CARP

D6.2: Static Verification of OpenCL[1]

Grant Agreement:	287767
Project Acronym:	CARP
Project Name:	Correct and Efficient Accelerator Programming
Instrument:	Small or medium scale focused research project (STREP)
Thematic Priority:	Alternative Paths to Components and Systems
Start Date:	1 December 2011
Duration:	36 months
Document Type[1]:	D (Deliverable)
Document Distribution[2]:	PU (Public)
Document Code[3]:	CARP-ICL-RP-006
Version:	v1.4
Editor (Partner):	Alastair F. Donaldson (ICL)
Contributors:	ICL, UT
Workpackage(s):	WP6
Reviewer(s):	ARM
Due Date:	31 October 2013
Submission Date:	8 November 2013
Number of Pages:	64

[1]MD = management document; TR = technical report; D = deliverable; P = published paper; CD = communication/dissemination.
[2]PU = Public; PP = Restricted to other programme participants (including the Commission Services); RE = Restricted to a group specified by the consortium (including the Commission Services); CO = Confidential, only for members of the consortium (including the Commission Services).

[3]This code is constructed as described in the Project Handbook.

[1]Revised and edited version for public distribution.

D6.2: Static Verification of OpenCL[2]

Alastair F. Donaldson[1], Adam Betts[1], Nathan Chong[1], Peter Collingbourne[1], Jeroen Ketema[1], Dan Liew[1], Paul Thomson[1], Marieke Huisman[2], Stefan Blom[2], Saeed Darabi[2], Matej Mihelcic[2]

[1]ICL, [2]UT

APPROVALS

Role	Name	Partner	Date
Workpackage Leader	A.F. Donaldson	ICL	8 November 2013
Project Manager	A.F. Donaldson	ICL	8 November 2013

[2]Revised and edited version for public distribution.

Contents

1 Executive Summary

This deliverable presents novel research and development results on techniques for static verification of software written using the OpenCL programming model. We report on related but complementary approaches based on: transformation to a sequential program verification task, and separation logic with permissions. We present these methods in detail and provide a comparison with techniques for kernel analysis proposed by other researchers.

2 Introduction

In recent years, massively parallel accelerator processors, primarily graphics processing units (GPUs) from companies such as AMD and NVIDIA, and based on designs from ARM, have become widely available to end-users. Accelerators offer significant 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 near future. Applications must exhibit portable correctness, operating correctly on any GPU accelerator. Software bugs in media processing domains can have serious financial implications. Moreover, GPUs are being used increasingly in domains such as medical image processing [14] where safety is critical. Thus there is an urgent need for verification techniques to aid construction of correct GPU software.

In this work we address the problem of static verification of GPU kernels written in kernel programming languages such as OpenCL [38], CUDA [53] and C++ AMP [50]. We place a specific emphasis on the OpenCL programming model because it is an industry standard implemented by all the main GPU providers, and is the target of novel compilation techniques being developed elsewhere in the CARP project (WP4).

In Chapter 3 we provide some background on GPU kernel programming, and detail two classes of bugs which make writing correct GPU kernels harder than writing correct sequential code: *data races* and *barrier divergence*.

We then describe two related but complementary methods for GPU kernel verification which we have investigated during the project so far, based on:

- Transformation to a sequential program verification task (Chapter 4)
- Separation logic with permissions (Chapter 5)

We then discuss related work on GPU kernel verification, presenting an experimental comparison with a comparable technique (Chapter 6).

We conclude with a discussion of open issues we hope to make progress on later in the project (Chapter 7).

Publications arising from work related to this deliverable

The techniques of Chapter 4 have led to publications at OOPSLA'12 [8], ESOP'13 [18] and OOPSLA'13 [15]. Instead of duplicating this material verbatim, we have attempted to present the approach in a less theoretically rigorous but more easily digestible manner.

The methods of Chapter 5 have led to a publication at BYTECODE'13 [36] and to the submission of an extended journal article.

3 Background on GPU kernels

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 widely used in parallel programming to accelerate tasks including (among many others, and citing only a small subset of works):

- Medical imaging [14]
- Computer vision [57]
- Computational fluid dynamics [32]
- DNA sequence alignment [46, 39]

Figure 3.1 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 document). 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.

A typical GPU-accelerated system is illustrated in Figure 3.2 (showing the case where host memory is separate from device global memory). The host application is responsible for copying data and code to the GPU for processing. Processing is achieved by invoking a *kernel* function which is executed by multiple threads running on parallel PEs on the GPU. If the application is massively parallel so that there are more threads than PEs then some threads will be blocked until other threads have run to completion. Some architectures also support interleaved execution of several threads on a single PE. When kernel invocation completes, the host application is responsible for copying data back into host memory for further processing. A GPU-accelerated application might launch a single kernel, or alternatively might launch a sequence of kernels to accelerate various phases of an application. In an iterative computation, it is common for a kernel invocation to be nested inside a host code loop. For example, a simulation application might iterate through a series of time steps, launching the same kernel again and again on each time step.

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

We can class races in GPU kernels 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

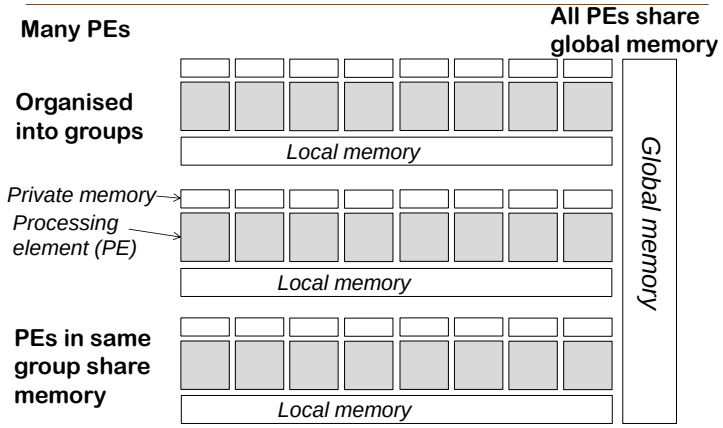


Figure 3.1: Overview of the structure of a typical GPU architecture.

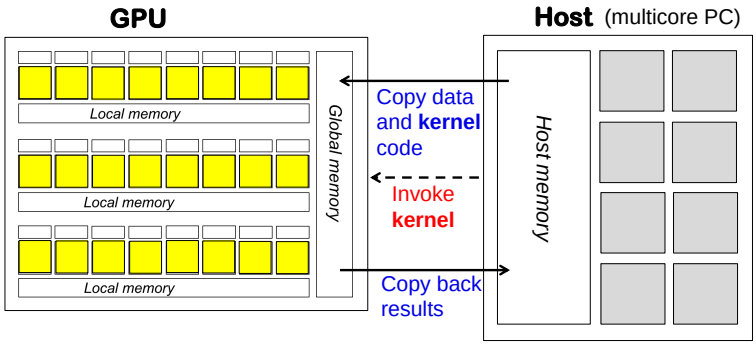


Figure 3.2: Illustration of a GPU-accelerated system.

respect to either global memory or the group's local memory. Figure 3.3 illustrates an inter-group data race, and a local memory intra-group data race.

Data races in GPU kernels are usually indicative of programmer errors, and lead to non-deterministic bugs that can be hard to reproduce and fix. A problem 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 one GPU architecture, thus evading 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 with data races in system-level CPU applications, which are often deliberate or regarded as benign [37], 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 Figure 3.4 shows a simple GPU kernel written in OpenCL C (a superset of a subset of the C99 language) [38]. The `__kernel` keyword indicates that the `add_neighbour` function is a kernel entry point. The kernel takes two arguments: `A`, which is a

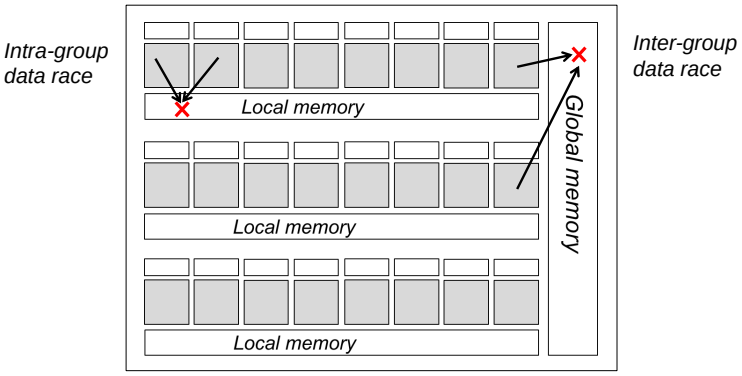


Figure 3.3: Illustration of intra- and inter-group data races.

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

Figure 3.4: A simple GPU kernel that exhibits a data race.

pointer to an array of integers resided 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.[1] A thread executing this kernel reads from `A` at offsets `tid` and `tid + offset`, sums the results together and writes them to `A` at offset `tid`. This example illustrates a read-write data race because, 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 synchronisation operation to separate these accesses.

Figure 3.5 shows how `A` evolves for two different thread schedules, assuming that `offset` is equal to 1, there are four threads, and the contents of `A` is initially $\{1,1,1,1,1\}$. The figure shows that these thread schedules lead to completely different results.[2]

Barrier synchronisation 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 syn-

[1] 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, and 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 deliverable we do not consider multi-dimensional kernels and simply use *tid* to refer to a thread's id, distinguishing between local and global id only when the distinction is important.

[2] Note that this example assumes a sequentially consistent memory model. In practice a GPU architecture is likely to exhibit a weakly consistent memory model, leading to an even higher degree of potential nondeterminism.

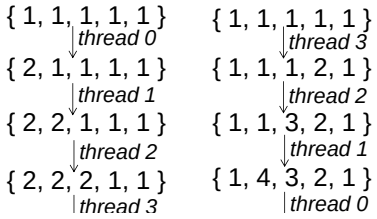


Figure 3.5: Examples of the very different results a kernel can compute for different thread schedules due to the nondeterminism introduced by a data race. Two possible executions of the kernel of Figure 3.4 are depicted, assuming that `offset` equals 1, four threads are used, and the contents of `A` is initially $\{1,1,1,1,1\}$.

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

Figure 3.6: Using a barrier to avoid the data race exhibited by the example of Figure 3.4.

chronisation within a kernel invocation is not possible.[3]

Barrier divergence The OpenCL 1.2 specification [38] lays down some rules on correct usage of barriers (in OpenCL, threads are referred to as *work items*):

> All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the barrier. This function must be encountered by all work-items in a work-group executing the kernel.
>
> If barrier is inside a conditional statement, then all work-items must enter the conditional if any work-item enters the conditional statement and executes the barrier.
>
> If barrier is inside a loop, all work-items must execute the barrier for each iteration of the loop before any are allowed to continue execution beyond the barrier.

The rules on barriers in conditionals make it clear, for example, that the following kernel snippet is not correct when executed by a work group consisting of multiple threads:

```
if(tid == 0) {
  barrier();
} else {
  barrier();
}
```

[3] Atomic operations can be used to achieve inter-group synchronisation on some architectures, but such a method relies on assumptions about the extent to which threads executing a kernel are scheduled concurrently, and thus is not portable.

because although all threads reach *some* barrier, thread 0 reaches a different barrier to other threads in the work group.

A more subtle example involving loops is as follows:

```
unsigned x = (tid == 0 ? 4 : 1);
unsigned y = (tid == 0 ? 1 : 4);

for(int i = 0; i < x; i++) {
  for(int j = 0; j < y; j++) {
L: barrier();
  }
}
```

In this example, thread 0 sets `x` and `y` to 4 and 1, respectively, while all other threads set `x` and `y` to 1 and 4, respectively. As a result, every thread should execute the barrier at label `L` 4 times. However, the example is incorrect according to the above excerpt from the OpenCL spec because thread 0 executes different numbers of outer and inner loop iterations compared with the other threads. In [8] we present experimental results showing that a kernel based on the above erroneous snippet computes different results on different architectures, and in [8, 18] we provide formal semantics for GPU kernels making precise the notion of barrier divergence.

4 Analysis of GPU Kernels through Sequential Program Verification

This chapter is joint work with Shaz Qadeer at Microsoft Research.

We describe our first approach to verification of OpenCL kernels, which involves exploiting 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, our approach has four main ingredients:

1. We focus race analysis on *barrier intervals*
2. We restrict analysis to consider a single, canonical thread schedule, avoiding the need to reason about a large number of schedules
3. We further restrict analysis 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. We apply predicated execution to handle loops and conditionals, and to precisely capture the conditions for barrier divergence

We discuss these ingredients in Section 4.1; ingredients 1-3 have been employed in other work on GPU kernel analysis [43, 17, 45], while ingredient 4 is a novel contribution of our work. 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. This constitutes a novel approach to GPU kernel verification.

To check for data races in practice using satisfiability modulo theories solvers, we encode access checks in a manner that avoids the use of quantifiers by exploiting nondeterminism. This is explained in Section 4.2.

After transformation and instrumentation, a GPU kernel can be verified for race- and divergence-freedom using techniques from sequential program verification. Our approach uses verification condition generation and automatic theorem proving, building on the Boogie framework [3]. To work for programs with loops and procedures, this relies on methods for invariant generation; we discuss our approach to invariant generation in Section 4.3.

Our basic verification approach considers just a pair of threads and uses a coarse abstraction to model further threads. This coarse abstraction does not always suffice for verification, and we have developed a novel shared state abstraction, *barrier invariants*, which we describe in Section 4.4. Barrier invariants allow the analysis of data-dependent GPU kernels.

We have implemented these ideas in a tool, GPUVerify.[1] We describe the architecture of this tool and discuss challenging practical issues related to dealing with unstructured control

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

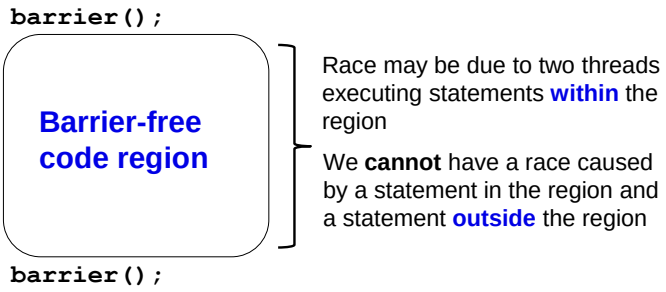


Figure 4.1: Illustration of a barrier interval. It is sufficient to restrict intra-group race analysis to pairs of instructions that lie within a common barrier interval.

flow graphs and accurately reporting errors in Section 4.5. We present results of an experimental evaluation using a large set of kernels in Section 4.6.

