⨟ q),\,u')$$

Aligning protocols $\boxed{\mathit{align} \colon \mathcal{W} \to \mathcal{A} \times \mathcal{U}}$

$$\mathit{align}(u\,;\mathsf{sync}) = (u\,;\mathsf{sync},\,\mathsf{skip}) \qquad \mathit{align}(p\,;q) = \mathit{align}(p) ⨟ \mathit{align}(q)$$

$$\frac{\mathit{align}(p) = (q,\,u_3) \qquad q_1 = u_1 ⨟ q[x := n] \qquad u = u_3\,;u_2}{\mathit{align}(u_1\,;\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;u_2\}) = (q_1\,;\mathsf{for}^{\mathsf{S}}\ x \in n{+}1..m\ \{u[x := x{-}1] ⨟ q\},\,u[x := m{-}1])}$$

Fig. 3: Aligning protocols.

syntactically. We note that the output of *align*, say $(q,\,u)$, can be trivially made into an aligned protocol: $q\,;u\,;\mathsf{sync}$. The case for synchronization is simple, *align* returns the input protocol as the first component of the pair and skip as the second component (the input protocol is already fully aligned). The case for sequence consists of the sequential composition of the pair aligned with unsynchronized code using operator $(⨟)$. Sequencing two pairs $(p,\,u) ⨟ (q,\,u')$ amounts to sequencing u to the outer-most piece of unsynchronized code present in q.

Dealing with synchronized loops is more involved. Given a loop $u_1\,;\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\,;u_2\}$, we produce a protocol consisting of the fragment preceding the loop and the synchronized part of its first iteration (q_1), an aligned loop starting at $n{+}1$, and the unsynchronized part of its last iteration $(u[x := m{-}1])$. See Listing 2.3 for an example of protocol aligning. We note that we can always unroll the loop because the analysis only considers non-empty synchronized loops; we discuss how to enforce this assumption in Section 5.

We now state two fundamental properties of barrier aligning: preserving and reflecting DRF (Theorem 1), and enabling compositional verification (Theorem 2). Theorem 1 states that verifying DRF of a well-formed protocol p is equivalent to verifying DRF of its aligned counterpart.

Theorem 1 (Correctness of Align). *If $p \in \mathcal{W}$ and $\mathit{align}(p) = (q,\,u)$, then p is DRF if and only if $q\,;u$ is DRF.*

To state our compositionality result, we introduce a language of contexts:

$$\mathcal{C} ::= [\,\_\,] \mid u\,;\mathsf{sync} \mid p\,;\mathcal{C} \mid \mathcal{C}\,;p \mid \mathcal{C}\,;\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\} \mid p\,;\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{\mathcal{C}\}$$

The base cases correspond to a hole $[\,\_\,]$ or an unsynchronized protocol (followed by sync). The other cases follow the structure of access memory protocols.

Syntax

$$\mathcal{L} \ni h ::= \mathsf{skip} \mid n{:}o[m] \mid h\,;h \mid \mathsf{var}\ x\ \mathsf{in}\ n..m; h$$

Product of histories $\boxed{H \otimes H}$

$$H_1 \otimes H_2 = [P_1 \cup P_2 \mid (P_1, P_2) \in H_1 \times H_2]$$

Big-step semantics $\boxed{h \Downarrow H}$

$$\frac{}{\mathsf{skip} \Downarrow [\emptyset]} \qquad \frac{n \downarrow i \qquad m \downarrow j}{n{:}o[m] \Downarrow [\{i{:}o[j]\}]} \qquad \frac{h_1 \Downarrow H_1 \qquad h_2 \Downarrow H_2}{h_1\,;h_2 \Downarrow H_1 \otimes H_2} \qquad \frac{(n \geq m) \downarrow \mathtt{true}}{\mathsf{var}\ x\ \mathsf{in}\ n..m; h \Downarrow [\emptyset]}$$

$$\frac{(n < m) \downarrow \mathtt{true} \qquad h[x := n] \Downarrow H_1 \qquad \mathsf{var}\ x\ \mathsf{in}\ n+1..m; h \Downarrow H_2}{\mathsf{var}\ x\ \mathsf{in}\ n..m; h \Downarrow H_1 \cdot H_2}$$

Projection $\boxed{\mathit{trace}\colon \mathcal{U} \to \mathcal{L}}$

$$\mathit{trace}(o[n]) = \mathsf{tid}{:}o[n] \qquad \mathit{trace}(\mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\}) = \mathsf{var}\ x\ \mathsf{in}\ n..m; \mathit{trace}(u)$$

$$\mathit{trace}(u_1\,;u_2) = \mathit{trace}(u_1)\,;\mathit{trace}(u_2) \qquad \mathit{trace}(\mathsf{skip}) = \mathsf{skip}$$

Splitting protocols $\boxed{\mathit{split}\colon \mathcal{A} \to [\mathcal{L}]}$

$$\mathit{split}(p\,;q) = \mathit{split}(p) \cdot \mathit{split}(q)$$

$$\frac{t_1, t_2 \text{ fresh} \qquad h_1 = \mathit{trace}(u)[\mathsf{tid} := t_1] \qquad h_2 = \mathit{trace}(u)[\mathsf{tid} := t_2]}{\mathit{split}(u\,;\mathsf{sync}) = [\mathsf{var}\ t_1\ \mathsf{in}\ 1..|\mathcal{T}|; \mathsf{var}\ t_2\ \mathsf{in}\ 0..t_1; h_1\,;h_2]}$$

$$\mathit{split}(p\,;\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{q\}) = \mathit{split}(p) \cdot [\mathsf{var}\ x\ \mathsf{in}\ n..m; h \mid h \in \mathit{split}(q)]$$

Fig. 4: Syntax and semantics of symbolic traces, and splitting of protocols.

