The Design and Implementation of a Verification Technique for GPU Kernels

ADAM BETTS, NATHAN CHONG, ALASTAIR F. DONALDSON, and JEROEN KETEMA, Imperial College London
SHAZ QADEER, Microsoft Research Redmond
PAUL THOMSON and JOHN WICKERSON, Imperial College London

We present a technique for the formal verification of GPU kernels, addressing two classes of correctness properties: data races and barrier divergence. Our approach is founded on a novel formal operational semantics for GPU kernels termed *synchronous, delayed visibility (SDV)* semantics, which captures the execution of a GPU kernel by multiple groups of threads. The SDV semantics provides operational definitions for barrier divergence and for both inter- and intra-group data races. We build on the semantics to develop a method for reducing the task of verifying a massively parallel GPU kernel to that of verifying a sequential program. This completely avoids the need to reason about thread interleavings, and allows existing techniques for sequential program verification to be leveraged. We describe an efficient encoding of data race detection and propose a method for automatically inferring the loop invariants that are required for verification. We have implemented these techniques as a practical verification tool, GPUVerify, that can be applied directly to OpenCL and CUDA source code. We evaluate GPUVerify with respect to a set of 162 kernels drawn from public and commercial sources. Our evaluation demonstrates that GPUVerify is capable of efficient, automatic verification of a large number of real-world kernels.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Theory, Verification

Additional Key Words and Phrases: Verification, GPUs, concurrency, data races, barrier synchronization

ACM Reference Format:
Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. 2015. The design and implementation of a verification technique for GPU kernels. ACM Trans. Program. Lang. Syst. 37, 3, Article 10 (May 2015), 49 pages.
DOI: http://dx.doi.org/10.1145/2743017

1. INTRODUCTION

In recent years, massively parallel *accelerator* processors, primarily graphics processing units (GPUs) from companies such as AMD, Nvidia, and ARM, have become widely available to end users. Accelerators offer significant compute power at a low cost,

This work was supported by the EU FP7 STREP project CARP (project number 287767), by EPSRC project EP/G051100/2, and by two EPSRC-funded PhD studentships. Part of the work was carried out while Alastair Donaldson was a Visiting Researcher at Microsoft Research Redmond. This article is a revised and extended version of Betts et al. [2012].
Authors' addresses: A. Betts, N. Chong, A. F. Donaldson, J. Ketema, P. Thomson, and J. Wickerson, Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, United Kingdom; emails: {a.betts, n.chong10, alastair.donaldson, j.ketema, paul.thomson11, j.wickerson}@imperial.ac.uk; S. Qadeer, Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA; email: qadeer@microsoft.com.

DOI: http://dx.doi.org/10.1145/2743017

and algorithms in domains such as medical imaging [Cates et al. 2004], computer vision [Salas-Moreno et al. 2013], computational fluid dynamics, [Harris 2004] and DNA sequence alignment [Li and Durbin 2009; Klus et al. 2012] can be accelerated to beat CPU performance, sometimes by orders of magnitude. Importantly, GPUs enable performance improvements not only in terms of runtime but also in terms of energy efficiency [Lokhmotov 2011].

GPUs present a serious challenge for software developers. A system may contain one or more of the plethora of devices on the market. Applications must exhibit *portable correctness*, operating correctly on any GPU accelerator. Software bugs in media, entertainment, and business domains can have serious financial implications, and bugs in safety critical domains, such as medical applications, can be more drastic still. Because GPUs are being used increasingly in such areas, there is an urgent need for verification techniques to aid the construction of correct GPU software.

In this work, we address the problem of static verification of GPU kernels written using mainstream programming models such as OpenCL [Khronos OpenCL Working Group 2012], CUDA [Nvidia 2012a], and C++ AMP [Microsoft Corporation 2012]. We focus on two classes of bugs that make writing correct GPU kernels harder than writing correct sequential code: *data races* and *barrier divergence*.

In contrast to the well-understood notion of data races, there does not appear to be a formal definition of barrier divergence for GPU programming. Our work gives a precise characterization of barrier divergence via an operational semantics based on predicated execution, which we call *synchronous, delayed visibility (SDV)* semantics. The SDV semantics models the execution of a kernel by multiple groups of threads, and provides operational definitions for inter- and intra-group data races as well as barrier divergence. While predicated execution has been used for code generation by GPU kernel compilers, our work is the first to use predicated operational semantics for the purpose of specification and verification.

Founded on the SDV semantics, we present a verification method which transforms a massively parallel GPU kernel into a sequential program such that correctness of the sequential program implies data race- and barrier divergence-freedom of the kernel. Our method achieves scalability by exploiting the fact that data races and barrier divergence are *pairwise* properties, allowing a reduction to an arbitrary pair of threads. For this reduction to be sound, it is necessary to use abstraction to model the possible effects of other threads. We consider two practical abstraction methods, the *adversarial* abstraction and the more refined *equality* abstraction, and prove formally that the reduction to a pair of threads is sound when combined with these abstractions.

Reducing the analysis of a highly concurrent program to the verification of a sequential program completely avoids the need to reason about thread interleavings, and enables reuse of existing verification techniques for sequential programs. We present novel heuristics for automatically inferring loop invariants required for verification.

We discuss the design and implementation of GPUVerify, a practical tool for verifying GPU kernels using our method, which can be applied directly to OpenCL and CUDA source code. GPUVerify builds on the Clang/LLVM compiler infrastructure[1], the Boogie verification system [Barnett et al. 2005] and the Z3 theorem prover [de Moura and Bjørner 2008].

We have used GPUVerify to analyze a set of 162 OpenCL and CUDA kernels. We summarize an experiment with the initial implementation of GPUVerify (which was limited to intragroup race checking [Betts et al. 2012]) where we divided these kernels into separate *training* and *evaluation* sets, tuned the capabilities of the tool to the training set, and then ran the tool *blindly* on the evaluation set (none of our team

[1]http://clang.llvm.org/.

were familiar with the kernels in the evaluation set). We found that fully automatic verification was achieved for 49 out of 70 kernels from the evaluation set (69%). We also present a comparison of the initial GPUVerify implementation with PUG, a formal analysis tool for CUDA kernels by Li and Gopalakrishnan [2010]. GPUVerify performs competitively with PUG when verifying correct kernels and rejects defective kernels in several cases where PUG reports false negatives. Additionally, the equality abstraction offered by GPUVerify is more refined than the shared state abstraction supported by PUG, allowing verification of real-world kernels for which PUG reports false positives.

We also report on a more recent version of GPUVerify, which supports intergroup race checking, to analyze our benchmarks, using a combination of improved invariant inference and manual invariant annotations to achieve verification across the entire benchmark set. We report on the performance of GPUVerify, assess the overhead associated with performing intergroup race checking on top of intragroup race checking, and compare the performance and precision of GPUVerify when the adversarial versus equality abstractions are used.

In summary, our article makes the following contributions:

—*Synchronous, delayed visibility* (SDV) semantics: a formal operational semantics for GPU kernels based on predicated execution, inter- and intragroup data-race freedom, and barrier divergence freedom.
—The *two-thread reduction*: an abstract version of the SDV semantics that models a pair of threads, and a soundness theorem showing that data race- and barrier divergence-freedom of a massively parallel kernel according to the SDV semantics can be established by proving data race- and barrier divergence-freedom for each pair of threads using the abstract semantics.
—A verification method for GPU kernels which exploits the SDV semantics and two-thread reduction to transform analysis to a sequential program verification task, allowing reuse of existing analysis techniques based on verification condition generation and automated theorem proving.
—A method for automatically inferring the invariants needed for our verification method.
—An extensive evaluation of our verifier on a collection of 162 publicly available and commercial GPU kernels.

Contributions in Relation to Prior Work

We make explicit how the current article significantly extends a previous conference version [Betts et al. 2012] and complements other papers that have arisen from the GPUVerify project [Collingbourne et al. 2013; Chong et al. 2013; Bardsley and Donaldson 2014; Bardsley et al. 2014a; Chong et al. 2014].

Treatment of Multigroup GPU Kernels. The original presentation of our verification method was restricted to the analysis of single-group GPU kernels. Here we treat the multigroup case by extending the formal semantics for kernels, implementing intergroup race checking in the GPUVerify tool, and conducting an experimental evaluation to assess the overhead associated with handling multiple groups versus just a single group.

A Full Soundness Proof for the Two-Thread Reduction. The scalability of our verification method hinges on reducing the analysis to consider an arbitrary pair of threads. Only a short proof sketch for the soundness of this "two-thread reduction" was presented in our prior work. In the current article, we present, for the first time, a detailed formal soundness proof. This is an important contribution because a similar reduction is also employed by other methods, such as the PUG verifier of Li and Gopalakrishnan [2010].

Mechanical Checking of the Language and Semantics Definitions. We have used the Isabelle proof assistant to type check all the definitions, semantic rules, and theorem statements appearing in the article. The result is published in the*Archive of Formal Proofs* [Wickerson 2014]. We have also conducted a partial mechanization of the soundness proof for the two-thread reduction.

New Experimental Evaluation. Rather than repeating the extensive experimental evaluation presented in our prior work [Betts et al. 2012], we instead summarize the results of this evaluation and present a new evaluation in which we assess the verification performance of a more recent version of the GPUVerify tool, as well as evaluating the overhead associated with intergroup race checking and performance characteristics of two-shared state abstractions used by the tool.

Relation to Other Papers Arising from the GPUVerify Project. In addition to the original version of the article [Betts et al. 2012], we have published three technical articles that extend the GPUVerify verification method, each in a distinct manner [Collingbourne et al. 2013; Chong et al. 2013; Bardsley and Donaldson 2014]. Each of these extensions is separate from—and complementary to—the new contributions made in the current article.

Following our prior work [Betts et al. 2012], the results presented in this article apply to structured GPU kernels expressed using **if** and **while** constructs. In related work, we have generalized the lock-step execution semantics used by our method to apply to unstructured control flow graphs, allowing the technique to be applied at the level of the intermediate representation used by a compiler [Collingbourne et al. 2013].

We have proposed *barrier invariants* [Chong et al. 2013] as a mechanism to allow users to improve on the precision offered by the *adversarial* and *equality* abstractions used in this work. The shared state abstractions discussed here are straightforward to apply automatically, but are inflexible. Barrier invariants are instead provided by a user; they offer a tunable, but heavyweight and manually driven, approach to GPU kernel verification.

These works, like the current contribution, are restricted to a barrier synchronizing programming model. We have conducted a preliminary investigation into lifting the method beyond this model to cater for kernels in which threads communicate using atomic operations and warp-based synchronization [Bardsley and Donaldson 2014].

As well as technical articles that extend the core verification method, a recent tool paper discusses engineering experience and insight gained during the GPUVerify project [Bardsley et al. 2014a], and an invited article provides a tutorial overview of the technique [Donaldson 2014]. We have also applied GPUVerify as a component in an automatic verification method for parallel prefix sum kernels [Chong et al. 2014].

Reproducibility

The GPUVerify tool is open source and available online at http://multicore.doc.ic.ac.uk/tools/GPUVerify.

On the tool web page we make available the versions of GPUVerify used for the experiments reported in this paper, together with the non-commercial benchmarks used for our evaluation.

2. GPU KERNEL PROGRAMMING

A typical GPU (see Figure 1) consists of a large number of simple *processing elements (PEs)*, sometimes referred to as *cores*. Subsets of the PEs are grouped together into *multiprocessors*, such that all PEs within a multiprocessor execute in lock-step, in single instruction multiple data (SIMD) fashion. Distinct multiprocessors on a GPU

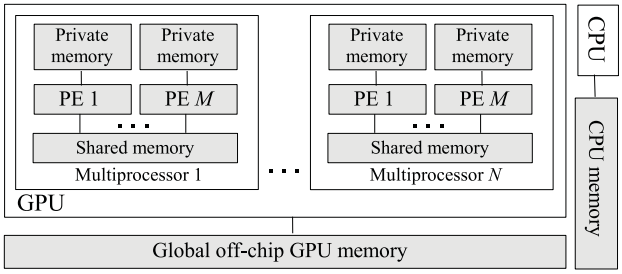


Fig. 1. Schematic overview of a typical GPU architecture.

Table I. Equivalent Terms for What We Shall Call *private*, *shared,* and *global* Memory in CUDA, OpenCL, and C++ AMP

Term	CUDA	OpenCL	C++ AMP
private	local	private	local
shared	shared	local	tile static
global	global	global	global

can execute independently. Each PE is equipped with a small private memory, and PEs located on the same multiprocessor can access a portion of *shared* memory dedicated to that multiprocessor. All PEs on the GPU have access to a large amount of off-chip memory known as *global* memory, which is usually separate from main CPU memory.

Today, there are three major GPU programming models: OpenCL, an industry standard proposed by the Khronos OpenCL Working Group [2012] and widely supported (in particular, OpenCL is AMD's primary high-level GPU programming model); CUDA, from Nvidia [2012a]; and C++ AMP, from Microsoft [2012]. Table I summarizes the terms used in the programming models to refer to *private*, *shared,* and *global* memory.

Threads and Groups. All three programming models provide a similar high-level abstraction for mapping computation across GPU hardware, centered around the notion of a *kernel* program being executed by many parallel *threads*, together with a specification of how these threads should be partitioned into *groups*. The kernel is a template specifying the behavior of an arbitrary thread, parameterized by thread and group ID variables. Expressions over these IDs allow distinct threads to operate on separate data and follow differing execution paths through the kernel. Threads in the same group can synchronize during kernel execution, while threads in distinct groups execute completely independently.

The runtime environment associated with a GPU programming model must interface with the driver of the available GPU to schedule execution of kernel threads across processing elements. Typically each group of threads is assigned to one of the multiprocessors of the GPU, so that distinct groups execute in parallel on different multiprocessors. If the number of threads in a group is N and the number of PEs in a multiprocessor is M, then a group is divided into $\lceil \frac{N}{M} \rceil$ *subgroups*, each consisting of up to M threads. Execution of a single group on a multiprocessor then proceeds by scheduling the subgroups in an interleaving manner. Each thread in a given subgroup is pinned to a distinct PE, and all threads in the same subgroup execute together in lock-step, following exactly the same control path. Distinct subgroups may follow different control paths.

Table II summarizes the specific terms used by the three main GPU programming models to refer to *threads*, *groups,* and (in the case of CUDA), *subgroups*. OpenCL and C++ AMP aim for portability across GPUs from multiple vendors, so they do not allow

Table II. Equivalent Terms for *thread*, *group,* and (Where Applicable) *subgroup* in CUDA, OpenCL, and C++ AMP

Term	CUDA	OpenCL	C++ AMP
thread	thread	work-item	thread
group	thread block	work-group	tile
sub-group	warp	N/A	N/A

Table III. Predicated forms for Conditionals and Loops

Program fragment	Predicated form

`if(lid > N)`
`  x = 0;`
`else`
`  x = 1;`
	`p = (lid > N);`
	`p ⇒ x = 0;`
	`!p ⇒ x = 1;`

`while (i < x) {`
`  i++;`
`}`
	`p = (i < x);`
	`while (∃ t :: t.p) {`
	`  p ⇒ i++;`
	`  p ⇒ p = (i < x);`
	`}`

a kernel to query the device-specific size or structure of thread subgroups.[2] As CUDA is Nvidia-specific, CUDA programmers *can* write kernels that make assumptions about the division of threads into subgroups. However, such kernels will not easily port to general-purpose GPU programming languages and may break when executed on future generations of Nvidia hardware using a different subgroup size. Thus, GPU kernels that do not make assumptions about the size of thread subgroups are preferable.

Predicated Execution. Recall that the PEs in a GPU multiprocessor execute in lock-step, such as a SIMD processor array. Threads within a subgroup occupy the PEs of a multiprocessor, and thus must also execute in lock-step. Conditional statements and loops through which distinct threads in the same subgroup should take different paths must, therefore, be simulated, and this is achieved using *predicated execution*.

Consider the conditional statement in the top left of Table III, where `lid` denotes the local ID of a thread within its group and `x` is a local variable stored in private memory. This conditional can be transformed into the straight-line code shown in the top right of the figure, which can be executed by a subgroup in lock-step. The meaning of a statement *predicate* ⇒ *command* is that a thread should execute *command* if *predicate* holds for that thread; otherwise, the thread should execute a no-op. All threads evaluate the condition `lid > N` into a local Boolean variable `p`, then execute both the *then* and *else* branches of the conditional, predicated by `p` and `!p`, respectively.

Loops are turned into predicated form by dictating that all threads in a subgroup continue to execute the loop body until the loop condition is false for *all* threads in the subgroup, with threads for whom the condition does not hold becoming disabled. This is illustrated for the loop in the bottom left of Table III (where `i` and `x` are local variables) by the code fragment shown in the bottom right of the figure. First, the condition `i < x` is evaluated into local variable `p`. Then the subgroup loops while `p` remains true for some thread in the subgroup, indicated by `∃ t : : t.p`. The loop body is predicated by `p`, and thus has an effect only for enabled threads.

We present precise operational semantics for predicated execution in Section 3.

Barrier Synchronization. When a thread t_1 writes to an address in shared or global memory, the result of this write is not guaranteed to become visible to another thread

[2]The recent OpenCL 2.0 extension specification [Khronos OpenCL Working Group 2014b, Section 9.17, p. 133] contains an optional extension to support subgroups of threads.

t_2 unless t_1 and t_2 *synchronize*. As noted earlier, there is *no* mechanism for threads in distinct groups to synchronize during kernel execution.[3] Threads in the same group can synchronize via *barriers*. Intuitively, a kernel thread belonging to group g waits at a *barrier* statement until every thread in g has reached the barrier. Passing the barrier guarantees that all writes to shared and global memory by threads in g occurring before execution of the barrier have been committed.

Our analysis through writing GPU kernels and talking to GPU developers is that there are two specific classes of bugs that make writing correct GPU kernels more difficult than writing correct sequential code: *data races* and *barrier divergence*.

2.1. Data Races

We distinguish between two kinds of data races in GPU kernels. An *intergroup data race* occurs if there are two threads t_1 and t_2 from *different* groups such that t_1 writes to a location in global memory and t_2 writes to or reads from this location. An *intragroup data race* occurs if there are two threads t_1 and t_2 from the same group such that t_1 writes to a location in global or shared memory, t_2 writes to or reads from this location, and no *barrier* statement is executed between these accesses. Races can lead to nondeterministic kernel behavior and computation of incorrect results.

2.2. Barrier Divergence

If threads in the same group *diverge*, reaching different barriers as in the following kernel fragment:

```
if ((lid%2) == 0)
  barrier();      //Even threads hit first barrier
else
  barrier();      //Odd threads hit second barrier
```

then kernel behavior is undefined. According to CUDA [Nvidia 2012a]:

"execution is likely to hang or produce unintended side effects."

While there is clarity across all programming models as to the meaning of barrier divergence in loop-free code, the situation is far from clear for code with loops. Consider the example kernel shown on the left of Figure 2. This kernel is intended to be executed by one group of four threads, and declares an array `A` of two shared buffers, each of size four. Private variable `buf` is an index into `A`, representing the *current* buffer. The threads execute a nest of loops. On each inner loop iteration, a thread reads the value of the current buffer at index `lid`+1 modulo 4 and writes the result into the noncurrent buffer at index `lid`. A barrier is used to avoid data races on `A`. Notice that local variables `x` and `y` are set to 4 and 1, respectively, for thread 0, and to 1 and 4, respectively, for all other threads. As a result, we expect thread 0 to perform four outer loop iterations, each involving one inner loop iteration, while other threads will perform a single outer loop iteration, consisting of four inner loop iterations.

