GPUVerify: A Verifier for GPU Kernels *

Adam Betts[1] Nathan Chong[1] Alastair F. Donaldson[1] Shaz Qadeer[2] Paul Thomson[1]

[1]Department of Computing, Imperial College London, UK [2]Microsoft Research, Redmond, USA

{abetts,nyc04,afd,pt1110}@imperial.ac.uk qadeer@microsoft.com

Abstract

We present a technique for verifying race- and divergence-freedom of GPU kernels that are written in mainstream kernel programming languages such as OpenCL and CUDA. Our approach is founded on a novel formal operational semantics for GPU programming termed *synchronous, delayed visibility* (SDV) semantics. The SDV semantics provides a precise definition of barrier divergence in GPU kernels and allows kernel verification to be reduced to analysis of a sequential program, thereby completely avoiding the need to reason about thread interleavings, and allowing existing modular techniques for program verification to be leveraged. We describe an efficient encoding for data race detection and propose a method for automatically inferring loop invariants required for verification. We have implemented these techniques as a practical verification tool, GPUVerify, which can be applied directly to OpenCL and CUDA source code. We evaluate GPUVerify with respect to a set of 163 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 F3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying & Reasoning about Programs

Keywords Verification, GPUs, concurrency, data races, barrier synchronization

1. Introduction

In recent years, massively parallel *accelerator* processors, primarily graphics processing units (GPUs) from companies such as AMD and NVIDIA, have become widely available to end-users. Accelerators offer tremendous compute power at a low cost, and tasks such as media processing, medical imaging and eye-tracking can be accelerated to beat CPU performance by orders of magnitude.

GPUs present a serious challenge for software developers. A system may contain one or more of the plethora of devices on the market, with many more products anticipated in the immediate future. Applications must exhibit *portable correctness*, operating correctly on *any* GPU accelerator. Software bugs in media processing domains can have serious financial implications, and GPUs are being used increasingly in domains such as medical image processing [37] where safety is critical. Thus there is an urgent need for verification techniques to aid construction of correct GPU software.

This paper addresses the problem of static verification of GPU kernels written in kernel programming languages such as OpenCL [17], CUDA [30] and C++ AMP [28]. We focus on two classes of bugs which 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 begins by giving a precise characterization of barrier divergence via an operational semantics based on predicated execution, which we call *synchronous, delayed visibility* (SDV) semantics. 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 design a verification method which reduces analysis of concurrent GPU threads to reasoning over a transformed sequential program. This completely avoids reasoning about thread interleavings, and enables reusing existing modular verification techniques for sequential programs. We present novel heuristics for automatically inferring loop invariants required for verification.

We have developed GPUVerify, a verifier for GPU kernels that can be applied directly to OpenCL and CUDA source code. We have used GPUVerify to analyse a set of 163 OpenCL and CUDA kernels. We divided this set into *training* kernels and *evaluation* kernels, such that none of

* This work was supported by the EU FP7 STEP 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.

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

our team had any experience with kernels in the evaluation set. We used the training benchmarks to design and tune GPUVerify's invariant inference and performance. We then ran GPUVerify blindly on the evaluation set, finding that fully automatic verification was achieved for 49 out of 71 kernels (69 %). We also compare GPUVerify experimentally with PUG, a recent formal analysis tool for CUDA kernels [21]. GPUVerify performs competitively with PUG for verification of correct kernels and rejects buggy kernels in several cases where PUG reports false negatives. Additionally, GPUVerify supports a finer shared state abstraction than PUG, allowing verification of real-world kernels for which PUG reports false positives.

GPUVerify, and all the non-commercial benchmarks used for our evaluation, are available online.[1]

In summary, our paper makes the following contributions:

- *Synchronous, delayed visibility* semantics: a formal operational semantics for GPU kernels based on predicated execution, data-race freedom, and divergence freedom
- A verification method for GPU kernels based on automatic abstraction followed by generation of verification conditions to be solved via automated theorem proving
- A method for inferring automatically the invariants needed for our verification method
- An extensive evaluation of our verifier on a collection of 163 publicly available and commercial GPU kernels; to our knowledge, this experimental evaluation is significantly larger, in terms of number of kernels, than any previously reported evaluation of a tool for GPU kernel analysis.

We begin by giving an overview of GPU kernel programming and the most pressing difficulties faced by kernel programmers.

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 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 Group and widely supported (in particular, OpenCL is AMD's primary high-level GPU programming model) [17];

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

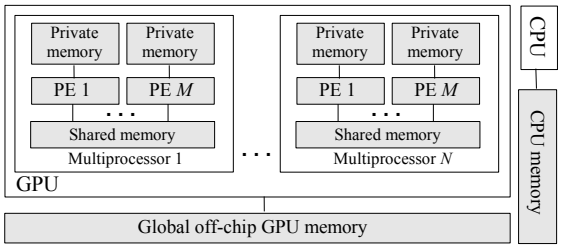


Figure 1: Schematic overview of a typical GPU architecture

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

Figure 2: Equivalent terms for *thread*, *group* and (where applicable) *sub-group* in CUDA, OpenCL and C++ AMP

CUDA, from NVIDIA [30]; and C++ AMP, from Microsoft [28].

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 GPU's multiprocessors, 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 $\lceil \frac{N}{M} \rceil$ *sub-groups*, each consisting of up to M threads. Execution of a single group on a multiprocessor then proceeds by round-robin execution of the sub-groups. Each thread in a given sub-group is pinned to a distinct PE, and all threads in the same sub-group execute together in lock-step, following exactly the same control path. Distinct sub-groups may follow different control paths.

Figure 2 summarizes the specific terms used by the three main GPU programming models to refer to *threads*, *groups* and (in the case of CUDA) *sub-groups*. OpenCL and C++ AMP aim for portability across GPUs from multiple vendors, so do not allow a kernel to query the device-

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);`
	`}`

Figure 3: Predicated forms for conditionals and loops

specific size or structure of thread sub-groups. As CUDA is NVIDIA-specific, CUDA programmers *can* write kernels that make assumptions about the division of threads into sub-groups. However, such kernels will not easily port to general-purpose GPU programming languages, and may break when executed on future generations of NVIDIA hardware that uses a different sub-group size. Thus GPU kernels that do not make assumptions about the size of thread sub-groups are preferable.

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

Consider the conditional statement in the top-left of Figure 3, 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 sub-group 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 sub-group continue to execute the loop body until the loop condition is false for *all* threads in the sub-group, with threads for whom the condition does not hold becoming disabled. This is illustrated for the loop in the bottom-left of Figure 3 (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 sub-group loops while `p` remains true for some thread in the sub-group, 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 t_2 unless t_1 and t_2 *synchronize*. As noted above, there is *no* mechanism for threads in distinct groups to synchronize during kernel execution.[2] 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 *inter-group 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 *intra-group 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.

In this work, we restrict our attention to intra-group data races because we consider them to be the more significant problem for GPU programmers. Threads across groups cannot synchronize and consequently the argument for absence of inter-group data races is usually based on globally disjoint memory access patterns. Threads within a group can synchronize in sophisticated ways using the barrier operation; consequently, the correctness argument is more complicated and errors more likely.

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 [30]: *"execution is likely to hang or produce unintended side effects"*.