Theorem 2 (Compositionality). *Let $\mathcal{C}$ be a context, s.t. $\mathcal{C}[\mathsf{skip}\,;\mathsf{sync}]$ is DRF. For all $p \in \mathcal{A}$, if p is DRF, $\mathit{fv}(p) \subseteq \{\mathsf{tid}\}$, then $\mathcal{C}[p] \in \mathcal{A}$ and $\mathcal{C}[p]$ is also DRF.*

Compositionality allows Faial to analyze each fragment of an aligned protocol independently, by splitting the given protocol into multiple symbolic traces.

4.2 Splitting Protocols into Symbolic Traces

The second verification step, *splitting*, consists in transforming an aligned protocol into *symbolic traces*, *i.e.*, symbolic representations of sets of memory accesses which occur between two synchronizations.

Symbolic traces Intuitively, symbolic traces are a thin abstraction over an SMT formula. We describe how to translate a symbolic trace to a formula in Section 5.

We give the syntax and semantics of symbolic traces in Figure 4. Expression skip terminates a trace. Expression $n{:}o[m]$ states that thread n accesses index m with mode o. Expression $h_1\,;h_2$ composes two symbolic traces using operator $\otimes$, also given in Figure 4. Expression $\mathsf{var}\ x\ \mathsf{in}\ n..m; h$ binds variable x in h, where variable x is an integer in the range induced from n and m. The semantics of a symbolic trace yields a history with a phase for each possible variable assignment. Expression skip yields a single empty phase. Expression $n{:}o[m]$ evaluates to a singleton set that contains the access value that results from evaluating the thread-identifier expression n and the index expression m. Sequencing histories $h_1\,;h_1$ consists of performing the product of phases (operator $\otimes$), which consists of merging every phase of H_1 with every phase of H_2. A variable binder behaves like a skip when the range of values is empty. Otherwise, we fork two histories H_1 and H_2. We assign the lower bound of the set in H_1, and we recursively evaluate a variable binder where we increment its lower bound in H_2.

Barrier splitting is the transformation from aligned protocols to symbolic traces, performed via functions *trace* and *split*, defined in Figure 4. Function *trace* extracts the symbolic trace of an unsynchronized program for a single thread. Memory accesses are tagged with the owner thread tid, and unsynchronized loops are converted into variable bindings. Function *split* returns a list of symbolic traces. The case for $p\,;q$ is trivial (operator $\cdot$ stands for list concatenation). The base case of *split* is for unsynchronized protocol fragment u, which produces a list containing a single symbolic trace. It introduces fresh variables t_1 and t_2 that represent two (distinct) symbolic thread identifiers. The rest of the trace consists of the trace of u instantiated to the first thread identifier t_1 followed by its instantiation to the second thread identifier t_2. The case for synchronized loops simply reinterprets the loop as a variable binder. Function *split* leads to an exponential blow up wrt. nesting of synchronized loops, but this has not posed problems in practice, *c.f.*, Claim 2.

Example 1. Let $\hat{p} = \mathsf{wr}[\mathsf{tid} + 1]; \mathsf{rd}[\mathsf{tid} + 2]; \mathsf{sync}$. We have that $\mathit{split}(\hat{p})$ returns:

$$\mathsf{var}\ t_1\ \mathsf{in}\ 1..|\mathcal{T}|; \mathsf{var}\ t_2\ \mathsf{in}\ 0..t_1; t_1{:}\mathsf{wr}[t_1{+}1]; t_1{:}\mathsf{rd}[t_1{+}2]; t_2{:}\mathsf{wr}[t_2{+}1]; t_2{:}\mathsf{rd}[t_2{+}2]$$

We show that barrier splitting preserves and reflects DRF.

Theorem 3. *Let $p \in \mathcal{A}$, such that $p \downarrow H_1$, and $H_2 = [H \mid h \in \mathit{split}(p) \wedge h \Downarrow H]$, then $\mathit{safe}(H_1)$ if and only if $\mathit{safe}(H_2)$.*

Hence we have established that aligning (Theorem 1) and splitting (Theorem 3) preserve and reflect data-races, *i.e.*, any and all data-races are found. Thus, the only source of approximation in our analysis stems from the inference of protocols from CUDA kernels, which we discuss in the next section. Theorem 3 highlights the compositionality of our analysis, as each symbolic trace resulting from function *split* can be analyzed independently.

5 Implementation

In this section we present our tool, Faial, that implements the steps described in Figure 1. Faial takes a CUDA kernel as input and produces results that either identify the kernel as DRF or list specific data-races. In this section, we describe the implementation of the protocol inference, well-formedness checks, and transformation to SMT.

Inference This step transforms a CUDA kernel into access memory protocols (one for each shared array). It uses `libclang` [23] to parse the kernel, a standard single static assignment (SSA) transformation to simplify the analysis of indices and arrays, and code slicing to only retain code related to *shared* array accesses. We note that Faial supports constructs of the CUDA programming model that are not directly modeled by access memory protocols, *e.g.*, unstructured loops, conditionals, function calls, and multi-dimensional arrays. To support multi-dimensional thread identifiers, we extend the language of protocols to support multiple thread identifiers, and adapt function *split* accordingly. The main challenges are related to loops and function calls.

Whenever possible loops are transformed to loops with a stride of 1 following ideas from loop normalization [24] and abstraction [30]. For instance, in **for**(int i=lb;i<ub;i+=s){S} we change the stride from s into 1 by executing the loop body S when the loop variable i is divisible by stride, *i.e.*, the loop becomes **for**(int i=lb;i<ub;i++) **if**((i+lb)%s==0){S}. Similarly, a loop ranging over powers of n, *e.g.*, **for**(int i=lb;i<ub;i*=s), becomes **for**(int i=lb;i<ub;i++) **if**(powerof(i,s)){S}, where function powerof(i,s) tests whether i is a power of base s. We approximate **while**s as a structured loop with an unknown upper bound.

