Electronic Communications of the EASST
Volume 70 (2014)

Proceedings of the
14th International Workshop on
Automated Verification of Critical Systems (AVoCS 2014)

The GPUVerify Method: a Tutorial Overview

Alastair F. Donaldson

16 pages

Guest Editors: Marieke Huisman, Jaco van de Pol
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

The GPUVerify Method: a Tutorial Overview

Alastair F. Donaldson[*]

Imperial College London

Abstract: I present a tutorial overview demonstrating the key technique used by GPUVerify, a static verification tool for graphics processing unit (GPU) kernels. The technique is a method for translating a massively parallel GPU kernel into a sequential program such that correctness of the sequential program implies data race-freedom of the parallel kernel.

Keywords: Formal verification, graphics processing units, predicated execution, concurrency

1 Introduction

GPUVerify is a formal verification technique and tool for the analysis of GPU kernels—programs designed to be executed in parallel on *graphics processing units*—with respect to two types of defects: *data races* and *barrier divergence* [BCD+12, BBC+14].[1] In this invited paper I present a tutorial overview of the key step undertaken by GPUVerify to allow scalable verification: the translation of a parallel GPU kernel into a sequential program such that analysis of the sequential program yields results about the original parallel kernel. The material presented here has been developed in the process of giving a number of seminars on GPUVerify, teaching about GPUVerify on the Software Reliability course at Imperial College London, and recording videos presenting the project.[2] My hope is that this exposition provides a clearer overview of how the method works compared with the technical presentation in the original GPUVerify paper [BCD+12].

After providing some background on GPU programming (Section 2), the bulk of the tutorial focuses on the key steps involved in the GPUVerify verification method (Section 3). This is followed by a brief overview of the GPUVerify tool chain (Section 4). The paper concludes with a survey of some related work in GPU kernel analysis and verification (Section 5).

2 Background on GPU programming

Originally designed to accelerate graphics processing, a graphics processing unit (GPU) has many parallel processing elements: graphics operations are inherently parallel. Early GPUs had limited functionality, tailored specifically towards graphics computations. Recently GPU designs have become more powerful and general purpose, and are now widely used in parallel programming to accelerate tasks including (among many others, and citing only a small subset of works):

[*] This work was supported by the EU FP7 STREP project CARP (project number 287767), and EPSRC project EP/K011499/1.

[1] GPUVerify can be accessed at http://multicore.doc.ic.ac.uk/tools/gpuverify.

[2] Introductory video on GPUVerify: https://www.youtube.com/watch?v=l8ysBPV8OvA

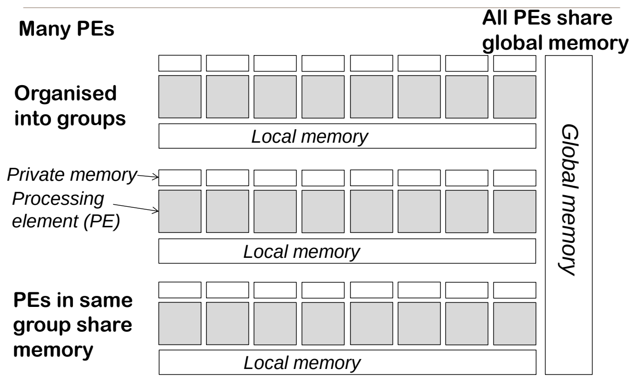


Figure 1: Overview of the structure of a typical GPU architecture.

medical imaging [CLW04], computer vision [SNS$^+$13], computational fluid dynamics [Har04] and DNA sequence alignment [LD09].

Figure 1 shows the structure of a typical GPU architecture (akin in essence to state-of-the-art architectures from NVIDIA and AMD). The chip consists of a number of *processing elements* (PEs) each equipped with a small amount of *private* memory. PEs are organised into *groups* such that PEs within a group share memory (called *local memory* in this paper). The set of groups of PEs is sometimes referred to as a *grid*. The GPU is also equipped with *global memory* shared among all PEs. In some implementations this global memory is physically separate from main memory; in other implementations global memory is part of the same physical memory.

Data races A *data race* occurs in a GPU kernel if:

- Two distinct threads access the *same* memory location
- At least one access is a *write*
- The accesses are not separated by a *barrier synchronisation* operation

Races in GPU kernels can be classified as *inter-group* or *intra-group*. Inter-group races are between threads executing on PEs in distinct groups. Necessarily, such races must occur with respect to global memory since this is the only memory that threads on distinct PEs can share. Intra-group races are between threads executing on PEs in the same group, and can be with respect to either global memory or the group's local memory.

Data races in GPU kernels are usually indicative of programmer errors, and lead to nondeterministic bugs that can be hard to reproduce and fix. A problem which is specific to data races in GPU kernels is that individual GPU architectures may be relatively deterministic: as there is no operating system running on the GPU, GPU threads are not preempted at unpredictable moments as is the case in concurrent CPU applications. This means that a data race may resolve deterministically and in an apparently bug-free manner on a particular GPU architecture, thus evading detection through testing, and then resolve differently, causing a crash or erroneous results, when an application is deployed on another platform, perhaps in a customer device. Unlike

```
kernel void add_neighbour(local int * A, int offset) {
  A[tid] = A[tid] + A[tid + offset];
}
```

Figure 2: A simple GPU kernel that exhibits a data race.

with data races in system-level CPU applications, which are often deliberate or regarded as benign, races in GPU kernels are almost always accidental and unwanted, thus there is a clear need for techniques and tools to help detect and eliminate them.