While there is clarity across all programming models for what barrier divergence means in loop-free code, the situation is far from clear for code with loops. Consider the example kernel shown on the left of Figure 4. This kernel is intended to be executed by a group of four threads, and

[2] Atomic operations on global memory are available in some GPU architectures, but cannot reliably implement inter-group synchronization 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);
}
```

Figure 4: Illustration of the subtleties of barriers in nested loops

declares an array `A` of two shared buffers, each of size four. Local 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 non-current 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 following:

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

However, consider the predicated version of the kernel shown in part on the right of Figure 4. This is the form in which the kernel executes on an NVIDIA GPU. The four threads comprise a single sub-group. 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 sub-group 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 [31], *"if any thread in a* [*sub-group*] *executes a* `bar` *instruction, it is as if all the threads in the* [*sub-group*] *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. This yields the following sequence of state-changes to A:

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

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 [17] gives a better, though still informal definition, stating: *"If a 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 4.

To investigate this issue in practice, we implemented the litmus test of Figure 4 in both CUDA and OpenCL and (with help from contacts in the GPU industry; see Acknowledgements) 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 Figure 5. Observe that the test result does not agree between any two vendors. The NVIDIA results match our above prediction. The AMD result also appears to stem from predicated execution. ARM's Mali architecture does *not* work using predicated execution [25], 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 Figure 5, which we attribute to cache effects.

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	$\{\{*,*,*,1\},\{3,0,1,2\}\}$

Figure 5: The litmus test of Figure 4 yields a range of results across varying platforms

The example of Figure 4 is contrived in order to be small enough to explain concisely and examine exhaustively. It does, however, illustrate that barrier divergence is a subtle issue, and that non-obvious 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 divergence-freedom.

3. Operational semantics for GPU kernels

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

For checking barrier divergence, the least conservative assumption we can safely make is that a thread group consists of a *single* sub-group, i.e., all threads in a group execute in lock-step. This will indeed be the case, e.g., for a group of 32 threads executing on an NVIDIA GPU. We can then define barrier divergence to occur if the 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 divergence-freedom for a kernel under this tight assumption, the kernel will also be divergence-free if thread groups are actually divided into sub-groups with a finer level of granularity.

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. This is the case on ARM's Mali GPU architecture [25] (so that essentially every sub-group consists of just a single thread). 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 which we call *synchronous, delayed visibility* (SDV). Under SDV, group execution is *synchronous*, allowing precise divergence checking. Each thread's shared memory accesses are logged, and the *visibility* of writes to shared memory by one thread to the group is *delayed* until a barrier is reached. 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 when threads synchronize at a barrier.

To describe the SDV semantics formally, we define Kernel Programming Language (KPL), which captures the essential features of mainstream languages for writing GPU kernels. KPL describes execution of a *single* group of GPU threads. Kernels in real GPU programming languages can have multiple groups, but it suffices for checking divergence- and intra-group race-freedom to model the execution of a single arbitrary group.

3.1 Syntax

The syntax for KPL is shown in Figure 6. A KPL kernel declares the total number of threads in the group that will execute the kernel (**threads**: number), and the group's id (**group**: number), catering for the fact that in practice, the group may be one of many. This is followed by a list of procedure declarations followed by a **main** statement. Each

kernel	::=	**threads**: number
		group: 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
	|	name(local_expr)
	|	**barrier**
	|	**break** | **continue** | **return**
basic_stmt	::=	loc := local_expr
	|	loc := **rd**(local_expr)
	|	**wr**(local_expr, local_expr)
local_expr	::=	gid | lid | loc
	|	constant literal of type Word
	|	local_expr op local_expr
loc	::=	name | V
name	::=	any valid C name

Figure 6: Syntax for Kernel Programming Language

procedure has a name, a single parameter and a body; for brevity we do not model multiple parameters or return values. The final element in the program is a *main* statement.

For simplicity, but without loss of generality, threads have access to a single shared array which we refer to as *shared memory*. Since our goal is to verify absence of only intra-group data races, we make no distinction between shared and global memory (c.f. Figure 1) in our programming language. We assume that every local variable and each indexable element of shared memory has type Word, the type of memory words. We assume that any value in Word can be interpreted as an integer and a boolean. In practice, Word will also represent floating point numbers, and structured data will be represented by sequences of values of type Word.

A thread may update one of its local variables by performing a local computation, or by reading from the shared state ($v := \mathbf{rd}(e)$, where e is an expression over local variables determining which index to read from). A thread may also update the shared state ($\mathbf{wr}(e_1, e_2)$, 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 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 few other statements: **barrier**, which is used to synchronize threads; **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 6 specifies two syntactic elements which should *not* appear directly in a KPL program; they are used in the semantic rules of Figure 8 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 the program is 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. An extra requirement important for our semantics is that the main statement must not contain a **return** and must end with a **barrier**.

We do not formalize features of GPU kernels such as multi-dimensional groups and arrays. However, our verification method and implementation, described in Section 4, handles both.

3.2 Semantics

Notation. Given a function $f : A \to B$ and elements $a \in A, b \in B$, we write $f[a := b]$ to denote the function $g : A \to B$ such that $g(x) = f(x)$ for all $x \in A \setminus \{a\}$, and $g(a) = b$. We abbreviate $f[a := b][c := d]$ to $f[a := b, c := d]$. By 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 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. We overload the @ operator and write $ss@tt$ for the concatenation of sequences ss and tt.

Thread and group states. To model the *delayed visibility* aspect of SDV, shared state is distributed: each thread is equipped with a *shadow* copy of shared memory. At the start of kernel execution, every thread's shadow memory is identical. During execution, a thread reads and modifies its shadow memory locally, and logs read and write sets recording which addresses in shared memory the thread has accessed. When a **barrier** statement is reached with all threads enabled, the read and write sets are checked for data races. If a race has occurred, execution aborts. Otherwise the write sets are used to build a consistent view of shared memory, the shadow memories are all reset to agree with this view, and the read and write sets are cleared.

In what follows let P be a KPL kernel.

Let n denote the number of threads in the group executing P, specified via **threads**: n in the definition of P. A *thread state* for P is a tuple (lid, l, sh, R, W) where:

- lid : Word is the thread's id within the group.
- $l : V \to$ Word is the storage for the thread's local variables (recall that V is a set of locations).
- $sh : \mathbb{N} \to$ Word is the thread's shadow copy of shared memory.
- $R, W \subseteq \mathbb{N}$ are the thread's read and write sets, recording the shared addresses the thread has accessed since the last barrier.

We use σ to denote a thread state, and $\sigma.lid$, $\sigma.l$, $\sigma.sh$, etc., to refer to the components of σ. The set of all thread states is denoted ThreadStates. For local expression e and thread state σ, we write e^σ for the result of evaluating e according to $\sigma.lid$ and $\sigma.l$. We do not provide a concrete definition of e^σ, which depends on the nature of the base type Word and the available operators, except we specify that $v^\sigma = \sigma.l(v)$ (where $v \in V$ is a storage location), $lid^\sigma = \sigma.lid$, and $gid^\sigma = x$ where **group**: x is specified in the definition of P.

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

A *group state* for P is a tuple (Σ, ss) where:

- $\Sigma = (\sigma_0, \ldots, \sigma_{n-1}) \in \mathsf{ThreadStates}^n$ records a thread state for each thread in the group
- $ss \in \mathsf{PredStmts}^*$ is an ordered sequence of program statements to be executed by the group

The set of all group states is denoted GroupStates. Given a tuple of thread states $\Sigma = (\sigma_0, \ldots, \sigma_{n-1})$, we use $\Sigma(i)$ to denote σ_i.

A group state (Σ, ss) is a valid initial state of P if:

- $ss = \langle (s, true) \rangle$, where s is declared in P via **main** : s.
- $\Sigma(i).lid = i$ and $\Sigma(i).R = \Sigma(i).W = \emptyset$ $(0 \le i < n)$.
- $\Sigma(i).l(v) = false$ for all $v \in V$ $(0 \le i < n)$.
- $\Sigma(i).sh = \Sigma(j).sh$ $(0 \le i, j < n)$.