Function calls that manipulate shared memory are uncommon in GPU programming. Additionally auxiliary functions that manipulate shared memory have a compiler annotation to inline their bodies, hence we can inline such calls easily. Faial cannot handle recursive functions, but these rarely occur in practice. Function calls that do not access shared memory are simply discarded.

Well-formedness This step ensures that kernels Faial analyzes meet the well-formedness conditions, *i.e.*, $p \in \mathcal{W}$, including the assumptions that synchronized loops iterate at least once, see Definition 1. First, Faial annotates loops with a synchronized/unsynchronized tag according to the presence of sync in the loop body, then adjusts the precedence of sequencing to group all unsynchronized code preceding a sync or a synchronized loops. Synchronized loops of well-formed protocols cannot manipulate thread-local variables (*i.e.*, tid), an assumption shared by the CUDA programming model. Hence, Faial flags such kernels as erroneous. Next, Faial adds assertions before/after synchronized loops to check that the loop range is non-empty, *i.e.*, loops execute at least once. Similarly to loops, conditionals are tagged as synchronized or unsynchronized. Then, Faial inlines synchronized conditionals, *i.e.*, when a synchronized conditional is found, two copies of the input program are created and each copy is prefixed by a global assertion corresponding to the condition. Faial does not support synchronized conditionals that appear within synchronized loops. We have not found real-world kernels that include such a construction.

Quantification This step transforms each symbolic trace (Figure 4) into an SMT formula, to check for *safety*, *c.f.*, Figure 2. The presented formalism assumes a decidable fragment. However, as CUDA programs may include multiplication in index expressions, Faial uses an undecidable logic (SMTLib's QF_LIA). Essentially, the generated formula guarantees that the indices of array accesses are distinct when there is at least one write. We illustrate this straightforward transformation with Example 2.

Example 2. The formula generated from the trace in Example 1 is given below:

$$\forall t_1, t_2 \colon 1 \leq t_1 < |\mathcal{T}| \land 0 \leq t_2 < t_1 \land (\mathsf{m}_1 = \mathsf{wr} \lor \mathsf{m}_2 = \mathsf{wr}) \implies \\ \big((\mathsf{idx}_1 = t_1 + 1 \land \mathsf{m}_1 = \mathsf{wr}) \lor (\mathsf{idx}_1 = t_1 + 2 \land \mathsf{m}_1 = \mathsf{rd})\big) \\ \land \big((\mathsf{idx}_2 = t_2 + 1 \land \mathsf{m}_2 = \mathsf{wr}) \lor (\mathsf{idx}_2 = t_2 + 2 \land \mathsf{m}_2 = \mathsf{rd})\big) \land \mathsf{idx}_1 \neq \mathsf{idx}_2$$

where each symbolic access is translated to a conjunction representing its index (idx) and access mode (m). Observe that the formula enforces that indices idx_1 and idx_2 (executed by distinct threads) are different.

For multi-dimensional arrays, we generate one pair of indices per dimension, and check that at least one pair is distinct.

6 Experimental Evaluation

We evaluate Faial over several datasets and show how it fares against existing approaches. We structure this evaluation in three claims.

Claim 1: Correctness. We claim that our approach finds more bugs and raises fewer false alarms than existing tools. To evaluate this claim, we compare Faial against four state-of-the-art kernel verification tools over 10 kernels that are known to be tricky to analyze.
Claim 2: Scalability. We claim that our approach scales better to larger programs. To evaluate this claim, we compare Faial against other tools over a set of synthetic benchmarks designed to test the limits of each tool, in terms of run time and memory usage.
Claim 3: Real-world usability. We claim that our approach is more usable than existing static verification tools on real-world CUDA programs. To evaluate this claim, we use a varied dataset of real-world DRF kernels and measure the false alarm rate, run time, and memory usage of Faial, GPUVerify, and PUG.

Benchmarking environment To make our evaluation reproducible, we developed a benchmarking framework to automate our experiments over the different tools and datasets. For Claim 1 and Claim 3, we designed a tool-agnostic file format for kernel functions and associated metadata (*e.g.*, expected result of DRF analysis, grid and block dimensions, and include directives). And for Claim 2, we created a tool that generates kernels according to given templates, *e.g.*, see Figure 7.

We evaluate Faial against the following verification tools: GPUVerify [5] v2018-03-22; PUG [24] v0.2; and, GKLEE [26] and SESA [27] v3.0. Experiments for Claim 1 use an Intel i5-6500 CPU, 7.7GiB RAM, and Fedora 33 OS, while Claim 2 and Claim 3 use an Intel i7-10510U CPU, 16GiB RAM, and Pop! OS.

Table 1: Results for Claim 1. DRF indicates that a (static analysis) tool reported a test case as DRF. NRR indicates that a (symbolic execution) tool did not report any data-race. Label x/y indicates that the tool reported y data-races, x of which are actual races. Label *timeout* indicates that the tool did not terminate within 90s. A test passes if the tool returns the expected result and all reported races are valid.

