Specification and Verification of Atomic Operations in GPGPU Programs

Afshin Amighi, Saeed Darabi, Stefan Blom, and Marieke Huisman

University of Twente, The Netherlands,
{a.amighi,s.darabi,s.blom,m.huisman}@utwente.nl

Abstract. We propose a specification and verification technique based on separation logic to reason about data race freedom and functional correctness of GPU kernels that use *atomic operations* as synchronisation mechanism. Our approach exploits the notion of *resource invariant* from Concurrent Separation Logic (CSL) to capture the behaviour of atomic operations. However, because of the different memory levels in the GPU architecture, we adapt this notion of resource invariant to these memory levels, i.e., *group resource invariants* capture the behaviour of atomic operations that access locations in local memory, while *kernel resource invariants* capture the behaviour of atomic operations that access locations in global memory. We show soundness of our approach and we provide tool support that enables us to verify kernels from standard benchmarks suites.

1 Introduction

General purpose GPU (GPGPU) programming enables programmers to use the power of massively parallel accelerator devices to solve computationally intensive problems with a significant speed up. However, massive parallelism also makes programming more error prone: data races might be difficult to detect, and moreover ensuring functional correctness becomes a challenge. To address this issue, different verification techniques for GPGPU programs have been developed [5, 3], based on separation logic and abstraction, respectively. However, these techniques do not support reasoning about functional properties of kernels using atomic operations. This paper discusses how the separation logic approach to reason about GPGPU programs is extended to reason about programs that use atomics for synchronisation.

GPU programming is based on the notion of *kernels*. A kernel consists of a large number (typically hundreds) of parallel threads that all execute the same instructions. The GPU execution model is an extension of the *Single Instruction Multiple Data* (SIMD) model[1], in which each thread executes the same instruction but on different data. For efficiency reasons, threads are grouped into *work groups*. Each work group has its own *local memory*, shared among all threads in

[1] To be precise, the GPU execution model is Single Instruction Multiple Thread (SIMT), which extends SIMD with more flexibility in the control flow.

the work group. Further, the kernel has a *global memory*, which is shared among all threads on the GPU device. Threads within a work group usually synchronise by *barriers*. Atomic operations provide asynchronous updates on shared memory locations (either in global or local memory) and are the only mechanism to support inter-group synchronisation in GPU programs. Moreover, atomic operations are also sometimes used for synchronisation within a work group, because they enable more flexible parallel behaviours than using barriers alone. For example, the *Parallel add* example in Section 3 and the *Histogram* example in the Parboil benchmark [15] benefit from the flexible parallel behaviour of atomic operations.

In earlier work, we used permission-based separation logic to reason about data race freedom and functional correctness of GPGPU kernels that use barriers as the only synchronisation construct [5]. This paper extends this logic to reason about kernels that also use atomic operations. The main idea of our work is to adapt the notion of *resource invariants*, as originally introduced for Concurrent Separation Logic (CSL) by O'Hearn, to reason about the behaviour of atomic operations w.r.t. the GPU memory hierarchy.

Resource invariants capture the properties of shared memory locations. These properties only may be violated by a thread that is in the critical section, and thus has exclusive access to the shared memory locations. Before leaving the critical section, the thread has to ensure that the resource invariants are re-established. Because of the GPU memory hierarchy, shared memory locations can be both in local memory (shared between threads in a single work group) and in global memory (shared between all threads). Therefore, in our approach we use *group resource invariants* that capture the properties for local shared memory locations, and *kernel resource invariants* to capture the properties for global shared memory locations. For each kernel, there always is a single kernel resource invariant, while for each work group there is a group resource invariant. However, by parametrising the group resource invariant with the group identifier *gid*, this can be specified with a single formula.

Note that we use the term shared memory locations instead of atomic variables, because the atomicity of a variable may change between different barrier intervals. Therefore, resource invariants should be re-established when a thread executes either an atomic operation or a barrier.