The first two requirements are straightforward. The third requirement ensures that local variables are initialized to the value in Word corresponding to $false$. The final requirement ensures that threads have a consistent but arbitrary initial view of the shared state. Our state representation does not include a single, definitive shared state component: the shared state is represented via the shadow copies held by individual threads, which are initially consistent, and made consistent again at each barrier.

Predicated group execution. The rules of Figure 7 define a binary relation

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

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

Rule T-DISABLED ensures that a predicated statement has no effect if the predicate does not hold, indicated by

$$\frac{\neg p^{\sigma}}{(\sigma, \mathsf{basic\_stmt}, p) \rightarrow_t \sigma} \text{ (T-DISABLED)}$$

$$\frac{p^{\sigma} \qquad l' = \sigma.l[v := e^{\sigma}]}{(\sigma, v := e, p) \rightarrow_t \sigma[l := l']} \text{ (T-ASSIGN)}$$

$$\frac{p^{\sigma} \qquad l' = \sigma.l[v := \sigma.sh(e^{\sigma})] \qquad R' = \sigma.R \cup \{e^{\sigma}\}}{(\sigma, v := \mathbf{rd}(e), p) \rightarrow_t \sigma[l := l', R := R']} \text{ (T-RD)}$$

$$\frac{p^{\sigma} \qquad sh' = \sigma.sh[e_1^{\sigma} := e_2^{\sigma}] \qquad W' = \sigma.W \cup \{e_1^{\sigma}\}}{(\sigma, \mathbf{wr}(e_1, e_2), p) \rightarrow_t \sigma[sh := sh', W := W']} \text{ (T-WR)}$$

Figure 7: Rules for predicated execution of basic statements

$\neg p^{\sigma}$ in the rule's premises; T-ASSIGN updates $\sigma.l$ according to the assignment; T-RD updates the thread's local store with an element from the thread's shadow copy of shared memory, and records the address that was read from; rule T-WR is analogous.

Figure 8 defines a binary relation

$$\rightarrow_g \subseteq \mathsf{GroupStates} \times (\mathsf{GroupStates} \cup \{error\}),$$

where *error* is a designated error state. This relation describes the evolution of a group as it executes a sequence of predicated statements. Collective execution of a predicated basic statement is achieved by every thread executing the statement, the order in which they do so is irrelevant (G-BASIC).

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

Intra-group races are detected via rule G-RACE. This rule states that if a group is due to execute a **barrier** statement and all threads are enabled, then when we compare the read and write sets computed by each thread, we must not find distinct threads i and j such that the write set for thread i intersects with either the read or write set for thread j. If this scenario occurs, the error state is reached. The predicate $\mathsf{races}(\Sigma)$ is defined as follows:

$$\mathsf{races}(\Sigma) = \exists\, 0 \leq i \neq j < n.\ (\Sigma(i).R \cup \Sigma(i).W) \cap \Sigma(j).W \neq \emptyset$$

The most intricate rule of Figure 8 is G-SYNC which captures the effect of a barrier synchronization in the absence of data races. A new thread state $\Sigma'(i)$ is constructed for each thread i, with the same local component l as before the barrier. The barrier enforces a consistent view of shared memory across the group by setting the shared shadow memories sh identically in each $\Sigma'(i)$. This is achieved using a function merge. If thread i has recorded a write to shared memory location z, i.e. $z \in \Sigma(i).W$, then $\mathsf{merge}(\Sigma)$ maps z to the value at address z in thread i's shadow memory, i.e. to

$$\frac{\forall\, 0 \leq i < n\ .\ (\Sigma(i), \mathsf{basic\_stmt}, p) \rightarrow_t \Sigma'(i)}{(\Sigma, (\mathsf{basic\_stmt}, p)@ss) \rightarrow_g (\Sigma', ss)} \text{ (G-BASIC)}$$

$$\frac{\exists\, 0 \leq i \neq j < n\ .\ p^{\Sigma(i)} \wedge \neg p^{\Sigma(j)}}{(\Sigma, (\mathbf{barrier}, p)@ss) \rightarrow_g error} \text{ (G-DIVERGENCE)}$$

$$\frac{\forall\, 0 \leq i < n\ .\ \neg p^{\Sigma(i)}}{(\Sigma, (\mathbf{barrier}, p)@ss) \rightarrow_g (\Sigma, ss)} \text{ (G-NO-OP)}$$

$$\frac{\forall\, 0 \leq i < n\ .\ p^{\Sigma(i)} \qquad \mathsf{races}(\Sigma)}{(\Sigma, (\mathbf{barrier}, p)@ss) \rightarrow_g error} \text{ (G-RACE)}$$

$$\frac{\begin{array}{c}\forall\, 0 \leq i < n\ .\ p^{\Sigma(i)} \qquad \neg\mathsf{races}(\Sigma) \\ \forall\, 0 \leq i < n\ .\ \Sigma'(i) = (\Sigma(i).lid, \Sigma(i).l, \mathsf{merge}(\Sigma), \emptyset, \emptyset)\end{array}}{(\Sigma, (\mathbf{barrier}, p)@ss) \rightarrow_g (\Sigma', ss)} \text{ (G-SYNC)}$$

$$(\Sigma, (S_1; S_2, p)@ss) \rightarrow_g (\Sigma, (S_1, p)@(S_2, p)@ss) \text{ (G-SEQ)}$$

$$\frac{\text{fresh } v \in V}{(\Sigma, (\mathbf{local}\ x\ S, p)@ss) \rightarrow_g (\Sigma, (S[x \mapsto v], p)@ss)} \text{ (G-VAR)}$$

$$\frac{\text{fresh } v \in V}{\begin{array}{c}(\Sigma, (\mathbf{if}\ e\ S_1\ \mathbf{else}\ S_2, p)@ss) \rightarrow_g \\ (\Sigma, (v := e, p)@(S_1, p \wedge v)@(S_2, p \wedge \neg v)@ss)\end{array}} \text{ (G-IF)}$$

$$\frac{\text{fresh } v \in V}{\begin{array}{c}(\Sigma, (\mathbf{while}\ e\ S, p)@ss) \rightarrow_g \\ (\Sigma, (\underline{\mathbf{while}}\ e\ belim(S, v), p \wedge \neg v)@ss)\end{array}} \text{ (G-OPEN)}$$

$$\frac{\begin{array}{c}\exists\, 0 \leq i < n\ .\ (p \wedge e)^{\Sigma(i)} \\ \text{fresh } u, v \in V \qquad q = p \wedge u \wedge \neg v\end{array}}{\begin{array}{c}(\Sigma, (\underline{\mathbf{while}}\ e\ S, p)@ss) \rightarrow_g \\ (\Sigma, (u := e, p)@(celim(S, v), q)@(\underline{\mathbf{while}}\ e\ S, p)@ss)\end{array}} \text{ (G-ITER)}$$

$$\frac{\forall\, 0 \leq i < n\ .\ \neg(p \wedge e)^{\Sigma(i)}}{(\Sigma, (\underline{\mathbf{while}}\ e\ S, p)@ss) \rightarrow_g (\Sigma, ss)} \text{ (G-DONE)}$$

$$\frac{\text{fresh } u, v \in V \qquad S = Body(f)[Param(f) \mapsto u]}{\begin{array}{c}(\Sigma, (f(e), p)@ss) \rightarrow_g \\ (\Sigma, (u := e; relim(S, v), p \wedge \neg v)@ss)\end{array}} \text{ (G-CALL)}$$

Figure 8: Rules for lock-step execution of a group

$\Sigma(i).sh(z)$. Formally, $\mathsf{merge}(\Sigma)$ is a map satisfying the following constraints:

$$\frac{z \in \Sigma(i).W \qquad 0 \leq i < n}{\mathsf{merge}(\Sigma)(z) = \Sigma(i).sh(z)} \qquad \frac{\forall\, 0 \leq i < n\ .\ z \notin \Sigma(i).W}{\mathsf{merge}(\Sigma)(z) = \Sigma(0).sh(z)}$$

Because $\mathsf{races}(\Sigma)$ is *false* (a premise of the rule), $\mathsf{merge}(\Sigma)$ is unique. Finally, the read and write sets of all threads are cleared.

The remaining rules in Figure 8 describe predicated execution for compound statements. Rule G-SEQ is straightforward. Rule G-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 the notation $S[x \mapsto v]$ to denote this substitution. Rule G-IF decomposes a conditional statement into a sequence of predicated statements:

the conditional's guard is evaluated into a new location v, the *then* branch S_1, is executed *by all threads* under predicate $p \wedge v$ (where p is the predicate of execution already in place on entry to the conditional), and the *else* branch S_2, is executed *by all threads* under predicate $p \wedge \neg v$.

The rules G-OPEN, G-ITER and G-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 G-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 has initial value *false*: 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 $v := true$. Furthermore, the predicate for the execution of the **while** loop becomes $p \wedge \neg v$ to model that statements in the loop have no effect subsequent to the execution of a **break** statement. A similar technique is used to model **break** statements in [14].

The G-ITER rule models execution of loop iterations, and handles **continue** statements. Two fresh local storage locations u and v are selected (both initialized to *false*). 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; v is initially *false* because no thread has executed a **continue** statement at the beginning of an iteration. First, the statement $u := e$ (executed under enclosing predicate p) evaluates the loop guard into u. Then 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 $p \wedge u \wedge \neg v$ (denoted q in rule G-ITER): a thread is enabled during the current iteration if the incoming predicate holds (p), the loop guard evaluated 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 G-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 G-ITER until, for for *every* thread, 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. When a) or b) is the case for all threads, loop exit is handled by rule G-DONE.

The rule G-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 the given 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 within the procedure body. Initially this location is set to $false$, and 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.

4. Verification method

Armed with the SDV semantics of Section 3, we now consider the problem of verifying that a GPU kernel is race- and divergence-free. For this purpose, we have designed a tool, GPUVerify, built on top of the Boogie verification system [2]. 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 [8]. We describe challenges associated with automatically translating OpenCL and CUDA kernels into a Boogie intermediate representation (Section 4.1), a technique for transforming this Boogie representation of the kernel into a standard sequential Boogie program whose correctness implies race- and divergence-freedom of the original kernel (Section 4.2), and a method for automatically inferring invariants and procedure contracts to enable automatic verification (Section 4.3).

4.1 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 [24] due to its existing support for both OpenCL and CUDA language extensions.

There were effectively three challenges with respect to this compilation. First, many industrial applications utilise 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 modelled them abstractly via uninterpreted functions. This sound over-approximation can in principle lead to false positives, but in our extensive evaluation (Section 5) we have only encountered one instance of this, where race-freedom depends on concrete floating point values. The third issue, namely handling of pointers, is more interesting technically and is now discussed in depth.

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);`

`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);`

`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);`

Figure 9: Translating pointer usage into Boogie

Modelling 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 integer arrays (in any memory space) and two integer pointers:

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

In this case GPUVerify generates the following types:

```
enum int_ptr_base = { A_base, B_base, null, none };

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

Thus an integer pointer is modelled as a pair consisting of a base array, or one of the special values `null` or `none` if the pointer is null or uninitialized, respectively, and an integer offset from this base. The offset is in terms of number of elements, not bytes.

Pointers `p` and `q` can be assigned to offsets from `A` or `B`, to `null`, or can be left uninitialized. Figure 9 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 writes and reads are modelled 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 corresponding accesses to such variables appropriately. This is made possible 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 disambiguate pointer dereferences. To avoid this, we have implemented Steensgaard's flow- and context-insensitive pointer analysis algorithm [36]. Although this over-approximates the points-to sets, our experience of GPU kernels is that aliasing is scarce and therefore precision is high. Returning to the above example, suppose points-to analysis determines 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)` case ensures that potential bugs in our points-to analysis do not lead to unsound verification.

4.2 Reducing race- and 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 10a, 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, 2-dimensional group of `SZ`×`SZ` threads. A thread's local id 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 10b shows the example kernel of Figure 10a after access logging instrumentation. Observe for example that the condition `gr[lidY][k] && gr[k][lidX]`[3] involves two read accesses to `gr`, thus is pre-pended by two calls to `LOG_RD_gr`.

Reduction to a pair of threads. After access logging, the kernel must be translated into a form which models the predicated execution of multiple threads in a group. Initially, we attempted to directly encode the SDV semantics of Section 3, modeling lock-step execution of *all* threads in a group. Unfortunately, modeling in this way required heavy use of quantifiers, especially for implementing the G-SYNC rule of Figure 8 and associated merge function. This led to Boogie programs outside the decidable theory supported by the Z3 theorem prover. As a result, verification of micro kernels took in the order of minutes, while verification attempts for large kernels quickly exhausted memory limits.

Both the properties of race- and divergence-freedom are stated *pairwise*: a race occurs when accesses by two threads conflict, and divergence occurs when a barrier is executed in a state where one thread is enabled and another disabled.

[3] For ease of presentation we treat the operands of `&&` as being evaluated simultaneously. In reality, short-circuit evaluation introduces an extra branch.

```
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