According to the guidance in the CUDA documentation, such a kernel appears to be valid: all threads will hit the barrier statement four times. Taking a snapshot of the array A at each barrier and at the end of the kernel, we might expect to see the

[3]In OpenCL 2.0, limited forms of interworkgroup synchronization can be implemented using atomic operations [Khronos OpenCL Working Group 2014a]. However, an interworkgroup barrier still cannot be reliably implemented due to lack of progress guarantees between groups.

```
shared int A[2][4];

void kernel() {
  int buf, x, y, i, j;
  x = (lid == 0 ? 4 : 1);
  y = (lid == 0 ? 1 : 4);
  buf = i = 0;
  while (i < x) {
    j = 0;
    while (j < y) {
      barrier();
      A[1-buf][lid]
        = A[buf][(lid+1)%4];
      buf = 1 - buf;
      j++;
    }
    i++;
  }
}
```

```
...
p = (i < x);
while (∃ t::t.p) {
  p ⇒ j = 0;
  q = p && (j < y);
  while (∃ t::t.q) {
    q ⇒ barrier();
    q ⇒ A[1-buf][lid]
          = A[buf][(lid+1)%4];
    q ⇒ buf = 1 - buf;
    q ⇒ j++;
    q ⇒ q = p && (j < y);
  }
  p ⇒ i++;
  p ⇒ p = (i < x);
}
```

Fig. 2. Illustration of the subtleties of barriers in nested loops.

following:

$$A = \{\{0,1,2,3\},\{-,-,-,-\}\} \rightarrow \{\{0,1,2,3\},\{1,2,3,0\}\} \rightarrow \{\{2,3,0,1\},\{1,2,3,0\}\} \rightarrow \{\{2,3,0,1\},\{3,0,1,2\}\} \rightarrow \{\{0,1,2,3\},\{3,0,1,2\}\},$$

under the assumption that, during each inner loop iteration, threads 1, 2, and 3 wait at the barrier until thread 0 reaches the barrier again by proceeding to the next iteration of the outer loop.

However, consider the predicated version of the kernel shown in part on the right of Figure 2. This is the form in which the kernel executes on an Nvidia GPU. The four threads comprise a single subgroup. All threads will enter the outer loop and execute the first inner loop iteration. Then thread 0 will become disabled (`q` becomes *false*) for the inner loop. Thus, the barrier will be executed with some, but not all, threads in the subgroup enabled. On Nvidia hardware, a barrier is compiled to a `bar.sync` instruction in the PTX (Parallel Thread Execution) assembly language. According to the PTX documentation [Nvidia 2012b],

> *"if any thread in a* [*subgroup*] *executes a* `bar` *instruction, it is as if all the threads in the* [*subgroup*] *have executed the* `bar` *instruction."*

Thus, threads 1, 2, and 3 will not wait at the barrier until thread 0 returns to the inner loop: they will simply continue to execute past the barrier, performing three more inner loop iterations. In effect, thread 0 is disabled, while threads 1, 2, and 3 perform these additional loop iterations, because in practice the threads execute in lock step. This yields the following sequence of state-changes to A:

$$A = \{\{0,1,2,3\},\{-,-,-,-\}\} \rightarrow \{\{0,1,2,3\},\{1,2,3,0\}\} \rightarrow \{\{0,3,0,1\},\{1,2,3,0\}\} \rightarrow \{\{0,3,0,1\},\{1,0,1,0\}\} \rightarrow \{\{0,1,0,1\},\{1,0,1,0\}\}.$$

Table IV. The Litmus Test of Figure 2 Yields a Range of Results across Varying Platforms

Architecture	Final state of A
Nvidia Tesla C2050	{{0, 1, 0, 1}, {1, 0, 1, 0}}
AMD Tahiti	{{0, 1, 2, 3}, {1, 2, 3, 0}}
ARM Mali-T600	{{0, 1, 2, 3}, {3, 0, 1, 2}}
Intel Xeon X5650	{{∗, ∗, ∗, 1}, {3, 0, 1, 2}}

After the inner loop exits, thread 0 becomes enabled, but all other threads become disabled, for a further three outer loop iterations, during each of which thread 0 executes a single inner loop iteration. The state of A thus remains {{0, 1, 0, 1}, {1, 0, 1, 0}}.

The OpenCL standard [Khronos OpenCL Working Group 2012] gives a better, though still informal definition, stating:

> *"If barrier is inside a loop, all* [*threads*] *must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier,"*

which at least can be interpreted as rejecting the example of Figure 2.

To investigate this issue in practice, we implemented the litmus test of Figure 2 in both CUDA and OpenCL and (with help from contacts in the GPU industry; see Acknowledgments) ran the test on GPU architectures from Nvidia, AMD and ARM, and on an Intel Xeon CPU (for which there is an OpenCL implementation). Our findings are reported in Table IV. Observe that the test result does not agree between any two vendors. The Nvidia results match our earlier prediction. The AMD result also appears to stem from predicated execution. ARM's Mali architecture does *not* use predicated execution [Lokhmotov 2011], so perhaps unsurprisingly gives the 'intuitive' result we might expect. For Intel Xeon, we found that different threads reported different values for certain array elements in the final shared state, indicated by asterisks in Table IV, which we attribute to cache effects.

The example of Figure 2 is contrived to be small enough to explain concisely and examine exhaustively. It does, however, illustrate that barrier divergence is a subtle issue, and that nonobvious misuse of barriers can compromise correctness and lead to implementation-dependent results. Clearly, a more rigorous notion of barrier divergence is required than the informal descriptions found in the CUDA and OpenCL documentation.

We give a precise, operational definition for barrier divergence in Section 3 which clears up this ambiguity. In essence, our definition states that if a barrier is encountered by a group of threads executing in lock-step under a predicate, the predicate must hold *uniformly* across the group, i.e., the predicate must be *true* for all threads, or *false* for all threads. This precise definition facilitates formal verification of barrier divergence-freedom and is incorporated in the GPUVerify tool we present in Section 5. GPUVerify can accurately analyze the kernel of Figure 2, flagging the barrier divergence issue.

3. OPERATIONAL SEMANTICS FOR GPU KERNELS

Our aim is to verify data race- and barrier divergence-freedom for GPU kernels. ITo do this, we require an operational semantics for GPU kernels that specifies precisely the conditions under which data races and barrier divergence occur.

For checking barrier divergence, the least conservative assumption we can safely make is that a thread group consists of a *single* subgroup, so that all threads in a group execute in lock-step. Thus, when analyzing barrier divergence-freedom for a kernel in which each thread group consists of N threads, we assume for the purposes of analysis that the subgroup size is also N (so that each thread group trivially contains just one subgroup), regardless of the subgroup size associated with any particular architecture.

We can then define barrier divergence to occur if and only if the lock-step thread group executes a barrier and the threads are not uniformly enabled: the current predicate of execution holds for some threads but not others. Clearly, if we can prove barrier divergence-freedom for a kernel under this strong assumption of lock-step behavior, the kernel will be free from barrier divergence if thread groups are divided into subgroups at a finer level of granularity.

For example, when analyzing barrier divergence for a kernel in which thread blocks have size 1,024, we assume that the threads execute in lock-step as a single subgroup size of 1,024. Recent GPUs from AMD, Nvidia, and ARM exhibit subgroup sizes of 64, 32, and 1, respectively [AMD 2013; Nvidia 2012a; Lokhmotov 2011]. Our worst-case assumption thus caters for the AMD scenario with the largest subgroup size, as well as the smaller subgroup sizes exhibited by Nvidia and ARM GPUs. Furthermore, the conservative assumption also caters for possible subgroup sizes associated with any other (present or future) architectures.

For race checking, the scenario is reversed: the least conservative safe assumption is that threads in the same group interleave completely asynchronously between pairs of barriers, with no guarantees as to the relative order of statement execution between threads (so that essentially every subgroup consists of just a single thread). This is the case, for example, on ARM's Mali GPU architecture [Lokhmotov 2011]. Clearly, if race-freedom can be proved under this most general condition, then a kernel will remain race-free if, in practice, certain threads in a group execute synchronously.

We propose a semantics that we call *synchronous, delayed visibility (SDV)*. Under SDV, group execution is *synchronous*, allowing precise barrier divergence checking. The shared memory accesses of all threads are logged, and when a thread writes to shared memory, the visibility of this write to other threads in the group is *delayed* until the group reaches a barrier. Delaying the visibility of writes ensures that threads do not see a synchronized view of shared and global memory between barriers, catering for the fact that execution might not *really* be fully synchronous. Logging accessed locations allows racing accesses to be detected. The SDV semantics does *not* attempt to directly capture the manner in which threads execute on real GPU architectures which exhibit a mixture of synchronous and interleaving execution in practice; instead, the semantics provides a basis for proving data race- and barrier divergence-freedom.

To describe the SDV semantics formally, we define the *Kernel Programming Language (KPL)*, which captures the essential features of mainstream languages for writing GPU kernels such as CUDA and OpenCL.

3.1. Syntax

The syntax for KPL is shown in Figure 3. A KPL kernel declares the number of groups (**groups**: number) and the number of threads in each group (**threads**: number) that will execute the kernel. Thus, the total number of threads that will execute the kernel is equal to the number of declared groups multiplied by the number of declared threads. The group and thread declarations are followed by a list of procedure declarations and a **main** statement. Each procedure has a name, a single parameter, and a body; for brevity, we do not model multiple parameters or return values.

For simplicity, but without loss of generality, threads have access to a single shared array *sh*, which we refer to as *global memory*. We make no distinction between shared and global memory (cf. Figure 1) in our programming language; from a verification perspective, it suffices to view shared memory as being part of global memory (by employing the IDs of groups in the calculation of memory addresses). We assume that every local variable and each indexable element of global memory has type Word, the type of memory words. Any value of type Word can be interpreted as an integer or a

```
kernel      ::=  groups: number
                 threads: number
                 proc*
                 main: stmt
proc        ::=  procedure name var stmt
stmt        ::=  basic_stmt | stmt; stmt
               |  local name stmt
               |  if local_expr stmt else stmt
               |  while local_expr stmt
               |  while local_expr stmt
               |  call name(local_expr)
               |  barrier
               |  break | continue | return
basic_stmt  ::=  loc := local_expr
               |  loc := sh[local_expr]
               |  sh[local_expr] := local_expr
local_expr  ::=  lid | gid | loc
               |  constant literal of type Word
               |  local_expr op local_expr
loc         ::=  name | V
name        ::=  any valid C name
number      ::=  any positive number
```

Fig. 3. Syntax for Kernel Programming Language.

Boolean. In practice, Word will also represent floating point numbers; structured data will be represented by sequences of values of type Word.

Local expressions are built from constant literals, locations, and the built-in variables *lid* and *gid*, which refer to the local ID of a thread within its group and the ID of the group in which the thread occurs, respectively. Compound expressions are build using an unspecified set of binary operators; the syntax could easily be extended to accommodate operators of other arities.

A thread may update one of its local variables by performing a local computation ($v := e$, where e is a local expression) or by reading from global memory ($v := sh[e]$, where e is a local expression determining which index to read from). A thread may also update global memory ($sh[e_1] := e_2$, where e_1, e_2 are expressions over local variables, with e_1 determining which index to write to, and e_2 the value to be written). For simplicity, we assume that all local variables are scalar.

Compound statements are constructed via sequencing, conditional branches, local variable introduction, loops, and procedure calls in the standard way. KPL provides a number of other statements: **barrier**, which is used to synchronize threads in a group; **break**, which causes execution to break out of the closest enclosing loop; **continue**, which causes execution to jump to the head of the closest enclosing loop; and **return**, which causes execution to return from the closest enclosing procedure call.

Figure 3 specifies two syntactic elements that should *not* appear directly in a KPL program; they are used in the semantic rules of Figure 6 which we will explain in Section 3.2. These are: a special **while** statement, used to model the dynamic semantics of while loops in which we have to distinguish between the first and subsequent iterations of a loop, and a set V of locations from which storage for local variables is allocated as they come into scope dynamically.

We assume that programs are well formed according to the usual rules, e.g., statements should only refer to declared variables, and variable introduction should not

hide a variable introduced earlier in an enclosing scope. In addition, we require the main statement not to contain a **return**; this nonconsequential restriction simplifies our presentation somewhat (it avoids having to apply the function *relim* of Figure 6, discussed in Section 3.2, to the main statement).

We do not formalize features of GPU kernels such as multidimensional groups and arrays. However, our verification method and implementation, described in Section 5, handles both.

3.2. Semantics

Given a function $f : A \to B$ and elements $a \in A, b \in B$, we write $f[a \mapsto b]$ to denote the function $g : A \to B$ such that $g(x) = f(x)$ for all $x \in A \setminus \{a\}$, and $g(a) = b$ otherwise. We abbreviate $f[a \mapsto b][c \mapsto d]$ with $a \neq c$ to $f[a \mapsto b, c \mapsto d]$. Viewing a tuple with named components as a function mapping names to element values, we also use this notation to specify updates to tuples. We use $\langle s_1, s_2, \ldots, s_k \rangle$ to denote a sequence of length k, and we write $\langle\rangle$ for the empty sequence. We write $s : ss$ for a sequence whose first element is s and whose remaining elements form the sequence ss; the operator ":" associates to the right.

In our semantics, each thread is equipped with a *shadow* copy of global memory. At the start of kernel execution, the shadow memory of all threads is identical. During execution, a thread reads and modifies its shadow memory locally, and maintains a read set and a write set recording those addresses in global memory that the thread has accessed. During each access, the read and write sets are checked for data races. If a race has occurred, execution aborts. When a **barrier** statement is reached with all threads in a group enabled, the write sets are used to build a consistent view of global memory between the threads in a group, the shadow memories are all updated to agree with this view, and the read and write sets are cleared with regard to the accesses of threads within groups.

In what follows, let P be a KPL kernel.

3.2.1. Thread, Group, and Kernel States. Let gs denote the number of groups executing P, specified via **groups**: gs in the definition of P. Similarly, let ts denote the number of threads in each group, specified via **threads**: ts. Let $\mathcal{G}$ denote the set $\{0, 1, \ldots, gs - 1\}$, and for each $i \in \mathcal{G}$, let $\mathcal{T}_i$ denote the set $\{0, 1, \ldots, ts - 1\}$.[4] Together, $\mathcal{G}$ and $\mathcal{T}$ specify the identifiers of all the threads running the kernel, via the function

$$tids(\mathcal{G}, \mathcal{T}) \triangleq \{(i, j) \mid i \in \mathcal{G} \wedge j \in \mathcal{T}_i\}.$$

We now define thread, group. and kernel states.

Thread States. A *thread state* is a tuple (l, sh, R, W) where:

—$l : V \to \mathsf{Word}$ is the local variable store of the thread (recall that V is a set of locations); included are locations *gid* and *lid* specifying, respectively, the ID of the group to which the thread belongs, and the ID of the thread within this group.
—$sh : \mathbb{N} \to \mathsf{Word}$ is the shadow copy of global memory owned by the thread.
—$R, W \subseteq \mathbb{N}$ are the read and write sets of the thread recording the global addresses the thread has accessed since the last barrier; these sets are used to detect intragroup data races.

We use τ to denote a thread state, and $\tau.l$, $\tau.sh$, $\tau.R$, $\tau.W$ to refer to the components of τ. The set of all thread states is denoted by $\mathsf{ThreadStates}$. Given a local expression e and a thread state τ, we write e^τ for the result of evaluating e according to $\tau.l$. We do not

[4]Observe that it is not strictly necessary to parameterize $\mathcal{T}$ in i, as $\mathcal{T}_i = \{0, 1, \ldots, ts-1\}$ for all $i \in \mathcal{G}$. However, the parameterization will simplify the presentation of the kernel abstractions presented later in the text.

provide a concrete definition of e^τ, which depends on the nature of the base type Word and the available operators, except we specify that, for a storage location v, $v^\tau = \tau.l(v)$.

Group States. A *group state* of a group $i \in \mathcal{G}$ is a tuple (γ_{ts}, R, W) where:

—$\gamma_{ts} : \mathcal{T}_i \to$ ThreadStates records a thread state for each thread in group i.
—$R, W \subseteq \mathbb{N} \times \mathcal{T}_i$ are the read and write sets of group i. These are used to detect *intergroup* data races. Each element (x, j) records a global address x accessed by thread j in the group since the start of the kernel. It is necessary to store the local ID j so that when we later reduce executions to just two threads, accesses by threads other than the two selected ones can be identified and removed from R and W.

We use γ (without a subscript) to denote a group state. Given $\gamma = (\gamma_{ts}, R, W)$, we use $\gamma(j)$ to refer to $\gamma_{ts}(j)$ and we refer to R and W by $\gamma.R$ and $\gamma.W$, respectively. A group state γ of a group i is *valid* if $\gamma(j).l(gid) = i$ and $\gamma(j).l(lid) = j$ for all $j \in \mathcal{T}_i$, that is, knowledge of the group id i is consistent across the threads in the group and each thread has a unique local id corresponding to its location in the group.

The set of all valid group states for a group i is denoted GroupStates$_i$, and we use GroupStates to denote $\bigcup_{i \in \mathcal{G}}$ GroupStates$_i$, the set of all valid group states.

Kernel States. A *predicated statement* is a pair (s, p), with $s \in$ stmt and $p \in$ local_expr. Intuitively, (s, p) denotes a statement s that should be executed if p holds, and otherwise should have no effect. The set of all predicated statements is denoted PredStmts.

A *kernel state* is a tuple (κ, ss) where:

—$\kappa : \mathcal{G} \to$ GroupStates records a group state for each group in the kernel.
—$ss \in$ PredStmts* is a sequence of predicated statements to be executed by the kernel.

A kernel state (κ, ss) is *valid* if, for each $i \in \mathcal{G}$, $\kappa(i) \in$ GroupStates$_i$, that is, each group has a unique id. The set of all valid kernel states is denoted KernelStates.

Observe that a kernel state does not include a single, definitive global memory component: global memory is represented via the shadow copies held by the individual threads in groups, which are initially consistent, and are made consistent again at barriers for threads within a group.

A kernel state $(\kappa, ss) \in$ KernelStates is a valid *initial state* of P if:

—$ss = \langle (s, true) \rangle$, where s is declared in P via **main** : s.
—$\kappa(i).R = \kappa(i).W = \emptyset$ for all $i \in \mathcal{G}$.
—$\kappa(i)(j).R = \kappa(i)(j).W = \emptyset$ for all $i \in \mathcal{G}$ and $j \in \mathcal{T}_i$.
—$\kappa(i)(j).sh = \kappa(i')(j').sh$ for all $i, i' \in \mathcal{G}$ and $j \in \mathcal{T}_i$, $j' \in \mathcal{T}_{i'}$.
—$\kappa(i)(j).l(v) = false$ for all $i \in \mathcal{G}$, $j \in \mathcal{T}_i$ and $v \in V$.

The first requirement ensures that the kernel executes the **main** statement. The second and third requirement guarantee that the read and write sets of both groups and threads are initially empty. The fourth requirement ensures that threads have a consistent but arbitrary initial view of global memory. We single out the final requirement as it impacts on our semantics in a somewhat subtle manner:

Remark 3.1 (*Local Storage Locations are Initially false*). The fifth requirement ensures that local storage locations are initialized to the value of type Word corresponding to *false*. Thus, whenever fresh variables are introduced by the semantic rules of Figure 6, these variables initially store the value *false*. This requirement simplifies our presentation of **break** and **continue** in the rules of Figure 6.