The chapter concludes with a discussion of the current limitations of and assumptions made by our verification approach (Section 4.7).

In what follows, we shall restrict our attention to the detection of *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. Nevertheless our implementation in GPUVerify (Section 4.5) implements inter-group race checking in full.

4.1 Transformation method

4.1.1 Focusing race detection to barrier intervals

Our first observation is that at any point of execution, two threads in the same group must be executing instructions that lie in a *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 4.1. In practice, a barrier interval may be more complex; for example, a barrier occurring inside a loop may form a barrier interval with itself. We can handle arbitrary barrier intervals by:

- Logging and checking all accesses to the shared state
- Resetting the logs each time a barrier is reached

4.1.2 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 in chunks. 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.

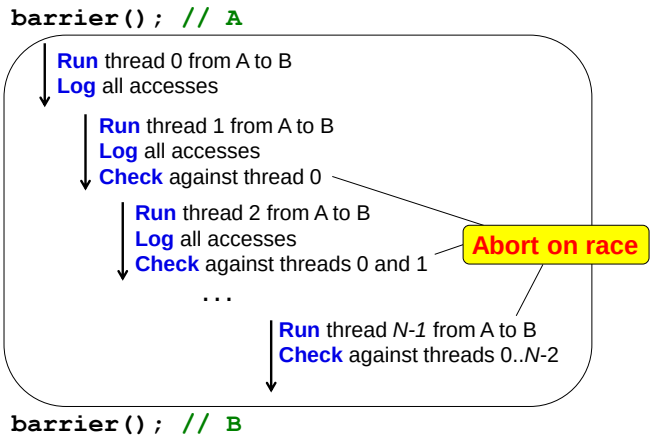


Figure 4.2: Restricting race analysis to consider a canonical thread schedule. The schedule shown is arbitrary: any other schedule will do, and in fact our approach uses a round-robin schedule in practice.

Our next observation is that if we check for data races occurring during the barrier interval for some fixed schedule then we will either detect a data race (in which case verification can abort with an error), or we show that this schedule exhibits no data races, in which case we can conclude that *no* schedule between these barriers can exhibit a data race. This technique has been used in several works [43, 17, 45, 8] and a proof that the reasoning is sound is presented in [44]. 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:

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

Thus considering an arbitrary canonical schedule 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.

4.1.3 The two thread abstraction

Restriction to a canonical schedule brings our analysis task into the realm of sequential program verification. 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.

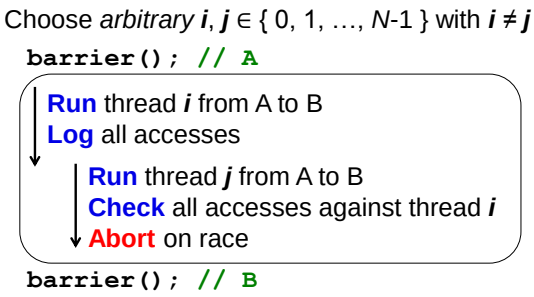


Figure 4.3: Reducing race analysis to consider an arbitrary *pair* of threads executing according to a canonical schedule.

We can avoid this sensitivity to the number of threads by exploiting the fact that data races and barrier divergence are both *pairwise* defects: 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. This is illustrated for the case of data races in Figure 4.3.

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 another 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 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. We call this the *adversarial* abstraction, and shall assume that this abstraction is being employed in the remainder of this presentation. A slightly richer abstraction, the *equality* abstraction, is discussed and compared with the adversarial abstraction in [8]. In Section 4.4 and [15] we also discuss *barrier invariants*, a tunable shared state abstraction technique for establishing richer properties.

4.1.4 Transformation for straight-line kernels

We now explain our program transformation in the case of single procedure straight-line kernels: kernels that do not use conditional or loop statements or call other procedures. We show how these additional features can be handled using *predicated execution* in Section 4.1.5.

Let us assume that a kernel has the following form:

```
__kernel void foo(<parameters, including __local arrays> ) {
```

<private variable declarations>

S_1;
S_2;
...
S_k;

}

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.

We also assume that all local array parameters to the kernel refer to *disjoint* arrays. We discuss this restriction further in Section 4.7.

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

- Models 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 In P we introduce two symbolic constants, 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:

```
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 we shall use "thread 1" or "the first thread" to refer to the thread whose id is recorded by tid$1, and "thread 2" or "the second thread" for the thread whose id is recorded by tid$2. It is important to note that here we are *not* referring 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. Our GPUVerify tool supports more refined abstractions: the equality abstraction [8] and barrier invariants [15] (also see Section 4.4); when these richer abstractions are used, the shared state is not discarded.

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 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, we assume that 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.

We also assume that P is 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

We do not yet specify how the sets R_A and W_A and the associated `LOG` and `CHECK` procedures are implemented. We discuss an efficient implementation from a verification perspective in Section 4.2.

We also assume that 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.

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 $a\$2 + \mathsf{tid}\$2 - x\$2$.

Table 4.1 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 (in Section 4.2 we shall see a method for implementing the effect of barriers).

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

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 we are abstracting the shared state, we have no knowledge of the values these variables should take after the read has occurred. Thus we use `havoc` 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 4.4.

4.1.5 Predicated execution, procedures and pointers

We now explain how loops and conditionals can be handled via the use of *predicated execution*. We then explain how our approach naturally extends to handle multiple procedures, and briefly comment on how pointers can be supported.

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:

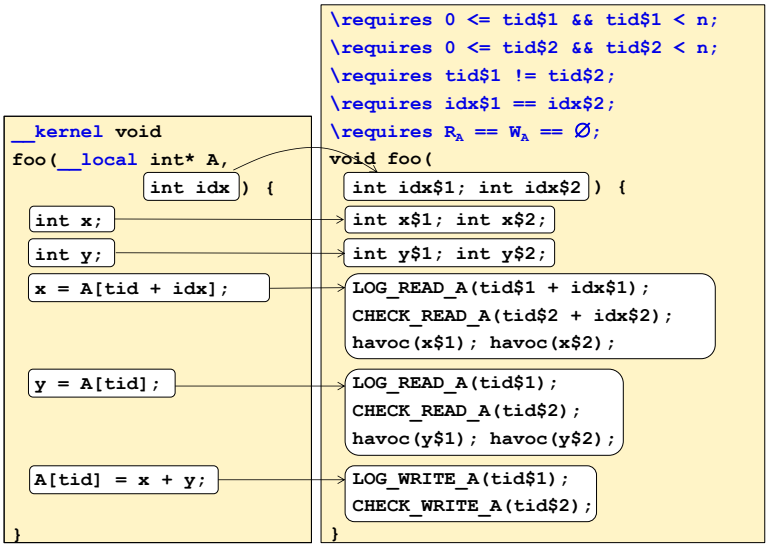


Figure 4.4: Illustration of the transformation process for a simple kernel.

```
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*).

We use predication in our 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.

We first adapt the `LOG` and `CHECK` procedures introduced in Section 4.1.4 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 on condition that the predicate holds
- `LOG_WRITE_A`: takes a Boolean predicate and an integer parameter; adds the parameter's value to W_A on condition that the predicate holds
- `CHECK_READ_A`: takes a Boolean predicate and an integer parameter; aborts 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 the predicate holds and the parameter's value belongs to either R_A or W_A

We also adapt the `barrier` procedure of P so that it additionally takes two predicate

parameters, one for each thread, and sets R_A and W_A to the empty set on condition that both predicate parameters are *true*. We discuss this form of `barrier` further in Section 4.2.

The rules of Table 4.2 show how a statement Stmt of K is translated to a statement $\mathsf{translate}(\mathsf{Stmt}, E)$ in P with respect to a predicate E. 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 4.1. 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.

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 *barrier divergence* is detected; we discuss this further in Section 4.2.

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 each 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 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.

Predicated execution for loops that exhibit additional control flow through `break` and `continue` statements can be achieved by extending the encoding presented here with additional instrumentation variables [8].

Handling multiple procedures Predicated program transformation elegantly extends to programs that use *non-recursive* procedures. The parameters of a procedure are duplicated, and an additional predicate parameter is added to every procedure for each thread, determining whether the thread is enabled during an invocation of the procedure. The procedure body is then transformed with respect to the predicate parameter. Procedure calls are translated by supplying the current predicate of execution for the predicate parameters, and by passing for every other parameter two values, one corresponding to the parameter value for the first thread, the other to the parameter value for the second thread.

This translation of procedures is shown in Table 4.3. For simplicity we do not show how

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($E\$1, E\$2$);	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($\mathsf{Stmt}_1, F$); translate($\mathsf{Stmt}_s, G$);	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($\mathsf{Stmt}_1, F$); $F\$1 = F\$1$ && $e\$1$; $F\$2 = F\$2$ && $e\$2$; }	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 4.2: 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*.

Procedure in K,	**Corresponding procedure in P**
`void` name($T_1\ a_1, \ldots, T_k\ a_k$) { Stmt }	`void` name(`bool` $E\$1$, $T_1\ a_1\$1, \ldots, T_k\ a_k\$1$, `bool` $E\$2$, $T_1\ a_1\$2, \ldots, T_k\ a_k\$2$) { translate(Stmt, E) }
Call statement in K, Stmt	**Corresponding call in P,** translate(Stmt, E)
name($e_1, \ldots, e_k$);	name($E\$1, e_1\$1, \ldots, e_k\$1, E\$2, e_1\$2, \ldots, e_k\$2$);

Table 4.3: Handling of procedures and procedure calls during predicated program transformation, extending the rules of Table 4.2.

to translate procedures that return values, but this can be handled in a straightforward manner through the use of output parameters.

The translation does not work for recursive procedures, as it leads to an infinite series of recursive calls. In practice, GPU kernels do not use recursion much as it was until recently forbidden in the main kernel programming languages [53, 38].

Support for pointers The aim of our transformation to a sequential program is to ease the task of verification. In practice we do this by compiling a kernel into the Boogie intermediate language [3], discussed further in Section 4.5. Boogie is a deliberately simple intermediate language, and does not support pointer data types natively. We have devised an encoding of pointers suitable for Boogie which we explain using an example.

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

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

We can model these pointers by generating 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 4.5 shows how uses of `p` and `q` are pre-processed so that explicit use of pointers is eliminated. This pre-processing takes place *before* the transformation to a sequential program is applied; thus translation into a two-threaded, predicated form takes place after pointers have been eliminated.

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

Source	After pre-processing
`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 4.5: Pre-processing to eliminate explicit pointer usage.

`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 private 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.

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 [58]. 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 Race instrumentation

We now describe an efficient method for implementing the `LOG` and `CHECK` procedures that we introduced abstractly in Sections 4.1.4 and 4.1.5.

Recall that the purpose of `LOG` is to insert a given element into a set, while the purpose of `CHECK` is to abort if a given element is contained in a set (both operations being conditional on a predicate). It is straightforward in a language such as Boogie [3] to model these operations by using sets directly, but this requires the use of *quantifiers* which are notoriously difficult to

reason about automatically [22, 20, 51]. Instead, we present a method that avoids the use of quantifiers. Our method can be viewed as a kind of quantifier elimination.

Logging reads and writes For an array A we introduce a Boolean and an integer to track reads from A:

- `READ_HAS_OCCURRED_A : bool`; *true* if and only if a read from A is being tracked
- `READ_OFFSET_A : int`; if `READ_HAS_OCCURRED_A` is *true* then `READ_OFFSET_A` indicates the offset associated with the read

We use a precondition on P to ensure that `READ_HAS_OCCURRED_A` is initially *false*.

Collectively, the pair (`READ_HAS_OCCURRED_A`, `READ_OFFSET_A`) will enable us to record *up to* one element of the set R_A by implementing `LOG_READ_A` as follows:

```
void LOG_READ_A(bool enabled, int offset) {
  if(enabled) {
    if(*) {
      READ_HAS_OCCURRED_A = true;
      READ_OFFSET_A = offset;
    }
  }
}
```

If the predicate `enabled` is *false*, the read is not logged. Otherwise, the nondeterministic expression `*` is used to choose between either logging the read, in which case `READ_HAS_OCCURRED_A` and `READ_OFFSET_A` are updated, or ignore the read. In the latter case, the instrumentation variables `READ_HAS_OCCURRED_A` and `READ_OFFSET_A` are unchanged. This means that if a read was already being tracked then it is *still* tracked.

With this implementation of `LOG_READ_A`, if `READ_HAS_OCCURRED_A` is *true* we know that the value of `READ_OFFSET_A` belongs to R_A. It may be the case that R_A contains other values; the instrumentation variables do not tell us this. If `READ_HAS_OCCURRED_A` is *false*, we know *nothing* about R_A. In particular, we *cannot* conclude that R_A is empty. This is because every call to `LOG_READ_A` has the option of leaving the instrumentation variables unchanged.

Similarly, we introduce two instrumentation variables to track writes on A:

- `WRITE_HAS_OCCURRED_A : bool`; *true* if and only if a write to A is being tracked
- `WRITE_OFFSET_A : int`; if `WRITE_HAS_OCCURRED_A` is *true* then `WRITE_OFFSET_A` indicates the offset associated with the write

and implement `LOG_WRITE_A` analogously to `LOG_READ_A`:

```
void LOG_WRITE_A(bool enabled, int offset) {
  if(enabled) {
    if(*) {
      WRITE_HAS_OCCURRED_A = true;
      WRITE_OFFSET_A = offset;
    }
  }
}
```

Checking reads and writes For an array A, the transformation described in Sections 4.1.4 and 4.1.5 uses procedure `CHECK_READ_A` to check that an offset being read from is not an element of W_A, and `CHECK_WRITE_A` to check that an offset being written to is not an element of $R_A \cup W_A$. With predicated execution, these checks are conditional on the current predicate being true.

With the above implementation of R_A and W_A, these checks are straightforward to implement:

```
void CHECK_READ_A(bool enabled, int offset) {
  // Check for write−read race
  assert(!(enabled && WRITE_HAS_OCCURRED_A && offset == WRITE_OFFSET_A))
}

void CHECK_WRITE_A(bool enabled, int offset) {
  // Check for read−write race
  assert(!(enabled && READ_HAS_OCCURRED_A && offset == READ_OFFSET_A));
  // Check for write−write race
  assert(!(enabled && WRITE_HAS_OCCURRED_A && offset == WRITE_OFFSET_A));
}
```

The `CHECK_READ_A` implementation aborts if the given predicate, `enabled`, is *true*, if a write is being logged (`WRITE_HAS_OCCURRED_A` holds), and if the offset of the logged write matches the offset being read from. The `CHECK_WRITE_A` implementation is similar, but checks for both read-write and write-write races.