```
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 that the pair of threads are distinct
  assume(lidX1 != lidX2 || lidY1 != lidY2);
  // Not shown: assume that thread ids lie in appropriate range
  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;
} }
```

(c) Kernel after transformation to 2-thread predicated form

Figure 10: Example illustrating how GPUVerify transforms a kernel into two-threaded, predicated form for verification

Based on this observation, we can consider transforming a kernel into a form where the predicated execution of only *two* threads is modeled. If we can prove a kernel race- and divergence-free for a pair of distinct but otherwise *arbitrary* threads, we can conclude correctness of the kernel. The design of the PUG verifier for CUDA kernels also hinges on this observation [21]. 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 allow us to leverage existing techniques for modular reasoning about sequential programs.

For this approach to be sound, we must *approximate* rule G-SYNC of Figure 8, abstracting the values written to the shared state by threads that are not modeled. This can be achieved in multiple ways. We have considered the following strategies:

- **Adversarial abstraction:** The shared state is completely removed; reads are replaced with non-deterministic assignments.
- **Equality abstraction:** Both 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.

We have found several example kernels (including the kernel of Figure 10a) where race-freedom hinges on threads agreeing on the value of certain shared locations. In these cases, adversarial abstraction is too strong for successful verification. However, in many such 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 prior work [21] which is limited to adversarial abstraction.

Using adversarial abstraction, when it suffices, proves to be more efficient than equality abstraction. GPUVerify applies abstraction on array-by-array basis. We have implemented an inter-procedural control dependence analysis to over-approximate those shared arrays whose values may influence control flow. Arrays which may influence control flow are handled using equality abstraction and all others using adversarial abstraction. We have found this heuristic typically leads to equality abstraction being applied only when it is required.

While the equality or adversarial abstractions suffice for verification of the vast majority of kernels we have studied, equality abstraction is not sufficient when correctness depends upon 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 kernels which perform a prefix sum operation into an array `B`, and then use `B` to index into an array `A` as shown above. We plan to investigate richer shared state abstractions to overcome this limitation in future work.

Figure 10c shows the result of transforming the access-instrumented version of the kernel (Figure 10b) into a form where the predicated execution of a pair of arbitrary, distinct threads is modeled, using the equality abstraction. (The transformation using adversarial abstraction is identical, ex-

cept that the arrays gr1 and gr2 are eliminated, and reads from these arrays are made nondeterministic.)

The id of the first thread is represented by the pair lidX1, lidY1, and similarly for the second thread. The assume statement dictates that at least one of lidX and lidY should differ between the threads; we omit an additional precondition ensuring that the id components lie in the range [0..SZ-1].

Local variable k is duplicated, and the assignment k = 0 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 10c, 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 (c.f. the incoming predicate p in the G-CALL rule of Figure 8). 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.

Proof sketch for two-thread reduction. First, consider an alternative semantics with a different version of the rule G-SYNC in which the shadow states for each thread are either set to a completely arbitrary value (adversarial abstraction) or an arbitrary but consistent value (equality abstraction). It is easy to see that either of these two alternative semantics is an abstraction of the original semantics: (1) all states reachable via race-free and divergence-free executions at barrier operations are preserved; (2) all divergences and data-races are preserved. Now suppose there is a data-race between two threads with indices i and j. Since the two-thread program is based on the abstract version of rule G-SYNC and the indices of the two threads in this program are arbitrary symbolic constants, the two-thread program can simulate the behavior of threads i and j all the way up to the data-race. Therefore, the two-thread program will also have a data-race. The argument for divergence is similar.

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 execution passed as *enabled* parameters.

Checking 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. This two-threaded version of rule G-DIVERGENCE (Figure 8) precisely matches the notion of barrier divergence presented formally in Section 3. We may wish to *only* check 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 divergence-free under this most general assumption is guaranteed to be divergence-free under any schedule of shared state modifications. If we prove divergence-freedom for a kernel under the equality abstraction, we can conclude a weaker property than divergence-freedom: that barrier divergence cannot occur unless a data race has occurred. Note that our divergence checking is stricter than that attempted by the PUG verifier [21] which merely requires threads which follow different conditional paths through a kernel to pass the same number of barriers.[4] While PUG reports micro-kernels exhibiting the divergence bugs discussed in Section 2 as successfully verified, such kernels are rejected by GPUVerify.

Race checking. The LOG_RD and LOG_WR procedures are responsible for manipulating a read and write set for each thread, for each of the kernel's shared arrays. According to the semantics of Section 3 (rule G-RACE of Figure 8), race checking then involves asserting inside barrier for each array A that the read and write sets for A do not conflict between threads. Alternatively, we can immediately assert race-freedom whenever an access is logged. GPUVerify employs this eager method, which we have found leads to faster analysis.

We encode read and write sets efficiently by exploiting nondeterminism, similar to a method used in prior work [9, 10]. For each shared array A with index type T we introduce the following variables for each thread i under consideration (where $i \in \{1, 2\}$):

- WR_exists_Ai : **bool**
- WR_elem_Ai : T
- RD_exists_Ai : **bool**
- RD_elem_Ai : T

Boolean WR_exists_Ai is set to *true* if and only if thread i's write set for A is non-empty. In this case, WR_elem_Ai represents one element of this write set: an index into A. The corresponding RD variables for read sets are similar.

Initially WR/RD_exists_Ai is *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

[4] In a subsequent paper on dynamic symbolic execution of CUDA kernels [23] the authors of [21] improve this check to restrict to textually aligned barriers.

on entry to the procedure (predicate parameter `en`i is *true*), then the thread nondeterministically chooses to do *nothing*, or to set WR_exists_Ai to *true* and WR_elem_Ai to the index being logged. Procedure `LOG_RD_A` operates similarly. This strategy ensures that if WR_exists_Ai holds, WR_elem_Ai 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_A1 ∧ WR_exists_A2∧
WR_elem_A1 == WR_elem_A2));

Procedure `LOG_WR_A` works analogously, and a similar assertion is used to check read-write races.

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. At a barrier, read and write sets are cleared via assuming that every WR/RD_exists is *false*, i.e., by terminating all execution paths along which read or written elements were logged.

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 above assertion to check that the values written are not equal.

4.3 Invariant inference

GPUVerify produces a Boogie program akin to the transformed kernel of Figure 10c, together with implementations of `barrier` and all `LOG_RD/WR` procedures. This program must be verified to prove race- and 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 and discussed in detail below, where threads do not tend to read or write from contiguous regions of memory, and also due to the predicate nature of the programs produced by our verification method.

Instead, we use the Houdini [12] algorithm as the basis for inferring invariants and contracts. Houdini is a method to find the largest set of inductive invariants from amongst a user-supplied pool of candidate invariants. Houdini works as a fixpoint procedure; starting with 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 discuss the relationship between Houdini and other invariant generation techniques briefly in Section 6.

Through manually deducing invariants for a set of kernels (the *training* set described in our experimental evaluation, Section 5) we have devised a number of candidate generation rules which we outline below. We emphasise 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 present such an evaluation in Section 5.1.

Candidate invariant generation rules. In what follows, `lid` and `SZ` denote a thread's local id, and the size of the thread group, respectively, and `==>` denotes implication. For clarity, we present the essence of each rule; the GPUVerify implementation is more flexible (e.g., being insensitive to the order of operands to commutative operations, and detecting when a thread's id has been copied into another local variable). For each of the rules associated with shared memory writes, there is an analogous rule for reads.

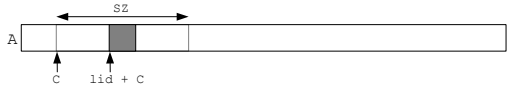
Rule: access at thread id plus offset.

Condition: `A[lid + C] = ...` occurs in loop

Generated candidate:

```
WR_exists_A ==> WR_elem_A - C == lid
```

Rationale: 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 as follows:



Rule: access at thread id plus strided offset.

Conditions: `A[lid + i*SZ + C] = ...` occurs in loop
`i` is live at loop head

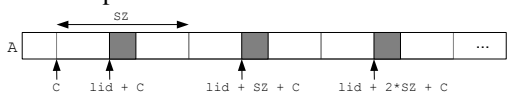
Generated candidate:

```
WR_exists_A ==> ((WR_elem_A - C) % SZ) == lid
```

Rationale: 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 the following example:

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

This access pattern is illustrated as follows:



Rule: access at thread id plus strided offset, with strength reduction.

Conditions: `i = lid` appears before loop
`A[i+C] = ...` occurs in loop
`i = i + SZ` appears in loop body
`i` is live at loop head

Generated candidates:

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

Rationale: Same as the previous rule. However, GPU programmers commonly apply the *strength reduction* operation manually, rewriting the above code snippet as follows:

```
for(i = lid; 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) == lid` is required in addition.

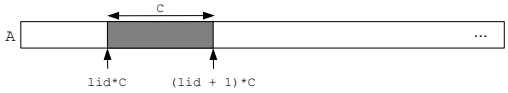
Rule: access contiguous range.

Conditions: `A[lid*C + i] = ...` occurs in loop
`i` is live at loop head

Generated candidates:

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

Rationale: It is common for threads to each be assigned a fixed-size chunk of an array to process. This access pattern is illustrated as follows:



Rule: variable is zero or a power of two.

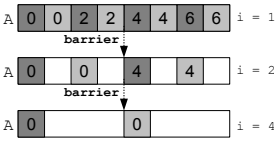
Conditions: `i = i*2` or `i = i/2` occurs in loop
`i` is live at loop head
`D` is the smallest power of 2 with $D \geq SZ$

Generated candidates:

```
i&(i-1)==0, i!=0, i<1, i<2, i<4, ..., i<D
```

Rationale: GPU kernels frequently perform tree reduction operations on shared memory, as in the code snippet below. Race-freedom is ensured through the use of a barrier, together with a guard ensuring that threads which have dropped out of the reduction computation do not write to shared memory. Verifying race-freedom requires an invariant that the loop counter is a power of two in a prefix-closed range, possibly including zero. The above rule generates a linear number of candidates which capture all relevant prefix-closed ranges. The access pattern for such a tree reduction with respect to a group of 8 threads (`SZ == 8`) is illustrated below. A grey square containing a thread id indicates a memory access by the associated thread; dark grey indicates both a read and a write, while light grey indicates a read only.

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



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 post-conditions 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.

5. Experimental evaluation

Benchmarks. We evaluate GPUVerify using four benchmark suites, comprising 163 kernels in total:

- **AMD SDK:** AMD Accelerated Parallel Processing SDK v2.6 [1], 71 publicly available OpenCL kernels
- **CUDA SDK:** NVIDIA GPU Computing SDK v2.0 [29], 20 publicly available CUDA kernels
- **C++ AMP:** Microsoft C++ AMP Sample Projects [27], 20 publicly available kernels, translated to CUDA
- **Basemark:** Rightware Basemark CL v1.1 [34], 52 commercial OpenCL kernels, provided to us under academic license

To our knowledge, this benchmark set makes our evaluation significantly larger, in terms of number of kernels analyzed, than any previously reported evaluation of a tool for GPU kernel analysis.

