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
[2] Royal Holloway, University of London

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 (anonymized name), a tool that outperforms the state-of-the-art. Notably, it can correctly verify at least 1.41× more real-world kernels, and it exhibits a linear growth in 4 out of 5 experiments, while others grow exponentially in all 5 experiments.

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 [25, 26, 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, 21] 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

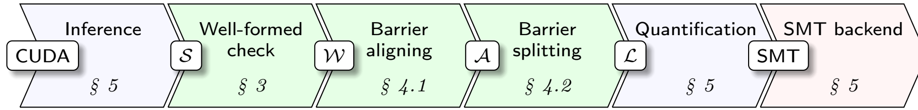

Fig. 1: Work-flow of the verification.

cope with larger programs and do not require array size information either miss data-races [23] or overwhelm the user with false alarms [38].

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, 23, 25, 26], the largest comparative study of GPU verification (5 tools in 260 kernels, 3 tools compared in 487 kernels). Faial verified 217 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.

Our paper is accompanied by an implementation (Faial, see § A), an evaluation framework (inc. datasets), and proof scripts (in Coq) for each theorem. Should the paper be accepted, these will be submitted for artifact evaluation.

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.

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

```
1 for (int r = 0; r < N; r++) {
2   for (int i = 0; i<M; i++)
3     { tile [ tid ] = ...;}
4   __syncthreads();
5   for (int j = 0; j<M; j++)
6     {... = tile [ tid+j];}}
```

76 ## 2 Overview

77 This section gives an overview of our approach by examining a data-race we
78 found in published work [16] and [34]. We discuss the challenges that such ex-
79 amples pose to the state-of-the-art of DRF analysis. Then we introduce a veri-
80 fication framework based on *access memory protocols*: behavioral types [1] that
81 codify the way threads interact via shared memory. Figure 1 gives an overview
82 of the verification pipeline. We start from CUDA kernels, from which we infer
83 access memory protocols. Protocols are then checked for well-formedness and
84 transformed in three steps into formulas that are verified by an SMT solver.

85 ### 2.1 Challenges of GPU Programming

86 **GPU programming model** The key component of GPU programming is the
87 *kernel* program, or just kernel, that runs according to the Single-Instruction-
88 Multiple-Thread (SIMT) execution model, where multiple threads run a single
89 instruction concurrently. A kernel is parameterized by a special variable that
90 holds a thread identifier, henceforth named tid. In parallel, each member of a
91 group of threads runs an instantiated copy of the kernel by supplying its identifier
92 as an argument. Threads communicate via shared memory (arrays) and mediate
93 communication via barrier synchronization (an execution point where all threads
94 must wait for each other before advancing further). Writes are only visible to
95 other threads after a barrier synchronization, *i.e.*, there is no guarantee that a
96 write of a thread can be read by another before a barrier synchronization.
97 GPU programming platforms usually group threads hierarchically in multi-
98 ple levels, across which no inter-groups synchronization is possible. In both the
99 literature [6, 23] and this work, the focus is on intra-group communication as
100 inter-group errors can be seen as a special case of intra-group errors.

101 **Challenges** We motivate the difficulty of analyzing data-races by studying a
102 programming error found in the wild, reported in Listing 2.1 (left). This excerpt
103 comes from a tutorial [34] on optimizing numeric algorithms for GPUs. The code
104 listing transposes a matrix N-times with an outer loop indexed by variable r.
105 Remarkably, the tutorial [34] does not inform the readers that Listing 2.1
106 contains a subtle *data-race*: one transpose-operation starts (the writes to tile
107 in line 3) without awaiting the termination of the previous transpose-operation

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

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

108 (the reads from tile in line 6), thus corrupting the data over time and possibly
109 skewing the timing of the optimization to appear faster than it should be. We
110 found a related data-race in [16], which reuses code from [34].

111 Our tool, Faial, successfully identifies the program state that triggers the
112 data-race in Listing 2.1: when r =1 and N =2. However, state-of-the-art tools
113 struggle to accurately analyze Listing 2.1, as evaluated in Section 6 (Claim 1:
114 Test 1). Symbolic execution tools, such as [25, 26], timeout for $N > 1$, and, in
115 general, cannot handle symbolic (unknown) bounds. GPUVerify [6], a tool based
116 on Hoare logic, reports a false alarm: a spurious data-race when r=0 and N=1.
117 And PUG [23] incorrectly identifies the example as DRF, as its analysis appears
118 to ignore data-races originating from different iterations of a loop.

119 2.2 Memory Access Protocols by Example

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

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

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

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

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

145 When $M > 1$, thread 0 (l.h.s) accesses $\mathsf{rd}[1]$ and thread 1 (r.h.s) accesses $\mathsf{wr}[1]$.
146 Thus, there is a data-race on index 1 of the array.
147 A fundamental challenge of static DRF verification is how to handle loops.
148 Symbolic execution approaches that unroll loops, *e.g.*, [25, 26], cannot handle
149 large nor symbolic iteration spaces. Static approaches that use Hoare logic,
150 *e.g.*, [5, 7, 21], require user-provided loop invariants. Another approach is to re-
151 duce loops to verifying the satisfiability of a corresponding universally quantified
152 formula, *e.g.*, [24, 30]. This has the advantage of being fast and not requiring in-
153 variants. However, its previous application to GPU programming, *i.e.*, PUG,
154 is unsound due to the interaction between barrier synchronizations and loops,
155 *e.g.*, PUG misses the data-race in Listing 2.1. We give more details in Section 6.

156 *Our Approach* A key contribution of our work is to identify conditions that allow
157 a kernel to be reduced to a first-order logic formula, by precisely characterizing
158 the effect of barrier synchronization in loops. To this end, the language of access
159 memory protocols distinguishes syntactically between protocol fragments that
160 synchronize from those that do not. For instance, the protocol in Listing 2.2 is
161 identified as *unsynchronized*, as it does not include any synchronization.
162 In Section 4, we show that the DRF analysis of unsynchronized protocols can
163 be precisely reduced to a first-order logic formula, where universally quantified
164 formulae represent loops, thus obviating the need to unroll them explicitly. For
165 instance, we reduce the verification of Listing 2.2 to asking whether for all M,
166 t_1, and t_2, where $t_1 \neq t_2$ are thread identifiers, the following holds:

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

167 This formula is *unprovable* since $\mathsf{rd}[t_1 + j_1]$ races with $\mathsf{wr}[t_2]$ when, e.g., $t_1 = 0$,
168 $t_2 = 1$, $j_1 = 1$, and $M > 1$. Hence, our technique flags Listing 2.2 as racy.