To conclude, the main contributions of this paper are:

- a specification and verification technique that adapts the notion of CSL resource invariants to the GPU memory model and enables us to reason about data race freedom and functional correctness of GPGPU kernels containing atomic operations;
- a soundness proof of our approach; and
- a demonstration of the usability of our approach by developing automated tool support for it[2].

The remainder of this paper is organised as follows. After some background information, Section 3 explains how the behaviour of GPGPU kernels with

[2] Our implementation supports the OpenCL programming language, but can easily be extended to other GPGPU programming languages such as CUDA and C++ AMP.

atomic operations is specified. Then, Section 4 formalizes our approach, while we conclude with related and future work in Sections 5 and 6.

2 Background

This section first gives a short overview of Concurrent Separation Logic, and then discusses how we use it to reason about GPGPU programs with barriers.

2.1 Atomic Operations in Concurrent Separation Logic

Separation Logic (SL) [13] is an extension of Hoare logic, originally developed to reason about imperative pointer-manipulating data structures. The basic predicate in classical SL is the *points-to* predicate $x \mapsto v$, meaning x points to a location on the heap, and this location contains the value v. These basic points-to predicates can be combined using the *separating conjunction* $\star$, which implicitly asserts disjointness of the locations: $\phi \star \psi$ holds for a heap h if formulas ϕ and ψ hold for *disjoint* subheaps of h.

O'Hearn introduced CSL as an extension of SL to reason about concurrent programs [12]. CSL allows one to verify threads in isolation, provided they do not interfere and operate on disjoint parts of the heap. In order to reason about programs with simultaneous reads, CSL has been extended with the notion of fractional permissions to denote the right to either read from or write to a location [7, 6]. The formula $\mathsf{Perm}(e, \pi)$ indicates that a thread holds an access right π to the heap location e, where any fraction of π in the interval $(0, 1)$ denotes a read permission and 1 denotes a write permission. Write permissions can be split into read permissions, while multiple read permissions can be combined into a write permission. For example, $\mathsf{Perm}(x, 1/2) \star \mathsf{Perm}(y, 1/2)$ indicates that a thread holds read permissions to access locations x and y, and these permissions are disjoint. If a thread holds $\mathsf{Perm}(x, 1/2) \star \mathsf{Perm}(x, 1/2)$, this can be merged into a write permission $\mathsf{Perm}(x, 1)$.

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, any verified program is free of data races.

When locations on the heap are shared, CSL expresses properties about this shared state as a *resource invariant*. Typically, a resource invariant captures the access permission to the shared location, but additionally it can also express a functional requirement on it. This leads to the following general judgement in CSL: $I \vdash \{P\}\ S\ \{Q\}$, which expresses that (1) shared state is specified with resource invariant I, (2) if the execution of S terminates, it turns a state satisfying precondition P into a state satisfying postcondition Q, and (3) I must be true before the execution, throughout the execution and after the execution.

One safe way to access shared locations is by using atomic operations, written $\mathsf{atomic}\{S\}$, which means that body S is executed in one atomic step. To reason about atomic operations, CSL uses the following proof rule [16]:

$$\frac{\mathsf{emp} \vdash \{I \star P\}\ S\ \{I \star Q\}}{I \vdash \{P\}\ \mathsf{atomic}\{S\}\ \{Q\}} \tag{1}$$

```
/*@  requires Perm(a[gtid],1) ** Perm(b[gtid],1);
     ensures  Perm(b[gtid],1) ** b[gtid] = (gtid+1) % gsize; @*/
kernel void rotate(global int a, global int b){
  a[gtid]=gtid;
  barrier(global){
  /*@  requires a[gtid]=gtid;
       ensures Perm(a[(gtid+1) % gsize],1/2) ** Perm(b[gtid],1);
       ensures a[(gtid+1) % gsize]=(gtid+1) % gsize;  @*/
  }
  b[gtid]=a[(gtid+1) % gsize];
}
```

List. 1. An example of a kernel with specifications

where emp is a predicate expressing that there is not any shared location in the heap, I is the resource invariant, P is a precondition that holds for the executing thread's local state before the atomic operation, Q is a postcondition that holds for the local state of the executing thread after the atomic operation, and S is the body of the atomic operation accessing the shared state expressed by I. This rule captures that a thread executing the body of an atomic operation obtains the associated resource invariant, which provides access to the shared state. Moreover, it may violate the resource invariant during the execution of S, but it has to re-establish the resource invariant before finishing the atomic operation. Section 3 explains how this CSL rule is adapted for GPU programs.