We consider the somewhat out-of-date v2.0 version of the NVIDIA SDK to facilitate a direct comparison of GPUVerify with PUG [21], the only existing verifier for CUDA kernels, since PUG is not compatible with more recent versions of the CUDA SDK. For this reason, we restrict our attention to the benchmarks from this SDK which were used to evaluate PUG in [21]. Because GPUVerify cannot directly analyse C++ AMP code, we retrieved the set of C++ AMP samples available online [27] on 3 February 2011, and manually extracted and translated the GPU kernel functions into corresponding CUDA kernels. This mechanical extraction and translation was straightforward.

We scanned each benchmark suite and removed kernels which 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 4.2 (12 kernels). We plan to investigate supporting atomic operations, and design richer shared state abstractions to handle double-indirection, in future work.

Experiments are performed on a PC with a 3.4GHz Intel Core i7-2600 CPU, 8GB RAM running Windows 7 (64-bit), using revision 2490 of Boogie, and Z3 v3.3. All times reported are averages over 3 runs.

GPUVerify, together with all our non-commercial benchmark kernels, are available from our web page.[5]

5.1 Evaluation of GPUVerify

Methodology. The practical utility of GPUVerify for proving correctness of GPU kernels depends largely on the effectiveness of the tool's invariant inference technique. Invariant inference must be precise enough to allow automatic verification of typical kernels, and semi-automatic verification of especially intricate kernels. Inference must not compromise efficient analysis: whether verification succeeds or fails (in the latter case due to the kernel being incorrect, or the inferred invariants being too weak), the runtime associated with verification should be as low as possible. Ideally, GPUVerify should be suitably efficient that it can run as a background process in an IDE such as Eclipse, to provide immediate feedback to GPU kernel developers.

We have 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 previously unknown to all members of our team. We chose the CUDA SDK, C++ AMP and Basemark benchmarks as the training set (92 kernels) and the AMD SDK benchmarks as the evaluation set (71): members of our team had looked previously at the CUDA SDK and C++ AMP benchmarks, but not at the AMD SDK or Basemark benchmarks; however, we wanted to make the evaluation set publicly available, ruling out Basemark.

We manually analysed all benchmarks in the training set, determining invariants sufficient for proving race- and 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 4.3. We implemented these heuristics in GPUVerify and tuned GPUVerify to maximise performance on the training set.

We then applied GPUVerify *blindly* to the evaluation set. We report below 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 tool's automatic capabilities.

Characteristics of the training and evaluation sets. Figure 11 provides an overview of the sizes of benchmarks in the training and evaluation sets. We indicate the number of effective lines of code (ELOC) (this *excludes* comments and whitespace, and counts a statement spanning multiple lines as a *single* effective line), number of procedures and number of loops. The largest kernels we analysed consist of 100 and 576 ELOC for the training and evaluation sets, respectively. The size and complexity of our benchmark kernels are representative of GPU kernels in practical use.

[5] http://multicore.doc.ic.ac.uk/GPUVerify

ELOC	**≤30**	**31-60**	**61-120**	**121-180**	**max=576**
Training	78	13	1	0	0
Evaluation	49	10	6	5	1
#procs	**1**	**2-3**	**4-5**	**6-7**	**max=8**
Training	69	18	2	3	0
Evaluation	55	10	5	0	1
#loops	**0**	**1**	**2**	**3**	**4-5**
Training	44	24	19	2	3
Evaluation	21	27	20	0	3

Figure 11: Summary of size, in terms of ELOC, number of procedures and number of loops, of benchmarks in the training and evaluation sets

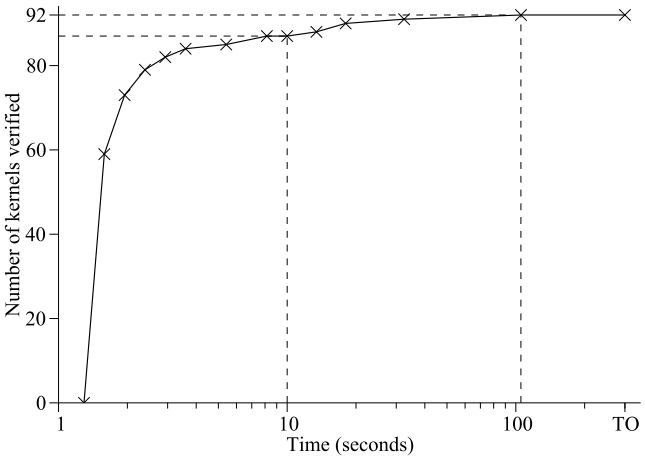


Figure 12: Cumulative histogram showing the time taken for successful verification with GPUVerify for the *training* set

In all experiments, we run GPUVerify using a timeout of five minutes per benchmark. Full inlining of procedures is used, as discussed in Section 4.3.

Results for the training set. Figure 12 is a cumulative histogram showing the performance of GPUVerify with respect to the training set. The x-axis plots time (in seconds, on a log scale), and the y-axis plots number of kernels. A point at position (x, y) indicates that for y of the kernels, verification took x seconds or fewer. The results show that GPUVerify is capable of rapidly analysing the vast majority of the training set kernels: 85 out of 92 were verified in 10 seconds or fewer. The longest verification time was 105 seconds; this is for a CUDA *PrefixSum* kernel which contains a complex bespoke invariant. In no cases did verification time out.

Running GPUVerify with race checking disabled, the tool was able to prove barrier divergence freedom for all training benchmarks, in under 10 seconds per kernel.

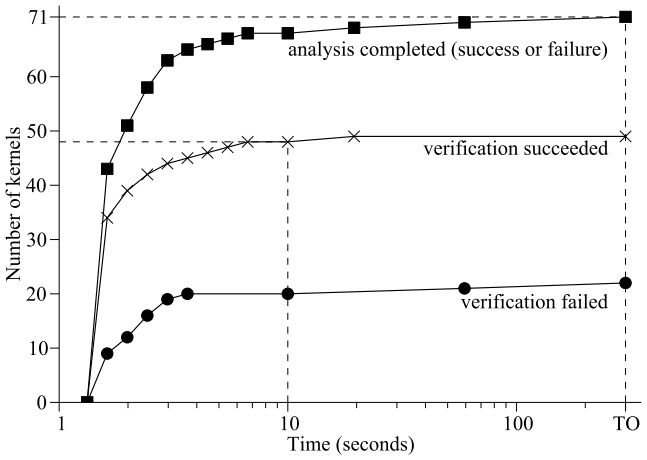


Figure 13: Cumulative histogram showing the times for successful and unsuccessful verification with GPUVerify for the *evaluation* set.

We had to add invariants manually to a number of the training set benchmarks to enable verification. In two cases these were bespoke invariants, as mentioned above. In the remaining cases we had to add a general sort of relational invariant commonly required for kernels that perform tree reductions. Tree reductions are often written in the following form (where `SZ` denotes group size and is assumed to be a power of two):

```
int offset = 1;
for (int d = SZ; d > 0; d >>= 1) {
  barrier();
  offset *= 2;
  if (lid < d) {
    // Write to shared array using function of 'offset' and 'lid'
  }
}
```

In this case, invariants asserting that `offset` and `d` are individually powers of two, as in the final rule discussed in Section 4.3, do not suffice. A relational invariant between `offset` and `d` is also required:

```
(d==SZ && offset==1 ) || (d==SZ/2 && offset==2) || ... ||
(d==1 && offset==SZ) || (d== 0 && offset==2*SZ))
```

We have not yet devised a general-purpose heuristic for inferring this sort of relational invariant.