Example GPU kernel GPUs are typically programmed by writing a *kernel*—a function to be executed simultaneously by many threads running across the processing elements of the chip—using a low-level programming model such as the industry standard Open Computing Language (OpenCL) [Khr12], or the NVIDIA-specific Compute Unified Device Architecture (CUDA) [NVI11]. Figure 2 shows a simple GPU kernel written in OpenCL C (a superset of a subset of the C99 language). The `kernel` keyword indicates that the `add_neighbour` function is a kernel entry point. The kernel takes two arguments: `A`, which is a pointer to an array of integers residing in *local* memory (indicated by the `local` keyword), and `offset`, an integer. Every thread executing the kernel runs the `add_neighbour` function, receiving identical values for the `A` and `offset` parameters.

In order to access distinct data values, a thread can use a built-in variable, *tid*, providing access to the thread's unique identifier.[3] A thread executing this kernel reads from `A` at offsets `tid` and `tid + offset`, sums these values and writes the result to `A` at offset `tid`. This example illustrates a read-write data race. For example, if `offset` is 1 then thread 0 will read from `A` at offset 1, thread 1 will write to `A` at offset 1, and there is no barrier synchronisation operation separating these accesses.

Barrier synchronisation and barrier divergence A **`barrier`** statement is used to synchronise threads in the same group: when a thread reaches a barrier it waits until *all* threads reach the barrier. When all threads have reached the barrier, the threads can proceed past the barrier, with the guarantee that reads and writes issued before the barrier have completed.

Barriers only allow synchronisation between threads in the same group; inter-group synchronisation within a kernel invocation is not possible.

The OpenCL 1.2 specification [Khr12] requires that threads should synchronise at syntactically identical barriers, making the following illegal:

```
if(tid == 0) barrier();
```

[3] In OpenCL, the `get_local_id` and `get_global_id` functions are in fact used to retrieve a thread's id within its work group and across all work groups, respectively. Because groups and grids of groups can be multi-dimensional these functions take arguments specifying in which dimension the id is required. For ease of presentation, in this paper multi-dimensional kernels are not considered, and we restrict attention to the case where there is a single group of threads, using *tid* to denote the id of a thread within this group. The GPUVerify tool supports reasoning about multi-dimensional kernels executed by multiple groups of threads.

```
kernel void add_neighbour(local int * A, int offset) {
  int temp = A[tid + offset];
  barrier();
  A[tid] = A[tid] + temp;
}
```

Figure 3: Using a barrier to avoid the data race exhibited by the example of Figure 2.

```
else barrier();
```

Furthermore, if a barrier is inside a loop then threads should hit the barrier with the same loop trip count. See [BCD+12] for discussion of an example which violates this requirement.

3 The GPUVerify kernel transformation method

The verification method employed by GPUVerify exploits the OpenCL programming model to transform a massively parallel kernel K into a sequential program P such that:

> P is correct (i.e. free from assertion failures) $\Rightarrow$ K is free from data races and barrier divergence.

At the conceptual level, the approach has four main ingredients:

1. Race analysis focuses on *barrier intervals*
2. Analysis is restricted to consider a single, canonical thread schedule, avoiding the need to reason about a large number of schedules
3. Analysis is further restricted to consider an arbitrary pair of threads executing the kernel, using abstraction to model the effects of further threads; this avoids the need to reason simultaneously about a large number of thread schedules
4. Predicated execution is applied to handle loops and conditionals, and to precisely capture the conditions for barrier divergence

Ingredients similar to 1–3 have been employed in other work on GPU kernel analysis [LG10, CCK14, LLS+12], while ingredient 4 is a novel contribution of the GPUVerify project [BCD+12]. Collectively, they allow kernel verification to be explicitly reduced to the analysis of a sequential program, allowing existing technology for sequential verification to be re-used. Careful re-use of sequential verification technology is a distinguishing characteristic of the GPUVerify method in comparison to related work (see Section 5).

Sections 3.1–3.3 elaborate on ingredients 1–3, respectively. Section 3.4 shows how these ideas can be used to perform transformation into a sequential program for kernels that do not exhibit conditional or looping code. Section 3.5 explains how conditionals and loops are handled using predicated execution.

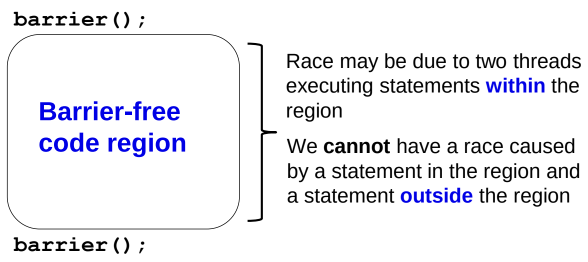

barrier();
Barrier-free code region
Race may be due to two threads executing statements within the region
We cannot have a race caused by a statement in the region and a statement outside the region
barrier();

Figure 4: Illustration of a barrier interval. It is sufficient to restrict intra-group race analysis to pairs of instructions that lie within a common barrier interval.