3.2.2. Predicated Execution. We now describe predicated execution of threads, groups, and kernels.

$$\frac{\neg p^{\tau}}{(\tau, \mathsf{basic\_stmt}, p) \rightarrow_t \tau}\ \text{(T-DISABLED)} \qquad \frac{p^{\tau} \qquad l' = \tau.l[v \mapsto e^{\tau}]}{(\tau, v := e, p) \rightarrow_t \tau[l \mapsto l']}\ \text{(T-ASSIGN)}$$

$$\frac{p^{\tau} \qquad l' = \tau.l[v \mapsto \tau.sh(e^{\tau})] \qquad R' = \tau.R \cup \{e^{\tau}\}}{(\tau, v := sh[e], p) \rightarrow_t \tau[l \mapsto l', R \mapsto R']}\ \text{(T-READ)}$$

$$\frac{p^{\tau} \qquad sh' = \tau.sh[e_1^{\tau} \mapsto e_2^{\tau}] \qquad W' = \tau.W \cup \{e_1^{\tau}\}}{(\tau, sh[e_1] := e_2, p) \rightarrow_t \tau[sh \mapsto sh', W \mapsto W']}\ \text{(T-WRITE)}$$

Fig. 4. Rules for predicated execution of basic statements by threads.

$$\frac{\forall\, j \in \mathcal{T}_i\ .\ (\gamma(j), \mathsf{basic\_stmt}, p) \rightarrow_t \gamma'(j) \qquad \mathsf{group\_race}_i(\gamma')}{(\gamma, \mathsf{basic\_stmt}, p) \rightarrow_{g(i)} \mathit{error}}\ \text{(G-RACE)}$$

$$\frac{\begin{array}{c}\forall\, j \in \mathcal{T}_i\ .\ (\gamma(j), \mathsf{basic\_stmt}, p) \rightarrow_t \gamma'(j) \qquad \neg\mathsf{group\_race}_i(\gamma') \\ \gamma'.R = \gamma.R \cup \bigcup_{j \in \mathcal{T}_i} (\gamma'(j).R \times \{j\}) \qquad \gamma'.W = \gamma.W \cup \bigcup_{j \in \mathcal{T}_i} (\gamma'(j).W \times \{j\})\end{array}}{(\gamma, \mathsf{basic\_stmt}, p) \rightarrow_{g(i)} \gamma'}\ \text{(G-BASIC)}$$

(a) Rules for basic statements

$$\frac{\forall\, j \in \mathcal{T}_i\ .\ \neg p^{\gamma(j)}}{(\gamma, \mathbf{barrier}, p) \rightarrow_{g(i)} \gamma}\ \text{(G-NO-OP)} \qquad \frac{\exists\, j \neq k \in \mathcal{T}_i\ .\ p^{\gamma(j)} \wedge \neg p^{\gamma(k)}}{(\gamma, \mathbf{barrier}, p) \rightarrow_{g(i)} \mathit{error}}\ \text{(G-DIVERGENCE)}$$

$$\frac{\forall\, j \in \mathcal{T}_i\ .\ p^{\gamma(j)} \qquad \forall\, j \in \mathcal{T}_i\ .\ \gamma'(j) = \gamma(j)[sh \mapsto \mathsf{merge}_i(\gamma), R \mapsto \emptyset, W \mapsto \emptyset]}{(\gamma, \mathbf{barrier}, p) \rightarrow_{g(i)} \gamma'}\ \text{(G-SYNC)}$$

(b) Rules for barriers

Fig. 5. Rules for lock-step execution of basic statements and barriers by groups.

Predicated Execution of Threads. The rules of Figure 4 define the binary relation

$$\rightarrow_t \subseteq (\mathsf{ThreadStates} \times \mathsf{PredStmts}) \times \mathsf{ThreadStates}$$

describing the evolution of one thread state to another under execution of a predicated basic statement. For readability, given a thread state τ and predicated statement (s, p), we write (τ, s, p) instead of $(\tau, (s, p))$.

Rule T-DISABLED ensures that a predicated statement has no effect if the predicate does not hold, indicated by $\neg p^{\tau}$ in the premise of the rule; T-ASSIGN updates $\tau.l$ according to the assignment; T-READ updates the local store of the thread with the appropriate element from the shadow copy of shared memory owned by the thread, and records the address that was read from; rule T-WRITE is analogous.

Predicated Execution of Groups. Figure 5 defines the binary relation

$$\rightarrow_{g(i)} \subseteq (\mathsf{GroupStates}_i \times \mathsf{PredStmts}) \times (\mathsf{GroupStates}_i \cup \{\mathit{error}\})$$

describing how a group state of a group i mutates into another as a result of executing a predicated basic statement or barrier, where *error* is a designated error state. As for threads, given a group state γ and predicated statement (s, p), we write (γ, s, p) instead of $(\gamma, (s, p))$.

Intragroup data races are detected via rule G-RACE. An intragroup race occurs if there is a thread j in group i such that the write set of j intersects with either the

read or write set of a thread k different from j but in the same group. The predicate $\mathsf{group\_race}_i(\gamma)$ used to detect intragroup races is formally defined as:

$$\mathsf{group\_race}_i(\gamma) \triangleq \exists\, j \neq k \in \mathcal{T}_i \,.\, \gamma(j).W \cap (\gamma(k).R \cup \gamma(k).W) \neq \emptyset\,.$$

Rule G-RACE states that if the execution of a basic statement by all threads in a group leads to a group state γ' in which $\mathsf{group\_race}_i(\gamma')$ holds, then the *error* state is reached.

Lock-step execution of basic statements by a group is achieved by having each thread in the group execute the statement; the order in which they do so is irrelevant due to delayed visibility (G-BASIC). The rule requires the execution of the basic statement to be free from intragroup data races. The rule also records which addresses of global memory were read from or written to by the threads in the group; this is achieved by propagating the addresses from the read and write sets of the individual threads into the read and write set of the group.

Execution of **barrier** with all threads in a group disabled has no effect (G-NO-OP). If the group is due to execute a **barrier** statement under predicate p but not all threads agree on the truth of p, then the *error* state is reached (G-DIVERGENCE). This *precisely* captures the notion of barrier divergence discussed in Section 2.

Rule G-SYNC captures the effect of barrier synchronization under predicate p in the case where all threads in the group agree that p is valid. A new group state γ' is constructed, in which for each thread j, the read and write sets of j are empty in $\gamma'(j)$ and the local component $\gamma'(j).l$ is identical to the local component for the thread before the barrier. The barrier also enforces a consistent view of shared memory across the group by setting the shadow memories of all threads to the same value. This is achieved by the function merge_i. If thread j has recorded a write to a shared memory location z, that is, $z \in \gamma(j).W$, then $\mathsf{merge}_i(\gamma)$ maps z to the value at address z in the shadow memory of thread j, that is, to $\gamma(j).sh(z)$.

Formally, $\mathsf{merge}_i(\gamma)$ is a map satisfying the following constraints:

$$\frac{j \in \mathcal{T}_i \quad z \in \gamma(j).W}{\mathsf{merge}_i(\gamma)(z) = \gamma(j).sh(z)} \qquad \frac{\forall\, j \in \mathcal{T}_i \,.\, z \notin \gamma.W}{\mathsf{merge}_i(\gamma)(z) = \gamma(0).sh(z)}$$

The value of $\mathsf{merge}_i(\gamma)$ is guaranteed to be unique, otherwise there exists a z belonging to $\gamma(j).W$ and $\gamma(k).W$ for $j \neq k$, in which case execution would have aborted earlier via G-RACE. Observe that, in the second constraint, $\gamma(0)$ is arbitrary, and could have been chosen to be any $\gamma(i)$; the value of $sh(z)$ is consistent across threads in this case, since no thread has written to the memory location z.

Predicated Execution of Kernels. Figure 6 defines the binary relation

$$\rightarrow_k \subseteq \mathsf{KernelStates} \times (\mathsf{KernelStates} \cup \{\mathit{error}\})\,,$$

where *error* is again a designated error state. This relation describes the evolution of a kernel as it executes a sequence of predicated statements.

Intergroup races are detected via rule K-INTER-GROUP-RACE. An intergroup race occurs if there is a group i such that the write set of i intersects with either the read or write set of a group j different from i (ignoring the particular thread responsible for each read and write). The predicate $\mathsf{kernel\_race}(\kappa)$ used to detect intergroup races is formally defined as:

$$\mathsf{kernel\_race}(\kappa) \triangleq \exists\, i \neq j \in \mathcal{G} \,.\, \mathit{fst}(\kappa(i).W) \cap \mathit{fst}(\kappa(j).R \cup \kappa(j).W) \neq \emptyset,$$

where $\mathit{fst}(X) = \{x \mid (x, y) \in X\}$. Rule K-INTER-GROUP-RACE states that if collective execution of a predicated basic statement by all groups leads to a state κ' in which $\mathsf{kernel\_race}(\kappa')$ holds, then the *error* state is reached. Rule K-INTRA-GROUP-RACE detects intragroup races by lifting the application of rule G-RACE to the kernel level.

$$\frac{\forall\, i \in \mathcal{G}\ .\ (\kappa(i), \mathsf{basic\_stmt}, p) \rightarrow_{g(i)} \kappa'(i) \qquad \mathsf{kernel\_race}(\kappa')}{(\kappa, (\mathsf{basic\_stmt}, p) : ss) \rightarrow_k \mathit{error}}\ \text{(K-INTER-GROUP-RACE)}$$

$$\frac{\exists\, i \in \mathcal{G}\ .\ (\kappa(i), \mathsf{basic\_stmt}, p) \rightarrow_{g(i)} \mathit{error}}{(\kappa, (\mathsf{basic\_stmt}, p) : ss) \rightarrow_k \mathit{error}}\ \text{(K-INTRA-GROUP-RACE)}$$

$$\frac{\forall\, i \in \mathcal{G}\ .\ (\kappa(i), \mathsf{basic\_stmt}, p) \rightarrow_{g(i)} \kappa'(i) \qquad \neg\mathsf{kernel\_race}(\kappa')}{(\kappa, (\mathsf{basic\_stmt}, p) : ss) \rightarrow_k (\kappa', ss)}\ \text{(K-BASIC)}$$

(a) Rules for basic statements

$$\frac{\exists\, i \in \mathcal{G}\ .\ (\kappa(i), \mathbf{barrier}, p) \rightarrow_{g(i)} \mathit{error}}{(\kappa, (\mathbf{barrier}, p) : ss) \rightarrow_k \mathit{error}}\ \text{(K-DIVERGENCE)}$$

$$\frac{\forall\, i \in \mathcal{G}\ .\ (\kappa(i), \mathbf{barrier}, p) \rightarrow_{g(i)} \kappa'(i)}{(\kappa, (\mathbf{barrier}, p) : ss) \rightarrow_k (\kappa', ss)}\ \text{(K-SYNC)}$$

(b) Rules for barriers

$$\frac{}{(\kappa, (S_1; S_2, p) : ss) \rightarrow_k (\kappa, (S_1, p) : (S_2, p) : ss)}\ \text{(K-SEQ)}$$

$$\frac{\text{fresh } v \in V}{(\kappa, (\mathbf{local}\ x\ S, p) : ss) \rightarrow_k (\kappa, (S[x \mapsto v], p) : ss)}\ \text{(K-VAR)}$$

$$\frac{\text{fresh } v \in V}{(\kappa, (\mathbf{if}\ e\ S_1\ \mathbf{else}\ S_2, p) : ss) \rightarrow_k (\kappa, (v := e, p) : (S_1, p \wedge v) : (S_2, p \wedge \neg v) : ss)}\ \text{(K-IF)}$$

$$\frac{\text{fresh } v \in V}{(\kappa, (\mathbf{while}\ e\ S, p) : ss) \rightarrow_k (\kappa, (\underline{\mathbf{while}}\ e\ \mathit{belim}(S, v), p \wedge \neg v) : ss)}\ \text{(K-OPEN)}$$

$$\frac{\exists\, i \in \mathcal{G},\ j \in \mathcal{T}_i\ .\ (p \wedge e)^{\kappa(i)(j)} \qquad \text{fresh } u, v \in V}{(\kappa, (\underline{\mathbf{while}}\ e\ S, p) : ss) \rightarrow_k (\kappa, (u := e, p) : (\mathit{celim}(S, v), p \wedge u \wedge \neg v) : (\underline{\mathbf{while}}\ e\ S, p) : ss)}\ \text{(K-ITER)}$$

$$\frac{\forall\, i \in \mathcal{G},\ j \in \mathcal{T}_i\ .\ \neg(p \wedge e)^{\kappa(i)(j)}}{(\kappa, (\underline{\mathbf{while}}\ e\ S, p) : ss) \rightarrow_k (\kappa, ss)}\ \text{(K-DONE)}$$

$$\frac{\text{fresh } u, v \in V \qquad S = \mathit{Body}(f)[\mathit{Param}(f) \mapsto u]}{(\kappa, (\mathbf{call}\ f(e), p) : ss) \rightarrow_k (\kappa, (u := e; \mathit{relim}(S, v), p \wedge \neg v) : ss)}\ \text{(K-CALL)}$$

(c) Rules for statements

Fig. 6. Rules for lock-step execution of statements by kernels.

Collective execution of a predicated basic statement is achieved by having every group execute the statement; the order in which they do so is irrelevant (K-BASIC). The rule requires that the collective execution was free from intergroup data races (via the $\neg$kernel_race(κ') condition) and intragroup data races (via the condition that each group state in κ transitions to a non-error group state in κ').

Rule K-DIVERGENCE lifts application of rule G-DIVERGENCE to the kernel level, that is, the kernel transitions to the *error* state if barrier divergence is detected in one of the groups. Collective execution of a **barrier** statement in the absence of barrier

divergence is achieved via rule K-SYNC where each group transitions to a new group state either by rule G-SYNC or rule G-NO-OP. Observe that the read and write sets of groups are not cleared because barriers are used solely for intragroup synchronization and do not achieve intergroup synchronization.

The remaining rules of Figure 6 describe predicated execution of compound statements. Rule K-SEQ is straightforward. Rule K-VAR creates storage for a new local variable x by allocating a fresh location v in V and substituting all occurrences of x in S by v; we use $S[x \mapsto v]$ to denote this substitution. Rule K-IF decomposes a conditional statement into a sequence of predicated statements: the guard of the conditional is evaluated into a new location v; the *then* branch S_1, is executed *by all threads in all groups* under predicate $p \wedge v$ (where p is the predicate already in place on entry to the conditional), and the *else* branch S_2, is executed *by all threads in all groups* under predicate $p \wedge \neg v$.

Rules K-OPEN, K-ITER, and K-DONE together model predicated execution of a **while** loop. In what follows, we say that a **break** or **continue** statement is *top-level* in a loop if the statement appears in the loop body but is not nested inside any further loops.

Rule K-OPEN converts a **while** loop into a **while** loop by creating fresh storage to model **break** statements. A fresh location v is selected; v records whether a thread has executed a **break** statement associated with the **while** loop. Like all local storage, v initially has the value *false* (see Remark 3.1): no thread has executed **break** on loop entry. The function *belim* is applied to the loop body. This function takes a statement S and a location v and replaces each top-level **break** statement inside S by the statement $v := true$. The predicate for the execution of the **while** loop becomes $p \wedge \neg v$ to model that the statements in the loop have no effect after the execution of a **break** statement. A similar technique to model **break** statements is used by Habermaier and Knapp [2012].

The K-ITER rule models execution of loop iterations and handles **continue** statements. The rule fires each time a loop iteration is executed. For a given loop iteration, two fresh local storage locations u and v are selected and (as per Remark 3.1) both are initialized to *false*.[5] Location u is used to store the valuation of the loop guard. Location v is used to record whether a thread has executed a top-level **continue** statement during the current loop iteration. The value of v is initially *false*, since no thread has executed a **continue** statement at the beginning of an iteration. The statement $u := e$ (executed under the enclosing predicate p) evaluates the loop guard into u. The function *celim* is applied to the loop body; this function takes a statement S and a location v and replaces each top-level **continue** statement inside S by the statement $v := true$. The loop body, after elimination of **continue** statements, is executed under the predicate $p \wedge u \wedge \neg v$: a thread is enabled during the current iteration if the incoming predicate holds (p), the loop guard evaluates to *true* at the start of the iteration (u) and the thread has not executed a **continue** statement ($\neg v$). Note that, due to rule K-OPEN, the incoming predicate p includes a conjunct recording whether the thread has executed a **break** statement. After the loop body, the **while** construct is considered again.

Thus, all threads continuously execute the loop body using K-ITER until, for *every* thread in *every* group, (a) the enclosing predicate p becomes *false*, either because this predicate was *false* on loop entry or because the thread has executed **break**, or (b) the loop condition no longer holds for the thread.[6] When (a) or (b) holds for all threads, loop exit is handled by rule K-DONE.

[5]Note that u and v are not reused between loop iterations: a fresh u and v are selected at the start of each loop iteration. Hence, there is no need to reset u and v at the end of a loop iteration.

[6]Execution of **continue** does not directly contribute to the conditions under which threads exit a loop. This is because on executing a **continue** statement a thread becomes disabled for the remainder of the current

The rule K-CALL models the execution of a call to a procedure f. This involves executing the statement corresponding to the body of the called procedure ($Body(f)$) after replacing all occurrences of its formal parameter ($Param(f)$) with a location storing the evaluation result of the actual parameter expression. All threads execute the entire body of a procedure in lock-step. A fresh storage location v is used to record whether a thread has executed a return statement. Initially, this location is set to *false*, and the function *relim* replaces each return statement in $Body(f)$ with the statement $v := true$. The procedure body is executed under the predicate $p \wedge \neg v$ (where p is the existing predicate of execution at the point of the call) so that execution of a return statement by a thread is simulated by the thread becoming disabled for the remainder of the procedure body.

3.3. Formalization in Isabelle

The definitions in this section have been formalized in the Isabelle proof assistant [Nipkow et al. 2002] and are available from the Archive of Formal Proofs [Wickerson 2014]. For the most part, the Isabelle definitions closely resemble those presented earlier, but there are three notable differences. First, we do not model the abuse of notation whereby $\gamma(j)$ abbreviates $\gamma_{ts}(j)$, since the use of such *type coercions* was found to necessitate many additional type annotations. Second, we extend kernel states with a third component, to record all the variables that have been used. This is necessary for calculating the fresh variables required by several of the rules in Figure 6, but is left implicit, in accordance with common practice, in the previous subsections. Third, because Isabelle does not allow *dependent types*, we cannot define functions between arbitrary sets (at least not without extensive use of `typedefs`, which can complicate proofs) such as $\gamma_{ts} : \mathcal{T}_i \rightarrow \mathsf{ThreadStates}$. In such situations, we instead declare γ_{ts} as a partial function from local IDs to ThreadStates and make $dom(\gamma_{ts}) = \mathcal{T}_i$ available as an assumption when proving theorems.

In Section 4.2, we discuss our experience using these formalized definitions to partially mechanize a proof of soundness for the *two-thread reduction*; this reduction is our next main contribution.