169 **Analysis of synchronized protocols** The protocol in Listing 2.3 (left) models
170 *all* the interactions over the shared array tile from Listing 2.1. This protocol
171 consists of one outer loop r that contains two inner loops separated by a barrier
172 synchronization (sync). The first inner loop writes ($\mathsf{wr}[\mathsf{tid}]$) to the array, while
173 the second reads ($\mathsf{rd}[\mathsf{tid} + \mathsf{j}]$) from the array.
174 This protocol illustrates how our language syntactically differentiates be-
175 tween protocols fragments that synchronize from those that do not. Concretely,
176 our language precludes an unsynchronized loop ($\mathsf{for}^{\mathsf{U}}\ x \in n..m\ \{u\}$) from calling
177 sync anywhere in u, and it requires that a synchronized loop ($\mathsf{for}^{\mathsf{S}}\ x \in n..m\ \{p\}$)

Listing 2.3: Access memory protocol (left) of array tile from Listing 2.1 and its aligned version (right).

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

aligns to

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

178 includes at least one occurrence of sync. The superscript U (resp. S) stands for
179 *s*ynchronized (resp. *u*nsynchronized). This distinction can be inferred automat-
180 ically and yields a compositional analysis, as we explain below.

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

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

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

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

204 The first two protocols are handled like Listing 2.2 because they are unsynchro-
205 nized. Representing a synchronized loop as a formula becomes possible when
206 the protocol is *aligned*: both threads must share the same value for r at each
207 iteration. Hence, we reduce the verification to asking whether for all N, M, t_1,

208 and t_2 where $t_1 \neq t_2$ and the following holds:

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

209 Our technique identifies Listing 2.3 as racy since this formula is *unprovable*, *i.e.*,
210 $\mathsf{rd}[t_1{+}j_1]$ races with $\mathsf{wr}[t_2]$ when $r = 1$, $t_1 = 0$, $t_2 = 1$, $j_1 = 1$, $N > 1$ and $M > 1$.

211 3 Access Memory Protocols

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

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

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

232 Evaluation of an unsynchronized protocol u by a thread identifier i, written
233 $u \downarrow_i P$, yields a *phase*, *i.e.*, a set $P \in \mathcal{P}$ of *access values* $\alpha \in \mathbb{A}$. Each access
234 value, or just access, notation $i{:}o[j]$, consists of its issuing thread identifier i,
235 an access mode o (read/write), and an index j. Protocol skip produces no ac-
236 cesses. A memory access $o[n]$ evaluates the index and creates a singleton phase.
237 Sequencing, branching, and looping are standard. Similarly to SVD, Rule $\mathcal{U}$-par
238 executes a copy of the unsynchronized code u for each thread $i \in \mathcal{T}$ by replacing
239 the special variable tid by the thread identifier, $u[\mathsf{tid} := i]$, which results in the
240 union of the accesses of all threads.

241 *Synchronized fragment* A protocol $p \in \mathcal{S}$ may perform barrier synchroniza-
242 tion sync, run unsynchronized code u, perform sequential composition, and loop.
243 Figure 2 gives the semantics of a protocol, notation $p \downarrow H$. Evaluation of a pro-
244 tocol p yields a *history* (ranged over by H), *i.e.*, a list of phases (P) that records
245 how memory was accessed. We use $::$ as list constructor and $\cdot$ for the usual list
246 concatenation operator. Histories are concatenated using the special $\odot$-operator.

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 $\mathcal{U}$ $\boxed{u \downarrow_i P}$ $\boxed{u \downarrow_{\mathcal{T}} S}$

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

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

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

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

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

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

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

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

Well-formed protocols $\boxed{p \in \mathcal{W}}$

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

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

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

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

247 A barrier synchronization creates two empty phases, corresponding to phases
248 before and after synchronization. Running an unsynchronized protocol yields a
249 single phase containing all accesses performed by that protocol. Sequencing two
250 synchronized protocols p with q merges the last phase of the former with the first
251 phase of the latter, as these two phases run concurrently. Running one iteration
252 of a synchronized loop sequences the execution of the first iteration with the
253 rest of the loop, by merging the last phase of the first iteration with the first
254 phase of the rest of the loop. Synchronized loops in access memory protocols are
255 nonempty, hence the base case is when there is one iteration left. This additional
256 requirement helps with the presentation of our theory as it implies that every
257 synchronized loop always executes at least one synchronization.

258 A protocol is well-formed, written $p \in \mathcal{W}$, if every unsynchronized fragment is
259 followed by a barrier synchronization, every synchronized loop includes a barrier
260 and is not branching on thread-local variables, *i.e.*, `tid`. We write $fv(p)$ (resp.
261 $fv(n)$) for the free variables of p (resp. n). We discuss how well-formedness is
262 enforced in Section 5.

263 DRF is formalized at the bottom of Figure 2. Two accesses are in a data-race
264 (or racy) when there exist two different threads that access the same index k,
265 and one of these accesses is a write. Phase P is *safe* iff each pair of access it
266 contains is not racy. History P is *safe* when all of its phases are safe.

267 4 DRF-Preserving Transformations of Protocols

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

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

274 4.1 Aligning Protocols

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

278 *Barrier aligning* (or just aligning) is performed by function *align*, given in
279 the bottom half of Figure 3. The function returns a pair whose first element is an
280 aligned and synchronized protocol, and whose second element is an unsynchro-
281 nized protocol. Intuitively, the pair represents a sequence which we delimitate
282 syntactically. We note that the output of *align*, say (q, u), can be trivially made
283 into an aligned protocol: $q\,;u\,;$`sync`. The case for synchronization is simple, *align*
284 returns the input protocol as the first component of the pair and `skip` as the
285 second component (the input protocol is already fully aligned). The case for
286 sequence consists of the sequential composition of the pair aligned with unsyn-
287 chronized code using operator $(\fatsemi)$. Sequencing two pairs $(p, u) \fatsemi (q, u')$ amounts
288 to sequencing u to the outer-most piece of unsynchronized code present in q.