Throughout, for ease of presentation, attention is restricted to the detection of *intra*-group data races; the implementation in GPUVerify (Section 4) implements inter-group race checking in full.

3.1 Focusing race detection to barrier intervals

Observe that at any point during execution of a GPU kernel, two threads in the same group must be executing instructions that lie in the same *barrier interval*: a sequence of instructions starting and ending with a barrier, but otherwise barrier-free.

The notion of a barrier interval is illustrated in Figure 4. In practice, a barrier interval may be more complex; for example, a barrier occurring inside a loop may form a barrier interval with itself. Arbitrary barrier intervals can be handled by:

- Logging and checking all accesses made by threads to the shared state, aborting if a data race is detected
- Resetting the access logs each time a barrier is reached

3.2 Restricting to a canonical thread schedule

Restricting analysis to barrier intervals has the potential to increase scalability, as it allows a kernel with multiple barriers to be analysed interval-by-interval. However, if there are n threads and a barrier interval contains k instructions then, assuming arbitrary thread interleavings can occur, there are $O(n^k)$ threads schedules. Exploring all such thread schedules to detect data races would not be feasible.

The next observation is that if checking for data races occurring during the barrier interval is performed for some fixed schedule then either a data race will be detected (in which case verification can abort with an error), or the schedule will be shown to exhibit no data races, in which case it can be concluded that *no* schedule between these barriers can exhibit a data race. This observation has been exploited in several works [LG10, CCK14, LLS$^+$12, BCD$^+$12] and a proof that the reasoning is sound is presented in [LLS$^+$12]. Informally, the argument is that for a barrier interval there exist a set of *earliest races*: a race is an earliest race if there exists some thread schedule such that the race is the first race to be observed during the schedule. If a schedule exhibits a data race then it must exhibit an earliest race, and a little reasoning shows that, for a thread schedule σ associated with barrier interval I:

$$\begin{gathered}\sigma \text{ exhibits an earliest race} \\ \Leftrightarrow \\ \text{every schedule for barrier interval } I \text{ exhibits an earliest race}\end{gathered}$$

Considering an arbitrary canonical schedule thus suffices for race analysis.

Restricting race analysis to a single schedule has two related and significant advantages. For a barrier interval of length k executed by n threads:

- The number of schedules that need to be considered is reduced from $O(n^k)$ to 1.
- The chosen schedule can be used to rewrite the barrier interval as a sequential program consisting of $n \cdot k$ instructions, one for each thread, plus instructions to perform race logging and checking. The instructions appear in the order dictated by the schedule.

3.3 The two thread abstraction

Restriction to a canonical schedule has the benefit of bringing the analysis task into the realm of *sequential* program verification, on which steady progress has been made now for several decades. However, verifying a barrier interval via a sequential program of length $n \cdot k$, where n is the number of threads and k the length of the barrier interval, is problematic if n is large. Because GPU kernels are often executed by thousands of threads, this is an issue in practice.

This sensitivity to the number of threads can be avoided by exploiting the fact that data races and barrier divergence are both defects that occur between *pairs* of threads: a data races occurs due to a conflict between precisely two threads; similarly barrier divergence occurs when two threads reach distinct barriers (or hit the same barrier having executed different numbers of iterations of loops enclosing the barrier).

Thus, when checking correctness of a barrier interval, it is sufficient to check the interval for every pair of threads separately. This can be achieved by checking that the barrier interval is free from races and divergence for an *arbitrary* pair of distinct threads.

If a data race exists then, as argued above, there must exist an earliest race between some pair of threads, s and t say. Attempting to check race-freedom for an arbitrary pair of threads i and j must include reasoning about the pair s and t, which will lead to discovery of the earliest race between s and t.

Care must be taken to extend this trick to the analysis of multiple barrier regions. After threads synchronise at a barrier, it is legitimate for a thread's execution to depend on a value computed by a different thread before the barrier. If just a pair of threads, i, j say, are modelled, then the effects across barriers of a thread k different from i and j will not be accurately represented.

This can be accounted for by *abstracting* the shared state, i.e. local and global memory. The simplest approach, termed *adversarial abstraction* [BCD+12], is to treat the shared state as being completely abstract, so that each time a thread reads from the shared state an arbitrary value is retrieved; in this case writes to the shared state need not be modelled at all. A slightly richer abstraction, the *equality* abstraction, is discussed and compared with the adversarial abstraction in [BCD+12], and *barrier invariants*, a tunable shared state abstraction technique for establishing richer properties, are proposed in [CDK+13]. Adversarial abstraction is considered throughout this paper.

3.4 Transformation for straight-line kernels

The program transformation performed by GPUVerify is now explained for the case of single procedure straight-line kernels. Handling of conditionals and loops is described in Section 3.5.

