GPUMC: A Stateless Model Checker for GPU Weak Memory Concurrency

Soham Chakraborty[1,2], S. Krishna[2], Andreas Pavlogiannis[3], and Omkar Tuppe[2(✉)]

[1] TU Delft, Delft, Netherlands
s.s.chakraborty@tudelft.nl
[2] IIT Bombay, Mumbai, India
{krishnas,omkarvtuppe}@cse.iitb.ac.in
[3] Aarhus University, Aarhus, Denmark
pavlogiannis@cs.au.dk

Abstract. GPU computing is embracing weak memory concurrency for performance improvement. However, compared to CPUs, modern GPUs provide more fine-grained concurrency features such as scopes, have additional properties like divergence, and thereby follow different weak memory consistency models. These features and properties make concurrent programming on GPUs more complex and error-prone. To this end, we present GPUMC, a stateless model checker to check the correctness of GPU shared-memory concurrent programs under scoped-RC11 weak memory concurrency model. GPUMC explores all possible executions in GPU programs to reveal various errors - races, barrier divergence, and assertion violations. In addition, GPUMC also automatically repairs these errors in the appropriate cases.

We evaluate GPUMC on benchmarks and real-life GPU programs. GPUMC is efficient both in time and memory in verifying large GPU programs where state-of-the-art tools are timed out. In addition, GPUMC identifies all known errors in these benchmarks compared to the state-of-the-art tools.

1 Introduction

In recent years GPUs have emerged as mainstream processing units, more than just accelerators [29,66,67,73]. Modern GPUs provide support for more fine-grained shared memory access patterns, allowing programmers to optimize performance beyond the traditional lock-step execution model typically associated with SIMT architectures. To this end, GPU programming languages such as CUDA and OpenCL [2,5], as well as libraries [3,4], have adopted C/C++ shared memory concurrency primitives.

Writing correct and highly efficient shared-memory concurrent programs is already a challenging problem, even for CPUs. GPU concurrency poses further challenges. Unlike CPU threads, the threads in a GPU are organized hierarchically and synchronize via barriers during execution. Moreover, shared-memory

R. Piskac and Z. Rakamarić (Eds.): CAV 2025, LNCS 15933, pp. 321–346, 2025.
https://doi.org/10.1007/978-3-031-98682-6_17

accesses are *scoped*, resulting in more fine-grained rules for synchronization, based on the proximity of their threads. Although these primitives and rules play a key role in achieving better performance, they are also complex and prone to errors.

GPU concurrency may result in various types of concurrency bugs – assertion violations, data races, heterogeneous races, and barrier divergence. While assertion violations and data race errors are well-known in CPU concurrency, they manifest in more complicated ways in the context of GPU programs. The other two types of errors, heterogeneous races and barrier divergence, are GPU specific. To catch these errors, it is imperative to explore all possible executions of a program.

The set of possible executions of a GPU concurrent program is determined by its underlying consistency model. State-of-the-art architectures including GPUs follow weak consistency, and as a result a program may exhibit extra behaviors in addition to the interleaving executions or more formally sequential consistency (SC) [49]. However, as the weak memory concurrency models in GPUs differ from the ones in the CPUs, the state-of-the-art analysis and verification approaches for programs written for CPUs do not suffice in identifying these errors under GPU weak memory concurrency. As a result, automated reasoning of GPU concurrency, particularly under weak consistency models, even though a timely and important problem, has remained largely unexplored.

To address this gap, in this paper we develop the GPUMC model checker for a scoped-C/C++ programming languages [61] for GPUs. Scoped-C/C++ has all the shared memory access primitives provided by PTX and Vulkan, and in addition, provide SC memory accesses. The recent work of [61] formalizes the scoped C/C++ concurrency in scoped-RC11 memory model (SRC11), similarly to the formalization of C/C++ concurrency in RC11 [48]. Consequently, GPUMC is developed for the SRC11 model. The consistency properties defined by SRC11, scoped C/C++ programming language follows catch fire semantics similar to traditional C/C++, that is, a program having a SRC11 consistent execution with a data race has undefined behavior. In addition, scoped C/C++ defines *heterogeneous race* [30,35,61,78] based on the scopes of the accesses, and a program having a SRC11-consistent execution with heterogeneous race also has undefined behavior.

Stateless Model Checking (SMC) is a prominent automated verification technique [23] that explores all possible executions of a program in a systematic manner. However, the number of executions can grow exponentially larger in the number of concurrent threads, which poses a key challenge to a model checker. To address this challenge, partial order reduction (POR) [24,32,68] and subsequently dynamic partial order reduction (DPOR) techniques have been proposed [28]. More recently, several DPOR algorithms are proposed for different weak memory consistency models to explore executions in a time and space-efficient manner [6,7,10,43,64,83]. For instance, GenMC-Trust [44] and POP [8] are recently proposed polynomial-space DPOR algorithms. While these techniques are widely applied for programs written for CPUs (weak memory) concur-

rency models [7,10,43–45,64], to our knowledge, DPOR-based model checking has not been explored for GPU weak memory concurrency.

GPUMC extends the GenMC-TruSt [44] approach to handle the GPU-specific features that the original GenMC lacks. More specifically, GPUMC implements an exploration-optimal, sound, and complete DPOR algorithm with linear memory requirements that is also parallelizable. Besides efficient exploration, GPUMC detects all the errors discussed above and automatically repairs certain errors such as heterogenous races. Thus GPUMC progressively transforms a heterogeneous-racy program to generate a heterogeneous-race-free version. We empirically evaluate GPUMC on several benchmarks to demonstrate its effectiveness. The benchmarks range from small litmus tests to real applications, used in GPU testing [51,77], bounded model checking [52], and verification under sequential consistency [39,40]. GPUMC explores the executions of these benchmarks in a scalable manner and identifies the errors. We compare GPUMC with DARTAGNAN [78], a bounded model checker for GPU weak memory concurrency [52]. GPUMC identifies races which are missed by DARTAGNAN in its benchmarks and also outperforms DARTAGNAN significantly in terms of memory and time requirements in identifying concurrency errors.

Contributions and Outline. To summarize, the paper makes the following contributions. Sections2 and 3 provide an overview of GPU weak memory concurrency and its formal semantics. Next, Sects. 4 and 5 discuss the proposed DPOR algorithm and its experimental evaluation. Finally, we discuss the related work in Sect. 6 and conclude in Sect. 7.

2 Overview of GPU Concurrency

A shared memory GPU program consists of a fixed set of threads with a set of shared memory locations and thread-local variables. Unlike in the CPU, the GPU threads are structured in hierarchies at multiple levels: cooperative thread array(CTA) (cta), GPU (gpu), and system (sys), where cta is a collection of threads and gpu is a group of cta, and finally sys consists of a set of gpus and threads of other devices such as CPUs. Thus, a thread can be identified by its (cta, gpu) identifiers and its thread identifier. The system (sys) is the same for all threads.

Shared memory operations are one of read, write, atomic read-modify-write (RMW), fence (fnc) or barrier (bar). Similar to the C/C++ concurrency [36,37], these accesses are non-atomic read or write, or atomic accesses with memory orders. Thus accesses are classified as: non-atomic (NA), relaxed (RLX), acquire (ACQ), release (REL), acquire-release (ACQ-REL), or sequentially consistent (SC). In increasing strength, $\text{NA} \sqsubset \text{RLX} \sqsubset \{\text{REL}, \text{ACQ}\} \sqsubset \text{ACQ-REL} \sqsubset \text{SC}$.

The shared memory accesses of the GPU are further parameterized with a scope $\mathsf{sco} \in \{\mathsf{cta}, \mathsf{gpu}, \mathsf{sys}\}$. The scope of an operation determines its role in synchronizing with other operations in other threads based on proximity. Thus,

shared memory accesses are of the following form where o_r, o_w, o_u, o_f denote the memory orders of the read, write, RMW, and fence accesses respectively.

$$r = X^{\mathsf{sco}}_{o_r} \mid X^{\mathsf{sco}}_{o_w} = E \mid r = \mathsf{RMW}^{\mathsf{sco}}_{o_u}(X, E_r, E_w) \mid \mathsf{fnc}^{\mathsf{sco}}_{o_f} \mid \mathsf{bar}^{\mathsf{sco}}(\mathsf{id})@l@$$

A read access $r = X^{\mathsf{sco}}_{o_r}$ returns the value of shared memory location/-variable X to thread-local variable r with memory order o_r selected from $\{\text{NA}, \text{RLX}, \text{ACQ}, \text{SC}\}$. A write access $X^{\mathsf{sco}}_{o_w} = E$ writes the value of expression E to the location X with memory order o_w selected from $\{\text{NA}, \text{RLX}, \text{REL}, \text{SC}\}$. The superscript sco refers to the scope. An RMW access $r = \mathsf{RMW}^{\mathsf{sco}}_{o_u}(X, E_r, E_w)$, atomically updates the value of location X with the value of E_w if the read value of X is E_r. On failure, it performs only the read operation. The memory order of an RMW is o_u selected from $\{\text{RLX}, \text{REL}, \text{ACQ}, \text{ACQ-REL}, \text{SC}\}$. A fence access fnc is performed with a memory order o_f selected from $\{\text{REL}, \text{ACQ}, \text{ACQ-REL}, \text{SC}\}$. GPUs also provide barrier operations where a set of threads synchronize and therefore affect the behaviors of a program. For a barrier operation $\mathsf{bar}^{\mathsf{sco}}(\mathsf{id})$, sco refers to the scope of the barrier and *id* denotes the barrier identifier. We model barriers as acquire-release RMWs ($\mathsf{RMW}^{\mathsf{sco}}_{\text{ACQ-REL}}$) parameterized with scope sco on a special auxiliary variable (similar to [46]).