Results for the evaluation set. Figure 13 summarises analysis times for GPUVerify applied to the evaluation set. This plot shows three cumulative histograms. The cumulative histogram whose points are crosses relates to benchmarks for which verification succeeded: a cross with coordinates (x, y) indicates that for y kernels, verification *succeeded* in x seconds or fewer. The cumulative histogram whose points are circles relates to benchmarks for which verification failed, either due to the kernel being incorrect, or due to invariant inference being too weak. A circle with coordinates (x, y) indicates that for y kernels, verification *failed* in x seconds or fewer. Finally, the cumulative histogram whose points are squares relates to benchmarks in either of the previous two categories. This indicates the *responsiveness* of GPUVerify: a square at (x, y) indicates that for y benchmarks, GPUVerify terminated, reporting either success or failure, within x seconds. Thus the y coordinate of each square is the sum of the y coordinates of the associated cross and circle: if verification succeeded and failed in x seconds or fewer for y_s and y_f kernels, respectively, then verification terminated in x seconds or fewer for $y_s + y_f$ kernels.

Using the inference techniques devised with respect to the training set (c.f. Section 4.3), GPUVerify was able to verify 49 out of the 71 evaluation set kernels (69 %) *fully automatically*. Of these kernels, 48 were verified in 10 seconds or fewer, and the longest verification time was 17 seconds. As for the training set, with race checking disabled, GPUVerify was able to prove divergence freedom fully automatically for all evaluation benchmarks, in under 10 seconds per kernel.

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. 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 71 evaluation kernels is a good result.

The results also show that, with one exception, the response time of GPUVerify, whether or not verification succeeds, is reasonable. The top cumulative histogram shows that verification terminated within 10 seconds for 68 of the kernels, and the response time for 70 out of the 71 kernels was less than 59 seconds. We believe that response time is critically important for practical uptake of the tool. Given that GPUVerify will frequently be applied to incorrect kernels, and accepting that invariant inference cannot be perfect, it is encouraging that GPUVerify's runtime is relatively impervious to whether verification succeeds or fails. Verification of one kernel timed out: this is a loop-free FFT implementation consisting of 576 ELOC. After translation to Boogie, reduction to a sequential program and full procedure inlining, the resulting Boogie program is almost 10,000 lines, resulting in a huge verification condition. Our current inference rules for procedure contracts were not sufficient to enable modular verification of this kernel.

Over all kernels that were successfully verified, 81 % of the candidate invariants speculated by GPUVerify proved to be true. This relatively high percentage indicates that our invariant generation rules (Section 4.3) are usually firing in cases where generated candidates turn out to be useful.

In addition to the FFT example, we have manually inspected the other 21 kernels for which GPUVerify reported verification failure. Figure 14 summarises 13 cases where verification would succeed with more sophisticated invariant inference. A further kernel verifies with a complex be-

Reason for verification failure	Solution
Slight variations of the access patterns recognised by our current inference rules are used (**7 kernels**)	Generalise existing inference rules
Access patterns should be recognised by our inference rules, but rules do not fire because components of an integer vector, rather than integer variables, are used for indexing (**3 kernels**)	Enhance existing inference rules to be sensitive to vector components
Kernel employs tree reduction in the form described above, and verifies with corresponding relational invariant (**3 kernels**)	Design inference rules for tree reduction invariants

Figure 14: Common causes of verification failure for evaluation set kernels, and planned improvements to inference

spoke invariant, which does not appear to conform to a general pattern. Verification of five kernels failed due to missing preconditions on kernel parameters. One example is a kernel whose race freedom depends upon the identity $a + b \geq a$, where a and b are positive values read from the shared state. If a and b are sufficiently large then, with 32-bit integers, this identity does not hold. Without a precondition stating that the contents of the shared state are suitably bounded, GPUVerify reports a data race. We believe that GPUVerify can be a useful as a tool to help GPU kernel programmers explicitly understand and state the preconditions their kernels assume. For one kernel GPUVerify reports a read-write race which we discovered to be benign: the writing thread is guaranteed to write the same value which the reading thread is about to read. Finally, we encountered one false positive arising from an array index being derived from floating point input data.

Detection of a bug in previous CUDA SDK example. Using GPUVerify we discovered a write-write data race in the N-body example that shipped with the CUDA SDK v2.3. This example uses multiple CUDA kernels to numerically approximate a system of N interacting bodies [33]. This is an ideal problem for parallelisation since 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 to a one-dimensional 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 optimisation for small values of N where threads are arranged in two-dimensional 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 sub-calculation, 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 master thread's final update, using the full summation, to be overwritten by partial results computed by other threads.

We reported this data race to Lars Nyland 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." [32]. NVIDIA had subsequently fixed this bug in v3.0 of the CUDA SDK.

5.2 Comparison of GPUVerify with PUG

We present a head-to-head comparison of PUG and GPUVerify on our CUDA benchmarks: the CUDA SDK and C++ AMP suites (40 kernels). Recall that we removed kernels from these suites which use atomic operations or write to the shared state using double-indirection: PUG is also incapable of reasoning about these features. The CUDA SDK benchmarks were previously used to evaluate PUG [21]. The C++ AMP benchmark suite consists of kernels we translated manually from C++ AMP to CUDA, which we then adapted to allow for documented limitations of PUG's front-end. We made a special effort not to modify these kernels in any way which would make them easier for GPUVerify to handle.

Both GPUVerify and PUG represent integers using bit-vectors. GPUVerify always uses 32 bits for variables of `int` type, as this is required by both CUDA and OpenCL. PUG allows the user to specify which bit width should be used for integers. In the evaluation of [21], custom bit widths were chosen on a benchmark-by-benchmark basis. To make the current comparison fair, we always run PUG in 32-bit mode: we believe that this is essential for verification purposes as smaller bit widths can change the semantics of kernels under analysis.

For all experiments, we use a five minute timeout for both PUG and GPUVerify.

Results for correct benchmarks. We found that PUG reported false positive for three kernels. These are kernels whose correctness depends upon threads agreeing on the contents of the shared state. GPUVerify is able to reason about these kernels using the *equality* abstraction (Section 4.2). PUG's shared state abstraction is equivalent to our *adversarial* abstraction, which is not sufficient for these kernels. Note that GPUVerify decides automatically which shared state abstraction to use.

Figure 15 compares the performance of GPUVerify and PUG on the 37 kernels that both tools could handle. A point with coordinates (x, y) corresponds to a kernel which took x and y seconds to be verified by GPUVerify and PUG respectively. Points lying above/below the diagonal correspond to kernels where GPUVerify performed better/worse

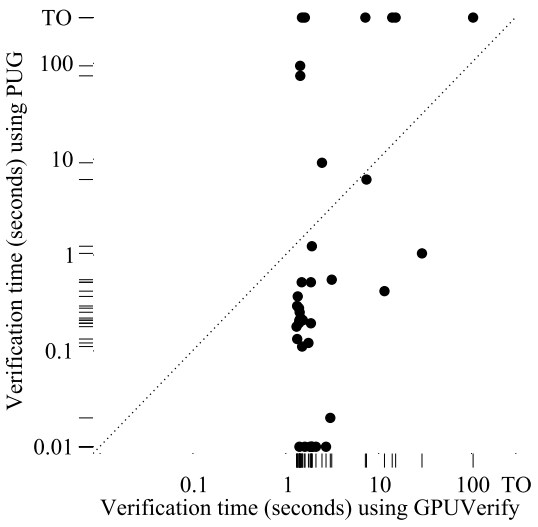


Figure 15: Scatter plot comparing the performance of GPUVerify and PUG on the CUDA SDK and C++ AMP kernels

than PUG. Points at the very top of the graph correspond to kernels for which PUG timed out. The axes use a log scale.

The plot shows that PUG is on average faster than GPUVerify, but that PUG's worst-case performance is significantly worse than GPUVerify's: the timeout of 5 minutes is reached by PUG for six kernels, but never reached by GPUVerify. We also find that PUG runs extremely quickly, taking only a tiny fraction of a second, for six kernels. We conjecture that in these cases PUG may be able to infer race freedom though cheap syntactic checks, without invoking its constraint solver.