Assume that a kernel has the following form:

```
kernel void foo(<parameters, including local arrays>) {
  <private variable declarations>
  S1; S2; ... Sk;
}
```

where each statement S_i has one of the following forms, where x denotes a private variable, e an expression over private variables, and A a `local` array:

- $x = e$ (private assignment)
- $x = A[e]$ (read from `local` array)
- $A[e] = x$ (write to `local` array)
- **`barrier`**`()` (barrier statement)

These assumptions ensure that a statement includes at most one read from local memory and at most one store to local memory. Pre-processing can be used to trivially transform statements that perform multiple loads and stores into this form. Furthermore, this is typical of the way a kernel is represented after compilation into a compiler intermediate representation such as LLVM bytecode.

Assume also that all local array parameters to the kernel refer to *disjoint* arrays.

The aim is to transform a kernel K into a sequential program P that:

- Captures execution of two arbitrary threads using some fixed schedule
- Detects data races
- Treats the shared state abstractly to model the effects of other threads

Introduction of distinct thread ids Two symbolic constants are introduced in P, tid$1 and tid$2, to represent the ids of two distinct but otherwise arbitrary threads. These conditions are encoded by the following preconditions on P, there `n` denotes the number of threads executing the kernel:

```
0 <= tid$1 && tid$1 < n
0 <= tid$2 && tid$2 < n
tid$1 != tid$2
```

The first and second conditions require that the ids of the threads under consideration are within the valid range of thread ids. The third condition requires that the ids are distinct.

In what follows, “thread 1” or “the first thread” refers to the thread whose id is recorded by $\mathsf{tid}\$1$, and “thread 2” or “the second thread” refers to the thread whose id is recorded by $\mathsf{tid}\$2$. It is important to note that this does *not* refer specifically to the threads whose ids are 1 and 2.

Removal of array parameters If K has a local array parameter then this parameter *does not* appear in P. This is because the shared stated of K will be represented abstractly in P by eliding writes, and replacing a reads into a private variable x with a non-deterministic assignment to x. After this abstraction, local arrays have no role.

Dualisation of non-array parameters If K has a non-array parameter, a of type T say, then P has two non-array parameters, $a\$1$ and $a\$2$, both with type T. Parameter $a\$1$ represents the first thread’s copy of the original parameter a, and likewise $a\$2$ represents the second thread’s copy. These parameters initially have identical values, enforced by the precondition $a\$1 \;==\; a\$2$.

Dualisation of private variables If K has a private, function-scope variable x of type T then P has two private, function-scope variables, $x\$1$ and $x\$2$, each of type T. Because private variables initially have arbitrary values which may be distinct between threads there is no precondition relating $x\$1$ and $x\$2$.

Race logging and checking routines For each local array A in K, P is equipped with two sets of integers, R_A and W_A, which record reads from A and writes to A respectively. A precondition ensures that initially both R_A and W_A are empty.

The program P is also equipped with four procedures:

- `LOG_READ_A`: takes an integer parameter and adds the parameter’s value to R_A
- `LOG_WRITE_A`: takes an integer parameter and adds the parameter’s value to W_A
- `CHECK_READ_A`: takes an integer parameter and aborts if the parameter’s value belongs to W_A
- `CHECK_WRITE_A`: takes an integer parameter and aborts if the parameter’s value belongs to either R_A or W_A

In addition, P is equipped with a `barrier` procedure, which has the effect of setting R_A and W_A to the empty set for every local array A.

In this paper a discussion of how the sets R_A and W_A, the associated `LOG` and `CHECK` procedures, and the `barrier` procedure are efficiently implemented is omitted. Two alternative implementations, both of which avoid the use of quantifiers, have been presented in prior work [BCD+12, BBC+14].

Translation of statements For an expression e over private variables and $i \in \{1,2\}$, we use $e\$i$ to denote the expression e with every occurrence of a private variable x replaced with $x\$i$. One can read $e\$i$ as “e in the context of thread i”. For instance, if e is $a + \mathsf{tid} - x$ then $e\$2$ is $a\$2 + \mathsf{tid}\$2 - x\$2$.

Statement in K	**Corresponding statement in P**	**Notes**
$x = e$;	$x\$1 = e\$1$; $x\$2 = e\$2$;	Each thread executes the assignment

$x = A[e]$;	`LOG_READ_A`($e\$1$);
	`CHECK_READ_A`($e\$2$);
	`havoc`($x\$1$);
	`havoc`($x\$2$);
		Thread 1 logs the read
		Thread 2 checks the read
		Each thread sets its copy of x to an arbitrary value

$A[e] = x$;	`LOG_WRITE_A`($e\$1$);
	`CHECK_WRITE_A`($e\$2$);
		Thread 1 logs the write
		Thread 2 checks the write
		Because A has been removed, the write itself is not modelled in P.

`barrier();`	`barrier();`	Barrier clears every R_A and W_A

Table 1: Translation of kernel statements into sequential program statements, in the absence of conditionals and loops.

Table 1 shows how the forms of statement for a straight line kernel K are translated into two-threaded form in the program P. Assignments and barriers are straightforward.