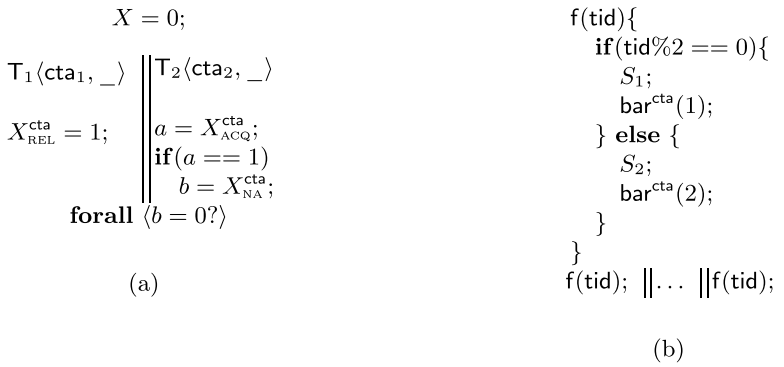


Fig. 1. Example of GPU concurrency errors. In (a), we have two threads T_1, T_2 from the CTAs $\mathsf{cta}_1, \mathsf{cta}_2$. In (b) all threads are in the same CTA.

2.1 GPU Concurrency Errors

Traditionally, two key errors in shared memory concurrency are assertion violations and data races. In addition, concurrent programs for GPUs may contain heterogeneous races and barrier divergence errors. The behavior of a program with data race or heterogeneous race is undefined, while divergence errors may lead to deadlocks [2,61,78, Section 16.6.2].

Assertion violation: In our benchmarks assertion violations imply weak memory bugs. Assertions verify the values of the variables and memory locations in a program. If the intended values do not match, it results in an assertion violation. Consider the program in Fig. 1a having the assertion **forall** $b = 0$? which checks whether, for all executions, b is 0. If the value of X read into a in T_2 is 1, then b cannot read a stale value 0 from X and the assertion fails.

Data Race: Two operations a and b in an execution are said to be in a data race [61] [78] if (i) a and b are concurrent, that is, not related by *happens-before*, (ii) they access the same memory location, (iii) at least one of the accesses is a write operation, and (iv) at least one of the accesses is a non-atomic operation. In Fig. 1a, if $\mathsf{cta}_1 = \mathsf{cta}_2$, the threads are in the same cta. In that case, if the acquire-read of X in the second thread reads from the release-write in the first thread, then it establishes synchronization. Hence, the release-write of X *happens-before* the non-atomic read of X, and the program has no data race.

Heterogeneous Race: Two operations a and b in an execution are in a heterogeneous race if (i) a and b are concurrent, (ii) they access the same memory location, (iii) at least one of the accesses is a write operation, and (iv) both accesses are atomic with non-inclusive scope, that is, the scopes of each access includes the thread executing the other access. Note that a heterogeneous race may take place between atomic accesses. In Fig. 1a, if $\mathsf{cta}_1 \neq \mathsf{cta}_2$ then the acquire-read and release-write do not synchronize and consequently are in a heterogeneous race. Then the program also has a data race between the non-atomic read of X and release-write of X.

Barrier Divergence: Given a barrier, the threads within the given scope of the barrier synchronize. During execution, while a thread reaches the barrier, it waits for all the other threads to reach the barrier before progressing the execution further. Consider the program in Fig. 1b, where all threads execute the function $\mathsf{f}()$. The threads with even thread identifiers synchronize to $\mathsf{bar}(1)$ and the thread with odd thread identifiers synchronize to $\mathsf{bar}(2)$. Hence the threads are diverging and not synchronizing to a single barrier. Modern GPUs consider it as a divergence error as the non-synchronizing threads may result in a deadlock. Following the definition from [2, Section 16.6.2], we report barrier divergence if at least one of the threads participating in the barrier is blocked at the barrier at the end of execution (no next instruction to execute).

3 Formal Semantics

In this section, we elaborate on the formal semantics of GPU concurrency. A program's semantics is formally represented by a set of *consistent* executions. An execution consists of a set of events and various relations between the events.

Events. An event corresponds to the effect of executing a shared memory or fence access in the program. An event $e = \langle id, \mathsf{tid}, ev, \mathsf{loc}, \mathsf{ord}, \mathsf{sco}, \mathsf{Val} \rangle$ is represented by a tuple where id, tid, ev, loc, ord, sco, Val denote the event identifier, thread identifier, memory operation, memory location accessed, memory order, scope, read or written value. A read, write, or fence access generates a read, write, or fence event. A successful RMW generates a pair of read and write events and a failed RMW generates a read event. A read event $\mathsf{R}^{\mathsf{sco}}_{o}(X, v)$ reads from location X and returns value v with memory order o and scope sco. A write event $\mathsf{W}^{\mathsf{sco}}_{o}(X, v)$ writes value v to location X with memory order o and scope sco. A fence event $\mathsf{F}^{\mathsf{sco}}_{o}$ has memory order o and scope sco. Note that for a fence event, $\mathsf{loc} = \mathsf{Val} = \bot$. The set of read, write, and fence events are denoted by R, W, and F respectively.

Relations. The events of an execution are associated with various relations. The relation program-order (po) denotes the syntactic order among the events. In each thread po is a total order. The relation reads-from (rf) relates a pair of same-location write and read events w and r having the same values to denote that r has read from w. Each read has a unique write to read from (rf^{-1} is a function). The relation coherence order (co) is a total order on the same-location write events. The relation rmw denotes a successful RMW operation that relates a pair of same-location read and write events r and w which are in *immediate-*po relation, that is, no other event a exists such that (r, a) and (a, w) are in po relations. We derive new relations following the notations below.

Notation on Relations. Given a binary relation B, we write B^{-1}, $B^{?}$, B^{+}, B^{*} to denote its inverse, reflexive, transitive, reflexive-transitive closures respectively. We compose two relations B_1 and B_2 by $B_1; B_2$. Given a set A, $[\mathrm{A}]$ denotes the identity relation on the set A. Given a relation B, we write $B_{=\mathsf{loc}}$ and $B_{\neq\mathsf{loc}}$ to denote relation B on same-location and different-location events respectively. For example, $\mathsf{po}_{=\mathsf{loc}}$ relates a pair of same-location events that are po-related. Similarly, $\mathsf{po}_{\neq\mathsf{loc}}$ relates po-related events that access different locations. Relation from-read (fr) relates a pair of same-location read and write events r and w'. If r reads from w and w' is co-after w then r and w' are in fr relation: $\mathsf{fr} \triangleq \mathsf{rf}^{-1}; \mathsf{co}$.

Execution and Consistency. An execution is a tuple $e.g.raph = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co}, \mathsf{rmw} \rangle$ consisting of a set of events E, and the sets of po, rf, co, and rmw relations. We represent an execution as a graph where the nodes represent events and different types of edges represent respective relations. A concurrency model defines a set of axioms or constraints based on the events and relations. If an execution satisfies all the axioms of a memory model then the execution is consistent in that memory model.

SRC11 Consistency Model. We first explain the relations of the RC11 model [48] which is extended to SRC11 [61] for GPUs, defined in Fig. 3.

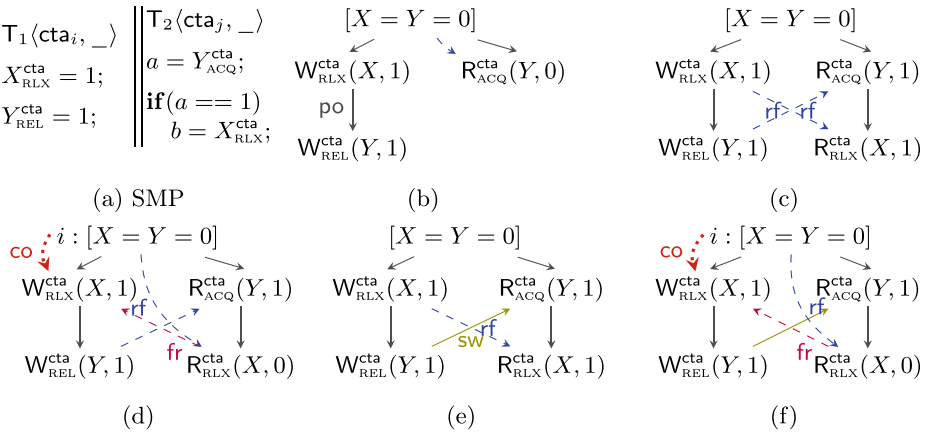


Fig. 2. Executions shown in (b) and (c) are independent of whether $i = j$ or not. (b) shows an execution where Y reads 0 from the initial location. (c) shows an execution where Y and X read 1 in T_2. (d) shows an execution where Y reads 1 from T_1 but cannot synchronize, as T_1 and T_2 are in different CTAs ($i \neq j$). If $i \neq j$, X may read 0 from initialization. (e) is a special case of execution shown in (c) where $i = j$. If $i == j$, then read and write on Y are in synchronization relation because these accesses on Y are scope-inclusive. (f) shows an execution where there is a synchronization on Y with an inclusion relation (so again $i = j$). Hence, X in T_2 cannot read value 0 from initialization, as it violates the coherence axiom; consequently, the execution is forbidden.