To see that our implementations of the `LOG` and `CHECK` procedures are sufficient for reasoning about data races, suppose that a race *can* occur between two statements in a kernel K, a read and a write on array A say, when executed by threads i and j respectively. Suppose that the read statement appears before the write statement in K. Then in the generated sequential program P, `LOG_READ_A` is called with some predicate and offset associated with thread 1. This is followed by a sequence of statements (possibly involving further calls to `LOG_READ_A`), followed by a call to `CHECK_WRITE_A` with some predicate and offset associated with thread 2. To verify race-freedom, it is necessary to establish that for all choices of thread 1 and thread 2, all possible inputs and all possible resolutions of nondeterminism, no assertion can fail. Thus it is necessary to consider thread i as thread 1, thread j as thread 2. In this case, the predicates passed to `LOG_READ_A` and `CHECK_WRITE_A` will both be *true* (because the read and the write are issued). One resolution of nondeterminism involves choosing to track the read in the call to `LOG_READ_A` and deciding *not* to track any other reads in subsequent calls to `LOG_READ_A`. In this case, when `CHECK_WRITE_A` is called, `READ_HAS_OCCURRED_A` and *enabled* will be *true*, and the offset being written to by thread j will match the offset that was read from by thread i, stored in `READ_OFFSET_A`. Hence a race is guaranteed to be detected. By a similar argument, if it can be shown that no assertions in the program P can fail, it follows that the original kernel K is race-free.

This method of encoding race checks was inspired by the *implicit encoding* used for analysis of direct memory access (DMA) races in programs running on accelerator cores in the Cell Broadband Engine architecture [23, 24, 25].

Barriers Recall from Section 4.1.5 and Table 4.2 that a barrier statement `barrier()` is translated into a call `barrier(`E`$1,`E`$2)` in P, where E is the predicate guarding execution. If the predicates do not match then *barrier divergence* has occurred and execution of P should

abort. Otherwise, if the predicates are both *true*, all of the read and write sets used in race analysis should be cleared.

A straightforward implementation of `barrier` in *P* is thus as follows:

```
void barrier(bool E$1, bool E$2) {
  assert(E$1 == E$2); // Check for barrier divergence
  if(E$1 && E$2) {
    // Barrier reached in enabled state, so clear all read/write sets
    // Clear read/write sets for array A
    READ_HAS_OCCURRED_A = false;
    WRITE_HAS_OCCURRED_A = false;
    // Clear read/write sets for array B
    READ_HAS_OCCURRED_B = false;
    WRITE_HAS_OCCURRED_B = false;
    // etc.
    ...
  }
}
```

This has the disadvantage that `barrier` modifies *all* of the Boolean variables used in race analysis. Thus if a barrier occurs in a loop, when reasoning about the loop we typically require an invariant that records the state of accesses for *every* array, even for arrays not touched by the loop.

A smarter encoding avoids this problem by exploiting the fact that there is always *one* execution in which *no* reads or writes are tracked – the execution where nondeterminism in the `LOG` procedures always results in no logging taking place. Thus `barrier` can be implemented by using `assume` statements to restrict execution to just this trace:

```
void barrier(bool E$1, bool E$2) {
  assert(E$1 == E$2); // Check for barrier divergence
  if(E$1 && E$2) {
    // Barrier reached in enabled state, so clear all read/write sets
    // Clear read/write sets for array A
    assume(!READ_HAS_OCCURRED_A);
    assume(!WRITE_HAS_OCCURRED_A);
    // Clear read/write sets for array B
    assume(!READ_HAS_OCCURRED_B);
    assume(!WRITE_HAS_OCCURRED_B);
    // etc.
    ...
  }
}
```

Here, `assume` is a special statement used in formal verification, and can be thought of as causing execution to gracefully terminate (without error) if the guard evaluates to *false*, or acting as a no-op otherwise. For a formal definition of the semantics of `assume`, see for example [5].

Though only subtly different, this encoding, which avoids side-effects, makes verification of data race-freedom significantly easier in practice.

Asymmetry of race checking Note that in the encoding presented above, all calls to `LOG` take parameters corresponding to the first thread, while calls to `CHECK` take parameters corresponding to the second thread. Thus the first thread does all the logging and the second thread all the checking. It may appear that this asymmetry in the encoding may lead to missed data races: for threads i and j, if i logs and j checks then we will *not* detect a race that involves first a write by j and then a write by i. However, the encoding is sound because we will consider *all* ordered pairs of threads, thus we will also consider the case where j logs and i checks, in which case we will detect this race.

4.3 Invariant generation

Our method of Section 4.1 transforms a massively parallel GPU kernel into a sequential program together with race instrumentation as described in Section 4.2. 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 algorithm [27] as the basis for inferring invariants and contracts. Houdini is a method to find the largest set of inductive invariants from a user-supplied pool of candidate invariants. Houdini works as a fix-point procedure; starting with the entire set of invariants, it tries to prove that the current candidate set is inductive. The invariants that cannot be proved are dropped from the candidate set and the procedure is repeated until a fix-point is reached.

Through manually deducing invariants for a set of kernels we have devised a number of candidate generation rules which we outline below. We emphasise that the candidate invariants generated by our method 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 the implementation of our tools cannot compromise the soundness of verification.

Our candidate generation rules are purely heuristic. The only fair way to evaluate these carefully crafted heuristics is with respect to a large set of unknown benchmarks. We present such an evaluation in Section 4.6.

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 implementation in our GPUVerify tool (Section 4.5) 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 variable). For each of the rules associated with shared memory writes, there is an analogous rule for reads.

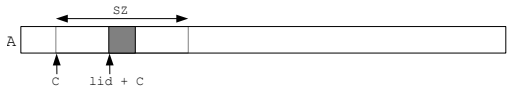
Rule: access at thread id plus offset

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

Generated candidate:

```
WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_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

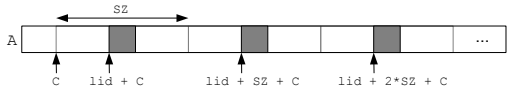
Generated candidate:

```
WRITE_HAS_OCCURRED_A ==> ((WRITE_OFFSET_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
WRITE_HAS_OCCURRED_A ==> ((WRITE_OFFSET_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

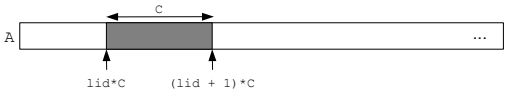
Conditions: `A[lid*C + i] = ...` occurs in loop
`i` is live at loop head

Generated candidates:

```
WRITE_HAS_OCCURRED_A ==> lid*C <= WRITE_OFFSET_A
```

`WRITE_HAS_OCCURRED_A ==> WRITE_OFFSET_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

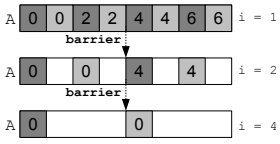
Conditions: `i = i*2` or `i = i/2` occurs in loop
`i` is live at loop head
`D` is the smallest power of 2 with `D` $\geq$ `SZ`

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.

4.4 Barrier invariants for reasoning about data-dependent kernels

The soundness and precision of the two-thread reduction hinges on the method used to over-approximate the behaviour of additional threads. The simplest approach, the *adversarial abstraction* discussed in Section 4.1.3, is to make *no* assumptions about the behaviour of additional threads, assuming that these threads may update the shared state arbitrarily. The adversarial abstraction is effective for verification of *data-independent* GPU kernels: kernels for which control flow and memory access patterns do not depend on the data stored in shared memory. While a large number of GPU kernels fall into this category, there are important and

interesting kernels that exhibit data-dependence. A key family of such kernels use *prefix sum* operations [10, 33] to perform compaction of data [9]. Data-dependent kernels are difficult to verify because they require reasoning about shared state manipulated by many parallel threads. The access pattern of a single thread may depend on the data or control flow of many other threads, making race checking challenging.

To see why data-dependence hinders verification using adversarial abstraction, consider the following simple kernel, where `A` and `B` are arrays in shared memory, `tid` denotes the id of a thread, and `f` is a side-effect free procedure which may read from the shared state and ensures that for distinct threads s and t that $\texttt{f}(s) \neq \texttt{f}(t)$:

```
A[tid] = f(tid); barrier(); B[A[tid]] = tid;
```

The kernel is data-dependent because array `B` is written to at an index which is computed by reading from the shared state. The kernel is clearly race-free. However, consider execution of the kernel fragment with respect to an arbitrary, distinct pair of threads, s and t, using the adversarial abstraction. Because $s \neq t$, execution of `A[tid]=f(tid)` by both threads will not result in a data race on `A`, and will lead to a state in which $\texttt{A}[s] \neq \texttt{A}[t]$. However, at the barrier, the adversarial abstraction dictates that `A` and `B` should be nondeterministically assigned. A possible nondeterministic assignment yields $\texttt{A}[s] = \texttt{A}[t] = 0$, which causes the statement `B[A[tid]]=tid` to result in a data race on `B` at index 0. This simple kernel is not amenable to verification using the two-thread reduction with adversarial abstraction: the adversarial abstraction is too coarse.

We have designed *barrier invariants*, a novel abstraction technique to allow verification of data-dependent GPU kernels using the two-thread reduction. The idea is that each barrier in a kernel can be annotated with a barrier invariant, stating a property of the shared state that must hold each time the barrier is reached. Barrier invariants retain the scalability of the two-thread reduction, but allow the precision lost by the adversarial abstraction to be recovered: when considering the execution of a barrier by an arbitrary pair of threads, instead of setting the shared state to an arbitrary value, the shared state is set to an arbitrary value *satisfying the barrier invariant*.

Barrier invariants can be specified manually by the user, and in principle they could be automatically inferred using techniques similar to the loop invariant inference methods described in Section 4.3. At present, the GPUVerify tool (see Section 4.5) does *not* support automatic generation of barrier invariants.

A barrier invariant can be added to the above example to capture the fact that the elements of `A` are distinct as follows (where x and y range over thread ids):

```
A[tid] = f(tid);
barrier() invariant(∀x ≠ y : A[x]≠A[y]);
B[A[tid]] = tid;
```

When considering the execution of an arbitrary pair of threads s and t, the invariant is established on entry to the barrier by checking that $\texttt{A}[s] \neq \texttt{A}[t]$: because s and t are arbitrary, this proves that the invariant holds for *all* pairs of threads. After the barrier, it is legitimate to assume that $\texttt{A}[x] \neq \texttt{A}[y]$ holds for all pairs of distinct threads x and y and, in particular, for the threads s and t under consideration, and thus the write to `B` at index `A[tid]` is verified to be race-free as required. In fact, after the barrier, it is sufficient to assume the invariant *only* for the pair (s,t). More complex kernels usually require the invariant to be assumed for multiple pairs. However, a small subset of all pairs usually suffices. This is important for kernels with

thousands of threads where assuming a barrier invariant for all pairs is not feasible without the use of quantifiers.

In the same way that loop invariants and procedure specifications are fundamental to modular reasoning about sequential programs, we believe that barrier invariants are fundamental to thread-modular reasoning about data-dependent GPU kernels which, until now, has been out-of-scope. Because GPU kernels are executed by large numbers of threads, barrier invariants are still necessary to reason about data-dependent properties even for loop- and call-free kernels.

In [15] we present full technical details of barrier invariants in terms of a concrete semantics for GPU kernels, and an abstract semantics that considers a pair of threads (s,t). For a kernel P and a pair of distinct threads (s,t), let us use $\mathscr{A}^{s,t}(P)$ to denote P interpreted with respect to this abstract semantics. The following theorem is proven in [15]:

Theorem 4.4.1 (Soundness)**.** *Let P be a kernel executed by n threads. If for every pair $0 \leq s \neq t < n$, no execution of $\mathscr{A}^{s,t}(P)$ from a valid initial state leads to an error state, then no execution of P from a valid initial state leads to an error state (where an error state indicates that a data race or barrier divergence has occurred).*

Hence, it suffices to prove data race-freedom, and validity of barrier invariants, with respect to the abstract semantics.

We describe the abstract semantics in an intuitive manner, referring the reader to [15] for formal details and a proof of Theorem 4.4.1.

The semantics models a pair of distinct threads (s,t) executing the kernel. When executing a sequence of statements between two barriers, it is as if s and t are the only threads executing the kernel. They perform private updates and access the shared state, recording all shared locations that are accessed in their read/write sets. The read/write sets are used to detect data races between s and t: if a data race occurs, execution aborts.

On reaching a barrier, a check is made to determine whether the invariant φ associated with the barrier holds for the pair (s,t). This check must take into account possible updates to the shared state made by additional threads executing the kernel. We handle this by considering whether a shared memory location v was accessed by s or t since the last barrier. There are two key cases:

- v **was accessed by at least one of** s,t**:** We say that v is a *known* location. In this case it is sound to assume that v has not been modified by any thread $r \notin \{s,t\}$. This is because our analysis considers *all possible pairs of threads*: if r can write to v then because some $x \in \{s,t\}$ accesses v a data race will be detected for the pair (x,r) and the kernel will not be deemed correct.

- v **was not accessed by** s **or** t**:** We say that v is an *unknown* location. In this case we must consider that v could have been modified by some thread $r \notin \{s,t\}$. A safe approximation is to assume that v contains an *arbitrary* value when evaluating the barrier invariant.

If there exists an assignment to unknown locations yielding a state in which φ does not hold for (s,t), execution aborts: there may be a concrete execution that would lead to this state. Thus with the two-thread reduction, user-provided barrier invariants are not taken on trust — they are **checked**.

If φ can be shown to hold for the pair (s,t) for all assignments to unknown locations then, because (s,t) is arbitrary, it is sound to assume φ holds for *all* pairs of distinct threads. Abstract

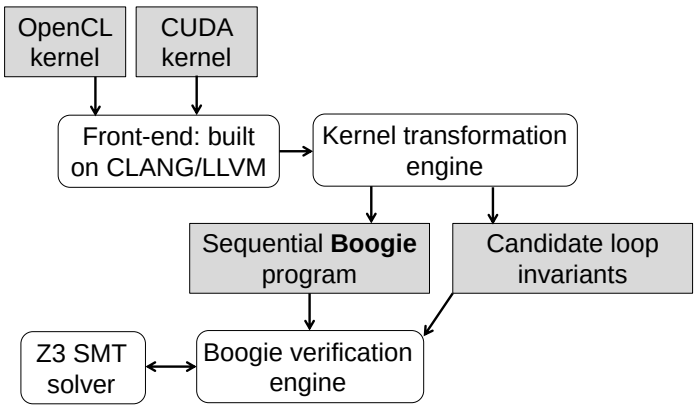


Figure 4.6: Architecture of the GPUVerify tool.

execution for the pair (s,t) continues after the barrier by transitioning to an abstract state in which:

- the private stores for (s,t) and the values of *known* locations are preserved;
- the read and write sets for (s,t) are cleared;
- φ is assumed to hold for all pairs of distinct threads.