Test		Expected	Faial	GPUVerify	PUG	GKLEE	SESA
1	transposeDiagonal	Racy	**1/1**	*0/2*	*DRF*	*timeout*	*timeout*
		DRF	**DRF**	*0/1*	**DRF**	*timeout*	*timeout*
2	first-iter	Racy	**1/1**	*0/1*	**1/1**	*timeout*	*timeout*
		DRF	**DRF**	*0/1*	*0/1*	*timeout*	*timeout*
3	last-iter	Racy	**1/1**	**1/1**	*0/1*	*timeout*	*timeout*
		DRF	**DRF**	*0/1*	**DRF**	*timeout*	*timeout*
4	last-iter-first-iter	Racy	**1/1**	*0/1*	*0/1*	*timeout*	*timeout*
		DRF	**DRF**	*0/1*	*0/1*	*timeout*	*timeout*
5	read-index	Racy	*0/1*	**1/1**	*0/1*	*NRR*	*NRR*
		DRF	*0/1*	**DRF**	*0/1*	**NRR**	**NRR**
Number of tests passed (of 5):			4	1	0	0	0

Excluded tools We excluded ESBMC-GPU [33] and Simulee [37] from the evaluation because we were unable to get them to run satisfactorily. Both tools have rudimentary support for verifying arbitrary CUDA kernels. ESBMC-GPU did not find a single data-race in our benchmarks, while Simulee produced false alarms for every DRF-kernel given.

Claim 1: Correctness

We have selected a set of tricky kernels to expose false alarms and missed data-races in Faial, GPUVerify, PUG, GKLEE, and SESA. Our results are reported in Table 1. The dataset consists of 5 tests, each consisting of two variations of the same kernel: one racy and one DRF. The racy version of Test 1 (*c.f.*, Listing 2.1) contains an inter-iteration data-races. The DRF version adds a sync after the second inner loop. Tests 2 to 4 expose various loop-related data-races. Their protocols are given in Figure 5. In the racy version of Test 2 $\mathsf{wr}[\mathsf{tid}+1]$ conflicts with $\mathsf{wr}[\mathsf{tid}]$ of the first iteration. Similarly, in the racy version of Test 3, $\mathsf{wr}[\mathsf{tid}+1]$ of the last iteration races with $\mathsf{wr}[\mathsf{tid}]$. In the racy version of Test 4 the last iteration of a nested loop races with the first iteration of the following loop. Test 5 exposes the abstraction gap between kernel and access memory protocols (which abstract away array elements), see Figure 6.

Faial passes more tests than any other tool. Failed Test 5 is caused by access memory protocols abstracting away from *what* data is being read from/written to arrays, *i.e.*, array elements. In each case, Faial reports one spurious data race (*0/1*). We report on performance trade-offs wrt. tracking array elements in Claim 2.

GPUVerify passes Test 5 because it tracks array elements, but fails the remaining 4 tests. Some reported false alarms are ill-formed, *e.g.*, on the racy component of Test 2, the report $(0 : \mathsf{wr}[\mathsf{tid}]; 16 : \mathsf{wr}[\mathsf{tid}])$ has disjoint indices.

```
// first-iter
wr[tid+1];
for^S x in 0..N {
  if (x > 0)
    { wr[tid] } ;
  sync}
```

```
// last-iter
for^S x in 0..N {
  sync;
  if (tid < |T|-1)
    { wr[tid+1] }};
wr[tid + |T|]
```

```
// last-iter-first-iter
for^S x in 1..N+1 {
  for^S y in 1..x+1 {
    sync; wr[tid+x+y]}};
for^S z in N*2..N*3 {
  wr[tid+z +1 ]; sync}
```

Fig. 5: Protocols for Tests 2 to 4, *c.f.*, Claim 1, where N is a free thread-global variable. Yellow shaded code only appears in the DRF version of first-iter and last-iter. Red shaded code only appears in the racy version of last-iter-first-iter.

```
// Racy kernel              // Protocol A
A[tid] = tid;               wr[tid];
int x = A[tid];             rd[tid];
A[x+1] = 0;                 wr[x+1]
```

```
// DRF kernel               // Protocol A
A[tid] = tid;               wr[tid];
int x = A[tid];             rd[tid];
A[x] = 0;                   wr[x]
```

Fig. 6: Kernels and protocols for Test 5 (read-index), *c.f.*, Claim 1; x becomes a free thread-local variable as protocols do not model array elements.

PUG obtains the worst score amongst static tools. Notably, the tool misses a data-race in Test 1, demonstrating its unsoundness, *c.f.*, Section 2.1.

GKLEE and SESA timeout for tests that include loops, as the loop bounds are unknown. Both tools miss the data-race in Test 5. Symbolic tools may be able to report data-races when the bound is known, *e.g.*, timeouts start in Test 1 when the bound is at least 2, in Test 2 when the bound is at least 23,000.

Claim 2: Scalability

We evaluate the scalability of our approach with a synthetic dataset that aims at demonstrating how different kernel constructs affect run time and memory usage of Faial, GKLEE, GPUVerify, PUG, and SESA. Our dataset is divided into five categories, one per syntactical construct in the language of access memory protocols, as well as conditionals, which are supported by our inference step, *c.f.*, Section 5. Figure 7 shows the protocols of the kernel patterns we generate in each category: (*i*) repeated accesses (read then write), (*ii*) repeated barrier synchronizations separated by writes, (*iii*) repeated conditionals, (*iv*) increasingly nested unsynchronized loops, and (*v*) increasingly nested synchronized loops. In each category, we vary the problem size by repeating a pattern from 1 to 50 times. Note that all kernels generated this way are DRF.

Figure 8 shows the average run time and memory usage over five runs on logarithmic and linear scales, respectively. For each run, we set a timeout of 90s and we exclude any run that times out or reports a false alarm. Cutoffs in the memory plots are determined by the cutoffs in the run time plots.