RC11 Relations. Relation extended-coherence-order (eco) is a transitive closure of the read-from (rf), coherence order (co), and from read (fr) relations, that is, $\mathsf{eco} \triangleq (\mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr})^+$. Note that the eco related events always access the same memory location.

Relation synchronizes-with (sw) relates a release event to an acquire event. For example, when an acquire read reads from a release write then the pair establishes an sw relation. In general, sw uses release-sequence rseq that starts at a release store or fence event and ends at an acquire load or fence event with an intermediate chain of rf-related rmw relations. Finally, relation happens-before (hb) is the transitive closure of the po and sw relations.

To relate the SC memory accesses and fences, the RC11 model defines the scb relation. A pair of events a and b is in scb relation in one of these cases: (1) (a, b) is in po, co, or fr relation. (2) a and b access the same memory location and are in hb relation, that is $\mathsf{hb}_{=\mathsf{loc}}(a, b)$ holds. (3) a has a different-location po-successor c, and event b has a different-location po-predecessor d, and (c, d) is in happens-before relation.

Based on the scb relation, RC11 defines $\mathsf{psc}_{\mathsf{base}}$ and $\mathsf{psc}_{\mathsf{F}}$. Relation $\mathsf{psc}_{\mathsf{base}}$ relates a pair of SC (memory access or fence) events and $\mathsf{psc}_{\mathsf{F}}$ relates a pair of SC fence events. Finally, RC11 defines psc relation by combining $\mathsf{psc}_{\mathsf{base}}$ and $\mathsf{psc}_{\mathsf{F}}$ relations.

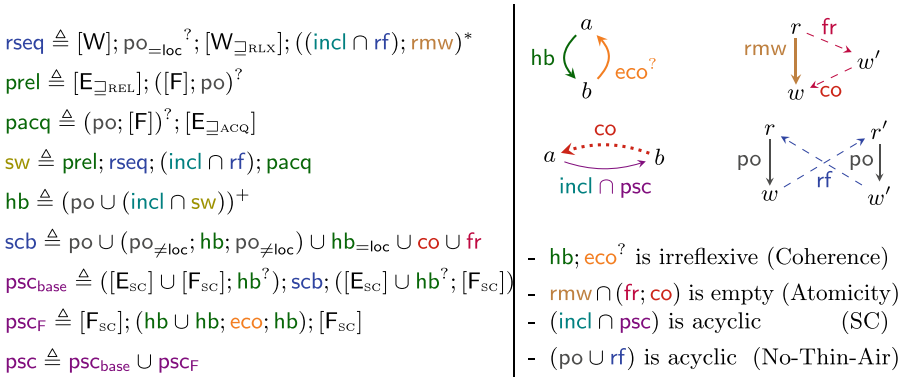


Fig. 3. SRC11 relations and axioms with some violation patterns.

RC11 to SRC11. The SRC11 model refines the RC11 relations with inclusion (incl). Relation $\mathsf{incl}(a, b)$ holds when (i) a and b are atomic events, (ii) if the scope of a or b includes the thread of b or a respectively, and (iii) if both a and b access memory then they access the same memory location. Note that the incl relations are non-transitive, that is, $\mathsf{incl}(a, b)$ and $\mathsf{incl}(b, c)$ *does not* imply an $\mathsf{incl}(a, c)$ relation. To see this, consider events a, b, c having scopes $\mathsf{cta}_1, \mathsf{gpu}_1$ and cta_2 respectively where $\mathsf{cta}_1, \mathsf{cta}_2$ belong to GPU gpu_1. Then we have $\mathsf{incl}(a, b)$ and $\mathsf{incl}(b, c)$ but not $\mathsf{incl}(a, c)$.

Based on the incl relation, the rseq, sw, and hb relations are extended in the SRC11 model. In SRC11, the rf relation in the rseq and sw relations must also be in the incl relation. Note that, even then, the sw related events may not be in the incl relation. Finally, hb in SRC11 is the transitive closure of the po and incl-related sw relations.

SRC11 **Axioms.** An execution in SRC11 is consistent when it satisfies the axioms in Fig. 3. The (Coherence) axiom ensures that the hb relation or the combination of hb and eco relations is irreflexive and does not create any cycle in the execution graph. The (Atomicity) axiom ensures that there is no intermediate event on the same memory location between a pair of events that are rmw-related. The *SC* axiom forbids any cycle between the SC events which are both in the psc relation and the incl relation. Finally, the (No-Thin-Air) axiom forbids any cycle composed of po and rf relations. These axioms essentially forbid the patterns shown in Fig. 3 in an execution graph. Among these scoped-RC11 axioms, (Atomicity) and (No-Thin-Air) are the same as those of RC11. The (Coherence) and (SC) axioms differ as they use more fine-grained incl relations for the scoped accesses.

Example. Consider the program and its execution graphs in Fig. 2. If $i \neq j$, then the accesses on Y do not synchronize, resulting in Fig. 2d. If $i = j$ then

the accesses on Y synchronize which results in Fig. 2c. The execution in Fig. 2f is forbidden as it violates the (Coherence) axiom.

4 GPUMC: Model Checking under SRC11

In this section we discuss the GPUMC approach in Sect. 4.1 followed by a running example in Sect. 4.2. Finally, in Sect. 4.3 we discuss the soundness, completeness, and optimality of the proposed exploration algorithm.

4.1 DPOR Algorithm

GPUMC extends GenMC-TruSt and is in the same spirit as other well known dynamic partial order reduction (DPOR) algorithms [7,10,28,43–45,64].

It verifies a program by exploring all its executions in a systematic manner, ensuring that no execution is visited more than once. Like [44], our algorithm also takes only polynomial space.

Outline Algorithm 1 invokes the Explore procedure to explore the executions of input program under SRC11. The Explore procedure uses Algorithm 2 to enable a read operation to read-from possible writes and thereby explore multiple executions, Algorithm 3 to ensure no execution is explored more than once, and Algorithm 4 to identify and fix errors.

Explore procedure The Explore procedure explores executions $\mathcal{G}$, starting from an empty execution $\mathcal{G}_\emptyset$ where $E = \emptyset$, as long as they are consistent for a given memory model, in this case SRC11 (see Lines 3 to 6 of Algorithm 1). Next, if some of the threads are waiting at a barrier, while all other threads have finished execution, then we observe a *barrier divergence*, and the execution is said to be Blocked. In a blocked execution, different threads may be waiting at different barriers. In this case (line 7), we report the divergence and terminate. Otherwise, we continue

Algorithm 1: DPOR($\mathcal{P}$)

```
   Input: program P
1  Explore(P, G_∅)
2  Procedure Explore(P, G)
3      if ¬PoRfAcyclic() then return
4      if ¬Coherent() then return
5      if ViolateAtomicity() then return
6      if ¬InclPscAcyclic() then return
7      if Blocked(G) then output
         "divergence in G"
8      switch e ← NextEvent(P, G) do
9          case assertion violation do
10             output "Error in G"
11         case ⊥ do
               // no next event
12             output "G"
13         case e = W(x, v) do
               //add W(x, v) to G
14             CheckAndRepairRace(G, e)
15             G' = addco(P, G, e)
16             Explore(P, G')
17             DelayedRFs(G, e)
18         case e =R(x, _) do
19             reversible(e) = true
               //W^x is set of writes on
                 x
20             for w ∈ W^x do
                   //add rf from w ∈ W^x
21                 G' = addRF(G, w, e)
22                 CheckAndRepairRace(G', e)
23                 Explore(P, G')
```

Algorithm 2: DelayedRFs($e.g.raph, w$)

```
1 let R be set of reversible reads in e.g.raph
2 for each r =R(x, _)∈ R s.t. r ∉ porf.w do
3     Deleted ← {e ∈ E | r <exe e ∧ e ∉ porf.w}
      //porf.w={e | ∃ a po, rf path in e.g.raph from e to w}
4     if CheckOptimal(e.g.raph, Deleted ∪ {r}, w, r) then
5         e.g.raph' ← addRF(e.g.raph|E\Deleted, w, r),
6         for each read r ∈ e.g.raph' ∩ R ∩ porf.w set reversible(r) = False
7         Explore(P,e.g.raph')
```

exploration by picking the next event (line 8). This schedules a thread and the next enabled event of that thread. We use the total order $<_{exe}$ to denote the order in which events are added to the execution.