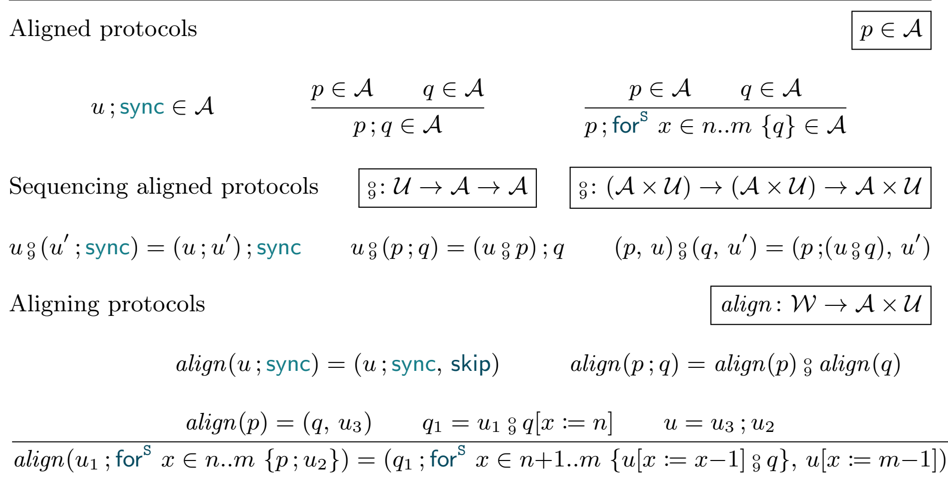

Fig. 3: Aligning protocols.

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 ($u[x \coloneqq m{-}1]$). See Listing 2.3 for an example of protocol aligning. We note that we can always unroll the loop because the analysis only considers nonempty 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). *Let $\mathit{align}(p) = (q, u)$ and $p \in \mathcal{W}$. If $p \downarrow H_1$ and $q\,; u \downarrow H_2$, then $\mathit{safe}(H_1)$ if and only if $\mathit{safe}(H_2)$.*

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.

Theorem 2 (Compositionality). *Let $\mathcal{C}$ be a context, s.t. $\mathcal{C}[\mathsf{skip}\,;\mathsf{sync}]$ is DRF, and $\mathcal{C}[\mathsf{skip}\,;\mathsf{sync}] \downarrow H$. For all $p \in \mathcal{A}$, if p is DRF, $p \downarrow H'$, and $\mathit{fv}(p) \subseteq \{\mathsf{tid}\}$, then $\mathcal{C}[p] \in \mathcal{A}$ and $\mathcal{C}[p]$ is also DRF.*

Syntax

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

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 $\boxed{h \Downarrow H}$

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

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

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

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

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

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

$$split(p\,; q) = split(p) \cdot split(q)$$

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

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

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

308
4.2 Splitting Protocols into Symbolic Traces

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

312 *Symbolic traces* Intuitively, symbolic traces are a thin abstraction over an SMT
313 formula. We describe how to translate a symbolic trace to a formula in Section 5.
314 We give the syntax and semantics of symbolic traces in Figure 4. Expres-
315 sion skip terminates a trace. Expression $n{:}o[m]$ states that thread n accesses
316 index m with mode o. Expression $h_1\,; h_2$ composes two symbolic traces using
317 operator $\otimes$, also given in Figure 4. Expression $\mathsf{var}\ x\ \mathsf{in}\ n..m; h$ binds variable x
318 in h, where variable x is an integer in the range induced from n and m. The
319 semantics of a symbolic trace yields a history with a phase for each possible vari-

320 able assignment. Expression skip yields a single empty phase. Expression $n{:}o[m]$
321 evaluates to a singleton set that contains the access value that results from eval-
322 uating the thread-identifier expression n and the index expression m. Sequencing
323 histories $h_1\,;h_1$ consists of performing the product of phases (operator $\otimes$), which
324 consists of merging every phase of H_1 with every phase of H_2. A variable binder
325 behaves like a skip when the range of values is empty. Otherwise, we fork two his-
326 tories H_1 and H_2. We assign the lower bound of the set in H_1, and we recursively
327 evaluate a variable binder where we increment its lower bound in H_2.

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

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

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

343 We show that barrier splitting preserves and reflects DRF.

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

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

352 5 Implementation

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

Inference This step transforms a CUDA kernel into access memory protocols (one for each shared array). It uses `libclang` [22] 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 [23] 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 ($p \in \mathcal{W}$) defined in Figure 2, as well as the assumptions that synchronized loops iterate at least once. 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 nonempty, *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. 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.

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

$$\forall t_1, t_2 \colon 1 \leq t_1 < 3 \land 0 \leq t_2 < t_1 \land (\mathsf{m}_1 = \mathsf{wr} \lor \mathsf{m}_2 = \mathsf{wr}) \implies \big((\mathsf{idx}_1 = t_1 + 1 \land \mathsf{m}_1 = \mathsf{wr}) \lor (\mathsf{idx}_1 = t_1 + 2 \land \mathsf{m}_1 = \mathsf{rd})\big) \land \big((\mathsf{idx}_2 = t_2 + 1 \land \mathsf{m}_2 = \mathsf{wr}) \lor (\mathsf{idx}_2 = t_2 + 2 \land \mathsf{m}_2 = \mathsf{rd})\big) \land \ \mathsf{idx}_1 \neq \mathsf{idx}_2$$

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

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

407 6 Experimental Evaluation

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

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

422 *Benchmarking environment* To make our evaluation reproducible, we developed
423 a benchmarking framework to automate our experiments over the different tools
424 and datasets. For Claim 1 and Claim 3, we designed a tool-agnostic file format for
425 kernel functions and associated metadata (*e.g.*, expected result of DRF analysis,
426 grid and block dimensions, and include directives). And for Claim 2, we created
427 a tool that generates kernels according to given templates, *e.g.*, see Figure 7.
428 We evaluate Faial against the following verification tools: GPUVerify [5] v2018-
429 03-22; PUG [23] v0.2; and, GKLEE [25] and SESA [26] v3.0. Experiments for
430 Claim 1 use an Intel i5-6500 CPU, 7.7GiB RAM, and Fedora 33 OS, while
431 Claim 2 and Claim 3 use an Intel i7-10510U CPU, 16GiB RAM, and Pop! OS.
432 *Excluded tools* We excluded ESBMC-GPU [33] and Simulee [38] from the evalu-
433 ation because we were unable to get them to run satisfactorily. Both tools have
434 rudimentary support for verifying arbitrary CUDA kernels. ESBMC-GPU did not
435 find a single data-race in our benchmarks, while Simulee produced false alarms
436 for every DRF-kernel given.