4. THE TWO-THREAD REDUCTION

The SDV operational semantics of Section 3 has all threads execute in lock-step. As such, our massively parallel N-threaded kernel can be encoded as a single sequential program, where each instruction is replicated N times, and where assertions are inserted to check for data races and barrier divergence. In principle, existing techniques for sequential program verification can be applied directly to this program and, as discussed in detail in Section 5, we have investigated the use of the Boogie verification system [Barnett et al. 2005] for this purpose. We have found that directly encoding massively parallel kernels as sequential programs leads to infeasibly large verification conditions if an explicit representation of threads is used. In Section 6.4, we illustrate this blow-up in verification condition size, and the corresponding effect this has on verification performance, using a simple example. The explicit representation can be replaced with quantification over all threads, but we have found this to lead to verification conditions that cannot be automatically handled by Z3, a state-of-the-art theorem prover [de Moura and Bjørner 2008], which Boogie uses by default.[7]

---

loop iteration, but the thread is *re-enabled* at the start of the next iteration, provided the loop guard still holds for the thread.

[7]The difficulty of reasoning about quantifiers is, of course, not specific to Z3, but a general obstacle faced by any automated theorem prover.

To overcome these practical limitations and achieve scalable reasoning, we describe an alternative encoding that involves reducing the number of threads under consideration from N to just *two*.

Observe that the properties of data race- and barrier divergence-freedom are *pairwise* properties: a race occurs when accesses by two threads conflict, and barrier divergence occurs when a barrier is executed in a state where one thread is enabled and another thread in the same group is disabled. Therefore, we can consider an operational semantics where the predicated execution of only *two* threads is modeled. If we can use such a semantics to prove a kernel data race- and barrier divergence-free for a pair of distinct but otherwise *arbitrary* threads, we can conclude correctness of the kernel. This is because the verification process considers all possible execution traces so that, because the two threads under consideration are arbitrary, interactions between all possible pairs of threads are implicitly considered.

Formally, if the original execution involves all of the threads in $tids(\mathcal{G}, \mathcal{T})$ (see Section 3.2.1), then we need to check data race- and barrier divergence-freedom for all restrictions of $(\mathcal{G}, \mathcal{T})$ to a pair of threads, that is, for restrictions to $(\mathcal{G}', \mathcal{T}')$ such that

—$\mathcal{G}' \subseteq \mathcal{G}$ and $\mathcal{T}_i' \subseteq \mathcal{T}_i$ for all $i \in \mathcal{G}'$, and
—$tids(\mathcal{G}', \mathcal{T}')$ consists of exactly two elements.

It is important to clarify that in practice we do not enumerating all possible two-thread reductions. Rather, as discussed in Section 5, when we discuss the implementation of GPUVerify, we use symbolic constants to model the identifiers of two arbitrary threads during analysis.

We now present the reduction to a pair of threads formally and prove that it is sound. Our GPUVerify verification technique and tool, described in Section 5, depend on this reduction. The design of the PUG verifier for CUDA kernels by Li and Gopalakrishnan [2010] also hinges on a reduction to a pair of threads. However, this is the first formal presentation and proof of soundness for the reduction method.

Observe that the definitions of (valid) group and kernel states carry over to the sets $\mathcal{G}'$ and $\mathcal{T}_i'$ as defined earlier and that the same holds for the kernel-level execution rules from Figure 6. However, for the aforementioned approach to be sound, we must *approximate* shared state handling, abstracting the values written to the shared state by threads that are not modeled. This can be achieved in multiple ways. We consider the following strategies:

Equality abstraction: Threads manipulate a shadow copy of the shared state. At a barrier, the shadow copies are set to be *arbitrary*, but *equal*. Thus, on leaving the barrier, the threads have a consistent view of shared memory.

Adversarial abstraction: The shared state is made irrelevant; reads from the shared state into local variables are replaced with nondeterministic assignments to local variables.

We have found several example kernels where race-freedom hinges on threads agreeing on the values read from certain shared locations. In these cases, the adversarial abstraction is too coarse for successful verification. However, in many cases, it does not matter what specific value is stored in shared memory, only that all threads see the same value. The equality abstraction suffices for such cases. Our use of equality abstraction allows us to improve upon the precision of the work of Li and Gopalakrishnan [2010], which is limited to the adversarial abstraction.

The execution relation for equality abstraction, written $\rightarrow_{k,\mathcal{E}(\mathcal{G}',\mathcal{T}')}$, is obtained by replacing rule G-SYNC in Figure 5 with rule G-SYNC-EQ from Figure 7(a), and replacing all references to $\mathcal{G}$ and $\mathcal{T}$ in all other semantic rules in Figures 5 and 6 with $\mathcal{G}'$ and $\mathcal{T}'$,

$$\frac{\forall\, j \in \mathcal{T}_i' \,.\, p^{\gamma(j)} \qquad \exists sh' \,.\, \forall\, j \in \mathcal{T}_i' \,.\, \gamma'(j) = \gamma(j)[sh \mapsto sh', R \mapsto \emptyset, W \mapsto \emptyset]}{(\gamma, \mathbf{barrier}, p) \rightarrow_{g(i)} \gamma'} \text{(G-SYNC-EQ)}$$

(a) Updated rule for the equality abstraction

$$\frac{p^{\tau} \qquad \exists\, v' \,.\, l' = \tau.l[v \mapsto v'] \qquad R' = \tau.R \cup \{e^{\tau}\}}{(\tau, v := sh[e], p) \rightarrow_t \tau[l \mapsto l', R \mapsto R']} \text{(T-READ-ADV)}$$

$$\frac{p^{\tau} \qquad W' = \tau.W \cup \{e_1^{\tau}\}}{(\tau, sh[e_1] := e_2, p) \rightarrow_t \tau[sh \mapsto sh', W \mapsto W']} \text{(T-WRITE-ADV)}$$

$$\frac{\forall\, j \in \mathcal{T}_i' \,.\, p^{\gamma(j)} \qquad \forall\, j \in \mathcal{T}_i' \,.\, \exists\, sh' \,.\, \gamma'(j) = \gamma(j)[sh \mapsto sh', R \mapsto \emptyset, W \mapsto \emptyset]}{(\gamma, \mathbf{barrier}, p) \rightarrow_{g(i)} \gamma'} \text{(G-SYNC-ADV)}$$

(b) Updated rules for the adversarial abstraction

Fig. 7. Rules for the adversarial and equality abstractions.

respectively. In rule G-SYNC-EQ, instead of merging the shared states of the threads in group i, an arbitrary value is chosen for the shared state and the shadow copy of each thread in the group is assigned this value. Clearing of read and write sets and checking for barrier divergence is left unchanged.

The execution relation for adversarial abstraction, written $\rightarrow_{k,\mathcal{A}(\mathcal{G}',\mathcal{T}')}$, is obtained by replacing rules T-READ and T-WRITE in Figure 4, and G-SYNC in Figure 5, with the rules T-READ-ADV, T-WRITE-ADV, and G-SYNC-ADV from Figure 7(b), respectively, as well as replacing all references to $\mathcal{G}$ and $\mathcal{T}$ in all other semantic rules in Figures 5 and 6 with $\mathcal{G}'$ and $\mathcal{T}'$, respectively. Rule T-READ-ADV makes the shared state irrelevant by updating the local store with an arbitrary value v' rather than one obtained from the shared state, although it still logs the read in the read set of the thread. Rule T-WRITE-ADV sets the shared state to an arbitrary value sh' but still logs the write in the write set of the thread.[8] Similar to G-SYNC-EQ, rule G-SYNC-ADV does away with merging the shared states of the threads in a group i; the rule sets the shared state of each thread to an arbitrary value, clears the read and write sets and checks for barrier divergence.

Comparing rules G-SYNC-EQ and G-SYNC-ADV, the premises are almost identical. The key difference is that in rule G-SYNC-EQ a single value is selected for the shared state, common to all threads, while in rule G-SYNC-ADV a value for the shared state is selected separately for each thread. Thus, with the adversarial abstraction, threads do not see a unified view of shared memory after a barrier synchronization.

4.1. Soundness of the Two-Thread Reduction

We first prove the soundness of the two-thread reduction when using the equality abstraction.

Throughout, let a *concrete execution* be a nonempty sequence of kernel states

$$\langle(\kappa_0, ss_0), \ldots, (\kappa_{n-1}, ss_{n-1})\rangle,$$

where (κ_0, ss_0) is a valid initial kernel state and where each successive pair is related by the $\rightarrow_k$ relation of Figure 6. An *execution under the equality abstraction with respect to*

[8]Technically, this rule does not need to change the shared state at all because the shared state is ignored by the T-READ-ADV. However, the rule in its current form is better able to simulate the concrete T-WRITE rule in the upcoming soundness proof.

$(\mathcal{G}', \mathcal{T}')$ is defined similarly, except that it uses the $\rightarrow_{k,\mathcal{E}(\mathcal{G}',\mathcal{T}')}$ relation instead. Recalling that $\mathcal{G} = \{0, 1, \ldots, gs - 1\}$ and that $\mathcal{T}_i = \{0, 1, \ldots, ts - 1\}$ for each $i \in \mathcal{G}$, we now have the following:

THEOREM 4.1 (SOUNDNESS: EQUALITY ABSTRACTION). *Let P be a KPL kernel executed by gs groups and ts threads per group. If no execution of P under the equality abstraction with respect to any $(\mathcal{G}', \mathcal{T}')$ (representing a pair of threads) leads to error, then no concrete execution of P leads to error.*

PROOF. We prove the contrapositive. Thus, let us assume that P has a concrete execution leading to *error*. We will show that there also exists an execution for P under the equality abstraction that leads to *error*.

Let the concrete execution of P that leads to *error* be:

$$\rho \triangleq \langle(\kappa_0, ss_0), \ldots, (\kappa_{n-1}, ss_{n-1})\rangle\,.$$

We now construct an execution of P under the equality abstraction with respect to some $(\mathcal{G}', \mathcal{T}')$ (representing a pair of threads) that also leads to *error*.

Observe that in the final state (κ_{n-1}, ss_{n-1}), either rule K-INTER-GROUP-RACE, K-INTRA-GROUP-RACE, or K-DIVERGENCE applies because only these rules can lead to *error*. In each case, exactly two threads are responsible for the rule being applicable. Assume that these threads are $s = (i_1, j_1)$ and $t = (i_2, j_2)$, with the first component of each pair being the group ID of the thread and the second component being the local ID. Define $\mathcal{G}'$ and $\mathcal{T}'$ as follows:

—if $i_1 = i_2$, then $\mathcal{G}' = \{i_1\}$ and $\mathcal{T}_{i_1} = \{j_1, j_2\}$; and
—if $i_1 \neq i_2$, then $\mathcal{G}' = \{i_1, i_2\}$, $\mathcal{T}_{i_1} = \{j_1\}$ and $\mathcal{T}_{i_2} = \{j_2\}$.

We now define a function $proj^{s,t}$ that takes a kernel state and filters out all components not pertaining to s and t:

$$proj^{s,t}(\kappa, ss) \triangleq (\kappa', ss),$$

where $dom(\kappa') = \mathcal{G}'$ and $\kappa'(i) = proj_i^{s,t}(\kappa(i))$ for all $i \in \mathcal{G}'$. For group states, we define:

$$proj_i^{s,t}(\gamma_{ts}, R, W) \triangleq (\gamma_{ts}\restriction_{\mathcal{T}_i'}, R \cap (\mathbb{N} \times \mathcal{T}_i'), W \cap (\mathbb{N} \times \mathcal{T}_i')),$$

where $f\restriction_A$ denotes the restriction of the domain of f to A. The intersection with $\mathbb{N} \times \mathcal{T}_i'$ restricts the read and write sets of the group to only those accesses by the thread(s) in $\mathcal{T}_i'$.

We next define a mapping $\mathcal{E}$ which, given a concrete execution ρ, will yield an execution $\mathcal{E}(\rho)$ under the equality abstraction. For the most part, it simply maps each successive kernel state (κ_i, ss_i) to the state $proj^{s,t}(\kappa_i, ss_i)$. Complexity arises when we reach a state that satisfies the following condition:

Rule K-ITER applies, but the predicate $p \wedge e$ holds for neither s nor t. (*)

The aforementioned situation occurs when, during the execution of a loop in a concrete execution, the guard no longer holds for threads s and t but does hold for at least one other thread. The appropriate rule in the concrete execution is K-ITER, but in the abstracted execution, we must invoke the K-DONE rule instead to exit the loop. Accordingly, when $\mathcal{E}$ reaches a state (κ_i, ss_i) where the condition (*) holds, it discards all of the following states up to and including the first state (κ_j, ss_j) where K-DONE applies and $ss_j = ss_i$. Here, the condition $ss_j = ss_i$ ensures that this is the end of the *current* loop, not the end of a loop nested in the current one.

The definition of $\mathcal{E}$ is as follows:

$$\mathcal{E}(\langle\rangle) \triangleq \langle\rangle$$
$$\mathcal{E}((\kappa, ss) : \rho) \triangleq \begin{cases} proj^{s,t}(\kappa, ss) : \mathcal{E}(dropUntil(loopEnd_{\kappa,ss}, \rho)) & \text{if } (\kappa, ss) \text{ satisfies (*)} \\ proj^{s,t}(\kappa, ss) : \mathcal{E}(\rho) & \text{otherwise} \end{cases}$$

where

$$dropUntil(p, \langle\rangle) \triangleq \langle\rangle$$
$$dropUntil(p, x : xs) \triangleq \begin{cases} xs & \text{if } p(x) \text{ holds} \\ dropUntil(p, xs) & \text{otherwise} \end{cases}$$

and where the predicate $loopEnd_{\kappa,ss}(\kappa', ss')$ holds if and only if $ss = ss'$ and K-DONE applies in state (κ', ss').

We prove that $\mathcal{E}(\rho)$ is an execution under the equality abstraction that leads to *error*, proceeding by induction on the length of ρ. Thus, suppose that $\rho = (\kappa, ss) : \rho'$. In the case where ρ' is empty—the base case—we obtain $\langle proj^{s,t}(\kappa, ss)\rangle$, which is clearly an execution of P under the equality abstraction. Moreover, since we have that (κ, ss) leads to *error*, it follows by our choice of s and t that so does $\langle proj^{s,t}(\kappa, ss)\rangle$.

For the induction step, let us first assume that the condition (*) does *not* apply to (κ, ss). We must now show that $proj^{s,t}(\kappa, ss) : \mathcal{E}(\rho')$ is an execution of P under the equality abstraction. Hence, assume that $\rho' = (\kappa', ss') : \rho''$. The induction hypothesis guarantees that $\mathcal{E}((\kappa', ss') : \rho'')$ is an execution under the equality abstraction leading to *error*. It remains to demonstrate that $proj^{s,t}(\kappa, ss)$ can take an abstract step to $proj^{s,t}(\kappa', ss')$. Since we know that (κ, ss) takes a concrete step to (κ', ss'), we can complete this part of the proof by a case distinction on the rules from Figure 6, that is, the abstract step simply invokes the equality-abstracted version of the rule invoked by the concrete step, where in the case of a barrier, the arbitrary shared state sh' in the rule G-SYNC-EQ is chosen to simulate the concrete kernel state.

If the condition (*) *does* apply to (κ, ss), then we must show that

$$\mathcal{E}(dropUntil(loopEnd_{\kappa,ss}, \rho'))$$

is an execution of P under the equality abstraction. To apply the induction hypothesis, we must know that the sequence $dropUntil(loopEnd_{\kappa,ss}, \rho')$ is nonempty. Thus, suppose for the moment that the sequence is empty. The condition (*) now implies that (κ, ss) is a state at the head of a loop and that threads s and t are disabled throughout the remainder of the loop execution. The sequence being empty means that the concrete execution reaches *error* before exiting the loop. Yet, since the transition to *error* is brought about by threads s and t, at least one must be enabled: contradiction. Hence, $dropUntil(loopEnd_{\kappa,ss}, \rho')$ is indeed nonempty, and it must therefore have a first state (κ'', ss''). We can now apply the induction hypothesis, to obtain that $\mathcal{E}(dropUntil(loopEnd_{\kappa,ss}, \rho'))$ is an execution of P under the equality abstraction and leads to *error*. It remains to show that we can take an abstract step from $proj^{s,t}(\kappa, ss)$ to $proj^{s,t}(\kappa'', ss'')$: this is justified by a single invocation of the K-DONE rule. The invocation is possible, as the shadow copies of the global memory as owned by threads s and t did not change between (κ, ss) and (κ'', ss''). If the shadow copies did change, either the condition (*) did not hold in (κ, ss), or rule G-SYNC was applied to a group in $\mathcal{G}'$ somewhere between (κ, ss) and (κ'', ss''); both of these contradict the assumption that the condition (*) holds. In particular, the rule G-SYNC can only apply if all threads in the group are enabled, which is not the case according to condition (*).

Finally, we note that since (κ_0, ss_0) is a valid initial kernel state, so is $proj^{s,t}(\kappa_0, ss_0)$ (the first state of $\mathcal{E}(\rho)$). Hence, $\mathcal{E}(\rho)$ is an execution of P under the equality abstraction leading to *error* and the result now follows. □

We shall now prove the soundness of the two-thread reduction when using the adversarial abstraction. Here, *execution under the adversarial abstraction with respect to* $(\mathcal{G}', \mathcal{T}')$ is defined to be a nonempty sequence of kernel states where each successive pair is related by the $\rightarrow_{k,\mathcal{A}(\mathcal{G}',\mathcal{T}')}$ relation.

THEOREM 4.2 (SOUNDNESS: ADVERSARIAL ABSTRACTION). *Let P be a KPL kernel executed by gs groups and ts threads per group. If no execution of P under the adversarial abstraction with respect to any $(\mathcal{G}', \mathcal{T}')$ (representing a pair of threads) leads to error, then no concrete execution of P leads to error.*

PROOF. By replaying the proof of Theorem 4.1, with G-SYNC-EQ replaced by G-SYNC-ADV and where in the case of K-BASIC, we observe that the arbitrary values chosen by rules T-READ-ADV and T-WRITE-ADV can be chosen in accordance with the shared state and evaluation of local expressions, respectively. □

Remark 4.3. Observe that as the contents of the shared state is not actually used by the adversarial abstraction, we can completely omit the shared state from the semantics. Our verification method presented in the next section exploits this observation.

Incompleteness of the Two-Thread Reduction. No completeness results corresponding to the aforementioned two soundness results hold, as the arbitrary choices employed in the abstractions allow for strictly more behavior. Moreover, as already observed by Habermaier and Knapp [2012], the abstractions allow for possible detection of races and barrier divergence in code not reachable during real kernel executions due to an infinite loop preceding the offending code. For instance, the following KPL fragment does *not* have a race on the shared location 42 because the failure of thread 0 to exit the loop prevents any other thread from writing to the location, but when only two arbitrary threads are considered, we cannot deduce that the loop does not terminate.

while $(lid = 0)$ **continue**;
$sh[42] := lid$

4.2. Use of the Isabelle Proof Assistant

As explained in Section 3.3, we used the Isabelle proof assistant [Nipkow et al. 2002] to formalize all the definitions presented in Section 3; we have also formalized the further definitions of this section using Isabelle. Even without conducting any proofs, the strict typing discipline imposed by Isabelle allowed us to find and fix several ambiguities in the original definitions.

We had intended to formalize the soundness results in Theorems 4.1 and 4.2 but found the task to be more challenging than anticipated, mainly due to the complexity of the KernelStates type and the sheer number of execution rules. The requirement for "fresh" variables in several of the execution rules proved particularly challenging; indeed, "if one wants to formalize such proofs in a theorem prover, then dealing with binders, renaming of bound variables, capture-avoiding substitution, etc., is very often a major problem" [Urban and Narboux 2009].