The exploration stops if an assertion is violated (line 10), or when all events from all threads are explored (line 12). The algorithm reports an error in the first case and in the second case outputs the graph $\mathcal{G}$.

If the exploration is not complete and the current event e is a write (line 13), then the procedure CheckAndRepairRace detects races due to events conflicting with e (line 14), and also offers to repair them. On detecting a race, the algorithm chooses one of the following based on user choice – (i) announce the race and stop exploration, or (ii) announce the race and continue exploration, or (iii) announce the race and repair the race.

Apart from calling Explore recursively (Line 16) after adding the necessary co edges (line 15) to $\mathcal{G}$, we check if e can be paired with any existing read in $\mathcal{G}$ (line 17). These reads are called "reversible" as we can reverse their order in the execution by placing them after the writes they read from. On a read event r, we consider all possible rfs for r and extend the execution $\mathcal{G}$ to a new execution $\mathcal{G}'$ (addRF, Line 21).

Algorithm 3: CheckOptimal($e.g.raph, Deleted, w, r$)

```
1 for each event e ∈ Deleted do
      // RF(e) is the write from which e reads
2     if e = R(x)∧ e <exe RF(e) ∧ RF(e) ∈ Deleted then return false
3     e' ← if e = W(x, v) then e else RF(e)
4     let Eset = {e'' | e'' <exe e ∨ e'' ∈ porf.w}
5     if e' co_x e'' for some e'' ∈ Eset then return false
6 return true
```

DelayedRFs procedure The procedure pairs all reversible reads r in $\mathcal{G}$ with all same-location write events w (line 1) provided r is not in the po ∪ rf prefix of w in $\mathcal{G}$ (line 2), to preserve the (No-Thin-Air) axiom. Moreover, a new execution $\mathcal{G}'$ is obtained from $\mathcal{G}$ where r reads from w (line 5), and all events between r and w which are not po ∪ rf before w are deleted (line 3).

CheckOptimal procedure To ensure that no execution is explored twice, the CheckOptimal procedure ensures that all writes in the deleted set are co-

maximal wrt their location, and all reads in the deleted set read from co-maximal writes. This is done by lines 2 to 5 in CheckOptimal.

CheckAndRepairRace procedure We check for races while adding each write w to the execution. For instance, assume that all the reads and writes have been explored (Line 1). For each event e' in this set which is not related to w by hb, we check if any one of them is non-atomic to expose a *data race*. If both have atomic accesses, we check if they are not scope-inclusive to report a *heterogeneous race* (Line 3). Likewise, for each read event added, we consider all explored writes (line 2), and repeat the same check to expose a *data race* or a *heterogeneous race*.

In addition, we also have an option of repair. In `Repair` (line 6, CheckAndRepairRace), we either skip and return to Explore, or do the following repairs and terminate. First, if e and e' respectively have atomic and non-atomic accesses with non-inclusive scopes, then we update their scope to make them inclusive: for instance, if e,e' are in different CTAs, we update their scopes to GPU-level. Second, if at least one of e, e' is a non-atomic access, then we update the non-atomic access to relaxed atomic, and update the scopes so that e, e' have the same scope to prevent a heterogeneous race between them later. However, currently, we do not repair on non-atomic location data types.

Algorithm 4: CheckAndRepairRace($e.g.raph, e$)

```
1 if e = W(x, v) then WR ← set of seen reads/writes on x
2 if e = R(x, v) then WR ← set of seen writes on x
3 for each e' ∈ WR s.t. e' ∉ hb.e ∧ e ∉ hb.e' do
4     if ¬(IsAtomic(e)) ∨ ¬(IsAtomic(e')) ∨ ¬(IsScopeInclusive(e.g.raph, e, e'))
        then
5         ReportRace(e, e')
6         Repair(P, e.g.raph, e, e')
```

Comparison with State-of-the-Art. We discuss how our algorithm differs from the existing DPOR algorithms. The first departure comes in the Explore procedure where we perform consistency checking: Lines 3 to 6 are specific to the Scoped RC11 model which is not handled by any of the existing algorithms including the most recent [8,44], since none of them handle scoped models. The DelayedRFs procedure is standard in all DPOR algorithms and checks if we can pair reads with eligible writes which have been explored later. Next we have CheckOptimal, which ensures that we are optimal while exploring executions: here, the optimality check of [8] is tailored for sequential consistency; we extend the optimality checking algorithm for RC11 [44] to SRC11. While optimality is achieved by ensuring co-maximality on writes [44], there could be optimal co orderings that are inconsistent in the non-scoped setting, which are consistent in the scoped case which need to be considered to achieve completeness. This needed careful handling to achieve polynomial space just as [44]. Finally, our CheckAndRepairRace algorithm is novel and differs from all existing approaches as it reports and also repairs heterogeneous races.

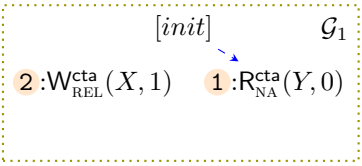
4.2 Exploring the Executions of SEG

We now illustrate the GPUMC algorithm on program SEG as a running example. The assertion violation to check is **exists**$(a = 1 \wedge b = 1)$. This program has 4 consistent executions under SRC11.

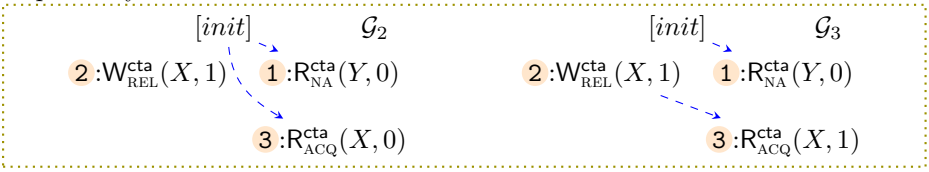
The exploration begins with the empty execution, with no events and all relations being empty. As we proceed with the exploration, we use numbers 1, 2, ... to denote the order in which events are added to the execution. Among the enabled events, we have the read from Y, namely, $a = Y_{\text{NA}}$ in thread T_2 and the write to X in T_1. We add two events for these accesses to the execution (lines 18, 21, 13 in EXPLORE). The read on Y has only the initial value 0 to read from; this is depicted by the rf edge to 1, obtaining $\mathcal{G}_1$. On each new call to EXPLORE, the partial execution is checked for consistency (lines 3-6). $\mathcal{G}_1$ is consistent.

$$
\begin{array}{l}
X = Y = 0; \\
\mathsf{T}_1\langle \mathsf{cta}_1, \_\rangle \\
X^{\mathsf{cta}}_{\text{REL}} = 1; \ @l@ \\
Y^{\mathsf{cta}}_{\text{REL}} = 1;
\end{array}
\quad \Big\| \quad
\begin{array}{l}
\mathsf{T}_2\langle \mathsf{cta}_1, \_\rangle \\
a = Y^{\mathsf{cta}}_{\text{NA}}; \ @l@ \\
b = X^{\mathsf{cta}}_{\text{ACQ}};
\end{array}
\quad @c@
$$

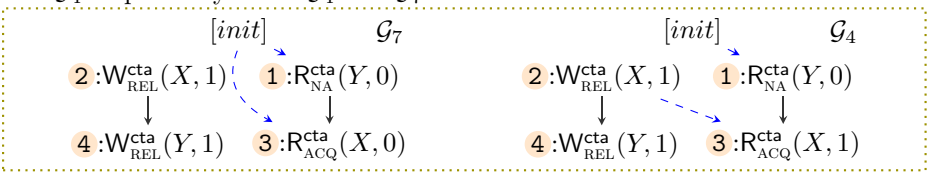
(SEG)



Next, the read event on X from T_2 is added (line 18) having two sources to read from X (line 20): the initial write to X, and the write event 2. This provides two branches to be explored, with consistent executions $\mathcal{G}_2$ and $\mathcal{G}_3$ respectively.



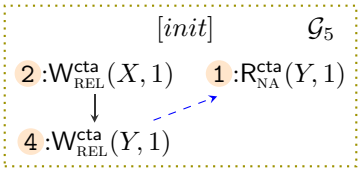
Next, we add write on Y from T_1 to $\mathcal{G}_2, \mathcal{G}_3$ which results in executions $\mathcal{G}_7$ and $\mathcal{G}_4$ respectively. Both $\mathcal{G}_4$ and $\mathcal{G}_7$ are consistent executions.