2.2 Reasoning about GPGPU Programs

In earlier work, we used permission-based separation logic to reason about GPU kernels with barriers [5]. Kernels, work groups, threads, and barriers are specified and verified modularly w.r.t. their specifications.

We illustrate the approach using the example in List. 1, which contains a kernel program annotated with a *thread specification*, plus a *barrier specification* for each barrier[3]. The specifications use the keywords gtid to denote the global thread identifier, and gsize to denote the number of threads in each work group, respectively. A thread specification specifies the permissions a thread should hold before (keyword **requires**) and after (keywords **ensures**) execution, together with the thread's functional behaviour. In the example, write permission to position gtid of both array a and b is required and it is ensured that position gtid of array b can be written and contains (gtid+1)% gsize. To illustrate the use of a barrier, the kernel is implemented in a non-standard way: first gtid is assigned to a[gtid] and then access to the array is rotated by synchronisation on a barrier, after which the thread reads a[(gtid+1) % gsize]. This rotation is specified with a *barrier specification*, which specifies (1) how permissions are redistributed over

[3] In our specification language we use ** for star conjunction because of the syntactic overlap with multiplication.

the threads in the work group, and (2) the functional pre- and postconditions that must hold before and after execution of the barrier.

There are two ways to specify the redistribution of permissions at a barrier in a work group. First, one can choose to redistribute all permissions available to the work group, assuming that each thread loses all permissions at a barrier. Second, one can force the user to explicitly specify which permissions are lost. Our original paper and the example use the first approach, which is efficient for proving data race freedom. In the rest of this paper, we use the second approach, which is more convenient for functional properties, as it ensures all functional properties are properly *framed* [5].

Given a thread specification which is parametrized by *gtid*, the *group specification* and *kernel specification* are defined as the *universal separating conjunction* of the thread specification over all threads in the same work group and over all threads in the GPU, respectively. Thus, group and kernel specifications are automatically derived from the thread specifications, and do not have to be explicitly given. Group specifications capture the resources in global memory that can be used by the threads in a particular work group, including its pre- and postcondition. 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. In the kernel specification, resources that are required from the host program along with the necessary preconditions and provided postconditions are specified. An invocation of a kernel by a host program is correct if the host program transfers the necessary resources and fulfils the kernel preconditions.

3 Specification

This section discusses two examples that illustrate our approach to the specification of kernels with atomic operations. The first example uses a single atomic add; the second example illustrates how we reason about kernels which use both barriers and atomic operations for synchronisation, and where the atomicity of a variable may change in different barrier intervals.

3.1 Specification of a Kernel with Parallel Addition

List. 2 contains an annotated parallel add kernel, where ltid indicates the local thread identifier. For simplicity, in this first example we assume that we have a single work group[4], later we extend our technique also to multiple work groups. We first explain the permission specifications, followed by an explanation of the functional properties (the highlighted annotations).