Nonetheless, the use of Isabelle in a "top-down" manner to explore the high-level structure of the proof rather than to prove each statement from first principles was very helpful. We were able to significantly improve the quality of an earlier hand proof of Theorem 4.1 and gain increased confidence in its correctness. For instance, to make the abstract semantics successfully simulate the concrete semantics, the R and W

components of each group state must tag each accessed location with the local ID of the accessing thread so that those locations not accessed by thread s or t can be identified and removed. This observation was made as a direct result of our failure to prove a lemma in Isabelle.

Our Isabelle proof script, which comprises the definitions presented in Sections 3 and 4, plus an outline proof of Theorem 4.1, is available online.[9]

5. THE DESIGN AND IMPLEMENTATION OF GPUVERIFY

Armed with the SDV semantics and abstractions of Section 3, we now consider the problem of verifying that GPU kernels are data race- and barrier divergence-free. For this purpose, we have designed a tool, GPUVerify, built on top of the Boogie verification system [Barnett et al. 2005]. Boogie takes a program annotated with loop invariants and procedure contracts, and decomposes verification into a set of formulas to be checked automatically by the Z3 theorem prover [de Moura and Bjørner 2008]. We describe the challenges associated with automatically translating OpenCL and CUDA kernels into a Boogie intermediate representation (Section 5.1), a technique for transforming this Boogie representation of the kernel into a standard sequential Boogie program whose correctness implies data race- and barrier divergence-freedom of the original kernel (Section 5.2), and a method for automatically inferring invariants and procedure contracts to enable automatic verification (Section 5.3).

5.1. Translating OpenCL and CUDA into Boogie

To allow GPUVerify to be applied directly to source code, we have implemented a compiler that translates GPU kernels into an intermediate Boogie form. Our compiler is built on top of the Clang/LLVM infrastructure, which supports both OpenCL and CUDA. We target CUDA 5.0 [Nvidia 2012a] and versions of OpenCL up to 1.2 [Khronos OpenCL Working Group 2012].

There were effectively three challenges with respect to compilation. First, many industrial applications utilize features of OpenCL and CUDA not present in vanilla C, such as declaring variables as vector or image types, or calling intrinsic functions; we therefore invested significant engineering effort into designing equivalent Boogie types and functions. Second, the Boogie language does not support floating point values directly, thus we modeled them abstractly via uninterpreted functions. This sound overapproximation can in principle lead to false positives, but we have not found this to be a problem in practice because floating point operators are not used to compute array indexing expressions or to determine synchronization-related control flow. The third issue, namely handling of pointers, is technically more interesting and is now discussed in depth.

Modeling Pointers. Boogie is a deliberately simple intermediate language and does not support pointer data types natively. We have devised an encoding of pointers in Boogie, which we explain using an example. For readability, we use C-like syntax rather than the Boogie input language.

Suppose a kernel declares exactly two character arrays (in any memory space) and two character pointers:

```
char A[1024], B[1024];
char *p, *q;
```

[9]http://multicore.doc.ic.ac.uk/tools/GPUVerify/IsabelleFormalisation/.

Table V. Translating Pointer Usage into Boogie

Source	Generated Boogie
`p = A;`	`p = int_ptr(A_base, 0);`
`q = p;`	`q = p;`
`foo(p);`	`foo(p);`
`p = q + 1;`	`p = int_ptr(q.base, q.offset + 1);`

`x = p[e];`	`if(p.base == A_base)`
	`  x = A[p.offset + e];`
	`else if (p.base == B_base)`
	`  x = B[p.offset + e];`
	`else`
	`  assert(false);`

`p[e] = d;`	`if(p.base == A_base)`
	`  A[p.offset + e] = d;`
	`else if (p.base == B_base)`
	`  B[p.offset + e] = d;`
	`else`
	`  assert(false);`

In this case, GPUVerify generates the following types:

```
enum int_ptr_base = { A_base, B_base, null } ;

struct int_ptr {
  int_ptr_base base;
  int offset;
 } ;
```

Thus, an integer pointer is modeled as a pair consisting of a base array, or the special value `null` if the pointer is null, and an integer offset from this base. To cater for arbitrary type-casting, the offset is in terms of bytes rather than array elements. GPUVerify incorporates an optimization to transform offsets to work at the level of array elements when it can be deduced using a simple type-based analysis that an array is always accessed at element-level granularity.

Pointers `p` and `q` can be assigned offsets from `A` or `B`, can be assigned `null`, or can be left uninitialized. An uninitialized pointer has an arbitrary value in the generated Boogie code. Table V shows how uses of `p` and `q` are translated into Boogie.

Statement `p = q + 1` demonstrates that pointer arithmetic is straightforward to model using this encoding. Pointer read and writes are modeled by a case split on all the possible bases for the pointer being dereferenced. If no base matches then the pointer is either uninitialized or `null`. These illegal dereferences are captured by an assertion failure. This encoding exploits the fact that in GPU kernels there are a finite, and usually small, number of explicitly declared pointer targets.

We deal with stack-allocated local variables whose addresses are taken by rewriting these variables as arrays of length one, and transforming the corresponding accesses to such variables appropriately. This suffices by the fact that GPU kernel languages do not permit recursion.

Points-to Analysis. The case-split associated with pointer dereferences can hamper verification of kernels with pointer-manipulating loops, requiring loop invariants that narrow down the permissible arrays to which pointers can refer. To avoid this in many cases, we have implemented the flow- and context-insensitive pointer analysis algorithm of Steensgaard [1996]. Although this overapproximates the points-to sets, our experience with GPU kernels is that aliasing is scarce and therefore precision is high. Returning to the aforementioned example, suppose the points-to analysis determines

```
void barrier();

shared bool gr[SZ][SZ];

void kernel() {
  int k = 0;
  while (k < SZ) {
    if (!gr[lidY][lidX]) {
      if (gr[lidY][k] && gr[k][lidX]) {
        gr[lidY][lidX] = true;
      }
    }
    barrier();
    k++;
  }
}
```

(a) Example kernel

```
void barrier();
void LOG_RD_gr(int y, int x);
void LOG_WR_gr(int y, int x);

shared bool gr[SZ][SZ];

void kernel() {
  int k = 0;
  while (k < SZ) {
    LOG_RD_gr(lidY, lidX);
    if (!gr[lidY][lidX]) {
      LOG_RD_gr(lidY, k);
      LOG_RD_gr(k, lidX);
      if(gr[lidY][k] && gr[k][lidX]) {
        LOG_WR_gr(lidY, lidX);
        gr[lidY][lidX] = true;
      }
    }
    barrier();
    k++;
  }
}
```

(b) Kernel after race instrumentation

Fig. 8. Example illustrating how GPUVerify instruments a kernel to detect races.

that `p` may only refer to array `A` (or be `null` or uninitialized). In this case, the assignment `p[e] = d` is translated to:

```
if(p.base == A_base)
  A[p.offset + e] = d;
else
  assert(false);
```

As well as checking for dereferences of `null` or uninitialized pointers, the `assert(false)` ensures that potential bugs in our implementation of the points-to analysis do not lead to unsound verification.

5.2. Reducing Data Race- and Barrier Divergence-Checking to Sequential Program Verification

Having compiled an OpenCL or CUDA kernel into corresponding Boogie form, GPUVerify attempts to verify the kernel. We describe the verification strategy employed by GPUVerify using a worked example.

Consider the kernel of Figure 8(a), adapted from part of a C++ AMP application that computes the transitive closure of a graph using Warshall's algorithm and simplified for ease of presentation. The kernel is written for a single, two-dimensional (2D) group of `SZ`×`SZ` threads. The local ID of a thread is 2D, with x and y components `lidX` and `lidY`, respectively. The kernel declares a 2D shared array of Booleans, `gr`, representing the adjacency matrix of a graph.

Access Logging Instrumentation. A kernel is first instrumented with calls to procedures that will log accesses to shared arrays. Figure 8(b) shows the example kernel of Figure 8(a) after access logging instrumentation. Observe for example that the

condition `gr[lidY][k] && gr[k][lidX]` involves two read accesses to `gr` and is thus preceded by two calls to `LOG_RD_gr`.[10]

Reduction to a Pair of Threads. After access logging, the kernel must be translated into a form that models the predicated execution of multiple threads. Initially, we attempted to encode the SDV semantics of Section 3 directly, modeling lock-step execution of all threads. Unfortunately, modeling in this way required heavy use of quantifiers, especially for implementing the G-SYNC rule of Figure 5 and associated merge_i function. This led to Boogie programs outside the decidable theory supported by the Z3 theorem prover. As a result, verification of small (micro-sized) kernels took in the order of minutes, while verification attempts for large kernels quickly exhausted memory limits.

To achieve better scalability, we instead use the two-thread reduction of Section 4 to transform a kernel into a form where the predicated execution of only two threads is modeled. The local and group identities of the two threads are represented by symbolic constants, constrained such that the threads are either in different groups or in the same group but with different local IDs.

Because a two-threaded predicated program with lock-step execution is essentially a *sequential* program consisting of parallel assignments to pairs of variables, reasoning about GPU kernels at this level completely avoids the problem of exploring interleavings of concurrent threads and allows us to leverage existing techniques for reasoning about sequential programs. Furthermore, because the IDs of the threads under consideration are symbolic, the sequential program can only be proven correct if correctness holds for all thread ID combinations. In practice, this is achieved by generating verification conditions that are discharged to an SMT solver.

GPUVerify supports both the adversarial and equality abstractions studied in Section 4. The equality abstraction is essential in verification of some kernels (including the kernel of Figure 8(a)) whose race-freedom hinges on threads agreeing on the values read from certain shared locations. In Section 6.2.3, we show that using the adversarial abstraction, when it suffices, typically proves to be more efficient than employing the equality abstraction. GPUVerify chooses the abstraction on array-by-array basis. We have implemented an inter-procedural taint analysis to overapproximate those shared arrays whose values may influence control flow. Arrays that may influence control flow are handled using the equality abstraction and all others using the adversarial abstraction. We study the precision of this heuristic experimentally in Section 6.2.3.

While the equality or adversarial abstractions suffice for verification of the vast majority of kernels we have studied, the equality abstraction is not sufficient when correctness depends on richer properties of the shared state. For instance, suppose a kernel declares shared arrays `A` and `B` and includes a statement:

```
A[B[lid]] = ...
```

Write-write race freedom of `A` requires that `B[i] != B[j]` for all distinct `i` and `j`. In practice, we have found that this prohibits verification of a number of kernels that perform a prefix sum operation into an array `B`, and then use `B` to index into an array `A` as just shown. A richer shared state abstraction is investigated by Chong et al. [2013].

Figure 9 shows the result of transforming the access-instrumented version of the kernel (Figure 8(b)) into a form where the predicated execution of a pair of arbitrary, distinct threads is modeled, using the equality abstraction. The transformation

[10]For ease of presentation, we ignore here the fact that, due to short-circuit evaluation, the read from `gr[k][lidX]` will not be issued if the value read from `gr[lidY][k]` is *false*. Our implementation does handle short-circuit evaluation correctly.

```
void barrier(bool en1, bool en2);
void LOG_RD_gr(bool en1, int y1, int x1, bool en2, int y2, int x2);
void LOG_WR_gr(bool en1, int y1, int x1, bool en2, int y2, int x2);
bool gr1[SZ][SZ], gr2[SZ][SZ];

void kernel() {
  int k1, k2;
  bool LC1, LC2, P1, P2, Q1, Q2; // Predicates
  // Assume the two group ids lie in appropriate range;
  // trivial in this example as there is only one group:
  assume(0 <= gid1 && gid1 < 1 && 0 <= gid2 && gid2 < 1);
  // Assume the thread ids lie in appropriate range:
  assume(0 <= lidX1 && lidX1 < SZ && 0 <= lidX2 && lidX2 < SZ);
  // Assume that if the two threads are in the same group (which, in this
  // single-group example, they must be) their local ids are distinct:
  assume((gid1 == gid2) ==> (lidX1 != lidX2 || lidY1 != lidY2));
  k1, k2 = 0, 0;
  LC1, LC2 = k1 < SZ, k2 < SZ;
  while (LC1 || LC2) {
    LOG_RD_gr(LC1, lidY1, lidX1, LC2, lidY2, lidX2);
    P1, P2 = LC1 && !gr1[lidY1][lidX1],
             LC2 && !gr2[lidY2][lidX2];
    LOG_RD_gr(P1, lidY1, k1, P2, lidY2, k2);
    LOG_RD_gr(P1, k1, lidX1, P2, k2, lidX2);
    Q1, Q2 = P1 && gr1[lidY1][k1] && gr1[k1][lidX1],
             P2 && gr2[lidY2][k2] && gr2[k2][lidX2];
    LOG_WR_gr(Q1, lidY1, lidX1, Q2, lidY2, lidX2);
    gr1[lidY1][lidX1], gr2[lidY2][lidX2] =
       Q1 ? true : gr1[lidY1][lidX1],
       Q2 ? true : gr2[lidY2][lidX2];
    barrier(LC1, LC2);
    k1, k2 = LC1 ? k1 + 1 : k1, LC2 ? k2 + 1 : k2;
    LC1, LC2 = LC1 && k1 < SZ, LC2 && k2 < SZ;
  }
}
```

Fig. 9. The kernel of Figure 8(a) after transformation to two-thread predicated form.

using the adversarial abstraction is identical, except that the arrays `gr1` and `gr2` are eliminated, and reads from these arrays are made nondeterministic.

The 2D local ID of the first thread is represented by `lidX1`, `lidY1`, and similarly for the second thread. The 1D group ID for the first thread is represented by `gid1`, and similarly for the second thread.

The first `assume` statement constrains the group IDs to lie within an appropriate range. As our worked example contains only a single group, the `assume` constrains each of `gid1` and `gid2` to be equal to zero.

The second `assume` statement constrains the local id components to lie within appropriate ranges. The 2D group of threads in our example has dimensions $\texttt{SZ}\times\texttt{SZ}$; thus, each local ID component must lie in the range $[0\,..\,\texttt{SZ}-1]$.

The final `assume` statement forces the group and thread IDs to be chosen such that if the threads are in the same group they cannot have identical local IDs. The left-hand-side of the implication holds in our example because, with only a single group, `gid1` and `gid2` must be equal; thus, the `assume` statement forces a difference in at least one of the pairs `lidX1`, `lidX2` or `lidY1`, `lidY2`.

Local variable `k` is duplicated, and the assignment `k = 0` is replaced with a parallel assignment, setting `k1` and `k2` to zero. The kernel declares fresh Boolean variables `LC`, `P`, and `Q` (duplicated for each thread). These are used to model predicated execution of the *while* loop (`LC`) and the outer and inner conditionals (`P` and `Q`, respectively). In the examples of Section 2 and in the operational semantics of Section 3, we specified that

under predicated execution, a *while* loop should continue to execute while there exists a thread for which the condition holds. In the presence of just two threads, existential quantification turns into disjunction, hence the loop condition `LC1 || LC2`.

In Figure 9, parameters to the `LOG_RD_gr` and `LOG_WR_gr` procedures are duplicated, with a parameter being passed for each thread. In addition, a *predicate* parameter, `en`, is passed for each thread, recording whether the thread is enabled during the call (cf. the incoming predicate p in rule K-CALL of Figure 6). If `LOG_RD_gr` is called with *false* as its `en1` parameter, this indicates that the first thread is not enabled, and thus a read should not be logged for this thread. Similarly, `barrier` is equipped with a pair of predicate parameters, `en1` and `en2`.

Handling Multiple Procedures. During the transformation to two-threaded form, the parameter list of each user-defined procedure is duplicated, and (as with the `LOG` and `barrier` procedures) enabled predicates are added for each thread. The procedure body is then translated to two-threaded, predicated form, with every statement guarded by the *enabled* predicate parameters. Correspondingly, actual parameters are duplicated at call sites, and the current predicates of the execution passed as *enabled* parameters.

Checking for Barrier Divergence. Under the two-thread encoding, inserting a check for barrier divergence is trivial: the `barrier` procedure merely asserts that its arguments `en1` and `en2` are equal if the group IDs of the two threads correspond. This two-threaded version of rules G-DIVERGENCE (Figure 5) and K-DIVERGENCE (Figure 6) precisely matches the notion of barrier divergence presented formally in Section 3. We may wish to only check barrier divergence-freedom for a kernel, if verifying race-freedom proves too difficult. This is sound under *adversarial* abstraction, where every read from the shared state returns an arbitrary value. A kernel that can be shown to be barrier divergence-free under this most general assumption is guaranteed to be barrier divergence-free under any schedule of shared state modifications. If we prove barrier divergence-freedom for a kernel under the equality abstraction, we can conclude a weaker property than barrier divergence-freedom: that barrier divergence cannot occur unless a data race has occurred. Note that our barrier divergence checking is stricter than that attempted by the PUG verifier of Li and Gopalakrishnan [2010], which merely requires threads that follow different conditional paths through a kernel to pass the same number of barriers.[11] While PUG reports microkernels exhibiting the barrier divergence bugs discussed in Section 2 as successfully verified, such kernels are rejected by GPUVerify.

Checking for Races. The `LOG_RD` and `LOG_WR` procedures are responsible for manipulating a read and write set for each thread and for each of the shared arrays of the kernel. According to the semantics of Section 3 (rules G-RACE and K-RACE of Figures 5 and 6, respectively), race checking involves asserting for each array A that the read and write sets for A do not conflict between threads.

We encode read and write sets efficiently by exploiting nondeterminism, similar to a method used in prior work by Donaldson et al. [2010, 2011]. For each shared array A with index type T, we introduce the following instrumentation variables for each thread i under consideration (where $i \in \{1, 2\}$):

$$\begin{aligned}
&\texttt{WR_exists_A}_i : \texttt{bool}\\
&\texttt{WR_elem_A}_i : \mathsf{T}\\
&\texttt{RD_exists_A}_i : \texttt{bool}\\
&\texttt{RD_elem_A}_i : \mathsf{T}
\end{aligned}$$

[11] In a subsequent paper on dynamic symbolic execution of CUDA kernels, Li et al. [2012b] improve this check to restrict to textually aligned barriers.

The Boolean WR_exists_A_i is set to *true* only if the write set of thread i for A is nonempty. In this case, WR_elem_A_i represents one element of this write set: an index into A. The corresponding RD variables for read sets are similar.

Initially WR_exists_A_i and RD_exists_A_i are false for each thread because the read /write sets are empty. The LOG_WR_A procedure then works as follows: for each thread i, if i is enabled on entry to the procedure (predicate parameter en$i$ is *true*), then the thread nondeterministically chooses to do nothing, or to set WR_exists_A_i to true and WR_elem_A_i to the index being logged. Procedure LOG_RD_A operates similarly. This strategy ensures that if WR_exists_A_i holds, then WR_elem_A_i is the index of an arbitrary write to A performed by thread i. Checking absence of write-write races can then be achieved by placing the following assertion in the LOG_WR_A procedure:

assert(!(WR_exists_A_1 && WR_exists_A_2 && WR_elem_A_1 == WR_elem_A_2))

A similar assertion is used to check read-write races; the procedure LOG_RD_A works analogously.

Because this encoding tracks an *arbitrary* element of each read and write set, if the sets can have a common, conflicting element this will be tracked by both threads along some execution trace, and the generated assertion will fail along this trace. If we can prove for every array that the associated assertions can *never* fail, we can conclude that the kernel is race-free.