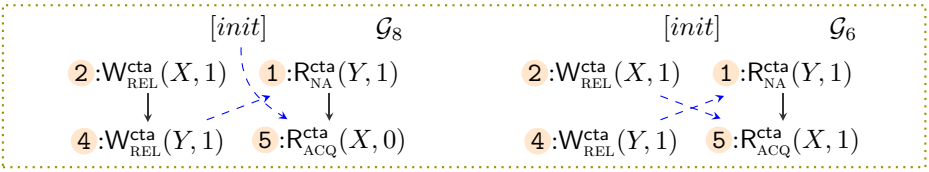


Reversible Reads. In $\mathcal{G}_4$, we observe that the read on Y (1) can also read from the write 4 which was added to the execution later. Enabling 1 to read from 4 involves swapping these two events so that the write happens before the corresponding read. Since 2 is po-before 4, both of these events must take

place before the read from Y (1) for the rf to be enabled. The read from X (3) however, has no dependence on the events in the first thread and happens after 1. Therefore, we can delete (line 3 in DelayedRFs) 3, and add the read from X later, after enabling the rf from 4 to 1 (line 5 in DelayedRFs). The optimality check (line 4 in DelayedRFs) is passed in this case (see also the paragraph on optimality below) and we obtain execution $\mathcal{G}_5$.

We continue exploring from $\mathcal{G}_5$, adding the read on X (Line 18 in Explore) from T_2. Here, X may read from (Line 20, Explore) the initial write or 2. This results in executions $\mathcal{G}_6$ and $\mathcal{G}_8$ which are both consistent.



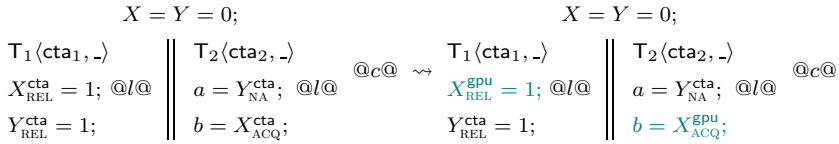


Optimality. From $\mathcal{G}_7$, we do not consider the possibility of Y reading from 4 as it would result in an execution identical to $\mathcal{G}_8$, and consequently violate optimality. The CheckOptimal procedure checks it to ensure that no execution is explored more than once. This check enforces a "co-maximality" criterion on the events that are deleted while attempting a swap between a read event and a later write event: this is exactly where $\mathcal{G}_4$ and $\mathcal{G}_7$ differ. In $\mathcal{G}_7$, while considering the later write on Y (4) to read from for the read event (1), the deleted (line 3, DelayedRFs) read event on X (3) reads from the initial write of X which is not co-maximal since it is co-dominated by 2 (lines 3-5 in CheckOptimal). Hence, the check-in line 5 of CheckOptimal fails. In $\mathcal{G}_4$ however, the deleted read on X (3) reads from a co_x-maximal write, and the test passes. Thus, the algorithm only considers the possibility of the Y reading from 4 in $\mathcal{G}_4$, avoiding redundancy.

Program Repair. The exploration algorithm detects the assertion violation in $\mathcal{G}_6$ (since both a, b read values 1) and detects a data race between 1 and 4.

If GPUMC exploration encounters a heterogeneous race between a pair of accesses then GPUMC automatically repairs the race. To do so, GPUMC changes the scope of the accesses to enforce an inclusion relation. After fixing a heterogeneous race GPUMC terminates its exploration.

Consider a variant of the SEG program where T_1 and T_2 are in different CTAs, GPUMC fixes the heterogeneous race by transforming the scope from cta to gpu.



4.3 Soundness, Completeness and Optimality

Theorem 1. *The DPOR algorithm for* SRC11 *is sound, complete and optimal.*

Soundness. The algorithm does not continue exploration from any inconsistent execution as ensured by Lines 3 to 6 in Algorithm 1, and is therefore sound.

Completeness. The DPOR algorithm is complete as it does not miss any consistent and full execution. We prove this in the following steps:

- We first show that starting from any consistent execution $\mathcal{G}$, we can uniquely roll back to obtain the previous execution $\mathcal{G}_p$ (see the supplement for the algorithm to compute $\mathcal{G}_p$ from $\mathcal{G}$). This is proved using the fact that we have a fixed order in exploring the threads, along with the conditions that allow a swap between a read and a later write to take place. To allow a swap of a read r on some variable (say x), all events in *Deleted* respect "co_x-maximality". This is enforced by CheckOptimal and allows us to uniquely construct the previous execution $\mathcal{G}_p$.
- Second, we show that Explore$(\mathcal{P}, \mathcal{G}_p)$ leads to the call of Explore$(\mathcal{P}, \mathcal{G})$. This shows that if $\mathcal{G}_p$ is reachable by the DPOR algorithm, then $\mathcal{G}$ is also reachable.
- In the final step, we show that walking backward from any consistent $\mathcal{G}$ we have a unique sequence of executions $\mathcal{G}_p, \mathcal{G}_{p-1}, \mathcal{G}_{p-2}, \ldots$, till we obtain the empty execution $\mathcal{G}_\emptyset$. Thus, starting from Explore$(\mathcal{P}, \mathcal{G}_\emptyset)$, we obtain $\mathcal{G}$.

Optimality. The algorithm is optimal as each full, consistent execution $\mathcal{G}$ is generated only once. Lines 23 and 15 of the Explore procedure ensure that each recursive call to Explore generates an execution that has a different rf edge or a different co edge. Also, during the DelayedRFs procedure, the swap of a read r with a write w is successful only when the deleted events respect "co_x-maximality". As argued in completeness, for every (partial) consistent execution $\mathcal{G}$, there exists a unique previous consistent execution $\mathcal{G}_p$.

If the algorithm explores $\mathcal{G}$ twice, it means that there are two different exploration sequences with respective previous executions $\mathcal{G}_p$ and $\mathcal{G}_q$. This is a contradiction as we have a unique previous execution.

Polynomial Space. The DPOR algorithm explores executions recursively in a depth-first manner, with each branch explored independently. Since the recursion depth is bounded by the size of the program, this approach ensures that the algorithm uses only polynomial space.

The proofs and related details are provided in the supplementary material [19].

4.4 Exploring the Reads-From Equivalence

For simplicity, we have focused our presentation on exploring executions that contain the co relation explicitly. However, Algorithm 1 can be easily adapted to explore executions where co is not given explicitly. This corresponds to exploring the reads-from partitioning [22], a setting that is also supported by GenMC [44]. This is often a desirable approach, because it may significantly reduce the search space: there can be exponentially many executions, differing only in their co, all of which collapse to a single class of the reads-from partitioning.

Exploring the reads-from partitioning requires that every time a new execution is explored, the algorithm performs a consistency check to derive a co, so as to guarantee that the execution is consistent. If the program has no SC accesses, this check is known to be efficient for RC11 [10,47], taking essentially linear time [79]. These results easily extend to scoped RC11, by adapting the computation of the happens-before relation so as to take the scope inclusion incl into consideration. On the other hand, the presence of SC accesses makes the problem intractable [31,62], though it remains in polynomial time with a bounded number of threads [9,31].

5 Experimental Evaluation

We implement our approach as a tool (GPU Model Checker GPUMC) capable of handling programs with scopes. GPUMC is implemented in GenMC-Trust [44], and takes scoped C/C++ programs as input and works at the LLVM IR level. Similar to existing approaches, we handle programs with loops by unrolling them by a user-specified number of times. We conduct all our experiments on an Ubuntu 22.04.1 LTS with Intel Core i7-1255U×12 and 16 GiB RAM.

We experiment with GPUMC on a wide variety of programs starting from litmus tests to larger benchmarks. We mainly compare its performance with Dartagnan [50,52], a state-of-the-art bounded model checker, which also handles programs with scope [78]. Dartagnan has recently integrated the PTX and Vulkan GPU consistency models into its test suite. Even though the consistency model considered by Dartagnan are different from SRC11, which GPUMC considers, Dartagnan is closest available tool to the kind of work we report in this paper. Two other tools that also handle programs with scopes are iGUARD [39] and Scord [40]. However, these tools do not reason about weak memory concurrency in GPUs. which makes their benchmarks not directly usable by GPUMC. In order to still experiment with them, we change their shared accesses to atomics.

Table 1. Data race detection: Evaluating on parameterized, single kernel code. Time Out (TO) = 30 min. (Time in Seconds and Memory in MB respectively). The number of events per execution is less than 120. In column Result, R denotes race detected and NR denotes no race. The * on two NR entries shows a wrong result in DARTAGNAN. In Grid column, X,Y represent X CTAs and Y threads per CTA.