In List. 2, each thread atomically adds its contribution (stored in values[ltid]) to the shared variable x. The **requires** and **ensures** clauses express a single thread's pre- and postconditions. The precondition specifies that each thread needs to

[4] The number of work groups is determined in the host code before launching the kernel.

```
/*@ given int cont[gsize];
 group invariant Perm(x,1)**Perm(cont[*],1/2)**x==(\sum cont[*]);
 requires Perm(values[ltid],1/2)**Perm(cont[ltid],1/2)**cont[ltid]==0;
 ensures Perm(values[ltid],1/2)**Perm(cont[ltid],1/2)**cont[ltid]==values[ltid];@*/
kernel void gpadd(local int x, local int values){
  atomic_add(x,values[ltid]) /*@ then { cont[ltid]=values[ltid]; } @*/;  }
```

List. 2. Specification of parallel add in a work group.

have read permission on its corresponding index of values. Additionally, we specify a group resource invariant for the local shared memory variable x, which expresses that the thread executing the atomic add operation has exclusive write access to x. With this specification, it is straightforward to prove that the program is free of data races, as it is guaranteed that there is only one thread executing the atomic operation and exclusively accessing the shared variable.

To reason about functional properties, the specification expresses the accumulative contributions of the threads on the shared variable. To track these contributions, we use an array cont[ ], added as a ghost parameter (line 1) to the kernel[5]. The idea is that the contribution of each thread (cont[ltid]) is 0 before it executes and values[ltid] after it finishes, while the invariant $\sum_{i=0}^{\text{gsize-1}} \text{cont}[i] = \text{x}$ is maintained in order to prove that the kernel computes the sum of the values. To make this work, the thread's precondition (line 3) states that each tread obtains a read permission on cont[ltid], in order to be able to use cont in the specifications. Each thread has to track its contribution towards the total in x in its own location in the cont array. This is done during the atomic operation by injecting an assignment statement as ghost code (specified as a then clause, see line 6). The thread executing atomic_add, first adds values[ltid] to x, and then executes the injected ghost code, *i.e.* cont[ltid]=values[ltid]. To achieve this, the group resource invariant is extended with a half permission on *all* elements of cont, written Perm(cont[∗],1/2)[6]. Thus, when thread ltid at the beginning of the atomic body obtains the resource invariants, it has *twice* a read permission Perm(cont[ltid],1/2), which can be combined into a single write permission Perm(cont[ltid],1).

3.2 Parallel Addition with Multiple Work Groups

As a next example, we discuss the specification of a kernel with multiple work groups, which employs both barriers and atomic operations for synchronisation. This is a common pattern to avoid making global memory access a bottleneck: first all threads in a work group compute an intermediate result in local memory, then the intermediate result is combined with the global result in global memory.

[5] A ghost variable (a.k.a. as auxiliary variable) is a specification-only variable, which does not change the control flow of the program and is used only for verification.

[6] This is syntactic sugar for universal quantification of the permissions over all the indices of cont[].

```
/*@ given global int sums[ksize]={0}; given local int cont[gsize]={0}, region=0;
  kernel invariant Perm(r,1)**Perm(sums[*],1/2)**r==(\sum sums[*]);
  group invariant Perm(region,1/(gsize+1))**Perm(x,region==0?1:1/2)**
                  Perm(cont[*],1/2)**x==(\sum cont[*]);
  requires Perm(region,1/(gsize+1))**Perm(values[gtid],1/2);
  requires Perm(cont[ltid],1/2)**cont[ltid]==0;
  requires ltid==0 ==> Perm(sums[gid],1/2)**sums[gid]==0;
  ensures  Perm(region,1/(gsize+1))**Perm(values[gtid],1/2);
  ensures  Perm(cont[ltid],1/4)**cont[ltid]==values[gtid];
  ensures  ltid==0 ==> Perm(cont[*],1/4)**Perm(sums[gid],1/2);
  ensures  ltid==0 ==> sums[gid]==(\sum cont[*]);  @*/
kernel void KParallelAdd(local int x, global int values, global int r){
  atomic_add(x,values[gtid]) /*@ then { cont[ltid]=values[gtid]; } @*/;
  barrier(local)/*@
    requires Perm(region,1/(gsize+1))**region==0}**Perm(cont[ltid],1/4);
    ensures Perm(region,1/(gsize+1))**region==1;
    ensures  ltid==0 ==> Perm(cont[*],1/4)**x==(\sum cont[*]);
  { region=1; } @*/;
  if(ltid==0)
    atomic_add(r,x)/*@ then { sums[gid]=x; } @*/;   }
```

List. 3. Specification of global parallel add.