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 x data-races, y 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

437 **Claim 1: Correctness**

438 We have selected a set of tricky kernels to expose false alarms and missed data-
439 races in Faial, GPUVerify, PUG, GKLEE, and SESA. Our results are reported
440 in Table 1. The dataset consists of 5 tests, each consisting of two variations
441 of the same kernel: one racy and one DRF. The racy version of Test 1 (*c.f.*,
442 Listing 2.1) contains an inter-iteration data-races. The DRF version adds a sync
443 after the second inner loop. Tests 2 to 4 expose various loop-related data-races.
444 Their protocols are given in Figure 5. In the racy version of Test 2 $\mathsf{wr}[\mathsf{tid}+1]$
445 conflicts with $\mathsf{wr}[\mathsf{tid}]$ of the first iteration. Similarly, in the racy version of Test 3,
446 $\mathsf{wr}[\mathsf{tid}+1]$ of the last iteration races with $\mathsf{wr}[\mathsf{tid}]$. In the racy version of Test 4 the
447 last iteration of a nested loop races with the first iteration of the following loop.
448 Test 5 exposes the abstraction gap between kernel and access memory protocols
449 (which abstract away array elements), see Figure 6.

450 Faial passes more tests than any other tool. Failed Test 5 (two false alarms)
451 is caused by access memory protocols abstracting away from *what* data is being
452 read from/written to arrays, *i.e.*, array elements. We report on performance
453 trade-offs wrt. tracking array elements in Claim 2.