Overall Faial is the most scalable tool. In 4 out of 5 categories, Faial has the slowest growth for all experiments, and verifies all tests within 0.46s. In the

// accesses	// barriers	// conditionals	// unsynchronized loops	// synchronized loops

rd[tid + n_1*|T|];
wr[tid + 1*|T|];
rd[tid + n_2*|T|];
wr[tid + 2*|T|];
// ...
	wr[tid];
	sync;
	wr[tid];
	sync;
	// ...
		if tid==0
		{wr[tid]};
		if tid==1
		{wr[tid]};
		// ...
			for^U i_1 in 0..N {
			wr[tid];
			for^U i_2 in 0..N {
			wr[tid];
			// ... }}
				for^S i_1 in 0..N {
				wr[tid]; sync;
				for^S i_2 in 0..N {
				wr[tid]; sync;
				// ... }}

Fig. 7: Synthetic protocols generated for Claim 2. N is a free thread-global variable, and n_1, n_2... are positive integer literals.

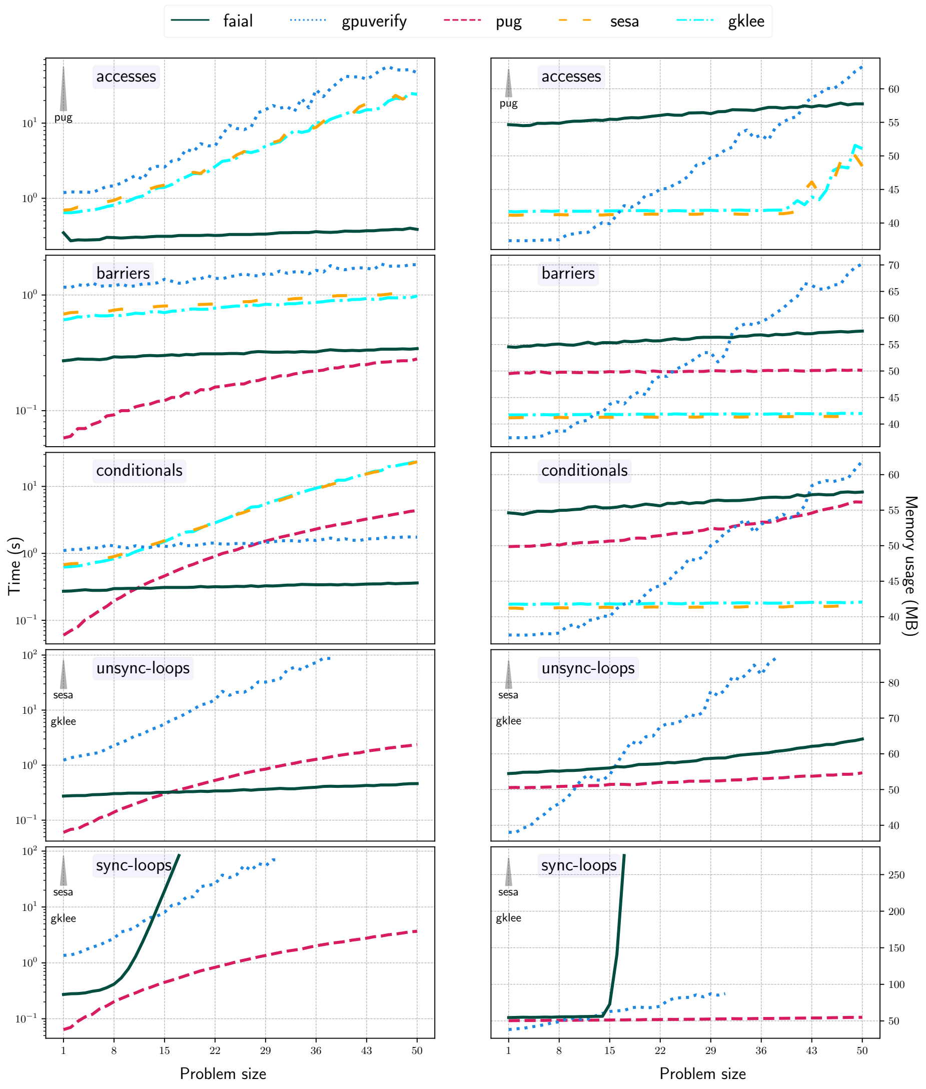


Fig. 8: Results for Claim 2. Run time (left plots) are given on a logarithmic scale, and memory (right plots) are given on a linear scale. Flatter and lower curve is better. Tools annotated with a triangle are excluded due to timeouts or errors.

largest problem sizes, our tool is the fastest in 3 categories (access, conditional, unsynchronized loop), 2nd for barriers, and 3rd for synchronized loops. Overall, the memory usage of Faial is competitive with other tools. Faial is the only tool with a near constant time/memory for up to 50 unsynchronized loops, indicating the scalability of reducing unsynchronized loops to universally quantified formulas. Faial only times out for kernels which consists of >17 nested synchronized loops. However such kernels are uncommon, *e.g.*, the levels of nested synchronized loops in the real-word kernels studied in Claim 3 are at most 3.

GPUVerify remains stable in the barrier and conditional categories but is affected negatively by loops and accesses. Loops are a known bottleneck in GPUVerify [2]. In the access category there is an exponential slowdown due to GPUVerify keeping track of what data is being written to/read from array.

PUG tool remains stable with the number of barrier synchronizations but is affected negatively by the number of conditionals and loops. PUG is the fastest tool with smaller inputs, but it raises false alarms in the access category, hence these measurements are omitted from the corresponding plots.