It is used, for example, in the parallel implementation of BFS in the Parboil benchmark [15]. The kernel in List. 3 is an extension of the previous example, using multiple work groups and a barrier, where ksize denotes the number of work groups. The kernel is implemented by the following steps: (1) each thread atomically adds its element of the global array values to its local accumulator, *i.e.* a locally shared variable x; (2) all threads within a work group are synchronized by a barrier (line 14); (3) after all threads have passed the barrier, one thread per work group (here ltid= 0) adds the work group's final value of x to a *globally* shared variable r (line 20). Eventually, r contains the collective contributions of all the threads in the kernel. Similar to the single work group example, to track the contributions at each step, the kernel program uses ghost arrays cont and sums, with all elements initialized with zero. We use cont to specify the current value of the local variable x. Similarly, array sums is used to sum up the total accumulated contributions of the work groups. Updating the local cont is explained in the previous example. In a similar way, using the ghost code at line 20, in each work group, the thread with ltid= 0 stores its contribution (the final value of x) to the global sums[gid], *i.e.* the index corresponding to the executing work group from the sums array.

In List. 3, there are two invariants that are maintained:

1. $\sum_{i=0}^{\text{gsize}-1} \text{cont}[i] = \text{x}$ for each work group; and
2. $\sum_{i=0}^{\text{ksize-1}} \text{sums}[i] = \text{r}$ for the kernel.

After termination of work group gid, we use the group invariant to conclude that:

$$\mathsf{sums[gid]} = \sum_{i=\mathsf{gsize}\times\mathsf{gid}}^{\mathsf{gsize}\times\mathsf{gid}+\mathsf{gsize}-1} \mathsf{values}[i] \ .$$

Hence after termination of all work groups we can prove that:

$$\mathsf{r} = \sum_{i=0}^{\mathsf{ksize\text{-}1}} \mathsf{sums}[i] = \sum_{j=0}^{\mathsf{ksize\text{-}1}} \sum_{i=j\times\mathsf{gsize}}^{(j+1)\times\mathsf{gsize}-1} \mathsf{values}[i]$$

Again, we first explain the permission specifications. The permission specifications for values are similar to the specifications in List. 2. The barrier divides the program into regions, and within a region the distribution of permissions over the threads and the resource invariants does not change. Only when all threads reach the barrier, permissions may be redistributed. This means in particular that a variable that is treated as a shared memory variable in one region, may become unshared in a next region (or vice versa). Thus, resource invariants often depend on the current barrier region. To keep track of the current barrier region, we use a ghost variable region initialised at 0 (line 1). Each thread at all times has read access to this region variable, and whenever all the threads go through the barrier, the region is updated (see line 18). The group resource invariant specifies that within region 0 (before the barrier instruction), variable x is a shared variable in local memory, while in region 1 (after the barrier), x is not shared any more. So, after the barrier x can be read concurrently by all the threads within a work group. The kernel resource invariant specifies that r is a shared variable in global memory, but that only threads with a local thread identifier 0 are able to correctly update r, because only threads with ltid= 0 can construct a write permission of sums[gid] (see lines 2 and 7) to store the contributions.

The barrier specification expresses that threads keep read access on region, and that the value of region is updated to 1. Moreover, the specification asserts that upon entering the barrier each thread gives up $1/4$ permission to access its contribution element, *i.e.* cont[ltid]. The barrier redistributes these permissions to the thread with ltid= 0, which ensures that the thread with ltid= 0 has sufficient permissions to frame (\sum cont[$*$]) in the barrier postcondition. Notice that when all threads have reached the barrier, all read accesses on region together (including the group resource invariant) can be combined into a write permission on region, thus enabling the update of this ghost variable within the barrier.