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

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

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

```
// 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.

463 **Claim 2: Scalability**

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

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

479 Overall Faial is the most scalable tool. In 4 out of 5 categories, Faial has
480 the slowest growth for all experiments, and verifies all tests within 0.46s. In the
481 largest problem sizes, our tool is the fastest in 3 categories (access, conditional,
482 unsynchronized loop), 2$^{\text{nd}}$ for barriers, and 3$^{\text{rd}}$ for synchronized loops. Overall,
483 the memory usage of Faial is competitive with other tools. Faial is the only tool
484 with a near constant time/memory for up to 50 unsynchronized loops, indicating
485 the scalability of reducing unsynchronized loops to universally quantified formu-
486 las. Faial only times out for kernels which consists of >17 nested synchronized
487 loops. However such kernels are uncommon, *e.g.*, the levels of nested synchro-
488 nized loops in the real-word kernels studied in Claim 3 are at most 3.

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

rd[tid + n₁*|T|];
wr[tid + 1*|T|];
rd[tid + n₂*|T|];
wr[tid + 2*|T|];
// ...
	wr[tid];
	sync;
	wr[tid];
	sync;
	// ...
		if tid==0
		{wr[tid]};
		if tid==1
		{wr[tid]};
		// ...
			for^U i₁ in 0..N {
			wr[tid];
			for^U i₂ in 0..N {
			wr[tid];
			// ... }}
				for^S i₁ in 0..N {
				wr[tid]; sync;
				for^S i₂ 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.

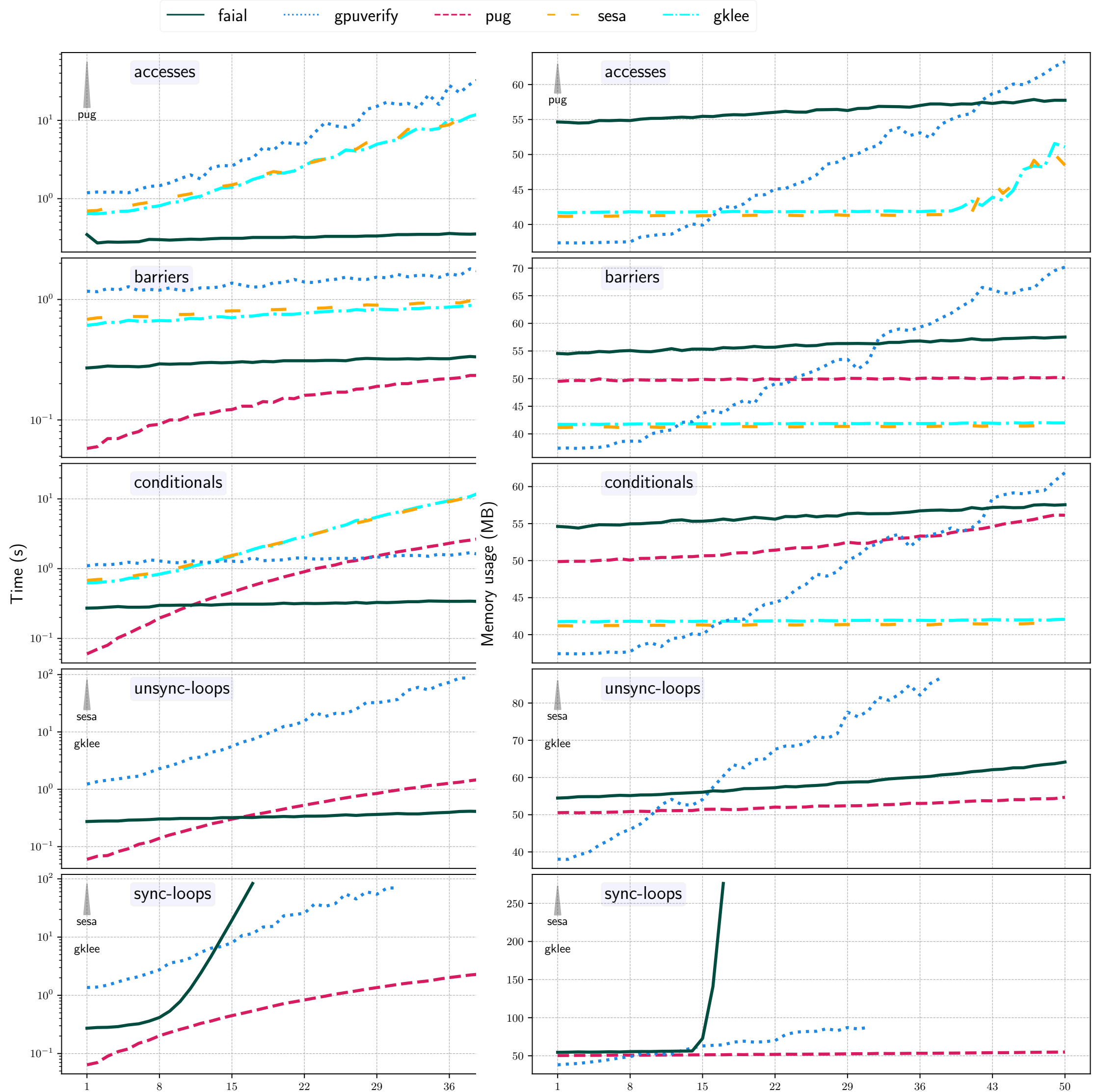

Fig. 8: Results for Claim 2. Run time (left plots) are given on a logarithmic scale, and memory (right plots) are given on a linear scale. Flatter and lower curve is better. Tools annotated with a triangle are excluded due to timeouts or errors.

489 GPUVerify remains stable in the barrier and conditional categories but is af-
490 fected negatively by loops and accesses. Loops are a known bottleneck in GPUVer-
491 ify [2]. In the access category there is an exponential slowdown due to GPUVerify
492 keeping track of what data is being written to/read from array.
493 PUG tool remains stable with the number of barrier synchronizations but is
494 affected negatively by the number of conditionals and loops. PUG is the fastest
495 tool with smaller inputs, but it raises false alarms in the access category, hence
496 these measurements are omitted from the corresponding plots.
497 We discuss GKLEE and SESA together since SESA processes GKLEE's NVCC
498 byte code output by concretizing variables, before passing it to GKLEE itself.
499 There are two main factors that affect negatively these symbolic execution tools:
500 (*i*) the number of loops, since they unroll each loop; and (*ii*) the amount of book-
501 keeping required to keep track of what is read from/written to memory. Figure 8
502 shows clear exponential curves for the access and barrier synchronization cate-
503 gories. Observe that these tools timeout immediately in the loop categories.

504 **Claim 3: Real-World Usability**

505 TCnoteR1: what is the unsupported kernel by our tool? We evaluate the usability
506 of our approach by comparing Faial with other static verification tools (GPUVerify
507 and PUG) on real-world kernels wrt. rate of false alarm and run time. We curated
508 a set of CUDA kernels from [2], which consists of 3 benchmark suites (totaling
509 227 CUDA kernels): NVIDIA GPU Computing SDK v2.0 (8 CUDA kernels);
510 NVIDIA GPU Computing SDK v5.0 (166 CUDA kernels); Microsoft C++ AMP
511 Sample Projects (20 kernels); gpgpu-sim benchmarks (33 kernels). All kernels are
512 DRF and have been pre-processed by the authors of [2] to facilitate verification.
513 Each kernel is in a distinct file, all dependencies are available, and kernels are
514 annotated with minimal pre-conditions to allow for automatic analysis (*e.g.*,
515 thread count is given).
516 As we aim to evaluate fully automatic verification of three tools, we removed
517 code annotations (pre-conditions and loop invariants) specific to GPUVerify. Ad-
518 ditionally, we made minor changes to some kernels to meet the limitations of
519 the front-end of Faial and PUG. For instance we converted nested array lookups
520 to use temporary variables and inlined functions calls that operate on arrays in
521 22 kernels. Another 8 kernels were modified to simplify their control flows. Our
522 curated dataset will be included in our artifact submission.
523 Figures 9a, 9b, and 9c give the correctness results of Faial, GPUVerify, and
524 PUG, respectively. Correct refers to the true-positive rate, *i.e.*, when the tool
525 correctly identifies the kernel as DRF. False Alarm refers to the false alarm rate,
526 *i.e.*, when the tool incorrectly identifies the kernel as racy. A kernel is Unsupported
527 if it makes the tool crash. A Timeout occurs when the tool exceeds the limit of
528 60s to verify a kernel. The values shown are an average calculated over five runs.
529 Figure 9d shows the average run time and memory usage of every true-positive
530 report (we omit invalid reports) across the three tools.
531 Overall Faial has the highest rate of true-positives at 96%. Our tool is second
532 in terms of run time and memory usage, showing a good compromise w.r.t. time

(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.

533 and space. Faial verifies most kernels within 1s, and all kernels that need more
534 time are only verified by Faial. GPUVerify shows lower memory usage at the
535 cost of a higher verification run time. PUG verifies the lowest number of kernels
536 (34.8%), as most kernels are unsupported (62.6%).

537 7 Related Work

538 *SMT-based DRF analyses* Li and Gopalakrishnan propose a direct encoding of
539 DRF analysis of GPU programs in SMT, with PUG [23,24]. Both PUG and Faial
540 follow a similar approach of barrier splitting: having a symbolic representation
541 of a canonical interleaving, and dividing up the analysis over barrier intervals.
542 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 [23, §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 [15].

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, 11] 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 [19,20] focuses on reasoning about the functional correctness of GPU programs using Hoare-logic. In [21] 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.*, [13, 17, 18, 27, 28, 32, 39, 40]. This technique only reports real data-races, but suffers from a slowdown of at least 10× and requires the kernel input data, which might be unavailable or unknown. Symbolic execution and model checking have been extended to detect data-races [9, 10, 25, 33, 38]. These techniques do without the kernel input 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 [14]. Dabrowski *et al.* mechanize the DRF-analysis of multithreaded programs [12]. Muller and Hoffmann present a logic to reason about the evaluation cost of CUDA kernels [31].

Session types [35] are a family of behavioral types that codify the message exchanges that take place over a given *session*, in terms of sends and receives. Access memory protocols are similar in that they codify the interactions that take place over a given shared *array*, in terms of reads and writes. Other behavioral types have been used to verify parallel and multithreaded systems that communicate via message-passing [29, 36, 37]. 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

586 compositional analysis of access memory protocols, along with fully mechanized
587 proofs and an implementation. Our evaluation explores challenging and diverse
588 benchmarks (229 real-world and 258 synthetic kernels) to demonstrate that our
589 approach is more precise (false alarms and missed alarms), scalable (time/mem-
590 ory growth), and usable (real-world kernels correctly verified) than other tools.

591 References

592 1. Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P.M.,
593 Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi,
594 V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida,
595 N.: Behavioral types in programming languages. Foundations and Trends in Pro-
596 gramming Languages **3**(2-3), 95–230 (2016). https://doi.org/10.1561/2500000031
597 2. Bardsley, E., Betts, A., Chong, N., Collingbourne, P., Deligiannis, P., Donaldson,
598 A.F., Ketema, J., Liew, D., Qadeer, S.: Engineering a static verification tool for
599 GPU kernels. In: Proceedings of CAV. vol. 8559, pp. 226–242. Springer (2014).
600 https://doi.org/10.1007/978-3-319-08867-9_15