We have applied barrier invariants to allow the verification of data-dependent kernels that use prefix sum operations [15].

4.5 Implementation in GPUVerify tool

We have implemented the techniques described in this chapter as a tool, GPUVerify. The architecture of GPUVerify is illustrated in Figure 4.6. As input, GPUVerify accepts a kernel written in CUDA or OpenCL. We have designed a front-end based on the Clang/LLVM framework which translates a kernel into a temporary Boogie representation which is fed to our kernel transformation engine. The transformation engine implements the transformation of Section 4.1 and race instrumentation of Section 4.2, generating a sequential Boogie program to be analysed. The tool also generates candidate loop invariants using the method described in Section 4.3. Verification is then delegated to the Boogie verification engine [3], an open source verifier developed primarily by Microsoft Research.[2] Boogie discharges verification conditions to an SMT solver which, by default, is Z3 [21]. We have also added support to Boogie for the CVC4 solver [6] to allow comparison between solvers.

The architecture of Figure 4.6 shows that GPUVerify exhibits significant *re-use*: we take advantage 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.

[2] `http://boogie.codeplex.com/`

We now comment on some notable aspects of the GPUVerify implementation required to make the tool work in practice on realistic examples. We also comment on our engineering efforts to ensure easy access to the tool and smooth deployment during the CARP project.

Unstructured control flow graphs Our initial design for GPUVerify's front-end worked at the level of Clang abstract syntax trees and was limited to structured kernels using `if`, `while` and `for` for control flow. Restricting to structured kernels allowed for a direct implementation of the transformation of Section 4.1 into structured Boogie. However, this meant that we could not handle kernels exhibiting more complex control flow, such as through the use of `switch` and short-circuit evaluation. Furthermore, we had to deal explicitly with many front-end issues related to pointer and array syntax and analysis of structures.

We then switched to a new front-end, Bugle,[3] which works at the level of LLVM intermediate representation (IR), which is equivalent to unstructured control flow graphs. To operate at this level we had to design a novel method for predicated execution of unstructured programs [18]. The new front-end allows us to apply GPUVerify to a larger set of real examples without being blocked by issues of syntax. These are handled by Clang which transforms intricate syntactic features into uniform IR. In particular, the new setup means that we can deal easily with CUDA kernels that use C++ templates, a feature of C++ for which front-end support is notoriously challenging.

Error reporting To make GPUVerify useful in practice, it is essential that errors reported by the tool are understandable to users. We have put significant effort into ensuring that errors detected at the Boogie representation level can be described with respect to the source locations in the original OpenCL or CUDA code. This involved leveraging support in Clang/LLVM which attaches debugging information to LLVM instructions. From this debugging information, file, directory, line and column information can be extracted and attached to the generated Boogie code. This makes it easy to report source information for single-statement errors such as assertion failures or barrier divergence. Accurately reporting data races is more challenging because *two* statements are involved. The second statement, which leads to the race being detected, is known directly so getting source information for this statement is easy. However, the first statement is not directly known – it results either from a `LOG` procedure call (see Section 4.2) recording an access, or from loop abstraction leading to race instrumentation variables being set to a conflicting state. We deal with this issue by extending the instrumentation of Section 4.2 so that a source location marker is also logged with each read and write. For an array `A` this involves adding variables `READ_SOURCE_A` and `WRITE_SOURCE_A`. When a read/write is logged, the associated `READ/WRITE_SOURCE` variable is set to a unique identifier corresponding to the syntactic statement associated with the access. This allows the necessary information to report the race to be recovered.

Annotation language Our aim is for GPUVerify to be a fully automatic tool. However, our invariant inference (see Section 4.3) is not fully mature, and cannot be complete. To aid expert users and to allow developers to specify preconditions necessary to eliminate false positives, we have provided front-end support for the specification of loop invariants and pre-/post-conditions. We have also provided constructs which allow the race instrumentation variables introduced in Section 4.2 to be directly referred to in specifications. This front-end support is achieved

[3] `git://git.pcc.me.uk/~peter/bugle.git`

by equipping Bugle with a *special function map* which maps the names of relevant functions to custom code generation routines. For example, a user can specify a kernel precondition ϕ by writing `__requires(`ϕ`)`. Clang compiles this to a regular function call when generating LLVM IR. However, when processing the IR, Bugle finds that `__requires` is in the special function map and invokes a routine designed to handle preconditions. This routine removes the `__requires` call from the body of the kernel and instead places a `requires` clause in the generated Boogie code.

Infrastructure to support development and uptake To encourage uptake of GPUVerify by industry, and to make it as easy as possible for other CARP partners to use the tool effectively, we have put significant engineering effort into infrastructure for the tool. This includes:

- A suite of (at time of writing) 296 OpenCL and 106 CUDA kernels for regression testing
- Nightly builds for Windows and Linux which automatically deploy GPUVerify to its web page for download[4]
- Documentation for users and developers designed using the Sphinx framework[5]
- Bug tracking, initially using Bugzilla and now through Codeplex[6]
- A web interface to GPUVerify through Microsoft Research's `rise4fun` web interface[7]
- YouTube videos describing the tool and underlying techniques[8]

4.6 Experimental evaluation

We now present an evaluation of GPUVerify which originally appeared in [8].

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 [52], 20 publicly available CUDA kernels
- **C++ AMP:** Microsoft C++ AMP Sample Projects [49], 20 publicly available kernels, translated to CUDA
- **Basemark:** Rightware Basemark CL v1.1 [56], 52 commercial OpenCL kernels, provided to us under academic license

[4] `http://multicore.doc.ic.ac.uk/tools/GPUVerify/download.php`
[5] `http://sphinx-doc.org/`
[6] `http://gpuverify.codeplex.com/`
[7] `http://rise4fun.com/GPUVerify-OpenCL`
[8] `http://www.youtube.com/watch?v=l8ysBPV8OvA`

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 [43], 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 [43]. Because GPUVerify cannot directly analyse C++ AMP code, we retrieved the set of C++ AMP samples available online [49] 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 which can only be handled using barrier invariants (Section 4.4) which we do not yet infer automatically (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.[9]

4.6.1 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

[9] 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 4.7: Summary of size, in terms of ELOC, number of procedures and number of loops, of benchmarks in the training and evaluation sets

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 4.7 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.

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 4.8 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.

Results for the evaluation set Figure 4.9 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

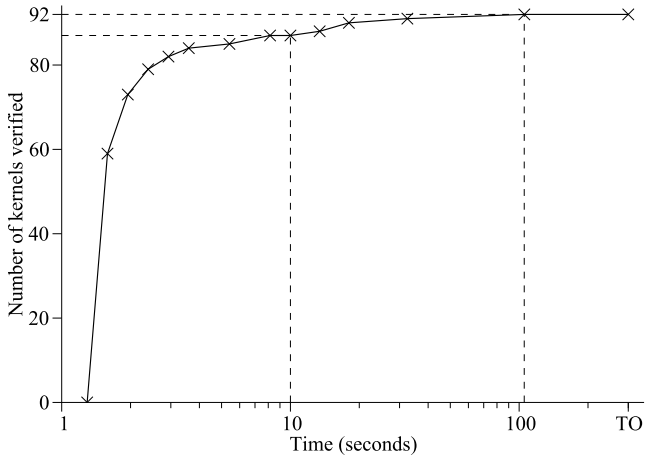


Figure 4.8: Cumulative histogram showing the time taken for successful verification with GPUVerify for the *training* set

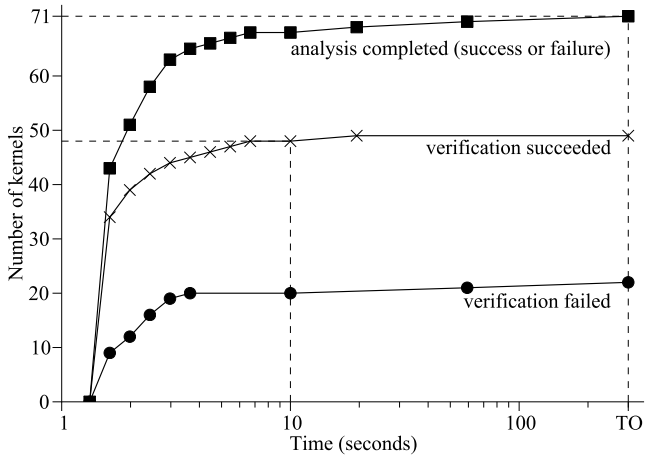


Figure 4.9: Cumulative histogram showing the times for successful and unsuccessful verification with GPUVerify for the *evaluation* set.

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.

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.

We have manually inspected the kernels for which GPUVerify reported verification failure, and provide a complete discussion in [8]. In [8] we also report on the detection of a bug in a previous CUDA SDK example.

4.7 Limitations and assumptions

The method described in this chapter is subject to various limitations and assumptions.

Limitations The following features are not supported by the current approach:

- Dynamic memory allocation: our memory model currently relies on there being a fixed number of shared arrays provided as arguments to the kernel.

- Recursion: our predication method for procedures does not work directly for recursive kernels, as briefly noted in Section 4.1.5. This limitation can be removed by adapting

predication so that a recursive call is *not* made if both threads under consideration are disabled.

- Function pointers: these are currently not allowed in the OpenCL programming model. However, if they are introduced in future versions of OpenCL then we could handle pointers to a fixed domain of target functions by transforming function pointer calls into case splits [19].
- Atomic operations: we are in the progress of adding rudimentary support for atomics, but atomic operations in general are a challenge for verification as they destroy the ability to restrict attention to an arbitrary thread schedule between barriers: valid nondeterminism can result from thread interleaving influencing the order in which atomic operations occur.
- Host code: currently GPUVerify checks only the code of a kernel function (and auxiliary functions called by the kernel). The tool and technique does not cater for verification of host code. In contrast, the methods for separation logic-based verification presented in Chapter 5 do not have this restriction.

Assumptions GPUVerify makes some unsound but pragmatic assumptions:

- GPUVerify assumes that all pointer parameters to a kernel refer to distinct, disjoint arrays. For example, if a kernel takes pointers `p` and `q` as parameters, GPUVerify does not consider the possibility of aliasing between `p` and `q`.
- GPUVerify assumes no out-of-bounds array accesses occur. Thus a data race that could occur by accessing an array `B` via an out-of-bounds access to array `A` will go undetected.

These assumptions are in place to lower the false positive rate of the tool. Without the first assumption, almost every kernel would appear to exhibit a data race when in practice input pointer parameters of real kernels *do* satisfy the disjointness property. Without the second assumption, the tool would have to prove memory safety as well as data race-freedom. This would add extra verification burden, requiring additional invariant generation techniques, and would increase the false positive rate of the tool. In future work we may consider revisiting both assumptions.

5 Separation Logic with Permissions

We now present a verification technique for general-purpose GPU (GPGPU) programs based on permission-based separation logic. Unlike the method described in Chapter 4, which focuses on establishing race- and divergence-freedom of the kernel part of a GPU-accelerated application, our method allows reasoning about both host and kernel code, and aims to allow verification of functional properties in addition to proving race-freedom.

The main inspiration for our verification approach is the use of permission-based separation logic to reason about multithreaded programs [11, 30, 29]. Key ingredient of the logic are read and write permissions. A location can only be accessed or updated if a thread holds the appropriate permission to access this location. Program annotations are *framed* by permissions: a functional property can only be specified and verified if a thread holds the appropriate permissions. Write permissions can be split into read permissions, while multiple read permissions can be combined into a write permission. Soundness of the logic guarantees that at most one thread at the time can hold a write permission, while multiple threads can simultaneously hold a read permission to a location. Thus, if a thread holds a permission on a location, the value of this location is *stable*, *i.e.*, it cannot be changed by another thread. Soundness of the logic also ensures that a program can only be verified if it is free of data races.

To adapt this idea to the GPGPU setting, for each kernel we specify all the permissions that are needed to execute the kernel. Upon invocation of the kernel, these permissions are transferred from the host code to the kernel. Within the kernel, the available permissions are distributed over the work groups, and within the work groups the permissions are distributed over the threads. Every time a barrier is reached, a barrier specification specifies how the permissions are redistributed over the threads available in the work group (similar to the barrier specifications of Hobor et al. [35]). The barrier specification also specifies functional pre- and postconditions for the barrier. Essentially this captures how knowledge about the state of global and local memory is spread over the different threads upon reaching the barrier.

The remainder of this chapter is organised as follows. Section 5.1 outlines our verification approach; Section 5.2 formally defines the kernel programming language, and its semantics; Section 5.3 presents the logic and its soundness proof. Section 5.4 discusses tool support for the logic, and Section 5.5 presents several verification examples. Finally, Section 6.1 discusses related work, while Section 5.6 presents conclusions and future work.

5.1 Reasoning about GPGPU Kernels

This section first briefly introduces permission-based separation logic, and then shows how we use it to reason about OpenCL kernels.

5.1.1 Permission-based Separation Logic

Separation logic [55] was originally developed as an extension of Hoare logic [34] to reason about programs with pointers, as it allows to reason explicitly about the heap. In classical Hoare logic, assertions are properties over the state and no distinction between variables on the heap on variables on the stack can be made, while in separation logic, the state is explicitly divided in the heap and a store related to the stack frame of the current method call. Separation logic is

also suited to reason modularly about concurrent programs [54]: two threads that operate on disjoint parts of the heap do not interfere, and thus can be verified in isolation.

However, classical separation logic requires use of mutual exclusion mechanisms for all shared locations, and it forbids simultaneous reads to shared locations. To overcome this, Bornat et al. [11] extended separation logic with fractional permissions. Permissions, originally introduced by Boyland [12], denote access rights to a shared location. A full permission 1 denotes a write permission, whereas any fraction in the interval $(0,1)$ denotes a read permission. Permissions can be split and combined, thus a write permission can be split into multiple read permissions, and sufficient read permissions can be joined into a write permission. In this way, data race freedom of programs using different synchronisation mechanisms can be proven. The set of permissions that a thread holds are often known as its *resources*.