Next, we discuss the functional property specifications. As we stated before, two resource invariants specify the values of the shared variables: (1) the local shared variable x must always express the accumulation of the contributions of the threads executing the first atomic operation (line 4), and (2) the global shared variable r must always express the accumulation of x's final value in each work, group which is stored in sums[gid] (line 2). To prove these invariants, each thread must ensure that it correctly stores its contribution as specified in line 9. Moreover, the barrier must ensure that the thread with ltid= 0 knows the

final value of x as specified by x==(\sum cont[∗]) in the barrier's postcondition. Finally, the thread with ltid= 0 must guarantee that the final value of x is stored in sums[gid] (line 11). Therefore, the verifier can prove that the value of r is the collective contributions of all the threads in the kernel.

4 Formalisation

The previous section illustrated how we specify permissions and functional properties of kernel programs in the presence of atomic operations and barriers on several examples. This section defines the approach formally. Rather than presenting this work on the full language, we will present it for a core kernel programming language. In our verification technique barrier divergence is not taken into consideration, *i.e.* if threads in a work group arrive at a barrier they all arrive at the *same one*. This is a realistic assumption: according to the OpenCL semantics, the behaviour of programs with barrier divergence is unspecified [11]. Moreover, in our earlier work [5], we proposed syntactical restrictions to determine whether a kernel programs is free of barrier divergence.

We first introduce syntax and semantics of our core kernel language, and also formally define the formula language to write the specifications. Then we present the Hoare logic rules used to reason about kernels with atomics, and we prove soundness of the proof rules. Finally, we also briefly discuss tool implementation.

4.1 Syntax and Semantics

Programming Language Figure 1 presents the syntax for our kernel programming language, which adapts the Kernel Programming Language (KPL) of [3] by extending it with atomic operations and changing the barrier statement. For simplicity, in this language, global and local memory are assumed to be single shared arrays. There are two local memory access operations: read from location e_1 in local memory ($v := \mathsf{rdloc}(e_1)$), and write e_2 to location e_1 in local memory ($\mathsf{wrloc}(e_1, e_2)$). Similarly, read and write operations in global memory are represented by $v := \mathsf{rdglob}(e)$ and $\mathsf{wrglob}(e_1, e_2)$, respectively. W.r.t. to the original KPL language, barriers are different. As in KPL, a barrier is labelled with a flag F, which denotes which memories it synchronises. That is, it always acts both as synchronisation between the threads in a work group and as a memory fence. Depending on the flag, it is either for local or for global memory. Additionally, a barrier is labelled with an identifier *bid*, which is used to distinguish different barrier instances, and it is extended with a block of statements to be executed while all threads are in the barrier. Further, we add an atomic block statement to the language, which a label to denote whether it accesses global or local shared memory. The (annotated) OpenCL atomic operations can be easily embedded into this atomic block statement.

The state of a kernel program consists of the state of the global memory, the states of the local memories and the state of all the threads. On these states, three steps are possible:

Reserved global identifiers (constant within a thread):
gtid Thread identifier with respect to the kernel
gid Group identifier with respect to the kernel
ltid Local thread identifier with respect to the work group
tc The total number of threads in the kernel
gs The number of threads per work group
ks The number of groups in the kernel

Kernel language:
b ::= boolean expression over global constants and private variables
e ::= integer expression over global constants and private variables
$S ::= v := e \mid v := \mathsf{rdloc}(e) \mid v := \mathsf{rdglob}(e) \mid \mathsf{wrloc}(e_1, e_2) \mid \mathsf{wrglob}(e_1, e_2)$
$\quad \mid \mathsf{nop} \mid S_1;S_2 \mid \mathsf{if}\, b\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \mid \mathsf{while}\, b\, \mathsf{do}\, S$
$\quad \mid \mathsf{atomic}(F)\{S\} \mid bid : \mathsf{barrier}(F)\{S\}$
$F ::= \mathsf{local} \mid \mathsf{global}$

Fig. 1. Syntax for Kernel Programming Language

1. A thread performs a non-atomic statement, see [5] for details of the operational semantics;
2. A thread *atomically* performs all statements in an $\mathsf{atomic}(F)\{S\}$ block. Its operational semantics is standard and can be defined easily, similar to [16].
3. All threads in the work group go through the barrier $bid : \mathsf{barrier}(F)\{S\}$. This can only happen if all threads in a group are waiting to execute S. The effect on the state is that all statements in S are performed, and all threads in the group consider *bid* as performed. The operational semantics of a barrier without a body is defined in [5]. However, its extension with a body is trivial as the body is executed atomically.