601 3. Bardsley, E., Donaldson, A.F., Wickerson, J.: KernelInterceptor: Automating GPU
602 kernel verification by intercepting kernels and their parameters. In: Proceedings of
603 IWOCL. pp. 1–5 (5 2014). https://doi.org/10.1145/2664666.2664673
604 4. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A
605 modular reusable verifier for object-oriented programs. In: Proceedings of FMCO.
606 p. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
607 5. Betts, A., Chong, N., Donaldson, A.F., Ketema, J., Qadeer, S., Thomson, P., Wick-
608 erson, J.: The design and implementation of a verification technique for GPU ker-
609 nels. Transactions on Programming Languages and Systems **37**(3), 1–49 (2015).
610 https://doi.org/10.1145/2743017
611 6. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUVerify: a
612 verifier for GPU kernels. In: Proceedings of OOPSLA. pp. 113–132. ACM (2012).
613 https://doi.org/10.1145/2384616.2384625
614 7. Blom, S., Huisman, M., Mihelčić, M.: Specification and verification of
615 GPGPU programs. Science of Computer Programming **95**(P3), 376–388 (2014).
616 https://doi.org/10.1016/j.scico.2014.03.013
617 8. Chatarasi, P., Shirako, J., Kong, M., Sarkar, V.: An extended polyhedral model
618 for SPMD programs and its use in static data race detection. In: Proceedings of
619 LCPC'16. pp. 106–120. Springer (2017). https://doi.org/10.1007/978-3-319-52709-
620 3_10
621 9. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic testing of OpenCL code. In:
622 Proceedings of HVC. pp. 203–218. Springer (2012). https://doi.org/10.1007/978-
623 3-642-34188-5_18
624 10. Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic crosschecking of floating-
625 point and SIMD code. In: Proceedings of EuroSys. pp. 315–328. ACM (2011).
626 https://doi.org/10.1145/1966445.1966475
627 11. Collingbourne, P., Donaldson, A.F., Ketema, J., Qadeer, S.: Interleaving and lock-
628 step semantics for analysis and verification of GPU kernels. In: Proceedings of
629 ESOP. pp. 270–289. Springer (2013). https://doi.org/10.1007/978-3-642-37036-
630 6_16
631 12. Dabrowski, F., Pichardie, D.: A certified data race analysis for a Java-
632 like language. In: Proceedings of TPHOL, pp. 212–227. Springer (2009).
633 https://doi.org/10.1007/978-3-642-03359-9_16

634 13. Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., Devietti, J.: BARRACUDA:
635 Binary-level Analysis of Runtime RAces in CUDA programs. In: Proceedings of
636 PLDI. pp. 126–140. ACM (2017). https://doi.org/10.1145/3062341.3062342
637 14. Ferrell, B., Duan, J., Hamlen, K.W.: CUDA au Coq: A framework for machine-
638 validating GPU assembly programs. In: Proceedings of DATE. pp. 474–479 (2019).
639 https://doi.org/10.23919/DATE.2019.8715160
640 15. Grosser, T., Ramanujam, J., Pouchet, L.N., Sadayappan, P., Pop, S.: Optimistic
641 delinearization of parametrically sized arrays. In: Proceedings of ICS. pp. 351–360.
642 ACM (2015). https://doi.org/10.1145/2751205.2751248
643 16. ul Hassan Khan Khan, A., Al-Mouhamed, M., Fatayer, A., Almousa, A.,
644 Baqais, A., Assayony, M.: Padding free bank conflict resolution for CUDA-
645 based matrix transpose algorithm. In: Proceedings of SNPD. pp. 1–6 (2014).
646 https://doi.org/10.1109/SNPD.2014.6888709
647 17. Holey, A., Mekkat, V., Zhai, A.: HAccRG: Hardware-accelerated data
648 race detection in GPUs. In: Proceedings of ICPP. pp. 60–69 (2013).
649 https://doi.org/10.1109/ICPP.2013.15
650 18. Kamath, A.K., George, A.A., Basu, A.: ScoRD: A scoped race detec-
651 tor for GPUs. In: Proceedings of ISCA. pp. 1036–1049. IEEE (2020).
652 https://doi.org/10.1109/ISCA45697.2020.00088
653 19. Kojima, K., Igarashi, A.: A Hoare logic for SIMT programs. In: Proceedings of
654 APLAS. vol. 8301, pp. 58–73. Springer (2013). https://doi.org/10.1007/978-3-319-
655 03542-0_5
656 20. Kojima, K., Igarashi, A.: A Hoare logic for GPU kernels. Transactions on Compu-
657 tational Logic **18**(1), 1–43 (2017). https://doi.org/10.1145/3001834
658 21. Kojima, K., Imanishi, A., Igarashi, A.: Automated verification of functional correct-
659 ness of race-free GPU programs. Journal of Automated Reasoning **60**(3), 279–298
660 (2018). https://doi.org/10.1007/s10817-017-9428-2
661 22. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program
662 analysis & transformation. In: Proceedings of CGO. pp. 75–88. IEEE (2004).
663 https://doi.org/10.1109/CGO.2004.1281665
664 23. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU
665 kernel functions. In: Proceedings of FSE. pp. 187–196. ACM (2010).
666 https://doi.org/10.1145/1882291.1882320
667 24. Li, G., Gopalakrishnan, G.: Parameterized verification of GPU ker-
668 nel programs. In: Proceedings of IPDPSW. pp. 2450–2459 (2012).
669 https://doi.org/10.1109/IPDPSW.2012.302
670 25. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE:
671 Concolic verification and test generation for GPUs. In: Proceedings of PPoPP.
672 vol. 47, pp. 215–224. ACM (2012). https://doi.org/10.1145/2370036.2145844
673 26. Li, P., Li, G., Gopalakrishnan, G.: Practical symbolic race checking
674 of GPU programs. In: Proceedings of SC. pp. 179–190. IEEE (2014).
675 https://doi.org/10.1109/SC.2014.20
676 27. Li, P., Ding, C., Hu, X., Soyata, T.: LDetector: A low overhead race detector for
677 GPU programs. In: Proceedings of WoDet (2014), http://wodet.cs.washington.
678 edu/wp-content/uploads/2014/02/wodet2014-final14.pdf
679 28. Li, P., Hu, X., Chen, D., Brock, J., Luo, H., Zhang, E.Z., Ding, C.: LD: Low-
680 overhead GPU race detection without access monitoring. Transactions on Architec-
681 ture and Code Optimization **14**(1), 1–25 (2017). https://doi.org/10.1145/3046678
682 29. López, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconce-
683 los, V.T., Yoshida, N.: Protocol-based verification of message-passing par-

684 allel programs. In: Proceedings of OOPSLA. pp. 280–298. ACM (2015).
685 https://doi.org/10.1145/2814270.2814302
686 30. Ma, H., Diersen, S.R., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic analysis
687 of concurrency errors in OpenMP programs. In: Proceedings of ICPP. pp. 510–516.
688 IEEE (2013). https://doi.org/10.1109/ICPP.2013.63
689 31. Muller, S.K., Hoffmann, J.: Modeling and analyzing evaluation cost of CUDA
690 kernels. Proceedings of the ACM on Programming Languages **5**(POPL) (2021).
691 https://doi.org/10.1145/3434306
692 32. Peng, Y., Grover, V., Devietti, J.: CURD: A dynamic CUDA
693 race detector. In: Proceedings of PLDI. pp. 390–403. ACM (2018).
694 https://doi.org/10.1145/3192366.3192368
695 33. Pereira, P., Albuquerque, H., Marques, H., Silva, I., Carvalho, C., Cordeiro, L.,
696 Santos, V., Ferreira, R.: Verifying CUDA programs using SMT-based context-
697 bounded model checking. In: Proceedings of SAC. pp. 1648–1653. ACM (2016).
698 https://doi.org/10.1145/2851613.2851830
699 34. Ruetsch, G., Micikevicius, P.: Optimizing matrix transpose in CUDA. NVIDIA
700 CUDA SDK Application Note **18** (2009), https://www.cs.colostate.edu/
701 ~cs675/MatrixTranspose.pdf
702 35. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing
703 system. In: Proceedings of PARLE. LNCS, vol. 817, pp. 398–413. Springer (1994).
704 https://doi.org/10.1007/3-540-58184-7_118
705 36. Vasconcelos, V.T.: Session types for linear multithreaded functional
706 programming. In: Proceedings of PPDP. pp. 1–6. ACM (2009).
707 https://doi.org/10.1145/1599410.1599411
708 37. Vasconcelos, V.T., Ravara, A., Gay, S.: Session types for functional mul-
709 tithreading. In: Proceedings of CONCUR. pp. 497–511. Springer (2004).
710 https://doi.org/10.1007/978-3-540-28644-8_32
711 38. Wu, M., Ouyang, Y., Zhou, H., Zhang, L., Liu, C., Zhang, Y.: Simulee: Detecting
712 CUDA synchronization bugs via memory-access modeling. In: Proceedings of ICSE.
713 pp. 937–948. ACM (2020). https://doi.org/10.1145/3377811.3380358
714 39. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GRace: A low-overhead mechanism
715 for detecting data races in GPU programs. In: Proceedings of PPoPP. pp. 135–146.
716 ACM (2011). https://doi.org/10.1145/1941553.1941574
717 40. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GMRace: Detecting data races in GPU
718 programs via a low-overhead scheme. Transactions on Parallel and Distributed
719 Systems **25**(1), 104–115 (2014). https://doi.org/10.1109/TPDS.2013.44

720 # A Demonstration of Faial

721 Here we present a demonstration of Faial in three examples.

722 *Example 3.* The following is command-line usage of Faial on Listing 2.3 (l.h.s).
723 In addition to CUDA kernels, Faial allows for the evaluation of access memory
724 protocols stored in the proto file type as shown. Barrier aligning is also shown.