Since kernel programs only have a single synchronisation mechanism, namely barriers, we can use a simplified permission system that only distinguishes between read-write and read-only permissions; rw and rd, respectively. Resource formulas in this simplified logic are first-order logic formulas, extended with the permission predicate, and the separating conjunction ($*$). The syntax of resource formulas R is defined as follows (where e is a first-order logic formula):

$$R ::= e \mid \texttt{Perm}(x,\pi) \mid R * R \mid e \Rightarrow R \mid \underset{\alpha:e}{*} R(\alpha) \qquad \pi \in \{\mathsf{rd},\mathsf{rw}\}$$

Note that the only operations allowed on the resource formulas of our logic are separating conjunction and implication from booleans to resources. This keeps the valid resources at any point in the program deterministic. This will make tool support much easier. An assertion $\texttt{Perm}(x,\pi)$ holds for a thread t if it has permission π to access the location pointed to by x[1]. A formula $\phi_1 * \phi_2$ holds if a heap can be split in two *disjoint* heaps such that the first heap satisfies ϕ_1, while the second heap satisfies ϕ_2. Finally, $*_{v:e} F(v)$ is the universal separating conjunction quantifier, which quantifies over the set of values for which the formula e is true. Notice that this is well-defined, because of the restriction to non-fractional permissions – for fractional permissions the semantics of quantification is only well-defined if the set is measurable.

A first-order formula A describing a functional property of a program is said to be *framed* by resource formula R if all resources necessary to evaluate A and the expressions in R are specified by R. Notice that a thread implicitly always holds full permissions to access local variables and method parameters. Framing is formally defined below, in Section 5.3.1.

5.1.2 Verification of GPGPU Kernels

The main goal of our logic is to prove (*i*) that a kernel does not have data races, and (*ii*) that it respects its functional behaviour specification. Kernels can exhibit two kinds of data races: (*i*) parallel threads within a work group can access the same location, either in global or in local memory, and this access is not ordered by an appropriate barrier, and (*ii*) parallel threads within different work groups can access the same locations in global memory, where in both cases one of the access operations must be a write. With our logic, we can verify the absence of both kinds of data races. Traditionally, separation logic considers a single heap for the program. However, to reason about kernels, we make an explicit distinction between global and local memory. To support our reasoning method, kernels, work groups and threads are specified as follows:

[1] In classical separation logic, this is usually written using the points-to predicate $\texttt{x} \overset{\pi}{\mapsto} v$, where additionally the location pointed to by x is known to hold v. Notice that $\texttt{x} \overset{\pi}{\mapsto} v$ is equivalent to $\texttt{Perm}(x,\pi) * x = v$.

- The *kernel specification* is a triple $(K_{res}, K_{pre}, K_{post})$. The resource formula K_{res} specifies all resources in global memory that are passed from the host program to the kernel, while K_{pre} and K_{post} specify the functional kernel pre- and postcondition, respectively. K_{pre} and K_{post} have to be framed by K_{res}. A kernel can only be invoked by a host program that transfers the necessary resources and respects the preconditions.

- The *group specification* is a triple $(G_{res}, G_{pre}, G_{post})$, where G_{res} specifies the resources in global memory that can be used by the threads in this group, and G_{pre} and G_{post} specify the functional pre- and postcondition, respectively, again framed by G_{res}. Notice that locations defined in local memory are only valid inside the work group and thus the work group always holds write permissions for these locations.

- Permissions and conditions in the work group are distributed over the work group's threads by the *thread specification* $(T^{res}_{pre}, T_{pre}, T^{res}_{post}, T_{post})$. Because threads within a work group can exchange permissions, we allow the resources before (T^{res}_{pre}) and after execution (T^{res}_{post}) to be different. The functional behaviour is specified by T_{pre} and T_{post}, which must be framed by T^{res}_{pre} and T^{res}_{post}, respectively.

- A *barrier specification* $(B_{res}, B_{pre}, B_{post})$ specifies resources, and a pre- and postcondition for each barrier in the kernel. B_{res} specify how permissions are redistributed over the threads (depending on the barrier flag, these can be permissions on local memory only, on global memory only, or a combination of global and local memory). The barrier precondition B_{pre} specifies the property that has to hold when a thread reaches the barrier. It must be framed by the resources that were specified by the previous barrier (considering the thread start as an implicit barrier). The barrier postcondition B_{post} specifies the property that may be assumed to continue verification of the thread. It should be *framed* by B_{res}.

Notice that it is sufficient to specify a single permission formula for a kernel and a work group. Since work groups do not synchronise with each other, there is no way to redistribute permissions over kernels or work groups. Within a work group, permissions are redistributed over the threads only at a barrier, the code between barriers always holds the same set of permissions.

Given a fully annotated kernel, verification of the kernel w.r.t. its specification essentially boils down to verification of the following properties:

- Each thread is verified w.r.t. the thread specification, *i.e.*, given the thread's code T_{body}, the Hoare triple $\{T_{res} * T_{pre}\}\ T_{body}\{T_{post}\}$ is verified using the permission-based separation logic rules defined in Section 5.3. Each barrier is verified as a method call with precondition $R_{\mathsf{cur}} * B_{pre}$ and postcondition $B_{res} * B_{post}$, where R_{cur} specifies all current resources.

- The kernel resources are sufficient for the distribution over the work groups, as specified by the group resources.

- The kernel precondition implies the work group's preconditions.

- The group resources and accesses to local memory are sufficient for the distribution of resources over the threads.

```
kernel demo {
  global int[gsize] a,b;
  void main(){
    a[tid]:=tid;
    barrier(global);
    b[tid]:=a[(tid+1) mod gsize];
  }
}
```

Figure 5.1: Basic example kernel

- The work group precondition implies the thread's preconditions.
- Each barrier redistributes only resources that are available in the work group.
- For each barrier the postcondition for each thread follows from the precondition in the thread, and the fenced conjuncts of the preconditions of all other threads in the work group.
- The universal quantification over all threads' postconditions implies the work group's postcondition.
- The universal quantification over all work groups' postconditions implies the kernel's postcondition.

Below these conditions will be formalised; here we will illustrate them with a small example.

Example 5.1.1. *Consider the kernel in Figure 5.1. For simplicity, it has a single work group. This kernel requires write permissions on arrays* a *and* b*. The kernel precondition states that the length of both arrays should be the same as the number of threads (denoted as gsize for work group size). The kernel postcondition expresses that afterwards, for any i in the range of the array,* $\mathtt{b}[i] = (i+1)\%gsize$*. Each thread i initially obtains a write permission at* $\mathtt{a}[i]$*, and moreover i is in the range of the arrays. When thread i reaches the barrier, the property* $\mathtt{a}[i] = i$ *holds; this is the barrier precondition. After the barrier, each thread i obtains a write permission on* $\mathtt{b}[i]$ *and a read permission on* $\mathtt{a}[(i+1)\%wgsize]$*, and it continues its computation with the barrier postcondition that* $\mathtt{a}[(i+1)\%gsize] = (i+1)\%gsize$*. From this, each thread i can establish the thread's postcondition* $\mathtt{b}[i] = (i+1)\%gsize$*, which is sufficient to establish the kernel's postcondition. See Fig. 5.8 for a tool-verified annotated version.*

Notice that the logic contains many levels of specification. However, typically many of these specifications can be generated, satisfying the properties above by construction. As discussed in Section 5.5 below, for the tool implementation it is sufficient to provide the thread and the barrier specifications.

5.2 Kernel Programming Language

This section defines syntax and semantics of a simple kernel language. The next section defines the logic over this simplified language, however we would like to emphasise that our tool can verify real OpenCL kernels.

Reserved global identifiers (constant within a thread):

tid	Thread identifier with respect to the kernel
gid	Group identifier with respect to the kernel
lid	Local thread identifier with respect to the work group
tcount	The total number of threads in the kernel
gsize	The number of threads per work group

Kernel language:

$$
\begin{array}{lcl}
b & ::= & \text{boolean expresion over global constants and private variables} \\
e & ::= & \text{integer expression over global constants and private variables} \\
S & ::= & \mathsf{v} := \mathsf{e} \mid v := \mathsf{rdloc}(e) \mid v := \mathsf{rdglob}(e) \mid \mathsf{wrloc}(e_1, e_2) \mid \mathsf{wrglob}(e_1, e_2) \\
 & \mid & \mathsf{nop} \mid S_1; S_2 \mid \mathsf{if}\, b\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \mid \mathsf{while}\, b\, \mathsf{do}\, S \mid bid : \mathsf{barrier}(F) \\
F & ::= & \emptyset \mid \{\mathsf{local}\} \mid \{\mathsf{global}\} \mid \{\mathsf{local}, \mathsf{global}\}
\end{array}
$$

Figure 5.2: Syntax for Kernel Programming Language

5.2.1 Syntax

Our language is based on the Kernel Programming Language (KPL) of Betts *et al.* [8]. However, the original version of KPL did not distinguish between global and local memory, while we do. As kernel procedures cannot recursively call themselves, we restrict the language to a single block of kernel code, without loss of generality. Fig. 5.2 presents the syntax of our language. Each kernel is merely a single statement, which is executed by all threads, where threads are divided into one or more work groups. For simplicity, but without loss of generality, global and local memory are assumed to be single shared arrays (similar to the original KPL presentation [8]). There are 4 memory access operations: read from location e_1 in local memory ($v := \mathsf{rdloc}(e_1)$); write e_2 to location e_1 in local memory ($\mathsf{wrloc}(e_1, e_2)$); read from global memory ($v := \mathsf{rdglob}(e)$); and write to global memory ($\mathsf{wrglob}(e_1, e_2)$). Finally, there is a barrier operation, taking as argument a subset of the flags local and global, which describes which of the two memories are fenced by the barrier. Each barrier is labelled with an identifier *bid*.

A common problem in kernel programming is that not all threads within the same work group reach the same barrier. In this case, the OpenCL specification states that the behaviour of the kernel is unspecified. Additionally, in barrier specifications, we cannot quantify a formula over all threads, if the formula uses private variables, unless we know their value in the other threads. Therefore, we add some additional syntactical restrictions that ensure that some private variables have the same value in all threads. With this restriction, our kernels do not suffer from *barrier divergence* and we can use these private variables in barrier specifications (see *e.g.*, the binomial coefficient example below).

Let $\mathscr{P}_{\mathsf{LS}}$ be the set of *lock-step-safe* private variables $\mathscr{P}$ that are updated in lock step within a work group. These are the reserved names gid, tcount, gsize, and all private variables that are assigned lock-step-safe expressions, *i.e.*, expressions built from purely functional operators and lock-step-safe variables. We consider two lock-step sensitive statements: barriers and assignment to a lock-step-safe variable. By requiring that conditions in conditionals and loops that contain lock-step sensitive statements are lock-step-safe, we can guarantee that the program is barrier divergence free and that lock-step-safe variables can be used in barrier preconditions.

Note that this restriction does not limit the expressiveness of specifications, as private, local and global ghost variables can be used to circumvent it. However, it does restrict how the

control flow of kernels may be written. We feel that our restriction is good practice that should be part of any coding convention for kernels. Moreover, techniques such as those employed in GPUVerify [8] can be adapted to implement a semantic check for barrier divergence and lock-step-safeness of expressions, rather than our syntactic check.

5.2.2 Semantics

To describe the behaviour of kernels, we present a small step operational semantics. In most GPU implementations, kernels operate in lock-step, *i.e.*, a subset of all the threads within a group execute all the same instruction. This results in the most efficient execution, because in the mean time, data that is used by the next subset of threads can be fetched from or written to memory. However, the specific details of this execution are hardware-specific. We intend our operational semantics to describe the most general behaviour possible, by considering all possible interleavings between two barriers. Soundness of our verification approach is proven w.r.t. to this most general behaviour, thus any verified property will hold for any possible implementation.

The logic requires for each thread to specify the permissions it holds between two barriers, and the verification rules for reading and writing ensure that these instructions can only be verified if the thread holds sufficient resources. Since the global behaviour is described as all possible interleavings of the threads between two barriers, it follows that for any state that is not at a barrier, a thread cannot make any assumptions about the state of other threads.

Throughout, we assume that we have sets Gid, Tid, and Bid of group, thread and barrier identifiers, with typical inhabitants *gid*, *tid*, and *bid*, respectively. As mentioned above, global and local memory are modelled as a single shared array. Private memory only contains scalar variables of type integer.

$$\begin{aligned}&\mathsf{GlobalMem} = \mathsf{LocalMem} = (\mathsf{Int} \to \mathsf{Int})\\ &\mathsf{PrivateMem} = (\mathsf{Var} \to \mathsf{Int})\end{aligned}$$

The state of a kernel KernelState consists of the global memory, and all its group states. The state of each group GroupState consists of local memory, and all its thread states. Finally, the state of a thread ThreadState consists of an instruction, its private state and a tag whether it is running (R), or waiting at barrier $bid \in \mathsf{Bid}$ ($\mathsf{W}(bid)$). Formally, this is defined as follows:

$$\begin{aligned}\mathsf{KernelState} &= \mathsf{GlobalMem} \times (\mathsf{Gid} \to \mathsf{GroupState})\\ \mathsf{GroupState} &= \mathsf{LocalMem} \times (\mathsf{Lid} \to \mathsf{ThreadState})\\ \mathsf{ThreadState} &= \mathsf{Stmt} \times \mathsf{PrivateMem} \times \mathsf{BarrierTag}\\ \mathsf{BarrierTag} &= \mathsf{R} \mid \mathsf{W}(bid)\end{aligned}$$

Below, updates to group and thread states are written using function updates, defined as follows: Given a function $f : A \to B$, $a \in A$, and $b \in B$:

$$f[a := b] = x \mapsto \begin{cases} b & , x = a \\ f(x), \text{ otherwise} \end{cases}$$