Inter- vs. Intragroup Race Checking. A novel contribution of this article over prior work is a treatment of both intra- and intergroup races; prior work focused only on the intragroup case. We briefly recall how these kinds of races are handled by our semantics (Section 3.2). Intragroup data races are detected by rule G-Race (Figure 5) which compares the read and write sets of individual threads; an intragroup race detected by G-Race is detected at the kernel level by rule K-Intra-Group-Race (Figure 6). The role of barrier operations in avoiding races within a work group is captured at the group level by rule G-Sync (Figure 5): this rule accumulates the read and write sets of individual threads to form group-wide read and write sets, and then clears the individual thread read and write sets to reflect the fact that synchronization has taken place. The group-level read and write sets capture the fact that data races between threads in distinct groups cannot be avoided using barriers; rule K-Inter-Group-Race of Figure 6 detects intergroup races by comparing these sets.

We capture the intent of these semantic rules in our implementation by distinguishing between the cases where the two threads under consideration are in the same or different groups. If the threads are in the same group, then the instrumentation variables (WR/RD_exists, WR/RD_elem, across all arrays) play the role of thread-level read/write sets for checking intragroup data races. These sets must be cleared at barriers, and there is no need to record additional group-level read/write sets (because no further threads are directly considered). If the threads are in different groups then the instrumentation variables play the role of group-level read/write sets for checking inter-group data races. These sets must *not* be cleared at barriers, and there is no need to record additional thread-level read/write sets.

Thus, at a barrier, the instrumentation variables are unaffected if the threads under consideration are in different groups. If the threads are in the same group, read and write sets are cleared by assuming (using Boogie-level assume statements) that every WR_exists and RD_exists is false, that is, by terminating all execution paths along which read or written elements were logged. This matches the thread-level read and write set handling of rule G-Sync (Figure 5) and K-Sync (Figure 6).

Tolerating Benign Write-Write Races. In practice, it is quite common for threads to participate in benign write-write races, where identical values are written to a common

location without synchronization. When equality abstraction is used, GPUVerify tolerates this kind of race by adding a conjunct to the aforementioned assertion to check that the values written are not equal.

5.3. Invariant Inference

GPUVerify produces a Boogie program akin to the transformed kernel of Figure 9, together with implementations of the `barrier` and all `LOG_RD/WR` procedures. This program must be verified in order to prove race- and barrier divergence-freedom of the original kernel. Verification hinges on finding inductive invariants for loops and contracts for procedures.

We have found that invariant generation using abstract interpretation over standard domains (such as intervals or polyhedra) is not effective in verifying GPU kernels. This is partly due to the data access patterns exhibited by GPU kernels, discussed in detail later in the text, where threads do not tend to read or write from contiguous regions of memory, and also due to the predicated form of the programs produced by our verification method.

Instead, we use the Houdini algorithm of Flanagan and Leino [2001] as the basis for inferring invariants and contracts. Houdini is a method to find the largest set of inductive invariants from among a user-supplied pool of candidate invariants. Houdini works as a fixpoint procedure; starting with the entire set of invariants, it tries to prove that the current candidate set is inductive. The invariants that cannot be proved are dropped from the candidate set, and the procedure is repeated until a fixpoint is reached. We briefly discuss the relationship between Houdini and other invariant generation techniques in Section 7.

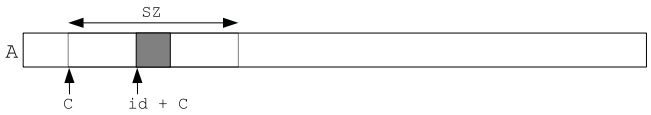
By manually deducing invariants for a set of kernels (the training set described in our experimental evaluation, Section 6), we have devised a number of candidate generation rules, which we outline in the following text. We emphasize that the candidate invariants generated by GPUVerify are just that: candidates. The tool is free to speculatively generate candidates that later turn out to be incorrect: these are simply discarded by Houdini. A consequence is that incorrect or unintended candidates generated due to bugs in GPUVerify cannot compromise the soundness of verification.

Our candidate generation rules are purely heuristic. The only fair way to evaluate these carefully crafted heuristics is to evaluate GPUVerify with respect to a large set of unknown benchmarks. We presented such an evaluation in [Betts et al. 2012], and we summarize the findings of this evaluation in Section 6.1.

Candidate Invariant Generation Rules. The following rules (except for the final "variable is zero or a power of two" rule) relate to the manner in which a thread accesses shared data using its thread id. In each case, the rule can be applied where thread ID denotes the local ID of a thread within its group, or the global ID of the thread across the kernel. We use `id` to generically describe both cases, and use `SZ` to denote the size of a thread group in the case that `id` refers to local ID, and the total number of threads in the case that `id` refers to global ID.

For clarity, we present the essence of each rule; the GPUVerify implementation is more flexible (e.g., being insensitive to the order of operands of commutative operations, and detecting when the ID of a thread has been copied into another local variable). For each of the rules associated with shared memory writes, there is an analogous rule for reads. We use ==> to denote implication.

Rule: access thread id plus offset. It is common for a thread to write to an array using its thread ID, plus a constant offset (which is often zero) as index; this access pattern is illustrated by the following diagram:



Observed pattern:

—`A[id + C] = ...` occurs in a loop

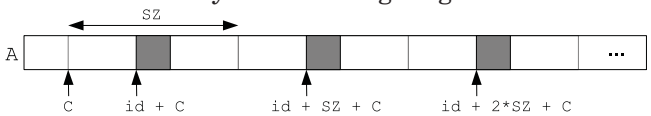
Generated candidate:

—`WR_exists_A ==> WR_elem_A - C == id`

Rule: access thread id plus strided offset. When processing an array on a GPU, it is typically efficient for threads in a group to access data in a coalesced manner as in:

```
for (i = 0; i < 256; i++)
  A[i*SZ + id + C] = ...;
```

This access pattern is illustrated by the following diagram:



Observed pattern:

—`A[id + i*SZ + C] = ...` occurs in a loop
—`i` is live at the loop head

Generated candidate:

—`WR_exists_A ==> ((WR_elem_A - C) % SZ) == id`

Rule: access at thread id plus strided offset, with strength reduction. This is similar to the previous rule. However, GPU programmers commonly apply the strength reduction operation manually, rewriting the aforementioned code snippet as follows:

```
for (i = id; i < 256*SZ; i += SZ)  A[i + C] = ...;
```

In this case, the write set candidate invariant will not be inductive in isolation: the invariant `(i % SZ) == id` is required in addition.

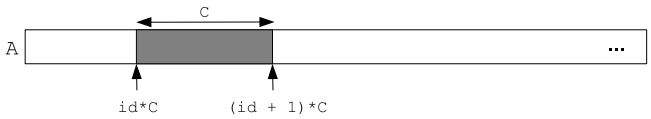
Observed pattern:

—`i = id` appears before a loop
—`A[i+C] = ...` occurs in the loop
—`i = i + SZ` appears in the loop
—`i` is live at the loop head

Generated candidates:

—`(i % SZ) == id`
—`WR_exists_A ==> ((WR_elem_A - C) % SZ) == id`

Rule: access contiguous range. It is common for threads to each be assigned a fixed-size chunk of an array to process. This access pattern is illustrated by the following diagram:



Observed pattern:

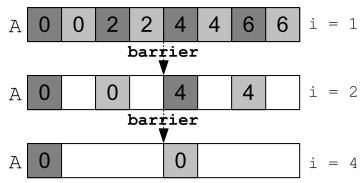
—`A[id*C + i] = ...` occurs in a loop
—`i` is live at the loop head

Generated candidates:

—`WR_exists_A ==> id*C <= WR_elem_A`
—`WR_exists_A ==> WR_elem_A < (id + 1)*C`

Rule: variable is zero or a power of 2. GPU kernels frequently perform tree reduction operations on shared memory, as in the following code snippet. Race-freedom is ensured through the use of a barrier, together with a guard ensuring that threads that have dropped out of the reduction computation do not write to shared memory. Verifying race-freedom requires an invariant stating that the loop counter is a power of 2, possibly allowing for the value to be zero. The access pattern for such a tree reduction with respect to a group of 8 threads (`SZ == 8`) is illustrated in the following. A gray square containing a thread id indicates a memory access by the associated thread; dark gray indicates both a read and a write, while light gray indicates a read only.

```
for(i = 1; i < SZ; i *= 2) {
  if((id % (2*i)) == 0) {
    A[id] += A[id + i];
     }
  barrier();
 }
```



Observed pattern:

—`i = i*2` or `i = i/2` occurs in a loop
—`i` is live at the loop head

Generated candidates:

—`i&(i-1)==0`
—`i!=0`

The first candidate caters for the case where `i` is either a power of 2 or zero; here, `&` denotes the bitwise-and operator. The second candidate encompasses the stronger condition where `i` is *not* zero.

GPUVerify includes a number of additional candidate generation rules that are intimately related to the details of our transformation of a kernel to a predicated sequential program. We omit details of these rules as they are very specific and less intuitive.

We have also designed rules to generate candidate pre- and postconditions for procedures. We do not discuss these rules: although they allow us to perform modular verification of some GPU kernels, we find that for our current benchmarks (which are representative of the sizes of today's GPU kernels), full procedure inlining yields superior performance to modular analysis.

6. EXPERIMENTAL EVALUATION

We have gathered a set of 162 kernels that have been used to drive the development of GPUVerify and to evaluate the capabilities of the tool in practice. The benchmarks come from four suites:

—**AMD SDK**[12]: AMD Accelerated Parallel Processing SDK v2.6 consisting of 71 publicly available OpenCL kernels.
—**CUDA SDK**[13]: Nvidia GPU Computing SDK v2.0 consisting of 20 publicly available CUDA kernels.
—**C++ AMP**[14]: Microsoft C++ AMP Sample Projects consisting of 20 publicly available kernels, translated to CUDA.
—**Basemark CL**[15]: Rightware Basemark CL v1.1 consisting of 51 commercial OpenCL kernels,[16] provided to us under an academic license.

We consider the somewhat out-of-date version 2.0 of the Nvidia SDK to facilitate a direct comparison of GPUVerify with the PUG tool by Li and Gopalakrishnan [2010], the only existing publicly available verifier for CUDA kernels, since PUG is not compatible with more recent versions of the CUDA SDK (the CUDA verification tool described by Leung et al. [2012] is not publicly available). We also restrict our attention to the benchmarks from this SDK which were used to evaluate PUG. Because GPUVerify cannot directly analyze C++ AMP code, we retrieved the set of C++ AMP samples available online on February 3, 2012, and manually extracted and translated the GPU kernel functions into corresponding CUDA kernels. This manual extraction and translation was straightforward.

We scanned each benchmark suite and removed kernels that are immediately beyond the scope of GPUVerify, either because they use atomic operations (7 kernels) or because they involve writes to the shared state using double-indirection as discussed in Section 5.2 (12 kernels). In recent work, we have considered prototype support for atomic operations in GPUVerify [Bardsley and Donaldson 2014]. A richer shared state abstraction to handle double-indirection is proposed by Chong et al. [2013], but requires significant manual effort to apply in practice. The number of kernels quoted above for each benchmark suite correspond to the sizes of these suites after removal of these kernels.

In Section 6.1, we summarize the results of an experiment using this benchmark set for a comparison of our initial implementation of GPUVerify with PUG and to evaluate the invariant inference capabilities of the initial implementation.

We then evaluate a more recent version of GPUVerify, equipped with new functionality for intergroup race checking (Section 6.2), assessing the performance of the tool across our benchmark set, the overhead associated with performing intergroup race checks, and the relative benefits in terms of precision and performance of the adversarial and equality abstractions.

Section 6.3 discusses a bug that we detected in an old CUDA SDK benchmark using GPUVerify. Section 6.4 demonstrates the effectiveness of the two-thread reduction.

[12]http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/.
[13]https://developer.nvidia.com/cuda-toolkit-archive
[14]http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx
[15]http://www.rightware.com/benchmarks/basemark-cl/.
[16]In [Betts et al. 2012] we reported results for 52 Rightware kernels, but we subsequently discarded one kernel from our benchmark set because of an issue, which we have reported to Rightware.

6.1. Summary of Previous Evaluation

In Betts et al. [2012], we presented a head-to-head comparison of the original GPUVerify implementation with the PUG tool of Li and Gopalakrishnan [2010] and evaluated the automatic invariant inference capabilities of GPUVerify. This evaluation focused on proving intragroup, but not intergroup, race-freedom, as well as barrier divergence-freedom. Instead of replicating this detailed evaluation here, we provide a brief summary of findings, preferring to focus on evaluating a more recent version of GPUVerify in Section 6.2.

The experimental evaluation we summarize here was performed on a PC with a 3.4GHz Intel Core i7-2600 CPU, 8GB RAM running Windows 7 (64-bit), with Z3 v3.3. All times reported are averages over three runs, using a 5-minute timeout per benchmark.

Comparing GPUVerify with PUG. We compared PUG and GPUVerify using the CUDA subset of our benchmarks: the CUDA SDK and C++ AMP suites (40 kernels), configuring PUG to use a 32-bit representation for integers, which is prescribed by the CUDA programming guide [Nvidia 2012a]. We used a synthetic bug injection procedure to randomly mutate each benchmark with an intragroup data race or barrier divergence bug to yield both correct and buggy versions of each benchmark.

Running the tools on correct kernels, we found that PUG reported a false positive in three cases, where correctness depended on threads agreeing on the contents of the shared state. GPUVerify is able to reason about these kernels using the equality abstraction (Section 4). The shared state abstraction of PUG is equivalent to our adversarial abstraction, which is not sufficient for these kernels. Note that GPUVerify decides automatically which shared state abstraction to use; we evaluate the heuristics GPUVerify uses to make this decision in Section 6.2.3. We found that PUG was, on average, faster than GPUVerify, and in six cases an order of magnitude faster. However, worst-case performance of PUG was significantly worse than that of GPUVerify: our timeout of 5 minutes was reached by PUG for six kernels.

Applying the tools to buggy versions of benchmarks, we found that the proof attempts by GPUVerify generally failed within around 5 seconds, whereas the proof attempts by PUG usually failed within half a second: an order of magnitude faster. However, for seven buggy kernels, we found that PUG reported false negatives: wrongly reporting correctness of the kernel. Of these false negatives, one mutation was an injected barrier divergence while the remaining six were intragroup data races. GPUVerify reported no false negatives.

Evaluating Automatic Invariant Inference. We used the following methodology to design and evaluate our invariant inference technique. We divided our benchmarks into two similarly sized sets: a *training set* and an *evaluation set*, such that details of the evaluation set were unknown to all members of our team. We chose the CUDA SDK, C++ AMP, and Basemark CL benchmarks as the training set (92 kernels), and the AMD SDK benchmarks as the evaluation set (70 kernels). Members of our team had looked previously at the CUDA SDK and C++ AMP benchmarks but not at the AMD SDK and Basemark CL benchmarks; however, we wanted to make the evaluation set publicly available, ruling out Basemark CL.

We manually analyzed all benchmarks in the training set, determining invariants sufficient for proving intragroup race- and barrier divergence-freedom. We then distinguished between "bespoke" invariants: complex, kernel-specific invariants required by individual benchmarks; and "general" invariants, conforming to an identifiable pattern that cropped up across multiple benchmarks. The general invariants led us to devise the invariant inference heuristics described in Section 5.3. We implemented these heuristics in GPUVerify and tuned GPUVerify to maximize performance on the training set.

We then applied GPUVerify blindly to the evaluation set to assess the extent to which our inference technique enabled fully automatic analysis of the AMD SDK kernels. We believe that this approach of applying GPUVerify unassisted to a large, unknown set of benchmarks provides a fair evaluation of the automatic capabilities of the tool.

Using the inference techniques devised with respect to the training set (cf. Section 5.3), GPUVerify was able to verify 49 of the 70 kernels from the evaluation set (69%) fully automatically. Of these kernels, 48 were verified in 10 seconds or less, and the longest verification time was 17 seconds. With race checking disabled, GPUVerify was able to prove barrier divergence-freedom fully automatically for all evaluation benchmarks, in under 10 seconds per kernel. In Betts et al. [2012], we present a detailed discussion of the reasons why verification failed for 22 of the evaluation kernels.

Many modern static analysis tools achieve low false alarm rates via a careful mixture of deliberately introduced unsoundness in the analysis and ad hoc warning suppression [Bessey et al. 2010]. GPUVerify does not follow this approach: the tool attempts to be a "real" verifier, and thus will report verification failure for a kernel unless it was possible to construct a proof of correctness in a sound manner under bit-level accuracy. With this in mind, we believe that being able to verify 49 out of 70 evaluation kernels is a good result.

6.2. Evaluation of GPUVerify

Building on the evaluation presented in Betts et al. [2012], we manually investigated each benchmark for which intra-group race-freedom could not be automatically verified. In each case, we either provided necessary invariants manually to allow the kernel to verify, or we improved the invariant inference capabilities of GPUVerify by adding additional candidate generation rules or generalizing existing rules. These additional and changed candidates all intimately relate to the details of our transformation from a kernel to a predicated sequential program, and details are omitted for that reason. We then turned on the inter-group race checking capabilities of GPUVerify that are novel in this work (see Section 5.2), and again manually supplied invariants and improved invariant inference capabilities until intergroup race-freedom could also be verified across the set of benchmarks.

In three cases, we had to make minor simplifications to allow correct kernels to be verified within our framework. In two binary search kernels from the AMD SDK, we had to insert *assume* statements to encode the precondition that the input array is strictly sorted. Another AMD SDK kernel that computes a quasi-random sequence exhibited an intricate, intentional benign read-write race, which is beyond the scope of the benign tolerance mechanism mentioned in Section 5.2; we inserted barriers to eliminate this race.

We also had to modify a number of examples to remove data races. In a histogram example from the AMD SDK, we found an arguably benign data race where threads nonatomically increment the same bucket of a histogram without synchronization. Such races can lead to increments to a bucket being lost nondeterministically, but in many uses of histograms, this does not matter. For purposes of analysis, we simplified this example so that threads write to disjoint locations; this changes the meaning of the kernel, but retains the features of the kernel that determine whether it is a challenging candidate for proving race-freedom. An AMD SDK kernel implementing a pass of the Floyd-Warshall algorithm adds together input values, which can cause an overflow-induced data race for extremely large inputs. Contacts at AMD regarded this data race report as a false alarm, which we suppressed through the use of an addition operator that assumes nonoverflowing behavior. Using GPUVerify, we found a *non-benign* data race bug in one AMD SDK benchmark, `DwtHaar1D`, arising from

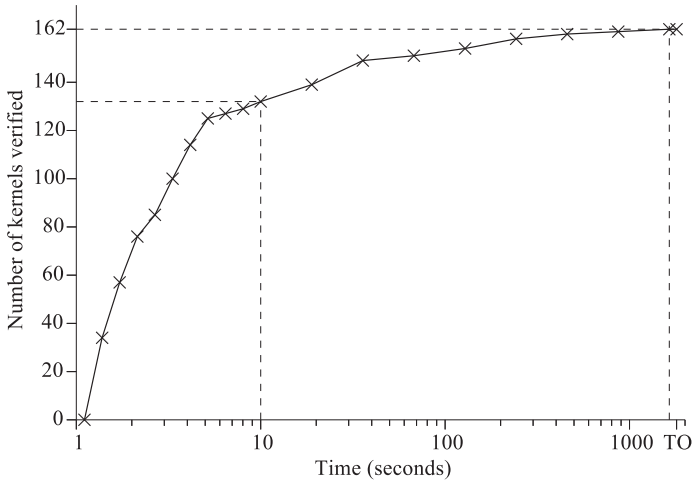


Fig. 10. Cumulative histogram showing the time taken to verify data race- and barrier divergence-freedom using GPUVerify across our benchmark set.

an erroneous array indexing expression, which has since been independently fixed by AMD in a more recent version of the SDK.