			DARTAGNAN			GPUMC		
Program	Grid	Threads	Result	Time	Memory	Result	Time	Memory
caslock	4,2	8	NR	1300	494	NR	50	85
caslock1	4,2	8	R	0.7	304	R	0.1	85
caslock1	6,4	24	R	2.5	670	R	0.1	85
caslock2	4,2	8	R	0.6	270	R	0.1	85
caslock2	6,4	24	R	2.3	680	R	0.1	85
ticketlock	4,2	8	–	TO	1062	NR	320	85
ticketlock1	4,2	8	R	0.7	340	R	0.1	84
ticketlock1	6,4	24	R	965	941	R	0.1	84
ticketlock2	4,2	8	R	0.9	290	R	0.1	84
ticketlock2	6,4	24	R	1020	952	R	0.1	84
ttaslock	3,2	6	–	TO	1116	NR	500	84
ttaslock1	4,2	8	R	0.7	285	R	0.1	84
ttaslock1	6,4	24	R	3.6	321	R	0.1	84
ttaslock2	4,2	8	R	0.7	324	R	0.1	84
ttaslock2	6,4	24	R	4	917	R	0.1	84
XF-Barrier	4,3	12	NR	29	4200	NR	28	85
XF-Barrier1	4,3	12	R	4	1380	R	0.1	85
XF-Barrier1	6,4	24	NR*	190	1476	R	0.2	85
XF-Barrier2	4,3	12	R	9	1399	R	0.1	85
XF-Barrier2	6,4	24	NR*	170	1505	R	0.2	85

5.1 Comparison with DARTAGNAN

We compare the performance of GPUMC with DARTAGNAN [52] on the implementation of four synchronization primitives (caslock, ticketlock, ttaslock, and XF-Barrier), taken from [52,81]. These benchmarks use relaxed atomics, which is a very important feature of real GPU APIs. All the 1 (caslock1, ticketlock1, ttaslock1, and XF-Barrier1) and 2 (caslock2, ticketlock2, ttaslock2, and XF-Barrier2) variants are obtained by transforming the release and acquire accesses to relaxed accesses, respectively. Moreover, the XF-Barrier benchmark uses CTA-level barriers for synchronization. Table 1 shows the results of the evaluation of these applications. We parameterize these applications by increasing the number of threads in the program, the number of CTAs, and the number of threads in a CTA. For comparing with DARTAGNAN, we focus on race detection.

In Table 1 the Grid and Threads columns denote the thread organization, and the total number of threads respectively. The Result column shows the observed result – whether a race was detected (R), or whether the program was declared safe and no race was reported (NR). The Time and Memory columns show the time taken in seconds and the memory consumed in MB taken by Dartagnan and GPUMC.

We observe that in all examples except XF-Barrier, GPUMC and Dartagnan produce the same results, and GPUMC outperforms Dartagnan significantly in time and memory requirements. For the benchmarks XF-Barrier1 and XF-Barrier2 with grid structure (6,4) respectively, GPUMC successfully detects the underlying data race within a fraction of a second. The time and memory requirements we have reported for Dartagnan is with loop bound 12 as Dartagnan is unable to find the race even after unrolling to loop bound 12. On increasing the loop bound to 13, Dartagnan kills the process after showing a heap space error. In conclusion, in all the benchmarks in Table 1, GPUMC significantly outperforms Dartagnan.

5.2 Verification of GPU Applications

We evaluate GPUMC on medium to large real GPU applications, particularly for heterogeneous race and barrier divergence errors.

Heterogeneous Races. We experiment with four GPU applications – OneDimensional Convolution (1dconv), Graph Connectivity (GCON), Matrix Multiplication (matmul) and Graph Colouring (GCOL) from [39,40]. Each program has about 250 lines of code. For our experiments, we transform the accesses in these benchmarks with SC memory order and gpu scope. Finally, all these transformed benchmarks have SC accesses except GCON which has only relaxed accesses. We do not execute Dartagnan on these programs, as they are multi-kernel and involve CPU-side code, which makes it unclear how to encode them in Dartagnan.

Table 2 shows the 4 variants of each program by varying the grid structure. For instance, ldconv12 represents the version having 12 CTAs. The last two columns show the time and memory taken by GPUMC in detecting the first heterogeneous race. The detection of the first heterogeneous races in the ldconv, GCON, GCOL, and matmul benchmarks takes 4, 455, 11, and 18 executions respectively. In all cases, GPUMC detects the first race within 6 min.

Barrier Divergence. Next, we evaluate GPUMC for detecting barrier divergence, with the results shown in Table 3. We consider four GPU applications – histogram [72], XF-Barrier, arrayfire:select-matches (arrayfire-sm) and arrayfire:warp-reduce(arrayfire-wr) [80,82], as well as GkleeTests1 and GkleeTests2 kernels from the GKLEE tests [55,80]. All these benchmarks except Histogram use SC accesses and have barrier divergence. Histogram has a mix

Table 2. Heterogenous race detection using GPUMC on GPU Applications. (Time in Seconds and Memory in MB respectively). Events column represents the maximum number of events across all executions.

Program	Grid	Threads	Events	Memory	Time
1dconv12	12,4	48	1135	85	8.9
1dconv15	15,4	60	1359	85	15.2
1dconv20	20,4	80	1662	85	28.7
1dconv25	25,4	100	1937	85	47.7
GCON4	4,2	8	493	126	2
GCON5	5,2	10	563	150	5
GCON7	7,2	14	697	176	25
GCON10	10,2	20	901	250	75
GCON15	15,2	30	1241	383	295
GCOL4	4,2	8	337	85	0.5
GCOL5	5,2	10	435	85	1.7
GCOL7	7,2	14	643	86	3.6
GCOL10	10,2	20	1000	85	14.5
GCOL15	15,2	30	1051	88	18
matmul4	4,3	12	1036	85	8
matmul5	5,3	15	1054	84	9
matmul7	7,3	21	1424	85	32
matmul10	10,3	30	2556	125	360
matmul15	15,3	45	2154	90	175

of SC and relaxed accesses. In our experiments, we introduce a barrier divergence bug in the original histogram program [72, Chapter 19]. We vary the grid structures, similar to the benchmarks created for experimenting with the heterogeneous race detection.

5.3 Race Repair

Apart from detecting, GPUMC also repairs heterogeneous races as shown in Table 4 on five micro-benchmarks and three GPU applications [39,40]. The #Race column shows the number of races detected and fixed and the #Fix column shows the number of lines of code changes required to fix the detected races. In all cases, GPUMC detects and repairs all races within 3 secs. After repair, we let GPUMC exhaustively explore all executions of corrected programs (bench1, bench2, bench5, matmul finish within 10 min and bench3, bench4, GCOL and 1dconv finish within 6 h). Finally, the Executions column shows the number of executions explored on running the corrected program, and the Events column shows the maximum number of events for all explored executions post-repair.

Table 3. Barrier Divergence using GPUMC on various grid-structured programs (Time in seconds, Memory in MB). Events column represents the maximum number of events seen across executions.

Program	Grid	Threads	Events	Memory	Time
histogram4	4,2	8	144	85	0.1
histogram6	6,4	24	104	85	0.1
XF-Barrier4	4,2	8	132	85	0.1
XF-Barrier6	6,4	24	369	85	0.7
arrayfire-sm	1,16	16	1400	88	13
arrayfire-wr	1,256	256	240	85	0.3
GkleeTests1	2,32	64	700	85	2.5
GkleeTests2	1,64	64	900	86	4

Table 4. Race Repair using GPUMC on various grid-structured programs. #Race denotes the number of races detected and #Fix represents the number of changes made to fix the race. Events column represents the maximum number of events seen across executions.

Program	Grid	Threads	Executions	Events	#Race	#Fix
bench1	2,3	6	720	77	1	2
bench2	2,3	6	205236	83	2	4
bench3	8,1	8	12257280	100	2	4
bench4	4,2	8	12257280	100	2	4
bench5	5,1	5	1200	65	3	3
GCOL	2,1	2	350242	459	3	6
matmul	3,1	3	2409	1153	2	1
1dconv	2,2	4	995328	361	1	1

5.4 Scalability

Figure 4 shows the scalability of GPUMC for increasing number of threads on three benchmarks – SB (store buffer) and two GPU applications 1dconv, GCON. For SB, we create 24 programs with increasing threads from 2 to 25. For 1dconv, we create 30 programs with increasing CTAs from 1 to 30 with four threads per CTA. For GCON, we create 50 programs with increasing threads from 1 to 50. Figure 4 shows the GPUMC execution time and the memory consumed to detect the heterogeneous race for 1dconv and GCON and the assertion violation in SB; the x-axis shows the total number of threads for GCON, SB and CTAs for 1dconv, and the y-axis measures the memory in megabytes (MB) and the time in seconds. We also experiment on the LB (load buffer) benchmark in Table 5. We create 21 programs with increasing threads (LB-2 to LB-22) and exhaustively explore all consistent executions. We observe that in all benchmarks GPUMC exhaustively explores more than 4 million executions within 5500 s.

Table 5. Scalability of GPUMC on safe benchmark LB (Time in Seconds and Memory in MB). Executions column represents the executions explored.