Note that because barriers are labelled in KPL, any program that exhibits barrier divergence will block forever and therefore does not terminate.

Formula Language The specifications of KPL programs can be written using the following formula language:

$E ::=$ expressions (in first-order logic) over global constants, private variables, $\mathsf{rdloc}(E)$, $\mathsf{rdglob}(E)$.

$R ::= \mathsf{true} \mid E \mid \mathsf{LPerm}(E, p) \mid \mathsf{GPerm}(E, p) \mid R_1 \star R_2 \mid E \Rightarrow R \mid \bigstar_{v:E(v)} R(v)$

where we use $\mathsf{LPerm}(E, p)$ and $\mathsf{GPerm}(E, p)$ as explicitly different permission statements to specify accesses to local and global memories, respectively. In addition to the separating conjunction of two resource formulas, we also have guarded resource formulas, and a universal separating conjunction quantifier, which quantifies over the set of values v for which $E(v)$ is true. Formalization of the specification language and validity of the formulas are elaborated in [5].

The behaviour of kernels, work groups, threads, and barriers are defined as $(K_{pre}, K_{post}, K_{rinv})$, $(G_{pre}, G_{post}, G_{rinv})$, (T_{pre}, T_{post}), and (B_{pre}, B_{post}), respectively. Note that the user only has to annotate a kernel resource invariant K_{rinv}, a group resource invariant G_{rinv} parametrized by group id, a thread's pre- and

postcondition T_{pre} and T_{post} and barrier's pre- and postcondition B^{bid}_{pre} and B^{bid}_{post}. We can derive the work groups' pre- and postconditions, *i.e.* G_{pre} and G_{post}, as the separating conjunction of the pre- and postconditions of all threads belonging to the work group and the work group's resource invariant. Similarly, the kernel's pre- and postcondition, *i.e.* K_{pre} and K_{post}, can be derived automatically as the separating conjunction of the pre- and postconditions of all work groups belonging to the kernel and the kernel's resource invariant.

4.2 Verification

Since we derive the contracts for work groups and kernels automatically, we can verify a kernel program by verifying all the threads belonging to a kernel. To verify a thread T, with body T_{body}, the following Hoare triple should be verified, using the verification rules defined in Figure 2:

$$K_{rinv}, G_{rinv}(gid) \ \vdash \ \{T_{pre}\}\ T_{body}\ \{T_{post}\}$$

In addition to the standard rules for sequential compositional, conditionals, loops, and weakening, Figure 2 shows the most important Hoare logic rules to reason about kernel threads. Rule **[Assign]** describes the updates to the thread's private memory. Rules **[LRead]** and **[LWrite]** specifies read and write of local memory[7]. The rules for global memory are defined similarly, but for space reasons are not presented here. The rules **[LAtomic]** for local and **[GAtomic]** for global atomic operations are simple instances of the CSL rule using the group resource invariant and kernel resource invariant, respectively.

The rule **[LBarrier]** reflects the functionality of the barrier with a flag indicating that it synchronises local memory. It acts similar to the CSL rule for the group resource invariant but at the same time it collects resources and knowledge from all threads and redistributes these resources and knowledge. To do so it requires that the block S can be executed given the resources provided by the invariant (G_{rinv}) and all threads in the work group ($R(t)$). Moreover, it ensures that all resources are given back ($E(t)$) and the invariant is re-established (G_{rinv}). The rule also says that the effect of passing through a barrier on a thread is to give up resources $R(t)$ and get $E(t)$ in return. Note that there is a side condition that S, R and E can refer to local memory only, as this would otherwise potentially create a data race: a local barrier functions as a memory fence for local memory, thus it can exchange information about local memory without any difficulties, but no order on global memory is guaranteed. The **[GBarrier]** rule is symmetric in the use of local vs. global memory and invariants. Note that the local/global flag affects memory only. Both uses of the barrier synchronise the threads within a single work group.