With respect to this refined benchmark set and tool chain, we now report on:

—The raw performance of GPUVerify across our benchmarks (Section 6.2.1)
—The overhead associated with performing intergroup race checking in addition to intragroup race checking (Section 6.2.2)
—The performance and precision differences associated with using the equality versus adversarial shared state abstractions (Section 6.2.3)

Experiments were performed on a PC with a 1.15GHz AMD Phenom CPU, 5.8GB RAM running Ubuntu (64-bit), using the nightly build of GPUVerify from August 8, 2013, and Z3 v4.3.1. All times reported are averages over 10 runs, and a timeout of 1,800 seconds (30 minutes) was used in all experiments.

6.2.1. Performance of GPUVerify. Figure 10 is a cumulative histogram showing the performance of GPUVerify with respect to the benchmark set. The x-axis plots the time (in seconds, on a log scale), and the y-axis plots the number of kernels. A point at position, (x, y) indicates that for y of the kernels, verification took x seconds or less. The results show that GPUVerify is capable of rapidly analyzing the vast majority of benchmark kernels: in 132 out of 162 cases, verification took less than 10 seconds. The longest verification time was 1,643 seconds, for analysis of an FFT kernel in the AMD SDK. Interestingly, the FFT kernel is loop-free, thus the high verification time stems solely from the cost of data race checking, and not from the inference of loop invariants.

6.2.2. Overhead of Intergroup Race Checking. A new addition to GPUVerify over [Betts et al. 2012] is the ability to perform intergroup race checking in addition to checking for races between groups. Intergroup race checks are represented by additional assertions in the Boogie program generated by GPUVerify, which in turn lead to larger verification conditions. Our hypothesis was that enabling intergroup race checking would uniformly lead to a drop in the performance of GPUVerify.

Figure 11 compares the performance of GPUVerify with and without intergroup race checking enabled across our benchmark set. Each point represents a benchmark. The x-axis plots the time in seconds taken to verify full (i.e., inter- and intragroup) race

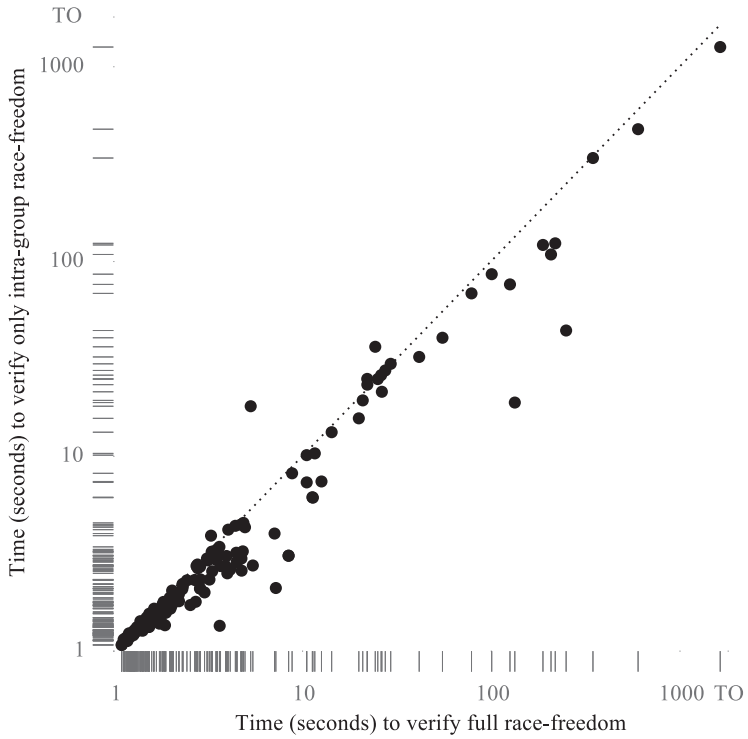


Fig. 11. Scatter plot comparing time taken to verify full race-freedom versus only intragroup race-freedom across our benchmark set.

freedom, and the y-axis plots the time in seconds taken to verify only intragroup race-freedom. Both axes are plotted using log scales. Thus, a point with coordinates (x, y) corresponds to a kernel for which verification of full race-freedom took x seconds and verification of only intragroup race-freedom took y seconds. Points lying below/above the diagonal support/rebut our hypothesis that there is computational overhead associated with adding inter-group race checks.

The data in Figure 11 strongly supports our hypothesis: in all but two cases, verification time either differs negligibly or slows down when intergroup race checking is enabled.

There are two outliers that do not follow this trend: verification of a histogram kernel from the CUDA SDK takes 3.4× *longer* when only intragroup races are analyzed than when full race checking is performed; similarly a bitonic sort kernel from the AMD SDK takes 1.5× longer. We attribute these outliers to quirks in the implementation of the Z3 theorem prover: we tried the same experiment using a different theorem prover, CVC4 [Barrett et al. 2011], finding that the histogram kernel is not an outlier when CVC4 is used, leaving the bitonic sort kernel as the only outlier, with a slightly reduced slowdown of 1.3×.

6.2.3. Adversarial vs. Equality Abstraction. In Section 4, we discussed two shared state abstractions: the very coarse adversarial abstraction and the slightly more refined equality abstraction. Treating an array using the adversarial abstraction has the potential advantage that the array can be completely removed from the Boogie program generated by GPUVerify. This reduces the extent to which the theorem prover must reason about arrays, which can be computationally expensive. Having observed this

advantage working in practice for specific examples, we implemented the taint analysis mentioned in Section 5.2, which GPUVerify employs to avoid use of the equality abstraction when it does not appear necessary.

We now evaluate both the effectiveness of GPUVerify in automatically deciding when the equality abstraction is necessary, and our hypothesis that it is desirable to avoid using the equality abstraction where possible.

To this end, we ran GPUVerify across the benchmark set in two modes, one where the adversarial abstraction is always used for every array, the other where the equality abstraction is always used for every array, adding to the data gathered in Section 6.2.1 where GPUVerify used its default heuristic to choose which abstraction to employ on an array-by-array basis.

Precision of Taint Analysis for Abstraction Choice. We found that for 155 out of 162 kernels, verification of full data race- and barrier divergence-freedom succeeded even when the adversarial abstraction was universally forced. In 130 of these cases, our taint analysis also led to the adversarial abstraction being used for all arrays, but in 25 cases, GPUVerify decided, unnecessarily, that at least one array should be modeled using the equality abstraction. This is because of the conservative nature of our taint analysis which is neither flow- nor path-sensitive; a more precise taint analysis could allow more aggressive application of the adversarial abstraction.

Among the seven kernels that required equality abstraction to be used for at least one array, our taint analysis succeeded in three cases to apply the equality abstraction sufficiently. In 4 cases, however, our taint analysis failed to choose the equality abstraction for certain arrays, leading to false-positive data race reports. These kernels were: the two binary search kernels discussed in Section 6.2, which depend on properties of the input array that the adversarial abstraction loses; the kernel implementing a pass of the Floyd-Warshall algorithm (discussed in relation to overflow in Section 6.2), for which race-freedom hinges on threads agreeing on the contents of a shared array; a parallel scan kernel exhibiting a benign data race that GPUVerify is only able to tolerate with the knowledge that a shared location is regarded as equal between threads. (In these cases, we used an option to override the taint analysis, using the equality abstraction for all arrays, when gathering data for the results in Section 6.2.1.)

Our evaluation shows that for our benchmark set, GPUVerify applies the equality abstraction relatively judiciously, capturing some but not all of the cases where it is useful.

Performance Comparison between Adversarial and Equality Abstractions. For the 155 kernels for which adversarial abstraction suffices for verification, we investigated our hypothesis that using the equality abstraction would lead to slower verification. We did this by comparing verification times for the extreme cases where either the adversarial abstraction is forced for every array, or the equality abstraction is forced for every array. (By default, GPUVerify chooses which of these abstractions to apply on an array-by-array basis.) Note that the equality abstraction is strictly finer than the adversarial abstraction, so that if a kernel can be verified using the adversarial abstraction for all arrays, it must also be possible to verify the kernel (given sufficient resources) using the equality abstraction for all arrays.

The scatter plot of Figure 12 uses log scales to plot the time in seconds taken for verification using the adversarial abstraction (x-axis) and equality abstraction (y-axis). Each point is a benchmark; thus, a point at coordinates (x, y) indicates that verification took x seconds using the adversarial abstraction and y seconds using the equality abstraction. Points with y coordinates at the top of the graph indicate that the timeout of 30 minutes was reached when using the equality abstraction. Points lying above the diagonal support our hypothesis that using the equality abstraction incurs a performance penalty in terms of verification time.

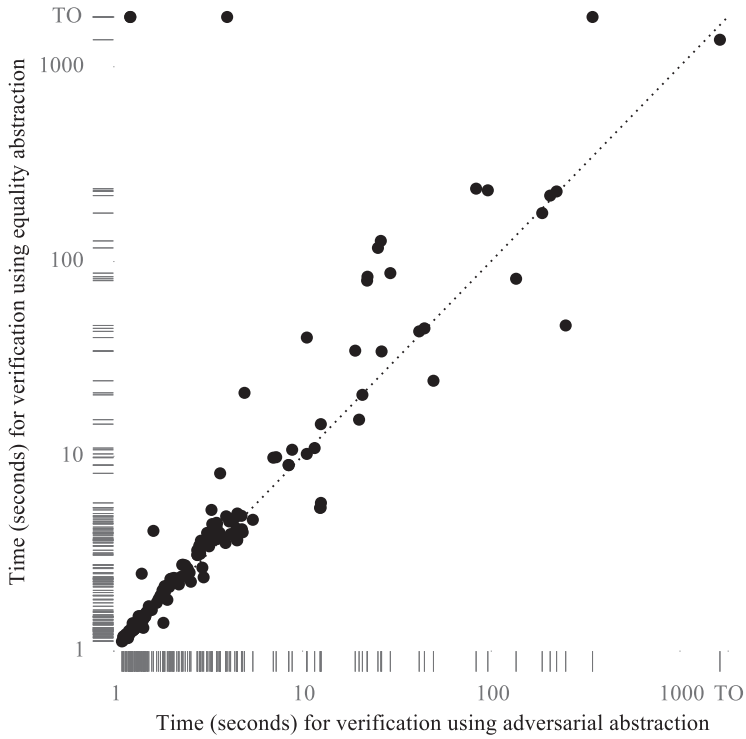


Fig. 12. Scatter plot comparing verification times using the adversarial and equality abstractions. Benchmarks for which verification using the adversarial abstraction fails (seven cases) are not included in the comparison.

Our hypothesis is mainly supported: ignoring the conglomeration of benchmarks clustered at the bottom left of Figure 12 that verify quickly with either abstraction, most benchmarks verify more efficiently when the adversarial abstraction is used. There are four cases where applying the equality abstraction leads to our timeout of 30 minutes being reached.

There are, however, a number of cases where we found verification to be faster when the equality abstraction is used. There are five outliers in Figure 12 for which verification was at least 2× faster using the equality abstraction than with adversarial abstraction; in the worst case, verification is 5.3× slower. As in our discussion of outliers when comparing full versus only intragroup race checking in Section 6.2.2, we attribute this to quirks of Z3. Again, we tried the same experiment using CVC4, obtaining a rather different set of outliers, which, although larger, included just two of the five outliers observed with respect to Z3. With sufficient processing power, we could guard against abstraction and solver quirks through portfolio solving, running multiple verification instances with different solvers in parallel.

6.3. Detection of a Bug in a Previous CUDA SDK Example

Using GPUVerify, we discovered a write-write data race in the N-body example that shipped with version 2.3 of the CUDA SDK. This example uses multiple CUDA kernels to numerically approximate a system of N interacting bodies [Nyland et al. 2007]. This is an ideal problem for parallelization because interactions between each pair of bodies can be calculated independently. The CUDA implementation of this example decomposes the N^2 pair-interactions into smaller $k \times k$ tiles, each of which is assigned

```
shared int A[N];

void kernel() {
  for(int d = N/2; d > 0; d = d / 2) {
    if(lid < d) {
      A[lid] += A[lid + d];
    }
    barrier();
  }
}
```

Fig. 13. A reduction kernel used to illustrate the effectiveness of the two-thread reduction.

to a 1D group of k threads. Within each group, every thread is assigned to a distinct body (a row of the tile) and sequentially considers the interactions associated with this body to compute an updated state for the body. The kernel implements an optimization for small values of N where threads are arranged in 2D groups, and multiple threads within a group are assigned to the same body. Consequently, the interactions calculated by threads assigned to the same body must be summed. A barrier ensures that each thread has completed its subcalculation, and then a conditional is used to ensure that a single "master" thread performs the summation. However, a data race could occur because a similar condition was not in place to ensure that only this master thread would perform a final update to the position and velocity of the body. As a result, it was possible for the final update by the master thread, using the full summation, to be overwritten by partial results computed by other threads.

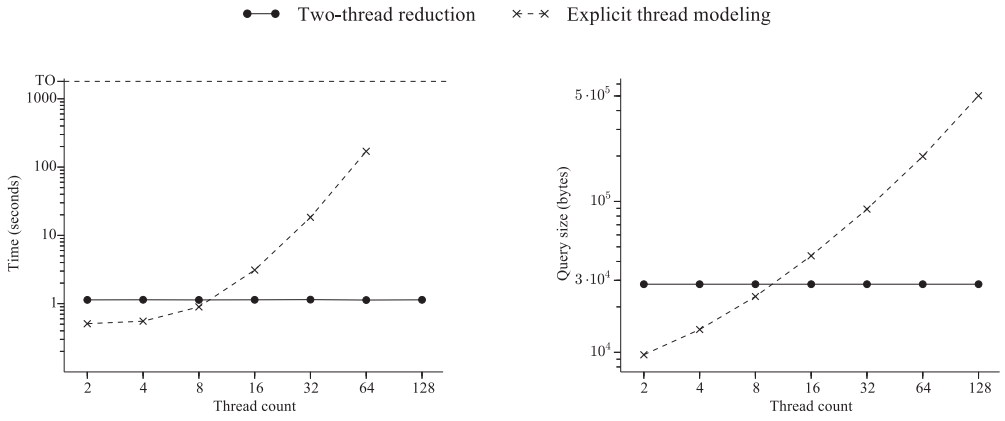
We reported this data race to Lars Nyland [2012] at Nvidia who confirmed that "it was a real bug, and it caused real issues in the results. It took significant debugging time to find the problem." Nvidia had subsequently fixed this bug in version 3.0 of the CUDA SDK.

6.4. A Demonstration of the Effectiveness of the Two-Thread Reduction

At the start of Section 4, we remarked that reasoning about GPU kernels using the SDV semantics of Section 3 directly—without reducing the number of threads that are modeled—does not scale. To justify this remark, we consider the kernel of Figure 13, which computes a reduction over an input array `A`. The kernel is designed to be executed by a single group of `N` threads, where `N` is a power of 2, with `lid` denoting the local ID of a thread.

We wrote a Python script that takes as input an integer N and outputs a Boogie program representing the kernel of Figure 13 modeled using SDV semantics, executed explicitly by N threads (i.e., without applying the two-thread reduction). For successively large values of N, we compared the performance of Boogie on these generated programs versus the performance of GPUVerify on the original kernel; GPUVerify does apply the two-thread reduction and uses Boogie in its backend. To ensure a fair comparison, we disabled invariant inference in GPUVerify and annotated both the generated Boogie programs and the kernel analyzed by GPUVerify with invariants necessary for verification. We used a 30-minute timeout for each verification task.

Figure 14(a) compares verification times. The x-axis plots the number of threads; the y-axis plots verification time in seconds. Both axes use a log scale. The solid line shows that verification with GPUVerify, using the two-thread reduction, remains virtually constant. On the other hand, the dashed curve shows that verification time grows rapidly (roughly exponentially) with explicit modeling as the thread count increases, exceeding our timeout with 128 threads.



(a) As the thread count increases, verification time using GPUVerify (solid line) is virtually unaffected; this is due to the two-thread reduction. In contrast, verification time increases dramatically according to thread count when threads are explicitly modeled (dashed line).

(b) The timing results of Figure 14(a) can be linked to the SMT queries associated with each verification task; these queries grow quadratically with thread count when explicit modeling is used, but remain constant when the two-thread reduction is employed.

Fig. 14. Comparing verification time and SMT query size when Boogie programs associated with the example kernel of Figure 13 are analyzed, with and without the two-thread reduction, for increasing thread counts.

This phenomenon is explained by Figure 14(b) in which the x-axis again plots the number of threads and the y-axis plots the size in bytes of the SMT query generated by Boogie for each verification problem. The formulas generated by GPUVerify (represented by the solid line) have identical size; they differ only in the constant that records thread count, which is a fixed-size bit-vector. In contrast, the formulas associated with explicit modeling (represented by the dashed line) become larger as the thread count increases. The growth is quadratic: a quadratic number of constraints are required to ensure that all the threads under consideration have distinct IDs. This growth in formula size places an increasing burden on the SMT solver.

7. RELATED WORK

There are numerous existing dynamic and static techniques for data-race detection in programs that use lock-based synchronization or fork-join parallelism; a full discussion of these techniques is beyond the scope of this article. We note, however, that this article is concerned with proving data race- and barrier divergence-freedom in data-parallel programs in which the primary challenges—barrier synchronization and disjoint access patterns based on clever array indexing—are different from those encountered in lock-based and fork-join programs. In the rest of this section, we discuss papers that explicitly handle data-parallel or GPU programs, and the relationship between the verification method employed by GPUVerify and comparable proof techniques used in protocol verification. We conclude the section with a brief discussion of invariant generation techniques.

Static Verification of GPU Kernels. The closest work to GPUVerify is the PUG analyzer for CUDA kernels by Li and Gopalakrishnan [2010]. Although GPUVerify and PUG have a similar goal, scalable verification of GPU kernels, the internal architecture of the two systems is very different. GPUVerify first translates a kernel into a

sequential Boogie program that models the lock-step execution of two threads; the correctness of this program implies data race- and barrier divergence-freedom of the original kernel. Next, it infers and uses invariants to prove the correctness of this sequential program. Therefore, we only need to argue soundness for the translation into a sequential program; the soundness of the verification of the sequential program follows directly from the soundness of contract-based verification. On the other hand, PUG performs invariant inference while simultaneously translating the GPU kernel into a logical formula. PUG provides a set of built-in loop summarization rules that replace loops exhibiting certain shared array access patterns with corresponding invariants. Unlike GPUVerify, which must prove or discard all invariants that it generates, the loop invariants inserted by PUG are assumed to be correct. While this approach works for simple loop patterns, it has difficulty scaling to general nested loops in a sound way, and this necessitates various restrictions on the input program required by PUG. In contrast, GPUVerify inherits flexible and sound invariant inference from Houdini, regardless of the complexity of the control structure of the GPU kernel.

A recent method for functional verification of GPU kernels by Blom et al. [2014] uses permissions-based separation logic. Here, a kernel must be annotated (currently manually) with an assignment of read or write permissions to memory locations on a per-thread basis. Race-freedom is proven by showing that having permission to write a location excludes other threads from having permission to read or write that location. To reason about communication at barriers, a barrier specification is required, to state how permissions are exchanged between threads at a barrier. The approach of Blom et al. [2014] does not employ any thread-reduction abstraction; instead, quantifiers are used to reason about all threads. This method has not yet been automated, so a systematic comparison with GPUVerify for realistic examples is not yet possible.