Notice that the operational semantics rules describing the behaviour of groups or threads can also update global or local memory. Therefore, the operational semantics of kernel behaviour is defined by the following three relations:

$$\begin{aligned}\to_{\mathscr{K}} &\subseteq (\mathsf{KernelState})^2\\ \to_{\mathscr{G},gid} &\subseteq (\mathsf{GlobalMem} \times \mathsf{GroupState})^2\\ \to_{\mathscr{T},tid} &\subseteq (\mathsf{GlobalMem} \times \mathsf{LocalMem} \times \mathsf{ThreadState})^2\end{aligned}$$

$$\frac{\Delta(gid) = (\delta,\Gamma) \quad (\sigma,\delta,\Gamma) \rightarrow_{\mathscr{G},gid} (\sigma',\delta',\Gamma')}{(\sigma,\Delta) \rightarrow_{\mathscr{K}} (\sigma',\Delta[gid := (\delta',\Gamma')])} \textbf{[kernel step]}$$

$$\frac{\Gamma(lid) = (S,\gamma,F) \quad (S,(\sigma,\delta,\gamma),F) \rightarrow_{\mathscr{T},gid\cdot gsize+lid} (S',(\sigma',\delta',\gamma'),F')}{(\sigma,\delta,\Gamma) \rightarrow_{\mathscr{G},gid} (\sigma',\delta',\Gamma[lid := (S',\gamma',F')])} \textbf{[group step]}$$

$$\frac{\forall lid \in \mathsf{Lid}.\Gamma(lid) = (S_{lid},\gamma_{lid},\mathsf{W}(bid))}{(\sigma,\delta,\Gamma) \rightarrow_{\mathscr{G},gid} (\sigma,\delta,lid \mapsto (S_{lid},\gamma_{lid},\mathsf{R}))} \textbf{[group barrier synchronise]}$$

$$\frac{}{(bid : \mathsf{barrier}(F),(\sigma,\delta,\gamma),\mathsf{R}) \rightarrow_{\mathscr{T},tid} (\varepsilon,(\sigma,\delta,\gamma),\mathsf{W}(bid))} \textbf{[barrier enter]}$$

$$\frac{}{(v := e,(\sigma,\delta,\gamma),\mathsf{R}) \rightarrow_{\mathscr{T},tid} (\varepsilon,(\sigma,\delta,\gamma[v := [e]^{tid}_{\gamma}]),\mathsf{R})} \textbf{[assign]}$$

$$\frac{}{(v := \textbf{rdglob}(e),(\sigma,\delta,\gamma),\mathsf{R}) \rightarrow_{\mathscr{T},tid} (\varepsilon,(\sigma,\delta,\gamma[v := \sigma([e]^{tid}_{\gamma})]),\mathsf{R})} \textbf{[global read]}$$

$$\frac{}{(v := \textbf{rdloc}(e),(\sigma,\delta,\gamma),\mathsf{R}) \rightarrow_{\mathscr{T},tid} (\varepsilon,(\sigma,\delta,\gamma[v := \delta([e]^{tid}_{\gamma})]),\mathsf{R})} \textbf{[local read]}$$

$$\frac{}{(\textbf{wrglob}(e_1,e_2),(\sigma,\delta,\gamma),\mathsf{R}) \rightarrow_{\mathscr{T},tid} (\varepsilon,(\sigma[[e_1]^{tid}_{\gamma} := [e_2]^{tid}_{\gamma}],\delta,\gamma)),\mathsf{R})} \textbf{[global write]}$$

$$\frac{}{(\textbf{wrloc}(e_1,e_2),(\sigma,\delta,\gamma),\mathsf{R}) \rightarrow_{\mathscr{T},tid} (\varepsilon,(\sigma,\delta[[e_1]^{tid}_{\gamma} := [e_2]^{tid}_{\gamma}],\gamma)),\mathsf{R})} \textbf{[local write]}$$

$$\frac{(S_1,(\sigma,\delta,\gamma),\mathsf{R}) \rightarrow_{\mathscr{T},tid} (S_1',(\sigma',\delta',\gamma'),\mathsf{R})}{(S_1;S_2,(\sigma,\delta,\gamma),\mathsf{R}) \rightarrow_{\mathscr{T},tid} (S_1';S_2,(\sigma',\delta',\gamma'),\mathsf{R})} \qquad \frac{(S_1,(\sigma,\delta,\gamma),\mathsf{R}) \rightarrow_{\mathscr{T},tid} (\varepsilon,(\sigma',\delta',\gamma'),\mathsf{R})}{(S_1;S_2,(\sigma,\delta,\gamma),\mathsf{R}) \rightarrow_{\mathscr{T},tid} (S_2,(\sigma',\delta',\gamma'),\mathsf{R})}$$

Figure 5.3: Small step operational semantics rules

Fig. 5.3 presents the rules defining these relations. As mentioned above, the operational semantics defines all possible interleavings. Therefore, the kernel state changes if one group changes its state. A group changes its state if one thread changes its state. A thread can change its state by executing an instruction according to the standard operational semantics rules for imperative languages, as long as its running. Figure 5.3 only gives the rules for sequential composition; the rules for conditionals and loops are standard. If a thread enters a barrier, it enters the "blocked at barrier" state. Once, at the group level, all threads have entered, the states are simultaneously switched back to running. The semantics of expression e over the private store γ in thread *tid* is denoted $[e]^{tid}_{\gamma}$; its definition is standard and not discussed further.

In the kernel's *initial states*, all memories are empty, and all threads contain the full kernel body as the statement to execute.

5.3 Program Logic

This section formally defines the rules to reason about OpenCL kernels. As explained above, we distinguish between two kinds of formulas: resource formulas (in permission-based separation logic), and property formulas (in first-order logic). Before presenting the verification rules, we first formally define syntax and validity of a resource formula for a given program state. Validity of the property formulas is standard, and we do not discuss this further.

$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(c) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(v) = (\emptyset,\emptyset) \quad \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(f(x_1,\cdots,x_n)) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(x_1) \cup \cdots \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(x_n)$$
$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{rdglob}(E)) = (\{[E]^{tid}_{(\sigma,\delta,\gamma)}\},\emptyset) \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E) \quad \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{rdloc}(E)) = (\emptyset,\{[E]^{tid}_{(\sigma,\delta,\gamma)}\}) \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E)$$
$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{true}) = (\emptyset,\emptyset) \quad \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R_1 \star R_2) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R_1) \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R_2)$$
$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{GPerm}(E,p)) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{LPerm}(E,p)) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E) \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(p)$$
$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E \Rightarrow R) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E) \cup ([E]^{tid}_{(\sigma,\delta,\gamma)})?(\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R)) : ((\emptyset,\emptyset))$$
$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\underset{v:E(v)}{*} R(v)) = (\bigcup\{\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E(v)) \mid v \in \mathbb{Z}\}) \cup (\bigcup\{\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R(v)) \mid [E(v)]^{tid}_{(\sigma,\delta,\gamma)}, v \in \mathbb{Z}\})$$

$$\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{true}) = (\emptyset,\emptyset) \quad \mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R_1 \star R_2) = \mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R_1) \cup \mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R_2)$$
$$\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(E \Rightarrow R) = ([E]^{tid}_{(\sigma,\delta,\gamma)})?(\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R)) : ((\emptyset,\emptyset))$$
$$\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{GPerm}(E,p)) = (\{[E]^{tid}_{(\sigma,\delta,\gamma)}\},\emptyset) \quad \mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{LPerm}(E,p)) = (\emptyset,\{[E]^{tid}_{(\sigma,\delta,\gamma)}\})$$
$$\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(\underset{v:E(v)}{*} R(v)) = \bigcup\{\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R(v)) \mid [E(v)]^{tid}_{(\sigma,\delta,\gamma)}, v \in \mathbb{Z}\}$$

Figure 5.4: Definition of footprint and provided resources

5.3.1 Syntax of Resource Formulas

Section 5.1.1 above defined the syntax of resource formulas. However, our kernel programming language uses a very simple form of expressions only, and the syntax explicitly distinguishes between access to global and local memory. Therefore, in our kernel specification language we follow the same pattern, and we explicitly use different permission statements for local and global memory.

As mentioned above, the behaviour of kernels, groups, threads and barriers is defined as tuples $(K_{res}, K_{pre}, K_{post})$, $(G_{res}, G_{pre}, G_{post})$, $(T^{res}_{pre}, T_{pre}, T^{res}_{post}, T_{post})$, and $(B_{res}, B_{pre}, B_{post})$, respectively, where the resource formulas are defined by the following grammar.

$$\begin{array}{lcl} E & ::= & \text{expressions over global constants, private variables, } \mathsf{rdloc}(E), \mathsf{rdglob}(E) \\ R & ::= & \mathsf{true} \mid \mathsf{LPerm}(E,p) \mid \mathsf{GPerm}(E,p) \mid E \Rightarrow R \mid R_1 \star R_2 \mid d \underset{v:E(v)}{*} R(v) \end{array}$$

Resource formulas can *frame* first-order logic formulas. To define this, we need the footprint of a formula, describing all global and local memory locations that are accessed to evaluate the formula. Moreover, for every resource formula we also need the resources that are *provided* by the formula. Figure 5.4 defines formally the footprint foot and the provided resources prov w.r.t. the thread identifier *tid* and the thread's current state (σ,δ,γ), where $\bigcup$ is lifted over the pair of global and local memory: $\bigcup\{(G_i,L_i) \mid i \in I\} = (\bigcup\{G_i \mid i \in I\}, \bigcup\{L_i \mid i \in I\})$. A first-order logic formula E is *framed* by a resource formula R if:

$$\forall \sigma, \Delta, tid \in \mathsf{Tid} : \ \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R) \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E) \subseteq \mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R)$$

Finally, pre- and postconditions are first-order logic formulas over E, correctly framed over the available resources.

5.3.2 Validity of Resource Formulas

To define validity of resource formulas, we have to extend the program state with permission tables for global and local memory (each thread always has full and exclusive access to its private memory). Above, we have defined global and local memory as a single array from indices to integers. Therefore, we define the *global* and *local permission table* as mappings from indices to a permission value in the domain $\mathsf{Perm} = \{\bot, \mathsf{rd}, \mathsf{rw}\}$:

$$\mathsf{GlobalPerm} = \mathsf{LocalPerm} = (\mathsf{Int} \to \mathsf{Perm})$$

Notice that we have the following order on the domain Perm: $\mathsf{rw} > \mathsf{rd} > \bot$.

Memory and permission tables are combined in a resource $\mathscr{R}$, defined as:

$$\mathscr{R} \in \mathsf{GlobalMem} \times \mathsf{LocalMem} \times \mathsf{GlobalPerm} \times \mathsf{LocalPerm}$$

For convenience, below we use appropriate accessor functions, such that $\mathscr{R} = (\mathscr{R}_{\mathsf{mg}}, \mathscr{R}_{\mathsf{ml}}, \mathscr{R}_{\mathsf{pg}}, \mathscr{R}_{\mathsf{pl}})$, and $\mathscr{R} = (\mathscr{R}_{\mathsf{mem}}, \mathscr{R}_{\mathsf{perm}})$.

Resources can be combined only if they are matching. Notice that because the logic supports quantification over arbitrary sets of integers, we define compatibility (and joining below) for arbitrary sets of arguments, rather than for just two arguments. We first define *compatibility* of memory and permission tables, denoted $\#_{\mathsf{m}}$ and $\#_{\mathsf{p}}$, respectively. Memories match if they store the same value for overlapping locations. Permission tables match if in case there is a write permission for a location, then they hold no other permissions for this location. Compatibility of resources, denoted #, is defined as compatibility of all resource components.

$$\begin{array}{rcl}
\#_{\mathsf{m}}\mathscr{M} & = & \forall v \in \mathsf{Int}. \exists m \in \mathscr{M}. m(v) \neq \bot \Rightarrow \forall m' \in \mathscr{M}. m'(v) \in \{\bot, m(v)\} \\
\#_{\mathsf{p}}\mathscr{P} & = & \forall v \in \mathsf{Int}. \exists p \in \mathscr{P}. p(v) = \mathsf{rw} \Rightarrow \forall p' \in \mathscr{P}. p \neq p' \Rightarrow p'(v) = \bot \\
\#(\mathscr{R}_c)_{c \in C} & = & \#_{\mathsf{m}}\{\mathscr{R}_{c\mathsf{mg}} \mid c \in C\} \wedge \#_{\mathsf{m}}\{\mathscr{R}_{c\mathsf{ml}} \mid c \in C\} \wedge \\
 & & \#_{\mathsf{p}}\{\mathscr{R}_{c\mathsf{pg}} \mid c \in C\} \wedge \#_{\mathsf{p}}\{\mathscr{R}_{c\mathsf{pl}} \mid c \in C\}
\end{array}$$

If resources are compatible, they can be combined. Again, we first define *joining* of memory and permissions, and then we define joining of resources.

$$\begin{array}{rcl}
\star_{\mathsf{m}}\mathscr{M} & = & \lambda v. \mathsf{if}\ \exists m \in \mathscr{M}. m(v) \neq \bot\ \mathsf{then}\ m(v)\ \mathsf{else}\ \bot \\
\star_{\mathsf{p}}\mathscr{P} & = & \lambda v. \mathsf{if}\ \exists p \in \mathscr{P}. P(v) \neq \bot\ \mathsf{then}\ p(v)\ \mathsf{else}\ \bot \\
\star(\mathscr{R}_c)_{c \in C} & = & (\star_{\mathsf{m}}\{\mathscr{R}_{c\mathsf{mg}} \mid c \in C\}, \star_{\mathsf{m}}\{\mathscr{R}_{c\mathsf{ml}} \mid c \in C\}, \\
 & & \star_{\mathsf{p}}\{\mathscr{R}_{c\mathsf{pg}} \mid c \in C\}, \star_{\mathsf{p}}\{\mathscr{R}_{c\mathsf{pl}} \mid c \in C\})
\end{array}$$

Last, in order to allow full permissions to be split into any (possibly infinite) number of read permissions, we define $\trianglerighteq$ as the greater or equal relation over permission tables, and then lift this to resources.

$$\begin{array}{rcl}
p_1 \trianglerighteq p_2 & \text{iff} & \forall v \in \mathsf{Int}. p_1(v) \geq p_2(v) \\
\mathscr{R}_1 \trianglerighteq \mathscr{R}_2 & \text{iff} & \mathscr{R}_{1\mathsf{pg}} \trianglerighteq \mathscr{R}_{2\mathsf{pg}} \wedge \mathscr{R}_{1\mathsf{pl}} \trianglerighteq \mathscr{R}_{2\mathsf{pl}}
\end{array}$$

Finally, validity of resource formula R is defined w.r.t. a typing environment Γ, whose definition is standard, and not discussed further; a *resource* $\mathscr{R}$, and a thread's private memory γ. Fig. 5.5 defines validity of the forcing relation $\Gamma \vdash \mathscr{R}; \gamma \models R$ by induction on the structure of the resource formula.