Results for buggy benchmarks. We have also compared GPUVerify and PUG to see how quickly they report proof failures when applied to buggy kernels. We randomly injected a mutation into each kernel in the CUDA SDK and C++ AMP benchmark suites. First, we used a script to choose, for each kernel, a random mutation and a random location within the kernel to apply the mutation. These mutations were chosen to elicit either a data race (for example, removing a barrier or adding a racy access) or barrier divergence (for example, adding a barrier where control flow is non-uniform). The script places its suggestions as comments within each kernel. Secondly, we took each kernel and examined the suggested mutation. If it was sensibly placed and would give rise to buggy behaviour we implement the mutation by hand; otherwise, we reran the script to generate a fresh mutation suggestion, repeating the process until a suitable mutation was generated.

We found that GPUVerify's proof attempts generally failed within around 5 seconds, whereas PUG's proof attempt failed usually 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 data races. GPUVerify reported no false negatives.

6. Related work

There are numerous existing dynamic and static techniques for data-race detection in programs using lock-based synchronization or fork-join parallelism; a full discussion of these techniques is beyond the scope of this paper. We note however that this paper is concerned with proving race- and 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. We conclude the section with a brief discussion of invariant generation techniques.

PUG. The closest work to GPUVerify is the PUG analyzer for CUDA kernels [21]. Although GPUVerify and PUG have 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 race- and 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 simultaneously with translation of the GPU kernel into a logical formula. PUG provides a set of built-in loop summarisation rules which 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 resulting in 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.

Formal semantics for GPU kernels. A recent paper studying the relationship between the lock-step execution model of GPUs and the standard interleaved semantics for threaded programs presents a formal semantics for predicated execution [14]. This semantics shares similarities with the SDV semantics we present in Section 3, but the focus of [14] is not on verification of GPU kernels.

Symbolic execution and bounded-depth verification. The GKLEE [23] and KLEE-CL [6] tools perform dynamic sym-

bolic execution of CUDA and OpenCL kernels, respectively, and are both built on top of the KLEE symbolic execution engine [5]. A method for bounded verification of barrier-free GPU kernels via depth-limited unrolling to an SMT formula is presented in [38]; lack of support for barriers, present in most non-trivial 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. A further advantage of GKLEE and KLEE-CL is that because they are based on KLEE, which works on LLVM bytecode, they can be applied to GPU kernels after optimization and thus have the potential to detect bugs that result from incorrect compiler optimizations. The major drawback to these methods is that they cannot verify freedom of defects for non-trivial kernels.

The GKLEE tool specifically targets CUDA kernels, and faithfully models lock-step execution of sub-groups of threads, or *warps* as they are referred to in CUDA (see Figure 2). This allows precise checking of CUDA kernels that deliberately exploit the warp size of an NVIDIA GPU to achieve high performance. In contrast, GPUVerify makes no assumptions about sub-group size, making it useful for checking whether CUDA kernels are portable, but incapable of verifying kernels whose correctness depends on implicit warp-level synchronization.

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 uses the notion of *parametric flows* to soundly restrict defect checking to consider only certain pairs of threads [22]. 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 inter-thread communication.

Dynamic analysis. Dynamic analysis of CUDA kernels for data race detection has been proposed [4]. A recent paper [20] reports a technique that combines dynamic and static data race analysis: a CUDA kernel is simulated with dynamic race checking. 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 which GPUVerify can verify only with the *equality* abstraction, due to threads testing input data, are not be amenable to analysis using the technique of [20].

Other approaches. A recent approach to construction of correct parallel programs is based on *thread contracts* [16]. A programmer specifies the coordination and data sharing strategy for their multi-threaded 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.

Invariant generation. As described in Section 4.3, we use the Houdini algorithm [12] to generate loop invariants for verification. Houdini was introduced as an annotation assistant for the Java Extended Static Checker [19]. Related template-based invariant generation techniques include [15, 18, 35]. As discussed in the related work section of [12], Houdini can be viewed under the framework of abstract interpretation [7] where the abstract domain is conjunctions of predicates drawn from the set of candidate invariants. Compared with standard predicate abstraction [13], 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 generate invariants in response to failed or partial verification attempts [3, 26], while the Daikon technique [11] allows program-specific invariants to be speculated through dynamic analysis. We plan to draw upon these techniques to improve GPUVerify's invariant inference in future work.

7. Conclusions

We have provided an operational semantics for GPU kernels, and used this semantics to design a novel technique for formal verification of race- and divergence-freedom. Through a large experimental evaluation we have demonstrated that our implementation of this technique, GPUVerify, is effective in verifying and falsifying real-world OpenCL and CUDA GPU kernels.

Future work will involve extending the reach of GPUVerify by supporting atomic operations, investigating more sophisticated strategies for inferring loop invariants and procedure specifications, and devising richer shared state abstractions.

Acknowledgements

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 for their insightful feedback on an earlier draft of this work.

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

References

[1] AMD. AMD Accelerated Parallel Processing (APP) SDK. `developer.amd.com/sdks/amdappsdk/pages/default.aspx`

[2] M. Barnett, B. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.

[3] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, 2007.

[4] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of CUDA programs. In *STMCS*, 2008.

[5] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[6] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic testing of OpenCL code. In *HVC*, 2011.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[8] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[9] A. Donaldson, D. Kroening and P. Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *TACAS*, 2010.

[10] A. Donaldson, D. Kroening and P. Rümmer. Automatic analysis of DMA races using model checking and *k*-induction. *Formal Methods in System Design* 39(1):83-113.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69(1-3):35-45.

[12] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, 2001.

[13] S. Graf and H. Sa idi. Construction of abstract state graphs with PVS. In *CAV*, 1997.

[14] A. Habermaier and A. Knapp. On the correctness of the SIMT execution model of GPUs. In *ESOP*, 2012.

[15] T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In *NFM*, 2011.

[16] R. K. Karmani, P. Madhusudan, and B. Moore. Thread contracts for safe parallelism. In *PPOPP*, 2011.

[17] Khronos OpenCL Working Group. The OpenCL specification, version 1.1, 2011. Document Revision: 44.

[18] S. K. Lahiri and S. Qadeer. Complexity and algorithms for monomial and clausal predicate. In *CADE*, 2009.

[19] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[20] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner. Verifying GPU kernels by test amplification. In *PLDI*, 2012.

[21] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *FSE*, 2010.

[22] G. Li, P. Li, G. Gopalakrishnan. Parametric flows: automated behaviour equivalencing for symbolic analysis of races in CUDA programs. In *SC*, 2012.

[23] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *PPOPP*, 2012.

[24] llvm.org. `clang`: a C language family frontend for LLVM. `clang.llvm.org`

[25] A. Lokhmotov. Mobile and embedded computing on Mali GPUs. In *UK GPU Computing Conference*, 2011.

[26] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[27] Microsoft Corporation. C++ AMP sample projects for download (MSDN blog). `blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx`

[28] D. Moth and Y. Levanoni. Microsoft's C++ AMP unveiled. `www.drdobbs.com/windows/231600761`

[29] NVIDIA. CUDA Toolkit Release Archive. `developer.nvidia.com/cuda-toolkit-archive`

[30] NVIDIA. CUDA C programming guide, v4.0, 2011.

[31] NVIDIA. PTX: Parallel thread execution ISA, v2.3, 2011.

[32] L. Nyland. Personal communication, April 2012.

[33] L. Nyland, M. Harris, and J. Prins. Fast N-body simulation with CUDA. *GPU Gems 3*, Chapter 31. Addison-Wesley, 2007.

[34] Rightware Oy. Basemark CL. `www.rightware.com/en/Benchmarking+Software/Basemark%99+CL`

[35] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, 2009.

[36] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[37] S. Stone, J. Haldar, S. Tsao, W. Hwu, B. Sutton, and Z. Liang. Accelerating advanced MRI reconstructions on GPUs. *J. Parallel Distrib. Comput.*, 68(10):1307–1318, 2008.

[38] S. Tripakis, C. Stergiou, and R. Lublinerman. Checking non-interference in SPMD programs. In *HotPar*, 2010.