```
725 // Source protocol
726 $ cat inter-iteration.proto
727 shared tile;
728 const global N,
729       global M,
730       local tid,
731       where distinct [tid] &&
732       N > 0 && M > 0;
733
734 foreach (r in 0.. N) {
735   foreach (i in 0.. M) { rw tile[tid]; }
736   sync;
737   foreach (j in 0.. M) { ro tile[tid + j]; }
738 }
739
740 // Step 3: aligned protocol
741 $ faial -A --steps 3 inter-iteration.proto
742 ; a-lang
743 ; a-prog 1
744 locations: tile;
745 globals: N, M;
746 locals: tid;
747 invariant: (proj($T1, tid) != proj($T2, tid) && N > 0) && M > 0;
748
749 code {
750     sync;
751     foreach (i in 0 .. M) {
752         rw tile[tid];
753     }
754     sync;
755     foreach* (r in 1 .. N) {
756         foreach (j in 0 .. M) {
757             ro tile[tid + j];
758         }
759         foreach (i in 0 .. M) {
760             rw tile[tid];
761         }
762         sync;
763     }
764     foreach (j in 0 .. M) {
765         ro tile[tid + j];
766     }
767     sync;
768 }
769 ; end of a-lang
770
771 // Final analysis
772 $ faial inter-iteration.proto
773 *** DATA RACE ERROR ***
774
775 Array:   tile[1]
776 T1 mode: R
777 T2 mode: W
778
779 ----------------
780  Globals  Value
781 ----------------
782  M        2
```

```
----------------
 N        2
----------------
 r        1
----------------

----------------
 Locals  T1  T2
----------------
 i       0   0
----------------
 j       1   0
----------------
 tid     0   1
----------------
```

Example 4. To show intermediate languages within Faial, we now run Example 1 from Section 4.2. Here we show the inlined protocol, aligned protocol, flattened phases, and generated booleans.