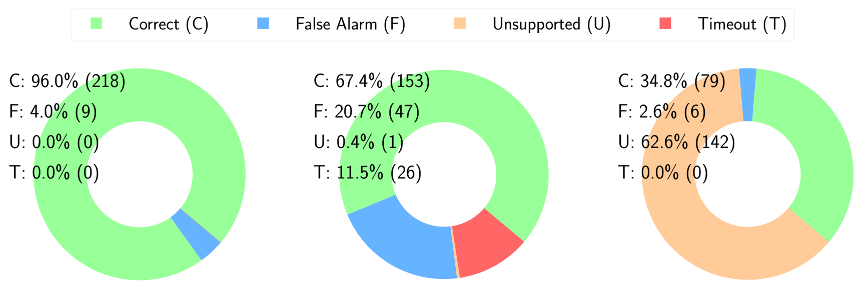
We discuss GKLEE and SESA together since SESA processes GKLEE's NVCC byte code output by concretizing variables, before passing it to GKLEE itself. There are two main factors that affect negatively these symbolic execution tools: (*i*) the number of loops, since they unroll each loop; and (*ii*) the amount of bookkeeping required to keep track of what is read from/written to memory. Figure 8 shows clear exponential curves for the access and barrier synchronization categories. Observe that these tools timeout immediately in the loop categories.

Claim 3: Real-World Usability

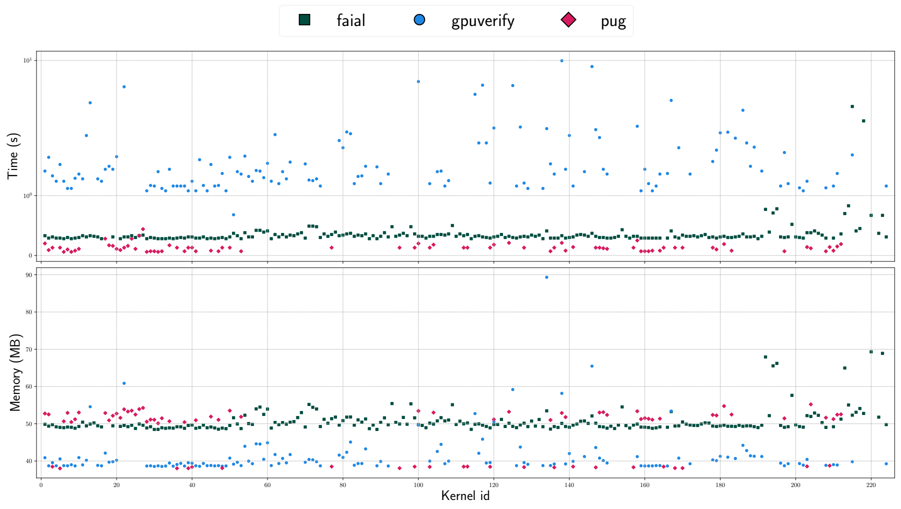
We evaluate the usability of our approach by comparing Faial with other static verification tools (GPUVerify and PUG) on real-world kernels wrt. rate of false alarm and run time. We curated a set of CUDA kernels from [2], which consists of 3 benchmark suites (totaling 227 CUDA kernels): NVIDIA GPU Computing SDK v2.0 (8 CUDA kernels); NVIDIA GPU Computing SDK v5.0 (166 CUDA kernels); Microsoft C++ AMP Sample Projects (20 kernels); gpgpu-sim benchmarks (33 kernels). All kernels are DRF and have been pre-processed by the authors of [2] to facilitate verification. Each kernel is in a distinct file, all dependencies are available, and kernels are annotated with minimal pre-conditions to allow for automatic analysis (*e.g.*, thread count is given).

As we aim to evaluate fully automatic verification of three tools, we removed code annotations (pre-conditions and loop invariants) specific to GPUVerify. Additionally, we made minor changes to some kernels to meet the limitations of the front-end of Faial and PUG. For instance we converted nested array lookups to use temporary variables and inlined functions calls that operate on arrays in 22 kernels. Another 8 kernels were modified to simplify their control flows. Our curated dataset will be included in our artifact submission.

Figures 9a, 9b, and 9c give the correctness results of Faial, GPUVerify, and PUG, respectively. Correct refers to the true-positive rate, *i.e.*, when the tool correctly identifies the kernel as DRF. False Alarm refers to the false alarm rate,



(a) Faial (b) GPUVerify (c) PUG



(d) Run time (top) and memory usage (bottom) of true-positives. Time (resp. memory) is cropped at 10s (resp. 100MB) and plotted on a logarithmic (resp. linear) scale.

Fig. 9: Results for Claim 3, on a set of 227 DRF CUDA kernels.

i.e., when the tool incorrectly identifies the kernel as racy. A kernel is Unsupported if it makes the tool crash. A Timeout occurs when the tool exceeds the limit of 60s to verify a kernel. The values shown are an average calculated over five runs. Figure 9d shows the average run time and memory usage of every true-positive report (we omit invalid reports) across the three tools.

Overall Faial has the highest rate of true-positives at 96%. Our tool is second in terms of run time and memory usage, showing a good compromise w.r.t. time and space. Faial verifies most kernels within 1s, and all kernels that need more time are only verified by Faial. GPUVerify shows lower memory usage at the cost of a higher verification run time. PUG verifies the lowest number of kernels (34.8%), as most kernels are unsupported (62.6%).

7 Related Work

SMT-based DRF analyses Li and Gopalakrishnan propose a direct encoding of DRF analysis of GPU programs in SMT, with PUG [24,25]. Both PUG and Faial follow a similar approach of barrier splitting: having a symbolic representation of a canonical interleaving, and dividing up the analysis over barrier intervals. The two major distinctions are that (1) PUG misses inter-thread data-races in synchronized loops, *e.g.*, Listing 2.1, and (2) the algorithms of PUG are unspecified and lack soundness proofs. In [24, §6.3] the authors identify the challenge of detecting inter-thread data-races, but do not elaborate a solution. Ma *et al.* [30] present a similar technique to detect data-races and deadlocks in OpenMP programs (CPU-based parallelism). However, their work does not guarantee DRF, and they do not formalize their algorithms. In [8], Prasanth *et al.* propose a polyhedral encoding of DRF for OpenMP programs, which is only applicable to programs with affine array accesses. However the prevalence of linearized array expressions in GPU kernels is known to stump polyhedral analysis [16].