A read is translated as a call to the appropriate `LOG_READ` procedure with thread 1's offset, and a call to the appropriate `CHECK_READ` procedure with thread 2's offset. Thus thread 1 takes responsibility for logging where it read from, and thread 2 makes sure that its read does not conflict with any write previously issued by thread 1. If x is the receiving variable for the read then both copies of this variable, $x\$1$ and $x\$2$, need to be updated in P. Because the shared state is abstracted, the values these variables should take after the read has occurred is unknown. Thus `havoc` is used to set each variable to an arbitrary value.

A write is similarly translated as a call to the appropriate `LOG_WRITE` procedure with thread 1's offset, and a call to the appropriate `CHECK_WRITE` procedure with thread 2's offset. Analogous to the case for reads, thread 1 takes responsibility for logging where it wrote to, and thread 2 makes sure that its write does not conflict with any write or read previously issued by thread 1. Due to shared state abstraction, there is no need to reflect the actual array write in P, because the array in question is not present.

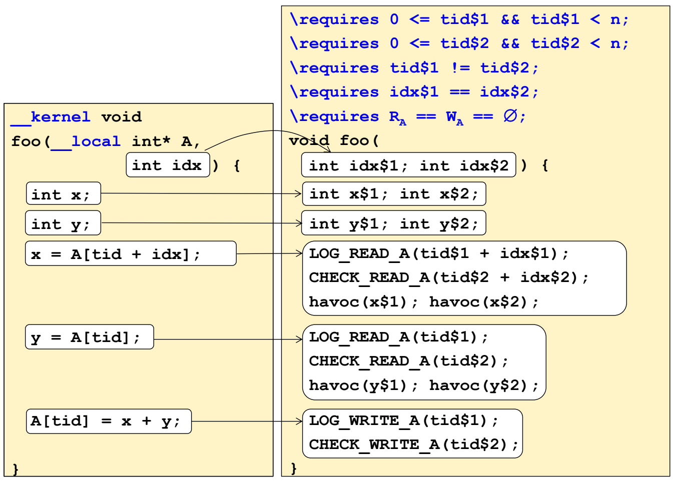
A complete example of the translation process is shown in Figure 5.

3.5 Handling conditionals and loops using predicated execution

Loops and conditionals are handled using *predicated execution*. The essence of predicated execution is to flatten conditional code into straight line code. For example, the following fragment of C code:

```
if(x < 100) {
  x = x + 1;
} else {
  y = y + 1;
}
```

can be flattened into straight line code through the introduction of two predicates, `P` and `Q`, where `P` records the truth of `x < 100` and `Q` the truth of `!(x < 100)`, as follows:



```
__kernel void
foo(__local int* A,
              int idx) {
  int x;
  int y;
  x = A[tid + idx];
  y = A[tid];
  A[tid] = x + y;
}
```

```
\requires 0 <= tid$1 && tid$1 < n;
\requires 0 <= tid$2 && tid$2 < n;
\requires tid$1 != tid$2;
\requires idx$1 == idx$2;
\requires R_A == W_A == ∅;
void foo(
  int idx$1; int idx$2) {
  int x$1; int x$2;
  int y$1; int y$2;
  LOG_READ_A(tid$1 + idx$1);
  CHECK_READ_A(tid$2 + idx$2);
  havoc(x$1); havoc(x$2);
  LOG_READ_A(tid$1);
  CHECK_READ_A(tid$2);
  havoc(y$1); havoc(y$2);
  LOG_WRITE_A(tid$1);
  CHECK_WRITE_A(tid$2);
}
```

Figure 5: Illustration of the transformation process for a simple kernel. The parallel OpenCL kernel on the left is transformed into the sequential program on the right.

```
P = (x < 100);
Q = !(x < 100);
x = P ? x + 1 : x;
y = Q ? y + 1 : y;
```

One can easily verify that, in this form, the code fragment computes the same result as the original. In effect, both sides of the conditional statement are executed, but the *then* side has no effect if `x < 100` does not hold (in which case `P` is *false*) and the *else* side has no effect if `x < 100` does hold (in which case `Q` is *false*).

Predication is used in the two-thread sequential encoding of a GPU kernel by flattening conditionals so that both threads execute both sides of every conditional, and modifying loops so that both threads execute the same number of iterations of each loop, but maintaining at all times a predicate for each thread determining whether the thread is enabled. If a thread is not enabled, it executes statements but these statements have no effect.

First, the the `LOG` and `CHECK` procedures introduced in Section 3.4 are adapted so that each is equipped with a Boolean predicate parameter.

- `LOG_READ_A`: takes a Boolean predicate and an integer parameter; adds the parameter's value to R_A if and only if the predicate holds
- `LOG_WRITE_A`: takes a Boolean predicate and an integer parameter; adds the parameter's value to W_A if and only if the predicate holds
- `CHECK_READ_A`: takes a Boolean predicate and an integer parameter; aborts if and only if the predicate holds and the parameter's value belongs to W_A
- `CHECK_WRITE_A`: takes a Boolean predicate and an integer parameter; aborts if and only if the predicate holds and the parameter's value belongs to either R_A or W_A