```
// Source protocol
$ cat example-1.proto
shared A;
const local tid
      where distinct [tid];

rw A[tid + 1];
ro A[tid + 2];
sync;

// Step 1: inline assignments and replace key-values
$ faial -A --steps 1 example-1.proto
 locations: A;
 globals: ;
 locals: tid;
 invariant: proj($T1, tid) != proj($T2, tid);

code {
     rw A[1 + tid];
     ro A[2 + tid];
     sync;
}

// Step 3: aligned protocol
$ faial -A --steps 3 example-1.proto
; a-lang
; a-prog 1
 locations: A;
 globals: ;
 locals: tid;
 invariant: proj($T1, tid) != proj($T2, tid);

code {
     sync;
     rw A[1 + tid];
     ro A[2 + tid];
     sync;
     sync;
}
; end of a-lang

// Step 6: flatten phases
$ faial -A --steps 6 example-1.proto
; flatacc
; acc 1
 location: A;
```

```
847   locals : tid;
848  pre: true;
849  {
850      rw[1 + tid] if proj($T1, tid) != proj($T2, tid);
851      ro[2 + tid] if proj($T1, tid) != proj($T2, tid);
852  }
853  ; end of flatacc
854
855  // Step 7: generate booleans
856  $ faial -A --steps 7 example-1.proto
857  ; symbexp
858  ; bool 1
859  array: A
860  predicates: ;
861  decls: tid$T2, tid$T1, $T2$mode, $T2$idx$0, $T1$mode, $T1$idx$0;
862  goal: ((tid$T1 != tid$T2 && ($T1$mode ==1 && $T1$idx$0 ==1 +tid$T1)) || (tid$T1 != tid$T2 &&
863         (($T1$mode ==0 && $T2$mode ==1) && $T1$idx$0 ==2 +tid$T1))) && (((tid$T1 != tid$T2
864         && ($T2$mode ==1 && $T2$idx$0 ==1 +tid$T2)) || (tid$T1 != tid$T2 && (($T2$mode ==0 &&
865         $T1$mode ==1) && $T2$idx$0 ==2 +tid$T2))) && $T1$idx$0 ==$T2$idx$0);
866  ; end of symbexp
867
868  // Final analysis
869  $ faial example-1.proto
870  *** DATA RACE ERROR ***
871
872  Array:   A[2]
873  T1 mode: W
874  T2 mode: R
875
876  ----------------
877   Locals  T1  T2
878  ----------------
879   tid     1   0
880  ----------------
```

881 *Example 5.* To demonstrate usage on a real-world CUDA kernel, Faial is run on
882 a matrix transpose from [34], *c.f.*, Listing 2.1 (l.h.s.). The inferred protocol is
883 also shown. Note that Faial handles the representation of multiple arrays in the
884 same protocol, but each array is analyzed independently.

```
885  // CUDA kernel source
886  $ cat transposeCoalesced.cu
887  #include <cuda.h>
888
889  #define TILE_DIM 16
890  #define BLOCK_ROWS 16
891
892  __global__ void kernel (float* odata, float * idata, int width, int height, int nreps) {
893
894    __requires(height == 2048);
895    __requires(width == 2048);
896
897    __shared__ float tile[TILE_DIM][TILE_DIM];
898
899    int xIndex = blockIdx.x * TILE_DIM +threadIdx.x;
900    int yIndex = blockIdx.y * TILE_DIM +threadIdx.y;
901    int index_in = xIndex + (yIndex)*width;
902
903    xIndex = blockIdx.y * TILE_DIM +threadIdx.x;
904    yIndex = blockIdx.x * TILE_DIM +threadIdx.y;
905    int index_out = xIndex + (yIndex)*height;
906
907    for (int r=0; r < nreps; r++) {
908      for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
909        tile [threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
910      }
```

```
911
912        __syncthreads();
913
914        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
915          odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
916        }
917      }
918    }
919
920    // Inferred protocol
921    $ faial -b 16,16 -g 64,64 -A --steps 0 transposeCoalesced.cu
922    arrays: tile, odata, idata;
923    scalars: width, nreps, height, gridDim.y, gridDim.x, blockIdx.y, blockIdx.x, blockDim.y,
924            blockDim.x;
925    pre: ((((proj($T1, threadIdx.x) != proj($T2, threadIdx.x) || proj($T1, threadIdx.y) != proj($T2,
926            threadIdx.y)) && blockIdx.x <gridDim.x) && threadIdx.x <blockDim.x) && blockIdx.y <
927            gridDim.y) && threadIdx.y <blockDim.y;
928
929    code {
930              local threadIdx.y;
931              local threadIdx.x;
932              assert (height == 2048)
933              assert (width == 2048)
934              local xIndex = (16 * blockIdx.x) + threadIdx.x;
935              local yIndex = (16 * blockIdx.y) + threadIdx.y;
936              local index_in = xIndex + (yIndex * width);
937              local xIndex = (16 * blockIdx.y) + threadIdx.x;
938              local yIndex = (16 * blockIdx.x) + threadIdx.y;
939              local index_out = xIndex + (yIndex * height);
940              foreach (r in 0 .. nreps) {
941                      foreach (i in 0 .. 16; i + 16) {
942                              rw tile[threadIdx.y + i, threadIdx.x];
943                              ro idata[index_in + (i * width)];
944                      }
945                      sync;
946                      foreach (i in 0 .. 16; i + 16) {
947                              rw odata[index_out + (i * height)];
948                              ro tile[threadIdx.x, threadIdx.y + i];
949                      }
950              }
951    }
952
953    // Final analysis
954    $ faial -b 16,16 -g 64,64 transposeCoalesced.cu
955    *** DATA RACE ERROR ***
956
957    Array:    tile[15, 14]
958    T1 mode: W
959    T2 mode: R
960
961    --------------------
962     Globals      Value
963    --------------------
964     blockIdx.x   0
965    --------------------
966     blockIdx.y   0
967    --------------------
968     nreps        2
969    --------------------
970     r            1
971    --------------------
972
973    ----------------------
974     Locals        T1  T2
975    ----------------------
976     i             0   0
977    ----------------------
978     i1            0   0
```

```
979 ---------------------
980  threadIdx.x  15  14
981 ---------------------
982  threadIdx.y  14  15
983 ---------------------
```