Hoare-logic-based DRF analyses The main drawback of Hoare-logic based tools is their high rate of false alarms. They also require code annotations from a concurrency expert to handle loops. GPUVerify [2, 3, 5, 6, 12] can verify CUDA and OpenCL kernels using Boogie [4] as a backend. GPUVerify also relies on a two-thread abstraction (pen and paper proof) — in this paper, we present the first *machine-checked* proof of the two-thread abstraction idea. VeriCUDA [20,21] focuses on reasoning about the functional correctness of GPU programs using Hoare-logic. In [22] the authors extend VeriCUDA to proving DRF. In a similar vein, VerCors [7] uses separation logic to prove the functional correctness and DRF of GPU kernels. Both VeriCUDA and VerCors expect a tool-specific language, hence cannot handle real-world kernels directly.

Data-race finders include: dynamic data-race detection, symbolic-execution, and model-checking. Such techniques are better suited for highly detailed analysis in smaller kernels, and typically are unable to prove DRF. Dynamic data-race detection executes a kernel to find data-races on a fixed input, *e.g.*, [14, 18, 19, 28, 32, 38, 39]. This technique only reports real data-races, but suffers from a slowdown of at least 10× compared to the non-instrumented program, and requires the kernel input data, which might be unavailable or unknown. Symbolic execution and model checking have been extended to detect data-races [10, 11, 26, 33, 37]. These techniques do without the kernel input data and can detect more data-races than dynamic data-race detection.

Miscellaneous Ferrel *et al.* introduce a machine-checked formalism to reason about the semantics of CUDA assembly [15]. Dabrowski *et al.* mechanize the DRF-analysis of multithreaded programs [13]. Muller and Hoffmann present a logic to reason about the evaluation cost of CUDA kernels [31].

Other behavioral types have been used to verify parallel and multithreaded systems that communicate via message-passing [29,35,36]. However these do not capture shared memory (only message-passing), thus cannot address data-races.

8 Conclusion

We tackle the problem of statically checking DRF in GPU kernels, with a new family of behavioral types, *i.e.*, access memory protocols. We provide a novel compositional analysis of access memory protocols, along with fully mechanized proofs and an implementation. Our evaluation explores challenging and diverse benchmarks (229 real-world and 258 synthetic kernels) to demonstrate that our approach is more precise (false alarms and missed alarms), scalable (time/memory growth), and usable (real-world kernels correctly verified) than other tools.

Acknowledgements We thank Rumyana Neykova, Stephen Chang, and the anonymous reviewers for their insightful feedback on earlier versions of this work.

References

1. Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P.M., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: Behavioral types in programming languages. Foundations and Trends in Programming Languages **3**(2-3), 95–230 (2016). https://doi.org/10.1561/2500000031
2. Bardsley, E., Betts, A., Chong, N., Collingbourne, P., Deligiannis, P., Donaldson, A.F., Ketema, J., Liew, D., Qadeer, S.: Engineering a static verification tool for GPU kernels. In: Proceedings of CAV. vol. 8559, pp. 226–242. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_15
3. Bardsley, E., Donaldson, A.F., Wickerson, J.: KernelInterceptor: Automating GPU kernel verification by intercepting kernels and their parameters. In: Proceedings of IWOCL. pp. 1–5 (5 2014). https://doi.org/10.1145/2664666.2664673
4. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Proceedings of FMCO. p. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
5. Betts, A., Chong, N., Donaldson, A.F., Ketema, J., Qadeer, S., Thomson, P., Wickerson, J.: The design and implementation of a verification technique for GPU kernels. Transactions on Programming Languages and Systems **37**(3), 1–49 (2015). https://doi.org/10.1145/2743017
6. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: Proceedings of OOPSLA. pp. 113–132. ACM (2012). https://doi.org/10.1145/2384616.2384625
7. Blom, S., Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs. Science of Computer Programming **95**(P3), 376–388 (2014). https://doi.org/10.1016/j.scico.2014.03.013
8. Chatarasi, P., Shirako, J., Kong, M., Sarkar, V.: An extended polyhedral model for SPMD programs and its use in static data race detection. In: Proceedings of LCPC'16. pp. 106–120. Springer (2017). https://doi.org/10.1007/978-3-319-52709-3_10
9. Cogumbreiro, T., Lange, J., Liew Zhen Rong, D., Zicarelli, H.: Checking Data-Race Freedom of GPU Kernels, Compositionally (Artifact) (2021). https://doi.org/10.5281/zenodo.4726300

10. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic testing of OpenCL code. In: Proceedings of HVC. pp. 203–218. Springer (2012). https://doi.org/10.1007/978-3-642-34188-5_18
11. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic crosschecking of floating-point and SIMD code. In: Proceedings of EuroSys. pp. 315–328. ACM (2011). https://doi.org/10.1145/1966445.1966475
12. Collingbourne, P., Donaldson, A.F., Ketema, J., Qadeer, S.: Interleaving and lock-step semantics for analysis and verification of GPU kernels. In: Proceedings of ESOP. pp. 270–289. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_16
13. Dabrowski, F., Pichardie, D.: A certified data race analysis for a Java-like language. In: Proceedings of TPHOL, pp. 212–227. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_16
14. Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., Devietti, J.: BARRACUDA: Binary-level Analysis of Runtime RAces in CUDA programs. In: Proceedings of PLDI. pp. 126–140. ACM (2017). https://doi.org/10.1145/3062341.3062342
15. Ferrell, B., Duan, J., Hamlen, K.W.: CUDA au Coq: A framework for machine-validating GPU assembly programs. In: Proceedings of DATE. pp. 474–479 (2019). https://doi.org/10.23919/DATE.2019.8715160
16. Grosser, T., Ramanujam, J., Pouchet, L.N., Sadayappan, P., Pop, S.: Optimistic delinearization of parametrically sized arrays. In: Proceedings of ICS. pp. 351–360. ACM (2015). https://doi.org/10.1145/2751205.2751248
17. ul Hassan Khan Khan, A., Al-Mouhamed, M., Fatayer, A., Almousa, A., Baqais, A., Assayony, M.: Padding free bank conflict resolution for CUDA-based matrix transpose algorithm. In: Proceedings of SNPD. pp. 1–6 (2014). https://doi.org/10.1109/SNPD.2014.6888709
18. Holey, A., Mekkat, V., Zhai, A.: HAccRG: Hardware-accelerated data race detection in GPUs. In: Proceedings of ICPP. pp. 60–69 (2013). https://doi.org/10.1109/ICPP.2013.15
19. Kamath, A.K., George, A.A., Basu, A.: ScoRD: A scoped race detector for GPUs. In: Proceedings of ISCA. pp. 1036–1049. IEEE (2020). https://doi.org/10.1109/ISCA45697.2020.00088
20. Kojima, K., Igarashi, A.: A Hoare logic for SIMT programs. In: Proceedings of APLAS. vol. 8301, pp. 58–73. Springer (2013). https://doi.org/10.1007/978-3-319-03542-0_5
21. Kojima, K., Igarashi, A.: A Hoare logic for GPU kernels. Transactions on Computational Logic **18**(1), 1–43 (2017). https://doi.org/10.1145/3001834
22. Kojima, K., Imanishi, A., Igarashi, A.: Automated verification of functional correctness of race-free GPU programs. Journal of Automated Reasoning **60**(3), 279–298 (2018). https://doi.org/10.1007/s10817-017-9428-2
23. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of CGO. pp. 75–88. IEEE (2004). https://doi.org/10.1109/CGO.2004.1281665
24. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Proceedings of FSE. pp. 187–196. ACM (2010). https://doi.org/10.1145/1882291.1882320
25. Li, G., Gopalakrishnan, G.: Parameterized verification of GPU kernel programs. In: Proceedings of IPDPSW. pp. 2450–2459 (2012). https://doi.org/10.1109/IPDPSW.2012.302

26. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: Concolic verification and test generation for GPUs. In: Proceedings of PPoPP. vol. 47, pp. 215–224. ACM (2012). https://doi.org/10.1145/2370036.2145844
27. Li, P., Li, G., Gopalakrishnan, G.: Practical symbolic race checking of GPU programs. In: Proceedings of SC. pp. 179–190. IEEE (2014). https://doi.org/10.1109/SC.2014.20
28. Li, P., Hu, X., Chen, D., Brock, J., Luo, H., Zhang, E.Z., Ding, C.: LD: Low-overhead GPU race detection without access monitoring. Transactions on Architecture and Code Optimization **14**(1), 1–25 (2017). https://doi.org/10.1145/3046678
29. López, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconcelos, V.T., Yoshida, N.: Protocol-based verification of message-passing parallel programs. In: Proceedings of OOPSLA. pp. 280–298. ACM (2015). https://doi.org/10.1145/2814270.2814302
30. Ma, H., Diersen, S.R., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic analysis of concurrency errors in OpenMP programs. In: Proceedings of ICPP. pp. 510–516. IEEE (2013). https://doi.org/10.1109/ICPP.2013.63
31. Muller, S.K., Hoffmann, J.: Modeling and analyzing evaluation cost of CUDA kernels. Proceedings of the ACM on Programming Languages **5**(POPL) (2021). https://doi.org/10.1145/3434306
32. Peng, Y., Grover, V., Devietti, J.: CURD: A dynamic CUDA race detector. In: Proceedings of PLDI. pp. 390–403. ACM (2018). https://doi.org/10.1145/3192366.3192368
33. Pereira, P., Albuquerque, H., Marques, H., Silva, I., Carvalho, C., Cordeiro, L., Santos, V., Ferreira, R.: Verifying CUDA programs using SMT-based context-bounded model checking. In: Proceedings of SAC. pp. 1648–1653. ACM (2016). https://doi.org/10.1145/2851613.2851830
34. Ruetsch, G., Micikevicius, P.: Optimizing matrix transpose in CUDA. NVIDIA CUDA SDK Application Note **18** (2009), https://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf
35. Vasconcelos, V.T.: Session types for linear multithreaded functional programming. In: Proceedings of PPDP. pp. 1–6. ACM (2009). https://doi.org/10.1145/1599410.1599411
36. Vasconcelos, V.T., Ravara, A., Gay, S.: Session types for functional multithreading. In: Proceedings of CONCUR. pp. 497–511. Springer (2004). https://doi.org/10.1007/978-3-540-28644-8_32
37. Wu, M., Ouyang, Y., Zhou, H., Zhang, L., Liu, C., Zhang, Y.: Simulee: Detecting CUDA synchronization bugs via memory-access modeling. In: Proceedings of ICSE. pp. 937–948. ACM (2020). https://doi.org/10.1145/3377811.3380358
38. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GRace: A low-overhead mechanism for detecting data races in GPU programs. In: Proceedings of PPoPP. pp. 135–146. ACM (2011). https://doi.org/10.1145/1941553.1941574
39. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GMRace: Detecting data races in GPU programs via a low-overhead scheme. Transactions on Parallel and Distributed Systems **25**(1), 104–115 (2014). https://doi.org/10.1109/TPDS.2013.44