Statement in K, Stmt	**Corresponding statement in** P, translate(Stmt, E)	**Notes**
$x = e$;	$x\$1 = E\$1$? $e\$1$: $x\$1$; $x\$2 = E\$2$? $e\$2$: $x\$2$;	Each thread executes the assignment if its predicate holds, otherwise the thread performs a no-op.
$x = A[e]$;	`LOG_READ_A`($E\$1, e\$1$); `CHECK_READ_A`($E\$2, e\$2$); `havoc`($tmp\$1$); `havoc`($tmp\$2$); $x\$1 = E\$1$? $tmp\$1$: $x\$1$; $x\$2 = E\$2$? $tmp\$2$: $x\$2$;	Thread 1 logs the read if its predicate holds Thread 2 checks the read if its predicate holds Each thread chooses an arbitrary value, then sets its copy of x to the arbitrary value if its predicate holds and performs a no-op otherwise.
$A[e] = x$;	`LOG_WRITE_A`($E\$1, e\$1$); `CHECK_WRITE_A`($E\$2, e\$2$);	Thread 1 logs the write if its predicate holds Thread 2 checks the write if its predicate holds Because A has been removed, the write itself is not modelled in P
`barrier();`	`barrier`($E\$1, E\$2$);	If both predicates hold then barrier clears every R_A and W_A.
Stmt$_1$; Stmt$_2$;	translate(Stmt$_1$, E); translate(Stmt$_2$, E);	Statements in a sequence are translated one by one.
if (e) { Stmt$_1$ } **else** { Stmt$_2$ }	$F\$1 = E\$1$ && $e\$1$; $F\$2 = E\$2$ && $e\$2$; $G\$1 = E\$1$ && !$e\$1$; $G\$2 = E\$2$ && !$e\$2$; translate(Stmt$_1$, F); translate(Stmt$_s$, G);	Each thread sets fresh predicates F and G to the values of e and !e respectively if enclosing predicate E holds, and to *false* otherwise. The *then* branch, Stmt$_1$, is translated w.r.t. F, and the *else* branch, Stmt$_2$, w.r.t. G.
while (e) { Stmt$_1$ }	$F\$1 = E\$1$ && $e\$1$; $F\$2 = E\$2$ && $e\$2$; **while** ($F\$1$ || $F\$2$) { translate(Stmt$_1$, F); $F\$1 = F\$1$ && $e\$1$; $F\$2 = F\$2$ && $e\$2$; }	Each thread sets fresh predicate F to the value of loop condition e if enclosing predicate E holds, and to *false* otherwise. Threads loop until *both* are disabled, but a thread performs no-ops if its predicate F is *false*. The predicate is re-evaluated on taking the loop back-edge.

Table 2: Complete rules for translation of kernel statements into sequential program statements, using predication to handle loops and conditionals. The top-level program statement sequence is translated with respect to the predicate *true*.

The `barrier` procedure of P is also adapted so that it additionally takes two predicate parameters, one for each thread. Execution of `barrier` aborts if and only if the predicate parameters differ. Otherwise the procedure sets R_A and W_A to the empty set if and only if both predicate parameters are *true*, leaving R_A and W_A untouched otherwise.

The rules of Table 2 show how a statement Stmt of K is translated with respect to predicate E to a statement translate(Stmt, E) in P. The predicate E determines whether each of the threads is enabled during execution of the statement. The top-level statement sequence of the kernel is translated with respect to the predicate *true* because initially both threads are enabled. The rules for conditionals and loops introduce stronger predicates on enabledness.

The rules for assignment, read and write mirror the straight-line rules of Table 1. The difference is that all components of the translated statement are guarded by the predicate E, so that if E does not hold for one of the threads then the thread performs no-ops. In the translation of reads, this difference requires the introduction of a special *tmp* variable in each thread which is used to temporarily store the nondeterministic result of a read. The value stored in *tmp* is then conditionally copied into receiving variable x depending on whether predicate E holds.

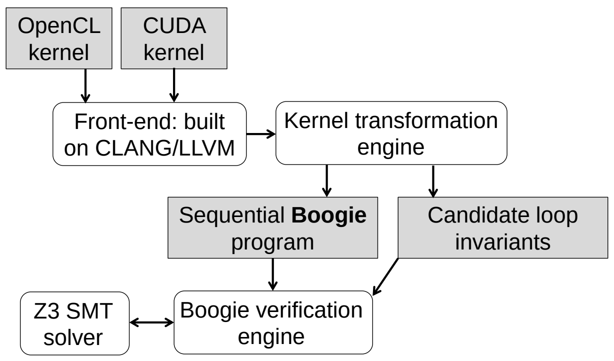

OpenCL kernel
CUDA kernel
Front-end: built on CLANG/LLVM
Kernel transformation engine
Sequential **Boogie** program
Candidate loop invariants
Z3 SMT solver
Boogie verification engine

Figure 6: Architecture of the GPUVerify tool.

A **`barrier`** is transformed into a call to the version of **`barrier`** that accepts predicate arguments. If both predicates are *true* then **`barrier`** does its usual job of clearing all read and write sets; if both predicates are *false* then **`barrier`** has no effect. If the predicates $E\$1$ and $E\$2$ disagree then execution aborts because *barrier divergence* has been detected.

Sequences of statements are translated in the obvious way. Conditionals are handled by introducing fresh predicates to record enabledness for each side of the conditional: if enclosing predicate E is *false* then both new predicates are also *false*, otherwise predicate F is set to the truth of the condition e, and G to its negation. Each side of the conditional is translated according to the appropriate predicate, and the conditional itself is eliminated.