In related work, we have extended the GPU verification method with barrier invariants [Chong et al. 2013], which go beyond the equality abstraction described in this work, allowing more sophisticated invariants about the shared state to be established and used in verification. While the emphasis of this article is on highly automatic verification of lightweight correctness properties, the focus of Chong et al. [2013] is on manual derivation of intricate barrier invariants to establish richer functional properties which can be used in race analysis of complex data-dependent kernels. We have also studied the generalization of lock-step execution from structured programs (as presented here) to unstructured control flow graphs [Collingbourne et al. 2013], and the current implementation of GPUVerify is based on this extension.

Formal Semantics for GPU Kernels. Habermaier and Knapp [2012] study the relationship between the Single Instruction Multiple Thread (SIMT) execution model of Nvidia GPUs and the standard interleaved semantics of threaded programs, presenting a formal semantics for predicated execution. This semantics shares similarities with the SDV semantics we present in Section 3, but the focus of Habermaier and Knapp [2012] is not on verification of GPU kernels. Recent work by Kojima and Igarashi [2013] provides a formulation of Hoare logic geared toward the SIMT model, in a similar spirit to Habermaier and Knapp [2012].

Symbolic Execution and Bounded-Depth Verification. The GKLEE tool by Li et al. [2012b] and KLEE-CL by Collingbourne et al. [2012] perform dynamic symbolic execution of CUDA and OpenCL kernels, respectively, and are both built on top of the KLEE symbolic execution engine [Cadar et al. 2008]. A method for bounded verification of barrier-free GPU kernels via depth-limited unrolling to an SMT formula is discussed by Tripakis et al. [2010]; lack of support for barriers, present in most nontrivial GPU kernels, limits the scope of this method. Symbolic execution and bounded unrolling

techniques can be useful for bug-finding—both GKLEE and KLEE-CL have uncovered data race bugs in real-world examples—and these techniques have the advantage of generating concrete bug-inducing tests. The major drawback to these methods is that they cannot verify freedom from defects for nontrivial kernels.

The GKLEE tool specifically targets CUDA kernels, and faithfully models lock-step execution of subgroups of threads, or *warps* as they are referred to in CUDA (see Table II). This allows precise checking of CUDA kernels that deliberately exploit the warp size of an Nvidia GPU to achieve high performance. By default, GPUVerify makes no assumptions about subgroup size, making it useful for checking whether CUDA kernels are portable; GPUVerify includes an option to enable warp-level reasoning [Bardsley and Donaldson 2014].

Both GKLEE and KLEE-CL explicitly represent the number of threads executing a GPU kernel. This allows for precise defect checking but limits scalability. A recent extension to GKLEE by Li et al. [2012a] uses the notion of parametric flows to soundly restrict defect checking to consider only certain pairs of threads. This is similar to the two-thread abstraction employed by GPUVerify and PUG, and leads to scalability improvements over standard GKLEE, at the expense of a loss in precision for kernels that exhibit interthread communication (GPUVerify and PUG also suffer from this loss in precision, as discussed under *Incompleteness of the Two-Thread Reduction* in Section 4.1).

The GKLEE tool has also been extended with support for reasoning about atomic operations [Chiang et al. 2013]. Atomic operations are challenging because they allow race-free nondeterminism between pairs of barriers, destroying the ability to reason about a single canonical thread schedule, an assumption on which PUG, GPUVerify, GKLEE, and KLEE-CL all rest. GKLEE supports bounded reasoning about atomics, geared toward bug-finding, through the use of *delay bounding* [Emmi et al. 2011]. GPUVerify includes prototype support for reasoning about kernels that manipulate data using atomic operations, through a refinement to the adversarial abstraction [Bardsley and Donaldson 2014]. Neither GPUVerify nor GKLEE provides the necessary support for sound reasoning about fine-grained concurrency in GPU kernels which, as demonstrated in recent work, can expose the relaxed memory consistency models of modern GPU architectures [Alglave et al. 2015].

Dynamic Analysis. Dynamic analysis of CUDA kernels for data race detection has been proposed by Boyer et al. [2008]. Leung et al. [2012] report on the following technique that combines dynamic and static data race analysis. They first simulate a CUDA kernel while dynamically checking for races. If no races are detected, flow analysis is used to determine whether the control flow taken during dynamic execution was dependent on input data; if not, the kernel can be deemed race free, otherwise the technique is inconclusive. It appears that this approach can handle kernels that are verifiable using our adversarial abstraction. Kernels that GPUVerify can verify only with the equality abstraction, due to threads testing input data, are not amenable to analysis using this technique. As discussed in Section 6, the tool associated with Leung et al. [2012] is not publicly available; hence, we do not present an experimental comparison between this work and GPUVerify.

A problem associated with static analysis can be the lack of adequate preconditions for reasoning about the correctness of a procedure. A pragmatic method for overcoming this problem when analyzing OpenCL kernels with GPUVerify is *kernel interception*, where dynamic analysis is used to capture the values of scalar parameters supplied to a kernel at runtime, as well as the specific configuration of threads and groups with respect to which the kernel executes [Bardsley et al. 2014b]. This dynamically collected information provides a very specific precondition under which a kernel can

be analyzed. This has the property of lowering the rate of false positives reported by GPUVerify, at the expense of making the verification result specific to the particular runtime configuration that was captured.

Thread Contracts. Karmani et al. [2011] present a practical approach for constructing correct parallel programs, based on *thread contracts*. A programmer specifies the coordination and data sharing strategy for their multithreaded program as a contract. Adequacy of the specification for ensuring race-freedom is then checked statically, while adherence to the specification by the implementation is ascertained via testing. Adapted to the setting of barrier synchronization rather than lock-based coordination, this technique might enable analysis of more complex GPU kernels for which automatic contract inference is infeasible.

Protocol Verification. A reduction to two processes, similar to the two-thread reduction employed in GPU kernel verification, is at the heart of a method for verifying cache coherence protocols known as CMP [Chou et al. 2004], which was inspired by the foundational work of McMillan [1999]. With CMP, verification of a protocol for an arbitrary number of processes is performed by model checking a system where a small number of processes are explicitly represented and a highly nondeterministic "other" process overapproximates the possible behaviors of the remaining processes. The unconstrained nature of the "other" process can lead to spurious counterexamples, which must be eliminated either by introducing additional explicit processes or by adding noninterference lemmas such that the actions of the "other" process more precisely reflect the possible actions of processes in the concrete system. The CMP method has been extended and generalized with message flows and message flow invariants by Talupur and Tuttle [2008]. This extension aids in the automatic derivation of noninterference lemmas by capturing large classes of permissible interactions between processes. Our approach, and that of PUG by Li and Gopalakrishnan [2010], uses the same high-level proof idea as the CMP method: we consider a small number of threads (two), and our default adversarial abstraction models the possible actions of all other threads, analogously to the "other" process. There are, however, many technical differences in the manner by which the high-level proof technique is applied in practice. Principally, we use predicated execution to turn the verification problem into a sequential program analysis task, completely eliminating the need to reason about concurrent threads, while the CMP method involves directly checking the parallel composition of a number of processes using model checking.

Invariant Generation. As described in Section 5.3, we use the Houdini algorithm of Flanagan and Leino [2001] to generate loop invariants for verification. Houdini was introduced as an annotation assistant for the Java Extended Static Checker [Leino et al. 2000]. Related template-based invariant generation techniques include Kahsai et al. [2011], Srivastava and Gulwani [2009], and Lahiri and Qadeer 2009]. As discussed in the related work section of Flanagan and Leino [2001], Houdini can be viewed under the framework of abstract interpretation [Cousot and Cousot 1977] where the abstract domain comprises conjunctions of predicates drawn from the set of candidate invariants. Compared with standard predicate abstraction [Graf and Saïdi 1997], which considers arbitrary Boolean combinations of predicates (and is thus more precise), verification using Houdini requires a linear instead of exponential number of theorem prover calls. In our context, the key advantage of the Houdini approach over traditional abstract interpretation using a fixed abstract domain is flexibility. We can easily extend GPUVerify with a richer language of predicates by adding further candidate invariant generation rules; there is no need for careful redesign of an abstract domain.

The main problem with our invariant generation method is that its success is limited by the scope of our candidate invariant generation rules. Interpolation and counterexample-guided abstraction refinement can be used incrementally to generate invariants in response to failed or partial verification attempts [McMillan 2006; Beyer et al. 2007], while the Daikon technique [Ernst et al. 2007] allows program-specific invariants to be speculated through dynamic analysis. It may be possible to draw upon these techniques to improve the invariant inference capabilities of GPUVerify.

8. CONCLUSIONS

We have presented the design and implementation of the GPUVerify method for statically verifying data race- and barrier divergence-freedom of GPU kernels. The foundations of the method are our novel SDV semantics and the two-thread reduction, which allows scalable verification to be built on the semantic foundations. We have presented a rigorous soundness proof for the two-thread reduction, partially mechanized using Isabelle, and have discussed practical issues associated with implementing the GPUVerify tool. Our large experimental evaluation demonstrates that GPUVerify is effective in verifying and falsifying real-world OpenCL and CUDA GPU kernels.

Our current and planned future activities are mainly focused on improving the automatic invariant inference facilities of GPUVerify to lower the rate of false positives reported by the tool; we believe this is key for industrial uptake of the method.

ACKNOWLEDGMENTS

Our thanks to Guodong Li and Ganesh Gopalakrishnan for providing us with the latest version of PUG, and for answering numerous questions about the workings of this tool.

Thanks to Matko Botinčan, Mike Dodds, Hristina Palikareva and the anonymous reviewers of the conference version of the article for their insightful feedback on an earlier draft of this work.

We are also grateful to the anonymous TOPLAS reviewers for their useful feedback on the extended version.

We thank several people working in the GPU industry who provided assistance with this work: Anton Lokhmotov (ARM) provided us with barrier divergence results for ARM's Mali architecture; Lee Howes (AMD) provided such results for AMD's Tahiti architecture; Teemu Uotila and Teemu Virolainen (Rightware) provided access to Basemark CL, and answered various queries about these kernels; Yossi Levanoni (Microsoft) provided early access to C++ AMP samples.

We are grateful to Peter Collingbourne for leading development of Bugle, an LLVM IR-based front-end for GPUVerify which replaced the Clang AST-based frontend described in the conference version of this article [Betts et al. 2012].

REFERENCES

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU concurrency: Weak behaviours and programming assumptions. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, 577–591.

AMD. 2013. OpenCL Programming Guide, Revision 2.7. Retrieved from http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf.

Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. 2014a. Engineering a static verification tool for GPU kernels. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*. Lecture Notes in Computer Science, Vol. 8559. Springer, Berlin, 226–242.

Ethel Bardsley and Alastair F. Donaldson. 2014. Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In *Proceedings of the 6th NASA Formal Methods Symposium (NFM'14)*. Lecture Notes in Computer Science, Vol. 8430. Springer, Berlin, 230–245.

Ethel Bardsley, Alastair F. Donaldson, and John Wickerson. 2014b. KernelInterceptor: Automating GPU kernel verification by intercepting kernels and their parameters. In *Proceedings of the 2014 International Workshop on OpenCL (IWOCL'14)*. ACM, New York, Article 7, 5 pages.

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *Revised Lectures of the 4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*. Lecture Notes in Computer Science, Vol. 4111. Springer, Berlin, 364–387.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference in Computer Aided Verification (CAV'11)*. Lecture Notes in Computer Science, Vol. 6806. Springer, Berlin, 171–177.

Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: A verifier for GPU kernels. In *Proceedings of the 27th ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, New York, 113–132.

Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. 2007. Path invariants. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, 300–309.

Stefan Blom, Marieke Huisman, and Matej Mihelčić. 2014. Specification and verification of GPGPU programs. *Science of Computer Programming* 95, 3 (2014), 376–388.

Michael Boyer, Kevin Skadron, and Westley Weimer. 2008. Automated dynamic analysis of CUDA programs. In *Proceedings of the 3rd Workshop on Software Tools for MultiCore Systems (STMCS'08)*. Retrieved from http://people.csail.mit.edu/rabbah/conferences/08/cgo/stmcs/.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 209–224.

Joshua E. Cates, Aaron E. Lefohn, and Ross T. Whitaker. 2004. GIST: An interactive, GPU-based level set segmentation tool for 3D medical images. *Medical Image Analysis* 8 (2004), 217–231. Issue 3.

Wei-Fan Chiang, Ganesh Gopalakrishnan, Guodong Li, and Zvonimir Rakamaric. 2013. Formal analysis of GPU programs with atomics via conflict-directed delay-bounding. In *Proceedings of 5th International NASA Formal Methods Symposium (NFM'13)*. Lecture Notes in Computer Science, Vol. 7871. Springer, Berlin, 213–228.

Nathan Chong, Alastair F. Donaldson, Paul Kelly, Shaz Qadeer, and Jeroen Ketema. 2013. Barrier invariants: A shared state abstraction for the analysis of data-dependent GPU kernels. In *Proceedings of the 28th ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'13)*. ACM, New York, 605–622.

Nathan Chong, Alastair F. Donaldson, and Jeroen Ketema. 2014. A sound and complete abstraction for reasoning about parallel prefix sums. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*. ACM, New York, 397–410.

Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. 2004. A simple method for parameterized verification of cache coherence protocols. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)* Lecture Notes in Computer Science, Vol. 3312. Springer, Berlin, 382–398.

Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. 2012. Symbolic testing of OpenCL code. In *Revised Selected Papers of the 7th International Haifa Verification Conference (HVC'11)* Lecture Notes in Computer Science, Vol. 7261. Springer, Berlin, 203–218.

Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. 2013. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *Proceedings of the 22nd European Symposium on Programming (ESOP'13)*. Lecture Notes in Computer Science, Vol. 7792. Springer, Berlin, 270–289.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. ACM, New York, 238–252.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. Lecture Notes in Computer Science, Vol. 4963. Springer, Berlin, 337–340.

Alastair F. Donaldson. 2014. The GPUVerify method: A tutorial overview. *Electronic Communications of the EASST* 70, Article 1 (2014), 16 pages. Retrieved from http://journal.ub.tu-berlin.de/eceasst/article/view/986.

Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. 2010. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*. Lecture Notes in Computer Science, Vol. 6015. Springer, Berlin, 280–295.

Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. 2011. Automatic analysis of DMA races using model checking and *k*-induction. *Formal Methods in System Design* 39, 1 (2011), 83–113.

Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. ACM, New York, 411–422.

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (2007), 35–45.

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe (FME'01)*. Lecture Notes in Computer Science, Vol. 2021. Springer, Berlin, 500–517.

Susanne Graf and Hassen Saïdi. 1997. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*. Lecture Notes in Computer Science, Vol. 1254. Springer, Berlin, 72–83.

Axel Habermaier and Alexander Knapp. 2012. On the correctness of the SIMT execution model of GPUs. In *Proceedings of the 21st European Symposium on Programming (ESOP'12)*. Lecture Notes in Computer Science, Vol. 7211. Springer, Berlin, 316–335.

Mark J. Harris. 2004. Fast fluid dynamics simulation on the GPU. In *GPU Gems*. Addison-Wesley, Boston, MA, 637–665.

Temesghen Kahsai, Yeting Ge, and Cesare Tinelli. 2011. Instantiation-based invariant discovery. In *Proceedings of 3rd International NASA Formal Methods Symposium (NFM'11)*. Lecture Notes in Computer Science, Vol. 6617. Springer, Berlin, 192–206.

Rajesh K. Karmani, P. Madhusudan, and Brandon M. Moore. 2011. Thread contracts for safe parallelism. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'11)*. ACM, New York, 125–134.

Khronos OpenCL Working Group. 2012. The OpenCL Specification, Version 1.2. Document revision 19. Retrieved from https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

Khronos OpenCL Working Group. 2014a. The OpenCL C Specification, Version 2.0. Document revision 26. Retrieved from https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf.

Khronos OpenCL Working Group. 2014b. The OpenCL Extension Specification, Version 2.0. Document revision 26. Retrieved from https://www.khronos.org/registry/cl/specs/opencl-2.0-extensions.pdf.

Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles S. H. Yeo, and Brian Y. H. Lam. 2012. BarraCUDA—A fast short read sequence aligner using graphics processing units. *BMC Research Notes* 5, Article 27 (2012), 7 pages.

Kensuke Kojima and Atsushi Igarashi. 2013. A Hoare logic for SIMT programs. In *Proceeding of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)*. Lecture Notes in Computer Science, Vol. 8301. Springer, Berlin, 58–73.

Shuvendu K. Lahiri and Shaz Qadeer. 2009. Complexity and algorithms for monomial and clausal predicate abstraction. In *Proceedings of the 22nd International Conference on Automated Deduction (CADE-22)*. Lecture Notes in Computer Science, Vol. 5663. Springer, Berlin, 214–229.

K. Rustan M. Leino, Greg Nelson, and James B. Saxe. 2000. *ESC/Java User's Manual*. Technical Report SRC Technical Note 2000-002. Compaq Systems Research Center.

Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU kernels by test amplification. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, 383–394.

Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. ACM, New York, 187–196.

Guodong Li, Peng Li, Geoffrey Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. 2012b. GKLEE: Concolic verification and test generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'12)*. ACM, New York, 215–224.

Heng Li and Richard Durbin. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* 25, 14 (2009), 1754–1760.

Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2012a. Parametric flows: Automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE, Los Alamitos, CA.

Anton Lokhmotov. 2011. Mobile and Embedded Computing on Mali GPUs. In *Proceedings of the 2nd UK GPU Computing Conference*.

Kenneth McMillan. 1999. Verification of infinite state systems by compositional model checking. In *Proceedings of the 10th Conference on Correct Hardware Design and Verification Methods (CHARME'99)*. Lecture Notes in Computer Science, Vol. 1703. Springer, Berlin, 219–234.

Kenneth L. McMillan. 2006. Lazy abstraction with interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. Lecture Notes in Computer Science, Vol. 4144. Springer, Berlin, 123–136.

Microsoft Corporation. 2012. C++ AMP: Language and Programming Model, Version 1.0. Retrieved from http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer, Berlin.

Nvidia. 2012a. CUDA C Programming Guide, Version 5.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

Nvidia. 2012b. Parallel Thread Execution ISA, Version 3.1.

Lars Nyland. 2012. Personal Communication.

Lars Nyland, Mark Harris, and Jan Prins. 2007. *Fast N-Body Simulation with CUDA*. Addison-Wesley, Upper Saddle River, NJ, 677–696.

Renato F. Salas-Moreno, Richard A. Newcombe, Hauke Strasdat, Paul H. J. Kelly, and Andrew J. Davison. 2013. SLAM++: Simultaneous localisation and mapping at the level of objects. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'13)*. IEEE, Los Alamitos, CA, 1352–1359.

Saurabh Srivastava and Sumit Gulwani. 2009. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, 223–234.

Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. ACM, New York, 32–41.

Murali Talupur and Mark R. Tuttle. 2008. Going with the flow: Parameterized verification using message flows. In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD'08)*. IEEE, Los Alamitos, CA.

Stavros Tripakis, Christos Stergiou, and Roberto Lublinerman. 2010. Checking equivalence of SPMD programs using non-interference. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*. Retrieved from http://static.usenix.org/events/hotpar10/.

Christian Urban and Julien Narboux. 2009. Formal SOS-proofs for the lambda-calculus. *Electronic Notes in Theoretical Computer Science* 247 (2009), 139–155.

John Wickerson. 2014. Syntax and semantics of a GPU kernel programming language. *Archive of Formal Proofs.* Retrieved from http://afp.sourceforge.net/entries/GPU_Kernel_PL.shtml.

Received April 2014; revised November 2014; accepted March 2015