$$
\begin{array}{rcl}
\Gamma \vdash \mathscr{R}; p \models e & \Leftrightarrow & [\![e]\!]_{\mathscr{R}_{\mathsf{mem}},p} \\
\Gamma \vdash \mathscr{R}; p \models \mathtt{Perm}(\mathbf{rdglob}(e), \pi) & \Leftrightarrow & [\![\pi]\!] \leq \mathscr{R}_{\mathsf{pg}}([\![e]\!]_{\mathscr{R}_{\mathsf{mem}},p}) \\
\Gamma \vdash \mathscr{R}; p \models \mathtt{Perm}(\mathbf{rdloc}(e), \pi) & \Leftrightarrow & [\![\pi]\!] \leq \mathscr{R}_{\mathsf{pl}}([\![e]\!]_{\mathscr{R}_{\mathsf{mem}},p}) \\
\Gamma \vdash \mathscr{R}; p \models R_1 \star R_2 & \Leftrightarrow & \exists \mathscr{R}_1, \mathscr{R}_2. \mathscr{R} \Rrightarrow \mathscr{R}_1 \star \mathscr{R}_2. \\
 & & \quad \Gamma \vdash \mathscr{R}_1; p \models R_1 \wedge \Gamma \vdash \mathscr{R}_2; p \models R_2 \\
\Gamma \vdash \mathscr{R}; p \models \bigstar_{v:E(v)} R(v) & \Leftrightarrow & \exists (\mathscr{R}_v)_{v \in \{v \mid [\![E(v)]\!]\}}. \mathscr{R} \Rrightarrow \star\{\mathscr{R}_v \mid [\![E(v)]\!]\}. \\
 & & \quad \forall v \in v. \Gamma \vdash \mathscr{R}_v; p \models R(c)
\end{array}
$$

Figure 5.5: Validity of Resource Formulas

$$\frac{}{\{R, P[v := e]\}\, \mathsf{v} := \mathsf{e}\, \{R, P\}}\ \textbf{[assign]}$$

$$\frac{}{\{R \star \mathsf{LPerm}(e,\pi), P[v := L[e]]\}\, v := \mathsf{rdloc}(e)\, \{R \star \mathsf{LPerm}(e,\pi), P\}}\ \textbf{[read local]}$$

$$\frac{}{\{R \star \mathsf{LPerm}(e_1, \mathsf{rw}), P[L[e_1] := e_2]\}\, \mathsf{wrloc}(e_1, e_2)\, \{R \star \mathsf{LPerm}(e_1, \mathsf{rw}), P\}}\ \textbf{[write local]}$$

$$\frac{}{\{R_{\mathsf{cur}}, B_{pre}(bid)\}\, bid : \mathsf{barrier}(F)\, \{B_{res}(bid), B_{post}(bid)\}}\ \textbf{[barrier]}$$

$$\frac{R_1 \Rrightarrow R_1' \quad P_1 \Rightarrow P_1' \quad \{R_1', P_1'\}\, S\, \{R_2', P_2'\} \quad R_2' \Rrightarrow R_2 \quad P_2' \Rightarrow P_2}{\{R_1, P_1\}\, S\, \{R_2, P_2\}}\ \textbf{[weakening]}$$

Figure 5.6: Hoare logic rules

5.3.3 Hoare Triples for Kernels

Since in our logic we explicitly separate the resource formulas and the first-order logic properties, we first have to redefine the meaning of a Hoare triple in our setting, where the pre- and the postcondition consist of a resource formula, and a first-order logic formula, such that the pair is properly framed.

$$
\begin{array}{l}
\{R_1, P_1\}\, S\, \{R_2, P_2\} = \\
\quad \forall \mathscr{R}\, \gamma. (\Gamma \vdash \mathscr{R}; \gamma \models R_1 \star P_1) \wedge (S, (\mathscr{R}_{\mathsf{mg}}, \mathscr{R}_{\mathsf{ml}}, \gamma), \mathsf{R}) \rightarrow^* (\varepsilon, (\sigma, \delta, \gamma'), F) \Rightarrow \\
\quad\quad \forall \mathscr{R}'. \mathscr{R}'_{\mathsf{mg}} = \sigma \wedge \mathscr{R}'_{\mathsf{ml}} = \delta. \Gamma \vdash \mathscr{R}'; \gamma' \models R_2 \star P_2
\end{array}
$$

Fig. 5.6 summarises the most important Hoare logic rules to reason about kernel threads; in addition there are the standard rules for sequential compositional, conditionals, and loops. Rule **assign** applies for updates to local memory. Rules **read local** and **write local** specifies lookup and update of local memory (where $L[e]$ denotes the value stored at location e in the local memory array, and substitution is as usually defined for arrays, cf. [2]):

$$L[e][L[e_1] := e_2] = (e = e_1)?e_2 : L[e]$$

Similar rules are defined for global memory (not given here, for space reasons).

The rule **barrier** reflects the functionality of the barrier from the point of view of one thread. First, the resources before (R_{cur}) are replaced with the barrier resources for the thread ($B_{res}(bid)$).

Second, the barrier precondition ($B_{pre}(tid)$) is replaced by the post condition ($B_{post}(tid)$). The requirement that the preconditions within a group imply the postconditions is not enforced by this rule; it must be checked separately.

5.3.4 Soundness

Finally, we can prove soundness of our verification technique.

Theorem 5.3.1. *Suppose we have a specified, lock step restricted, kernel program*

$$\langle P, K_{res}, K_{pre}, K_{post}, G_{res}, G_{pre}, G_{post}, T^{res}_{pre}, T_{pre}, T^{res}_{post}, T_{post} \rangle$$

such that:

1. *the Hoare triple* $\{T^{res}_{pre}, T_{pre}\}P\{T^{res}_{post}, T_{post}\}$ *can be derived;*
2. *all global proof obligations hold,* i.e.*,*

$$K_{res} \circledast \mathop{*}_{gid \in \mathsf{Gid}} G_{res}(gid) \qquad \forall gid \in \mathsf{Gid}.G_{res}(gid) \circledast \mathop{*}_{tid \in \mathsf{Tid}(gid)} T^{res}_{pre}(tid)$$
$$\forall gid \in \mathsf{Gid}, bid \in \mathsf{Bid} : G_{res}(gid) \circledast \mathop{*}_{tid \in \mathsf{Tid}(gid)} B_{res}(bid, tid)$$
$$K_{pre} \Rightarrow (\forall gid \in \mathsf{Gid} G_{pre}(gid)) \qquad \forall gid.(G_{pre}(gid) \Rightarrow \forall tid \in \mathsf{Tid}(gid).T_{pre}(tid)$$
$$(\forall gid \in \mathsf{Gid}.G_{post}(gid)) \Rightarrow K_{post} \quad \forall gid.((\forall tid \in \mathsf{Tid}(gid).T_{post}(tid)) \Rightarrow G_{post})$$
$$\forall gid.(G_{pre}(gid) \Rightarrow \forall tid \in \mathsf{Tid}(gid).T_{pre}(tid)$$
$$\forall gid.((\forall tid \in \mathsf{Tid}(gid).T_{post}(tid)) \Rightarrow G_{post})$$

3. *all properties are properly framed.*

Then every execution of the kernel, starting in a state that satisfies K_{pre} *and has exclusive access to the resources* K_{res}*, will: (i) never encounter a data race; and (ii) upon termination satisfies* K_{post}*.*

Proof sketch. Work groups execute completely independent from each other, so w.l.o.g., we assume that there is only one work group.

We prove the result by induction on the number of barrier synchronisations in the trace. If there are no barrier synchronisations then the known Hoare logic proof is applicable. Otherwise, consider the trace up to and following the first barrier synchronisation. For the trace up to the barrier, the known proof applies. Since the barrier resources properly divide the group resources, the resources required by the second part of the trace are available. Since the barrier preconditions imply the postconditions, the functional properties required for the second part of the trace hold. For the second part of the trace after the barrier, the induction hypothesis proves the result. □

5.4 Tool Support

This section discusses how our logic for the functional verification of kernels, outlined in the previous section, is implemented in the VerCors tool set. It can be tried online at at `http://fmt.ewi.utwente.nl/puptol/vercors-verifier/`. The VerCors tool set is originally developed as a tool to reason about multithreaded Java programs. It encodes multithreaded Java

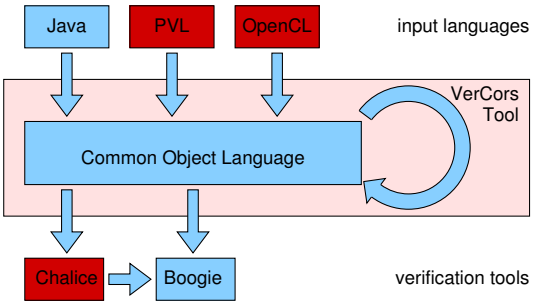


Figure 5.7: Overall architecture VerCors tool set

programs in several program transformation steps into Chalice [41]. Chalice is a verifier for an idealised multithreaded programming language, using permission-based separation logic as a specification language. Chalice in turn gives rise to an encoding in Boogie [4], which gives rise to SMT-compliant proof obligations. To support the verification of OpenCL kernels, we have added an extra input option to the VerCors tool set and we have also extended the toy language PVL with kernel syntax. Figure 5.7 sketches the overall architecture of the tool set (in some sequential cases, the VerCors tool directly generates a Boogie encoding).

Encoding of Kernels and their Specifications To verify a kernel, our method as discussed above gives rise to the following proof obligations:

1. global properties to ensure the correct relation between the different levels of specifications (*e.g.*, all kernel resources are properly distributed over a work group, and the universally quantified barrier precondition implies the universally quantified barrier postcondition);

2. correctness of a single arbitrary thread w.r.t. its specifications; and

3. ensuring correct framing of each pre- and postcondition.

To encode the first verification problem, for each global verification condition of the form "ϕ implies ψ", a Chalice method with an empty body is generated, with precondition ϕ and postcondition ψ.

Example 5.4.1. *Consider again the kernel in Example 5.1.1. It has a single work group, which has exactly the same resources as the kernel. To verify that the group resources are properly distributed over the threads at the barrier, the following method is generated:*

```
requires (\∀* int tid;0<=tid&&tid<gsize;Perm(this.a[tid],100));
requires (\∀* int tid;0<=tid&&tid<gsize;Perm(this.b[tid],100));
ensures (\∀* int tid;0<=tid&&tid<gsize;Perm(this.a[((tid+1)%gsize)],10));
ensures (\∀* int tid;0<=tid&&tid<gsize;Perm(this.b[tid],100));
void main_resources_of_1(int tcount,int gsize,int gid){}
```

The complete generated encoding for this example is available online.

Finding out the necessary conditions for the barrier checks is difficult. Therefore the tools uses the following sound approximations. (*i*) For each barrier and each group, the derived group-level resources should imply the resource conjunction of the barrier's post-resources. (*ii*) For each barrier and each thread, the derived group-level resources together with the private

```
kernel demo {
  global int[gsize] a;
  global int[gsize] b;

  requires perm(a[tid],100) * perm(b[tid],100);
  ensures perm(b[tid],100) * b[tid] = (tid+1) mod gsize;
  void main(){
    a[tid]:=tid;
    barrier(global){
      requires a[tid]=tid;
      ensures perm(a[(tid+1) mod gsize],10) * perm(b[tid],100);
      ensures a[(tid+1) mod gsize]=(tid+1) mod gsize; }
   b[tid]:=a[(tid+1) mod gsize];
  }
}
```

Figure 5.8: Tool input for the running example.

knowledge about un-fenced variables and the local knowledge about fenced variables from the barrier precondition should imply the barrier postcondition of the thread.

Next, the second verification problem essentially is a verification problem of a sequential thread. However, some special treatment is needed to encode the barrier invocations. In Chalice, we keep track of the last barrier visited by the thread, to allow to treat the barrier specification as a method contract. Specifically, this allows to specify the permissions that are handed in when reaching a barrier as the following method contract:

```
requires resources(last_barrier);
ensures resources(i);
int barrier_call(last_barrier, i)
```

Therefore, in the Chalice encoding, the code of the thread starts with the declaration `int last_barrier=0;`, and each call to barrier `i` is replaced with

```
last_barrier=barrier_call(last_barrier,#i#)
```

Finally, the third verification problem is handled by the built-in footprint checks of Chalice.

Generation of Kernel Specifications To make the verification easier, our tool also is able to generate many specifications. In particular, if a user specifies the following: (*i*) a thread's initial resources, precondition, and postcondition; and (*ii*) for each barrier, the barrier's pre- and postcondition, and the resources returned by the barrier, the work group and kernel specifications can be established from the thread's specification by universal quantification. We believe that in many cases, the barrier's postcondition can be established by restricting the universal quantification of the barrier's precondition to the resources returned by the barrier (*i.e.*, its frame). It is future work to investigate this further. Clearly, all generated specifications respect the corresponding proof obligations by construction.

Finally, the tool generates the resources that a thread hands in when reaching a barrier. The tool must do this because it replaces barrier statements, which implicitly take away all permissions, with a barrier method that must explicitly require them. To make the resulting contract valid, we also compute the purely non-deterministic abstraction of the control flow

```
1  kernel binomial {
2    global int[gsize] bin;
3    local int[gsize] tmp;
4
5    requires gsize > 1 * perm(bin[tid],100) * perm(tmp[tid],100);
6    ensures perm(bin[tid],100) * bin[tid]=binom(gsize-1,tid);
7    void main(){
8      int temp;
9      int N:=1;
10     bin[tid]:=1;
11     invariant perm(ar[tid],100) * perm(tmp[tid],100);
12     invariant tid<N ? ar[tid]=binom(N,tid) : ar[tid]=1;
13     while(N<gsize-1){
14       tmp[tid]:=ar[tid];
15       barrier(1,{local}){
16         ensures perm(ar[tid],100) * perm(tmp[(tid-1) mod gsize],10);
17         ensures 0<tid & tid<=N -> tmp[(tid-1) mod gsize]=binom(N,tid-1);
18       }
19       N := N+1;
20       if(0<tid & tid<N){
21         temp:=tmp[(tid-1) mod gsize];
22         ar[tid]:=temp+ar[tid];
23       }
24       barrier(2,{}){
25         ensures perm(ar[tid],100) * perm(tmp[tid],100);
26       }
27     }
28   }
29 }
```

Figure 5.9: Kernel program for binomial coefficients

of the kernel between two barriers (or between the barrier and the thread's end) and add that information to the barrier contract.

Example 5.4.2. *Figure 5.8 gives the running example in the PVL language used by the tool. All other specifications are generated by the tool.*

5.5 Example: Binomial Coefficient

Finally we discuss the verification of a more involved kernel, to illustrate the power of our verification technique. The full example is available online and can be tried in the online version of our tool set.

The kernel program in Fig. 5.9 computes the binomial coefficients $\binom{N-1}{0} \cdots \binom{N-1}{N-1}$ using N threads forming a single work group. Due to space restrictions, only the critical parts of the specifications have been given. The actual verified version has longer and more tedious specifications.