Program	Events	Memory	Executions	Time
LB-3	36	84	7	0.3
LB-7	76	84	127	0.4
LB-12	126	84	4095	1.3
LB-18	186	101	262143	228
LB-22	226	127	4194303	5647

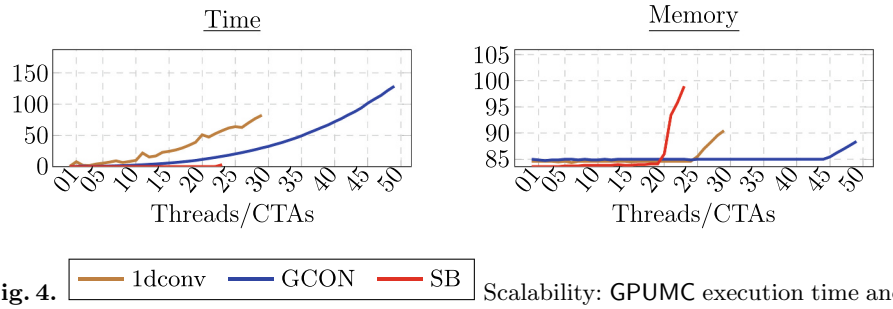


Fig. 4. Scalability: GPUMC execution time and the memory consumed to detect the heterogeneous race for 1dconv and GCON and the assertion violation in SB. The x-axis shows the total number of threads for GCON and SB, and CTAs for 1dconv. The y-axis measures the memory in megabytes (MB) and the time in seconds.

6 Related Work

Semantics. Weak memory concurrency is widely explored in programming languages for CPUs and GPUs [1,15,16,21,35,41,48,65,76], compilers [20,70], and CPU and GPU architectures [12,13,33,60,61,71]. Although GPUMC follows scoped-RC11 semantics [61], it is possible to adapt our approach to several other GPU semantic models. However, developing a DPOR model checker for GPUs with all guarantees that explore executions with $\mathsf{po} \cup \mathsf{rf}$ cycle is a nontrivial problem, in general [21,38,41,63], which is future work.

GPU Testing. Testing of GPU litmus programs is used to reason about GPU features [12,74,77], reveal errors [74], weak memory behaviors [12], and various progress properties [42,75–77]. Complementarily, our model checker explores all executions to check the correctness of the GPU weak-memory programs.

Verification and Testing of Weak Memory Concurrency. There are several DPOR algorithms for the verification of shared memory programs under weak memory such as TSO, PSO, release-acquire (RA) and RC11 [6,7,10,18,43,64,83]. DPOR algorithms have also been developed for weak consistency models such as CC, CCv and CM [11]. These are sound, complete, and optimal, although they

incur an exponential memory usage. Recently, [44,45] proposed a DPOR algorithm applicable to a range of weak memory models incurring only polynomial space while being also sound, complete and optimal. On the testing front, we have tools such as C11-tester [59], and tsan11rec [58] for several variants of C11 concurrency. However, these tools do not address the verification of programs with scopes.

GPU Analysis and Verification. Several tools propose analysis and verification of GPU programs including GPUVerify [17], G-Klee [55], GPUDrano [14], Scord [40], iGUARD [39], Simulee [80], SESA [56] for checking data races [17,27,34,39,40,57,69,84,85], divergence [17,26,27]. Other relevant GPU tools are PUG [53,54] and Faial [25]. However, these do not handle weak memory models.

7 Conclusion

We present GPUMC, a stateless model checker developed on theories of DPOR for GPU weak memory concurrency, which is sound, complete, and optimal, and uses polynomial space. GPUMC scales to several larger benchmarks and applications, detects errors, and automatically fixes them. We compare GPUMC with state-of-the-art tool Dartagnan, a bounded model checker for GPU weak memory concurrency. Our experiments on Dartagnan benchmarks reveal errors that remained unidentified by Dartagnan.

Acknowledgements. This work was partially supported by a research grant (VIL42117) from VILLUM FONDEN.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Vulkan memory model. https://github.com/KhronosGroup/Vulkan-MemoryModel
2. Cuda C++ programming guide (2024). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
3. Cuda core compute libraries (CCCL) (2024). https://github.com/NVIDIA/cccl
4. Cutlass 3.6.0 (2024). https://github.com/NVIDIA/cutlass
5. The opencl^{TM} specification (2024). https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html
6. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 353–367. Springer, Heidelberg (2015)
7. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: a foundation for optimal dynamic partial order reduction. J. ACM **64**(4), 25:1–25:49 (2017). https://doi.org/10.1145/3073408

8. Abdulla, P.A., Atig, M.F., Das, S., Jonsson, B., Sagonas, K.: Parsimonious optimal dynamic partial order reduction. In: International Conference on Computer Aided Verification, pp. 19–43. Springer (2024)
9. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. **3**(OOPSLA), 150:1–150:29 (2019). https://doi.org/10.1145/3360576
10. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. **2**(OOPSLA) (2018). https://doi.org/10.1145/3276505
11. Abdulla, P.A., Atig, M.F., Krishna, S., Gupta, A., Tuppe, O.: Optimal stateless model checking for causal consistency. In: Proceedings of TACAS 2023, vol. 13993, pp. 105–125. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_6
12. Alglave, J., et al.: GPU concurrency: weak behaviours and programming assumptions. In: PASPLOS 2015, pp. 577–591. https://doi.org/10.1145/2694344.2694391
13. Alglave, J., Deacon, W., Grisenthwaite, R., Hacquard, A., Maranget, L.: Armed cats: formal concurrency modelling at arm. ACM Trans. Program. Lang. Syst. **43**(2) (2021). https://doi.org/10.1145/3458926
14. Alur, R., Devietti, J., Navarro Leija, O.S., Singhania, N.: GPUDrano: detecting uncoalesced accesses in GPU programs. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification, pp. 507–525. Springer, Cham (2017)
15. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and OpenCL. In: POPL 2016, pp. 634–648. ACM (2016). https://doi.org/10.1145/2837614.2837637
16. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL 2011, pp. 55–66. ACM (2011). https://doi.org/10.1145/1926385.1926394
17. Betts, A., Chong, N., Donaldson, A.F., Qadeer, S., Thomson, P.: GPUverify: a verifier for GPU kernels. In: Leavens, G.T., Dwyer, M.B. (eds.) OOPSLA 2012, pp. 113–132 (2012). https://doi.org/10.1145/2384616.2384625
18. Bui, T.L., Chatterjee, K., Gautam, T., Pavlogiannis, A., Toman, V.: The reads-from equivalence for the TSO and PSO memory models. Proc. ACM Program. Lang. **5**(OOPSLA), 1–30 (2021). https://doi.org/10.1145/3485541
19. Chakraborty, S., Krishna, S., Pavlogiannis, A., Tuppe, O.: Supplementary material for GPUMC (2025). https://doi.org/10.6084/m9.figshare.29143991.v1, https://figshare.com/articles/dataset/Supplementary_material_for_GPUMC/29143991
20. Chakraborty, S., Vafeiadis, V.: Formalizing the concurrency semantics of an LLVM fragment. In: CGO 2017, pp. 100–110 (2017)
21. Chakraborty, S., Vafeiadis, V.: Grounding thin-air reads with event structures **3**(POPL) (2019). https://doi.org/10.1145/3290383
22. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Data-centric dynamic partial order reduction. Proc. ACM Program. Lang. **2**(POPL) (2017). https://doi.org/10.1145/3158119
23. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite state concurrent systems using temporal logic specifications: a practical approach. In: Proceedings of POPL 1983, pp. 117–126. ACM Press (1983)
24. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. Int. J. Softw. Tools Technol. Transf. **2**(3), 279–287 (1999). https://doi.org/10.1007/s100090050035

25. Cogumbreiro, T., Lange, J., Rong, D., Zicarelli, H.: Checking data-race freedom of GPU kernels, compositionally. In: Silva, A., Leino, K. (eds.) Computer Aided Verification, pp. 403–426. Springer, Cham (2021)
26. Coutinho, B., Sampaio, D., Pereira, F.M., Meira, W., Jr.: Profiling divergences in GPU applications. Concurrency Comput. Pract. Exp. **25**(6), 775–789 (2013)
27. Eizenberg, A., Peng, Y., Pigli, T., Mansky, W., Devietti, J.: Barracuda: binary-level analysis of runtime races in Cuda programs. In: PLDI 2017. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3062341.3062342
28. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) POPL 2005, pp. 110–121. ACM (2005). https://doi.org/10.1145/1040305.1040315
29. Francis, E.: Autonomous cars: no longer just science fiction (2014)
30. Gaster, B.R., Hower, D., Howes, L.: HRF-relaxed: adapting HRF to the complexities of industrial heterogeneous memory models. ACM Trans. Arch. Code Optim. (TACO) **12**(1), 1–26 (2015)
31. Gibbons, P.B., Korach, E.: Testing shared memories. SIAM J. Comput. **26**(4), 1208–1244 (1997). https://doi.org/10.1137/S0097539794279614
32. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60761-7
33. Goens, A., Chakraborty, S., Sarkar, S., Agarwal, S., Oswald, N., Nagarajan, V.: Compound memory models. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591267
34. Holey, A., Mekkat, V., Zhai, A.: HAccRG: hardware-accelerated data race detection in GPUs. In: 2013 42nd International Conference on Parallel Processing, pp. 60–69 (2013). https://doi.org/10.1109/ICPP.2013.15
35. Hower, D.R., et al.: Heterogeneous-race-free memory models. In: ASPLOS 2014, pp. 427–440 (2014)
36. ISO/IEC 14882: Programming language C++ (2011)
37. ISO/IEC 9899: Programming language C (2011)
38. Jeffrey, A., Riely, J.: On thin air reads towards an event structures model of relaxed memory. In: LICS 2016, pp. 759–767 (2016). https://doi.org/10.1145/2933575.2934536
39. Kamath, A.K., Basu, A.: Iguard: in-GPU advanced race detection. In: SOSP 2021, pp. 49–65 (2021). https://doi.org/10.1145/3477132.3483545
40. Kamath, A.K., George, A.A., Basu, A.: ScoRD: a scoped race detector for GPUs. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 1036–1049 (2020). https://doi.org/10.1109/ISCA45697.2020.00088
41. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL 2017, pp. 175–189 (2017). https://doi.org/10.1145/3009837.3009850
42. Ketema, J., Donaldson, A.F.: Termination analysis for GPU kernels. Sci. Comput. Program. **148**, 107–122 (2017). https://doi.org/10.1016/J.SCICO.2017.04.009
43. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. **2**(POPL), 17:1–17:32 (2018). https://doi.org/10.1145/3158105
44. Kokologiannakis, M., Marmanis, I., Gladstein, V., Vafeiadis, V.: Truly stateless, optimal dynamic partial order reduction. Proc. ACM Program. Lang. **6**(POPL), 1–28 (2022). https://doi.org/10.1145/3498711

45. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: Proceedings of PLDI 2019, pp. 96–110. ACM (2019)
46. Kokologiannakis, M., Vafeiadis, V.: Bam: efficient model checking for barriers. In: International Conference on Networked Systems, pp. 223–239. Springer (2021)
47. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: ICALP 2015, pp. 311–323 (2015). https://doi.org/10.1007/978-3-662-47666-6_25
48. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI 2017, pp. 618–632 (2017). https://doi.org/10.1145/3062341.3062352, https://plv.mpi-sws.org/scfix/full.pdf
49. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439
50. Ponce-de León, H., Haas, T., Meyer, R.: Dartagnan: SMT-based violation witness validation (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 418–423. Springer International Publishing, Cham (2022)
51. Levine, R., Cho, M., McKee, D., Quinn, A., Sorensen, T.: GPUHarbor: testing GPU memory consistency at large (experience paper). In: ISSTA 2023, pp. 779–791 (2023). https://doi.org/10.1145/3597926.3598095
52. Ponce de León, H.: Dat3m (2024). https://github.com/hernanponcedeleon/Dat3M
53. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: FSE 2010, Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 187–196. Association for Computing Machinery, New York (2010). https://doi.org/10.1145/1882291.1882320
54. Li, G., Gopalakrishnan, G.: Parameterized verification of GPU kernel programs. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pp. 2450–2459 (2012). https://doi.org/10.1109/IPDPSW.2012.302
55. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: PPOPP 2012, pp. 215–224 (2012). https://doi.org/10.1145/2145816.2145844
56. Li, P., Li, G., Gopalakrishnan, G.: Practical symbolic race checking of GPU programs. In: SC 2014: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 179–190 (2014). https://doi.org/10.1109/SC.2014.20
57. Li, P., et al.: LD: low-overhead GPU race detection without access monitoring. ACM Trans. Arch. Code Optim. (TACO) **14**(1), 1–25 (2017)
58. Lidbury, C., Donaldson, A.F.: Dynamic race detection for C++11. In: POPL 2017, pp. 443–457 (2017). https://doi.org/10.1145/3009837.3009857
59. Luo, W., Demsky, B.: C11Tester: A Race Detector for C/C++ Atomics, pp. 630–646. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3445814.3446711
60. Lustig, D., Cooksey, S., Giroux, O.: Mixed-proxy extensions for the NVIDIA PTX memory consistency model: industrial product. In: ISCA 2022 (2022). https://doi.org/10.1145/3470496.3533045
61. Lustig, D., Sahasrabuddhe, S., Giroux, O.: A formal analysis of the NVIDIA PTX memory consistency model. In: ASPLOS 2019, pp. 257–270. ACM (2019). https://doi.org/10.1145/3297858.3304043

62. Mathur, U., Pavlogiannis, A., Viswanathan, M.: The complexity of dynamic data race prediction. In: LICS 2020, pp. 713–727. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3373718.3394783
63. Moiseenko, E., Kokologiannakis, M., Vafeiadis, V.: Model checking for a multi-execution memory model. Proc. ACM Program. Lang. **6**(OOPSLA2) (2022). https://doi.org/10.1145/3563315
64. Norris, B., Demsky, B.: A practical approach for model checking C/C++11 code. ACM Trans. Program. Lang. Syst. **38**(3) (2016). https://doi.org/10.1145/2806886
65. Orr, M.S., Che, S., Yilmazer, A., Beckmann, B.M., Hill, M.D., Wood, D.A.: Synchronization using remote-scope promotion. In: ASPLOS 2015, pp. 73–86 (2015). https://doi.org/10.1145/2694344.2694350
66. Özerk, Ö., Elgezen, C., Mert, A.C., Öztürk, E., Savaş, E.: Efficient number theoretic transform implementation on GPU for homomorphic encryption. J. Supercomput. **78**(2), 2840–2872 (2022)
67. Pandey, M., et al.: The transformational role of GPU computing and deep learning in drug discovery. Nat. Mach. Intell. **4**(3), 211–221 (2022)
68. Courcoubetis, C. (ed.): CAV 1993. LNCS, vol. 697. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56922-7
69. Peng, Y., Grover, V., Devietti, J.: Curd: a dynamic Cuda race detector. In: PLDI 2018, pp. 390–403. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3192366.3192368
70. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290382
71. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. PACMPL **2**(POPL), 19:1–19:29 (2018). https://doi.org/10.1145/3158107
72. Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., Tian, X.: Data Parallel C++: Programming Accelerated Systems Using C++ and SYCL, 2 edn. Apress, Berkeley, CA (2023). https://doi.org/10.1007/978-1-4842-9691-2
73. Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., Kepner, J.: Survey and benchmarking of machine learning accelerators. In: 2019 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–9 (2019). https://doi.org/10.1109/HPEC.2019.8916327
74. Sorensen, T., Donaldson, A.F.: Exposing errors related to weak memory in GPU applications. In: Krintz, C., Berger, E.D. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13–17, 2016, pp. 100–113. ACM (2016). https://doi.org/10.1145/2908080.2908114
75. Sorensen, T., Donaldson, A.F., Batty, M., Gopalakrishnan, G., Rakamarić, Z.: Portable inter-workgroup barrier synchronisation for GPUs. In: OOPSLA 2016, Association for Computing Machinery, pp. 39–58 (2016). https://doi.org/10.1145/2983990.2984032
76. Sorensen, T., Evrard, H., Donaldson, A.F.: GPU schedulers: how fair is fair enough? In: Schewe, S., Zhang, L. (eds.) 29th International Conference on Concurrency Theory, CONCUR 2018, September 4–7, 2018, Beijing, China. LIPIcs, vol. 118, pp. 23:1–23:17 (2018). https://doi.org/10.4230/LIPICS.CONCUR.2018.23
77. Sorensen, T., Salvador, L.F., Raval, H., Evrard, H., Wickerson, J., Martonosi, M., Donaldson, A.F.: Specifying and testing GPU workgroup progress models. Proc. ACM Program. Lang. **5**(OOPSLA), 1–30 (2021). https://doi.org/10.1145/3485508

78. Tong, H., Gavrilenko, N., Ponce de Leon, H., Heljanko, K.: Towards unified analysis of GPU consistency. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 4, pp. 329–344. ASPLOS 2024, Association for Computing Machinery, New York (2025). https://doi.org/10.1145/3622781.3674174
79. Tunç, H.C., Abdulla, P.A., Chakraborty, S., Krishna, S., Mathur, U., Pavlogiannis, A.: Optimal reads-from consistency checking for C11-style memory models. Proc. ACM Program. Lang. **7**(PLDI), 137:761–137:785 (2023). https://doi.org/10.1145/3591251
80. Wu, M., Ouyang, Y., Zhou, H., Zhang, L., Liu, C., Zhang, Y.: Simulee: detecting Cuda synchronization bugs via memory-access modeling. In: ICSE 2020, pp. 937–948 (2020). https://doi.org/10.1145/3377811.3380358
81. Xiao, S., Feng, W.C.: Inter-block GPU communication via fast barrier synchronization. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–12. IEEE (2010)
82. Yalamanchili, P., et al.: ArrayFire - a high performance software library for parallel computing with an easy-to-use API (2015). https://github.com/arrayfire/arrayfire
83. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI '15, Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 250–259. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2737924.2737956
84. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: Grace: a low-overhead mechanism for detecting data races in GPU programs. SIGPLAN Not. **46**(8), 135–146 (2011). https://doi.org/10.1145/2038037.1941574
85. Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: GMRace: detecting data races in GPU programs via a low-overhead scheme. IEEE Trans. Parallel Distrib. Syst. **25**(1), 104–115 (2014). https://doi.org/10.1109/TPDS.2013.44