Arguably the most interesting translation rule is the rule for loops. Unlike in the case for conditionals, predication does not *eliminate* loops, but rather transforms a loop into a form where each thread is guaranteed to execute the same number of iterations, so that in the generated sequential program the threads being modelled can enter and leave the loop uniformly. This is achieved by using a fresh predicate F to record whether each thread is still enabled during loop execution. This predicate is set to *false* if the enclosing predicate E does not hold, and otherwise is set to the value of the loop condition. The loop iterates until F becomes *false* for *both* threads. While F remains *true* for one of the threads, both threads execute the loop body, but their execution is predicated with respect to F. This means that if one of the threads has become disabled, execution of the body by this thread has no effect. On iterating the loop, F is recomputed: it is strengthened by the current value of the loop guard. Notice that once F has been set to *false* it cannot become *true*, thus once a thread has become disabled the thread cannot become enabled until loop exit.

4 Implementation in GPUVerify tool

The transformation described in this tutorial is at the heart of the GPUVerify tool. The architecture of GPUVerify is illustrated in Figure 6. As input, GPUVerify accepts a kernel written in

CUDA or OpenCL. A custom-built front-end based on the Clang/LLVM framework[4] parses the kernel into an intermediate form which is fed to the kernel transformation engine. The transformation engine implements the transformation of Section 3 and applies race instrumentation as described in [BCD$^+$12, BBC$^+$14], generating a sequential program expressed in the Boogie intermediate verification language [Lei08]. The tool also generates candidate loop invariants using heuristics described in [BCD$^+$12]. Verification of the sequential program is then delegated to the Boogie verification engine [BCD$^+$05], an open source verifier developed primarily by Microsoft Research.[5] Boogie uses the Houdini algorithm [FL01] to perform invariant inference over the candidate loop invariants and discharges verification conditions to an SMT solver. GPUVerify supports the Z3 [MB08] and CVC4 solvers [BCD$^+$11].

The architecture of Figure 6 shows that GPUVerify exhibits significant *re-use*: advantage is taken of Clang/LLVM, Boogie and Z3, which are widely used, robust components actively developed by expert research teams. This has significantly reduced the implementation effort required to make GPUVerify work, as well as increasing the reliability of the tool by virtue of other researchers and users continually improving the 3rd party components.

Two large experimental evaluations have demonstrated the scalability of GPUVerify, especially in terms of the number of threads that can be considered when verifying a kernel [BCD$^+$12, BBC$^+$14]. Notable aspects of the engineering effort associated with getting GPUVerify to work on realistic examples are also discussed in [BBC$^+$14].

5 Related work

Extensions to and applications of GPUVerify The core GPUVerify method was presented in [BCD$^+$12], with details on how to support unstructured control flow graphs, key to re-use of the Clang/LLVM infrastructure (see Figure 6), presented in [CDKQ13]. A richer shared state abstraction based on *barrier invariants* has been proposed and implemented in the tool [CDK$^+$13]. GPUVerify has also been extended to support for warp-based execution and specific use cases for atomic operations [BD14]. Engineering details associated with the project have appeared [BBC$^+$14], and the tool forms a key component in an automated verification method for parallel prefix sums [CDK14].

Other verification methods The closest work to the GPUVerify method is the PUG analyser for CUDA kernels [LG10]. Although GPUVerify and PUG have a similar goal, scalable verification of race-freedom for 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 it is only necessary 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

[4] http://llvm.org/

[5] http://boogie.codeplex.com/

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 [LG10]. In contrast, GPUVerify inherits flexible and sound invariant inference from the Houdini invariant inference algorithm [FL01] regardless of the complexity of the control structure of the GPU kernel.

A verification approach for CUDA kernels [LGA$^+$12] uses dynamic analysis to find data races at runtime. Then, if no data races are found, static analysis is used to determine whether control flow decisions at runtime were input-dependent. If not, the kernel is guaranteed to be data race-free. This method is in principle highly automatic for verifying race-freedom of input-independent kernels (though the implementation of the associated tool is not publicly available). However, it cannot be used to verify more complex examples where control flow can be input-dependent.

A permission-based separation logic has been developed for verifying GPU kernels, with a proof of race-freedom for a kernel being established through a consistent set of permission annotations for the kernel [HM13]. This approach provides a method for establishing richer functional properties than data race-freedom. However, the problem of automating the generation of permission annotations has not yet been studied in this line of work.

Symbolic execution and bounded-depth verification The GKLEE [LLS$^+$12] and KLEE-CL [CCK14] tools perform dynamic symbolic execution of CUDA and OpenCL kernels, respectively, and are both built on top of the KLEE symbolic execution engine [CDE08]. A method for bounded verification of barrier-free GPU kernels via depth-limited unrolling to an SMT formula is presented in [TSL10]; 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 optimisation 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.

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 [LLG12]. 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.