The intended output is the global array `bin`. The local array `tmp` is used for exchanging data between threads. The algorithm proceeds in $N-1$ iterations and in each iteration `bin` contains

a row from Pascal's triangle as the first part, and ones for the unused part.

On line 10 the entire `bin` array is initialized to 1. This satisfies the invariants on line 11/12 that states that the array `bin` contains the N^{th} row of Pascal's triangle, followed by ones. The loop body first copies the `bin` array to the `tmp` array, then using a barrier that fences the local variable. These values are then transmitted to the next thread and the write permission on `tmp` is exchanged for a read permissions. Then, for the relevant subset of threads, the equation

$$\binom{N}{k} = \binom{N-1}{k-1} + \binom{N-1}{k}$$

is used to update `bin`, and the second barrier returns write permission on `tmp`.

Note that the first barrier fences the local variables, which is necessary to ensure that the next thread can see the values. The second barrier does not fence any variables because it is only there to ensure that the value has been read and processed, making it safe to write the next value in `tmp`.

Also note the use of the two private variables, `N` and `temp`. The former is a lock step variable (assign 1 and then increment by one), but the latter is not. Therefore, the condition of the while loop is lock-step-safe, using only `N` and *gsize*. However, the condition of the if uses *tid*, which is not lock-step-safe, but as the conditional does not contain a barrier, or update a lock step variable, this does not cause any problems.

5.6 Summary and and Future Work

This chapter presents a verification technique for GPGPU kernels, based on permission-based separation logic. The main specifics are that (*i*) for each kernel and work group we specify all permissions that are necessary to execute the kernel, (*ii*) the permissions in the kernel are distributed over the work groups, (*iii*) the permission in the work group are distributed over the threads, and (*iv*) at each barrier the permissions are redistributed over the threads. Verification of individual threads uses standard program verification techniques, where barrier specifications are treated as method calls, while additional verification conditions check consistency of the specifications. We have shown validity of our approach on a non-trivial example, but need further tool development to apply our technique on larger examples.

Our approach naturally can support host code verification. To achieve this, it is sufficient to specify the behaviour of the API methods that are used in the host to initialise the kernel, and then to use a verification method for concurrent C programs using permission-based separation logic (such as Gotsman *et al.* [28]). In particular, the specification of the host method that invokes the kernel ensures that the host gives up the permissions that are transferred to the kernel. This is similar to fork-join reasoning for standard multithreaded programs [30]. It is future work to specify these methods, and to support this in our tool set.

Our specification method in principle is very verbose; specifications at many different levels are required. As discussed, many of the specifications can be generated by the tool. It is future work to see whether methods for generation of permission annotations (*e.g.*, by Ferrara and Müller [26]) can be used to further increase automation of our tool set.

Finally, we also plan to study verified optimisation of kernels. The idea is to start with a very simple and direct kernel implementation that can be verified directly, and then to optimise this into an efficient kernel by applying a collection of verified optimisations to implementation and specification, in such a way that correctness is preserved.

Acknowledgement We are very grateful to Christian Haack, who helped clarifying many of the formal details of the logic.

6 Related Approaches

In this chapter we discuss related work on static verification of GPU kernels in the literature (Section 6.1) and present a direct experimental comparison between the GPUVerify technique of Chapter 4 and the PUG verification tool (Section 6.2).

6.1 Other related work

PUG The closest work to the GPUVerify technique of Chapter 4 is the PUG analyzer for CUDA kernels [43]. 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 [31]. This semantics shares similarities with the SDV semantics used by GPUVerify [8] but the focus of [31] is not on verification of GPU kernels. A recent studying Hoare logic for GPU kernels is in a similar vein [40].

Symbolic execution and bounded-depth verification The GKLEE [45] and KLEE-CL [17] tools perform dynamic symbolic execution of CUDA and OpenCL kernels, respectively, and are both built on top of the KLEE symbolic execution engine [13].

A method for bounded verification of barrier-free GPU kernels via depth-limited unrolling to an SMT formula is presented in [60]; 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. 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 [47]. 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.

Combining static and dynamic analysis A verification approach for CUDA kernels [42] 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.

Protocol verification A reduction to two processes, similar to the two-thread reduction employed in Chapter 4, is at the heart of a method for verifying cache coherence protocols known as CMP [16] (CMP is derived from the surnames of the authors of this paper), which was inspired by the foundational work of McMillan [48]. With CMP, verification of a protocol for an arbitrary number of processes is performed by model checking a system where a small number of processes are explicitly represented and a highly nondeterministic 'other' process over-approximates the possible behaviours of the remaining processes. The unconstrained nature of the 'other' process can lead to spurious counterexamples, which must be eliminated either by introducing additional explicit processes, or by adding *non-interference lemmas* so that the actions of the 'other' process more precisely reflect the possible actions of processes in the concrete system. The CMP method has been extended and generalised with *message flows* and *message flow invariants* [59], which aid in the automatic derivation of non-interference lemmas by capturing large classes of permissible interactions between processes.

The GPUVerify approach uses the same high-level proof idea as the CMP method: we consider a small number of threads (two), and our default adversarial abstraction models the possible actions of all other threads, analogously to the 'other' process. The purpose of barrier invariants (Section 4.4) is to refine the adversarial abstraction so that the possible behaviours of additional threads are represented more precisely, thus barrier invariants can be seen as analogues to non-interference lemmas. However, the techniques aim to solve different problems: non-interference lemmas in the CMP method describe interaction sequences between processes in a message-based protocol, while barrier invariants for GPU kernel verification capture properties of the shared state in shared memory parallel programs, and thus there are many technical differences in the manner by which the high-level proof technique is applied in practice.

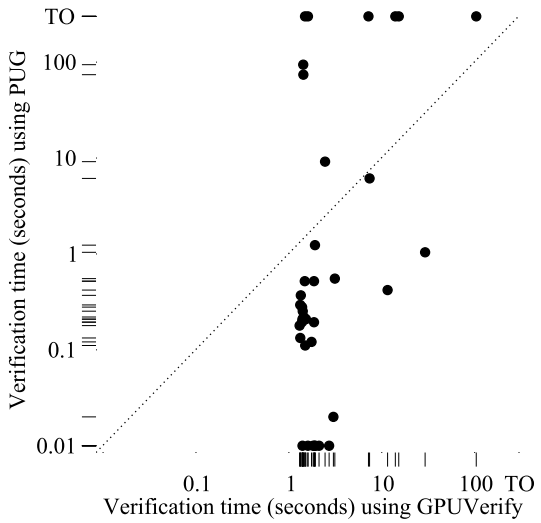


Figure 6.1: Scatter plot comparing the performance of GPUVerify and PUG on the CUDA SDK and C++ AMP kernels

6.2 Comparison of GPUVerify with PUG

We present a head-to-head comparison of PUG and GPUVerify on the CUDA benchmarks described in Section 4.6.1: 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 [43]. 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 [43], 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 because it uses a slightly more refined abstraction of the shared state than that employed by PUG.

Figure 6.1 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 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 single mutation into each kernel in the CUDA SDK and C++ AMP benchmark suites (we did not try multiple mutations simultaneously within a kernel, nor the same kernel separately with different mutations). 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; from a theoretical perspective this is not a surprise as the techniques on which GPUVerify is built are based on sound over-approximations, this evaluation provides some confidence that these techniques have been correctly implemented.

7 Future Work and Open Problems

We have presented complementary approaches to statically analysing OpenCL applications, based on: transformation to a sequential verification task; separation logic with permissions.

Comparison and integration between techniques We are now at a point where meaningful comparisons can be made between the techniques we have developed. We plan to look in detail at the relevant strengths and weaknesses associated with the race instrumentation method described in Chapter 4 and the permission-based approach of Chapter 5 for proving data race freedom. One difference is that the permission-based approach can also verify functional behaviour, while the race instrumentation method abstracts away from data.

While the methods we have investigated are complementary, we may also consider ways in which they could be combined. For instance, the separation logic-based method of Chapter 5 provides a framework for reasoning about both host and kernel code, something which the GPUVerify method of Chapter 4 is lacking and could thus draw upon.

Better invariant generation The main limitation of the GPUVerify approach is that the automatic invariant generation technique described in Section 4.3 is limited, incorporating only a small set of memory access patterns. We plan to look at dramatically increasing the tool's capability in this area, allowing the generation of a much wider class of invariants through the template-based method. The challenge here will be to increase the precision of invariant inference without sacrificing performance too much. To achieve this we plan to use dynamic analysis to quickly eliminate incorrect candidate invariants, and to employ parallel processing to accelerate the Houdini algorithm.

Automatic inference of specifications Relatedly, the method of Chapter 5 that uses permission-based separation logic is dependent on a number of specifications, for kernels, groups, threads and barriers. Currently we have derived these manually for examples. As tool support for this aspect of our work improves, we will consider the possibility of deriving such specifications automatically or semi-automatically.

Precise bug detection The methods of Chapters 4 and 5 aim to verify correctness of GPU software, and do not emphasise bug-finding specifically. If a proof of correctness fails, this may be indicative of a bug, but may also be a false positive due to insufficient invariants or permission specifications. Methods based on symbolic execution have more potential for bug-finding [45, 17]. We plan to investigate further the role of symbolic execution, and perhaps also dynamic analysis, in finding bugs suggested by static verification techniques.

Atomic operations and relaxed memory The techniques we have investigated so far are restricted to the barrier-synchronising computational model common to OpenCL and CUDA. However, these programming models also feature atomic operations which allow threads to communicate *between* barriers. The OpenCL 2.0 specification draft describes a fully-fledged memory model for OpenCL, based on the C++11 standard, equipped with a rich set of atomic operations. We are already working with members of the Khronos group on formalising and

clarifying the draft memory model, building on a successful formalisation for C++ [7]. We plan to investigate support for verifying kernels that use atomic operations in future work.

Bibliography

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

[2] K. Apt. Ten years of Hoare's logic: A survey − Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, oct 1981.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 364 – 387. Springer-Verlag, 2005.

[5] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87. ACM, 2005.

[6] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Computer Aided Verification - 23rd International Conference (CAV'11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[7] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In T. Ball and M. Sagiv, editors, *POPL*, pages 55–66. ACM, 2011.

[8] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPU kernels. In *OOPSLA*, pages 113–132, 2012.

[9] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *HPG*, pages 159–166, 2009.

[10] G. E. Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1990.

[11] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

[12] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer-Verlag, 2003.

[13] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

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

[15] N. Chong, A. F. Donaldson, P. Kelly, S. Qadeer, and J. Ketema. Barrier invariants: a shared state abstraction for the analysis of data-dependent GPU kernels. In *OOPSLA*, 2013.

[16] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*, pages 382–398, 2004.

[17] P. Collingbourne, C. Cadar, and P. H. J. Kelly. Symbolic testing of OpenCL code. In *Haifa Verification Conference (HVC 2011)*, 01/2012 2011.

[18] P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *ESOP*, pages 270–289, 2013.

[19] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload - automating code migration to heterogeneous multicore systems. In Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, editors, *HiPEAC*, volume 5952 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2010.

[20] L. M. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, pages 183–198, 2007.

[21] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.

[22] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

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

[24] A. F. 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, 2011.

[25] A. F. Donaldson, D. Kroening, and P. Rümmer. SCRATCH: a tool for automatic analysis of DMA races. In C. Cascaval and P.-C. Yew, editors, *PPOPP*, pages 311–312. ACM, 2011.

[26] P. Ferrara and P. Müller. Automatic inference of access permissions. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, LNCS, pages 202–218. Springer-Verlag, 2012.

[27] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[28] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *Proceedings of the 5th Asian conference on Programming languages and systems*, APLAS'07, pages 19–37. Springer-Verlag, 2007.

[29] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, volume 5356 of *LNCS*, pages 171–187. Springer-Verlag, 2008.

[30] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 199–215. Springer-Verlag, 2008.

[31] A. Habermaier and A. Knapp. On the correctness of the simt execution model of GPUs. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 316–335. Springer, 2012.

[32] M. Harris. Fast fluid dynamics simulation on the GPU. In *GPU Gems*, volume 1, chapter 38. Addison-Wesley, 2004.

[33] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison-Wesley, 2007.

[34] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[35] A. Hobor and C. Gherghina. Barriers in concurrent separation logic. In *20th European Symposium of Programming (ESOP 2011)*, LNCS, pages 276–296. Springer-Verlag, 2011.

[36] M. Huisman and M. Mihelčić. Specification and verification of GPGPU programs using permission-based separation logic. In *BYTECODE*, 2013.

[37] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with Portend. In T. Harris and M. L. Scott, editors, *ASPLOS*, pages 185–198. ACM, 2012.

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

[39] P. Klus, S. Lam, D. Lyberg, M. Cheung, G. P. G, I. McFarlane, G. Yeo, and B. Lam. BarraCUDA - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5, 2012.

[40] K. Kojima and A. Igarashi. A Hoare logic for SIMT programs. In *APLAS*, 2013.

[41] K. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Lecture notes of FOSAD*, volume 5705 of *LNCS*, pages 195–222. Springer-Verlag, 2009.

[42] A. Leung, M. Gupta, Y. Agarwal, et al. Verifying GPU kernels by test amplification. In *PLDI*, pages 383–394, 2012.

[43] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *FSE'10*, pages 187–196. ACM, 2010.

[44] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and 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.

[45] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *PPOPP*, pages 215–224. ACM, 2012.

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

[47] P. Li, G. Li, and G. Gopalakrishnan. Parametric flows: automated behavior equivalencing for symbolic analysis of races in CUDA programs. In J. K. Hollingsworth, editor, *SC*, page 29. IEEE/ACM, 2012.

[48] K. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–234, 1999.

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

[50] Microsoft Corporation. Overview of C++ accelerated massive parallelism (C++ AMP). `http://msdn.microsoft.com/en-us/library/hh265136%28v=VS.110%29.aspx`.

[51] M. Moskal. Programming with triggers. In *SMT*, pages 20–29, 2009.

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

[53] NVIDIA. NVIDIA CUDA C programming guide, version 4.0, 2011.

[54] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[55] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

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

[57] R. F. Salas-Moreno, R. A. Newcombe, H. Strasdat, P. H. J. Kelly, and A. J. Davison. SLAM++: Simultaneous localisation and mapping at the level of objects. In *CVPR*, pages 1352–1359. IEEE, 2013.

[58] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

[59] M. Talupur and M. R. Tuttle. Going with the flow: Parameterized verification using message flows. In *FMCAD*, pages 1–8, 2008.

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