Formal semantics for GPU kernels The relationship between the lock-step execution model of NVIDIA GPUs and the standard interleaved semantics for threaded programs presents a formal semantics for predicated execution has been studied [HK12]. This semantics shares similarities with a "synchronous delayed visibility" semantics used to present GPUVerify [BCD+12] but the focus of [HK12] is not on verification of GPU kernels. A recent paper studying Hoare logic for GPU kernels is in a similar vein [KI13].

Acknowledgements: I am grateful to the following for their contributions to the GPUVerify project: Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Jeroen Ketema, Egor Kyshtymov, Dan Liew, Shaz Qadeer, Paul Thomson, John Wickerson. Thanks to Andrei Lascu for providing corrections to an earlier draft of this material. The related work section draws on material from [BCD+12] co-authored with Shaz Qadeer.

This work was supported by the EU FP7 project CARP and EPSRC grant EP/K011499/1.

Bibliography

[BBC+14] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, S. Qadeer. Engineering a Static Verification Tool for GPU Kernels. In *CAV*. LNCS 8559, pp. 226–242. 2014.

[BCD+05] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*. LNCS 4111, pp. 364–387. 2005.

[BCD+11] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, C. Tinelli. CVC4. In *CAV*. LNCS 6806, pp. 171–177. 2011.

[BCD+12] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, P. Thomson. GPUVerify: a Verifier for GPU Kernels. In *OOPSLA*. Pp. 113–132. 2012.

[BD14] E. Bardsley, A. F. Donaldson. Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels. In *NFM*. LNCS 8430, pp. 230–245. 2014.

[CCK14] P. Collingbourne, C. Cadar, P. H. J. Kelly. Symbolic Crosschecking of Data-Parallel Floating-Point Code. *IEEE Trans. Software Eng.* 40(7):710–737, 2014.

[CDE08] C. Cadar, D. Dunbar, D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. Pp. 209–224. 2008.

[CDK+13] N. Chong, A. F. Donaldson, P. Kelly, S. Qadeer, J. Ketema. Barrier Invariants: a Shared State Abstraction for the Analysis of Data-Dependent GPU Kernels. In *OOPSLA*. Pp. 605–622. 2013.

[CDK14] N. Chong, A. F. Donaldson, J. Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *POPL*. Pp. 397–410. 2014.

[CDKQ13] P. Collingbourne, A. F. Donaldson, J. Ketema, S. Qadeer. Interleaving and Lock-Step Semantics for Analysis and Verification of GPU kernels. In *ESOP*. Pp. 270–289. 2013.

[CLW04] J. E. Cates, A. E. Lefohn, R. T. Whitaker. GIST: an interactive, GPU-based level set segmentation tool for 3D medical images. *Medical Image Analysis* 8:217–231, 2004.

[FL01] C. Flanagan, K. R. M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *FME*. LNCS 2021, pp. 500–517. 2001.

[Har04] M. Harris. Fast Fluid Dynamics Simulation on the GPU. In *GPU Gems*. Volume 1, chapter 38. Addison-Wesley, 2004.

[HK12] A. Habermaier, A. Knapp. On the Correctness of the SIMT Execution Model of GPUs. In *ESOP*. LNCS 7211, pp. 316–335. 2012.

[HM13] M. Huisman, M. Mihelčić. Specification and Verification of GPGPU Programs using Permission-Based Separation Logic. In *BYTECODE*. 2013.

[Khr12] Khronos OpenCL Working Group. The OpenCL Specification, Version 1.2. 2012. Document Revision: 19.

[KI13] K. Kojima, A. Igarashi. A Hoare Logic for SIMT Programs. In *APLAS*. LNCS 8301, pp. 58–73. 2013.

[LD09] H. Li, R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* 25(14):1754–1760, 2009.

[Lei08] K. R. M. Leino. This is Boogie 2. Technical report, June 2008.
http://research.microsoft.com/apps/pubs/default.aspx?id=147643

[LG10] G. Li, G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *FSE*. Pp. 187–196. 2010.

[LGA$^+$12] A. Leung, M. Gupta, Y. Agarwal et al. Verifying GPU Kernels by Test Amplification. In *PLDI*. Pp. 383–394. 2012.

[LLG12] P. Li, G. Li, G. Gopalakrishnan. Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs. In *SC*. P. 29. 2012.

[LLS$^+$12] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, S. P. Rajan. GKLEE: Concolic Verification and Test Generation for GPUs. Technical report, School of Computing, University of Utah, Salt Lake City, Utah, March 12.
http://www.cs.utah.edu/formal_verification/GKLEE/gklee_tr.pdf

[LLS$^+$12] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *PPOPP*. Pp. 215–224. ACM, 2012.

[MB08] L. M. de Moura, N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*. LNCS 4963. 2008.

[NVI11] NVIDIA. NVIDIA CUDA C Programming Guide, Version 4.0. 2011.

[SNS+13] R. F. Salas-Moreno, R. A. Newcombe, H. Strasdat, P. H. J. Kelly, A. J. Davison. SLAM++: Simultaneous Localisation and Mapping at the Level of Objects. In *CVPR*. Pp. 1352–1359. IEEE, 2013.

[TSL10] S. Tripakis, C. Stergiou, R. Lublinerman. Checking Non-Interference in SPMD Programs. In *HotPar*. 2010.