[7] $L[e]$ denotes the value stored at location e in the local memory array, and substitution is as usually defined for arrays, cf. [1]:

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

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \{R[v := e]\}\ v\ :=\ e\ \{R\}}\ \textbf{[Assign]}$$

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \{\mathsf{LPerm}(e,\pi) \star R[v := L[e]]\}\ v\ :=\ \mathsf{rdloc}(e)\ \{\mathsf{LPerm}(e,\pi) \star R\}}\ \textbf{[LRead]}$$

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \{\mathsf{LPerm}(e_1,1) \star R[L[e_1] := e_2]\}\ \mathsf{wrloc}(e_1,e_2)\ \{\mathsf{LPerm}(e_1,1) \star R\}}\ \textbf{[LWrite]}$$

$$\frac{\begin{array}{l}S \text{ refers to local memory only.}\\ K_{rinv} \vdash \{P(t) \star G_{rinv}(gid)\}\ S\ \{G_{rinv}(gid) \star Q(t)\}\end{array}}{K_{rinv}\ , G_{rinv}(gid) \vdash\ \{P(t)\}\ \mathsf{atomic(local)}\{S\}\ \{Q(t)\}}\ \textbf{[LAtomic]}$$

$$\frac{\begin{array}{l}S \text{ refers to global memory only.}\\ G_{rinv}(gid) \vdash \{P(t) \star K_{rinv}\}\ S\ \{K_{rinv} \star Q(t)\}\end{array}}{K_{rinv}\ , G_{rinv}(gid) \vdash\ \{P(t)\}\ \mathsf{atomic(global)}\{S\}\ \{Q(t)\}}\ \textbf{[GAtomic]}$$

$$\frac{\begin{array}{l}S,\ R, \text{ and } E \text{ refer to local memory only.}\\ K_{rinv} \vdash \{\bigstar_{t\in[0..gs)} R(t) \star G_{rinv}(gid)\}\ S\ \{G_{rinv}(gid) \star \bigstar_{t\in[0..gs)} E(t)\}\end{array}}{\begin{array}{c}\{P(t) \star R(t)\}\\ K_{rinv}\ , G_{rinv}(gid) \vdash\ \textbf{barrier}(\mathsf{local})\quad \textbf{req}\ R(t);\quad \textbf{ens}\ E(t);\ \{S\}\\ \{P(t) \star E(t)\}\end{array}}\ \textbf{[LBarrier]}$$

$$\frac{\begin{array}{l}S,\ R, \text{ and } E \text{ refer to global memory only.}\\ G_{rinv}(gid) \vdash \{\bigstar_{t\in[0..gs)} R(t) \star K_{rinv}\}\ S\ \{K_{rinv} \star \bigstar_{t\in[0..gs)} E(t)\}\end{array}}{\begin{array}{c}\{P(t) \star R(t)\}\\ K_{rinv}\ , G_{rinv}(gid) \vdash\ \textbf{barrier}(\mathsf{global})\quad \textbf{req}\ R(t);\quad \textbf{ens}\ E(t);\ \{S\}\\ \{P(t) \star E(t)\}\end{array}}\ \textbf{[GBarrier]}$$

Fig. 2. Important Hoare logic rules

4.3 Soundness

Finally, we prove soundness of our verification technique.

Theorem 1. *Given a barrier divergence free kernel, for which the thread level Hoare triples are provably correct. Then every possible execution of the kernel starting in a state that satisfies the kernel precondition is data race free and ends in a state that satisfies the kernel postcondition.*

Proof. We are given a finite trace of executions.

In this trace every thread $t_{gid,ltid}$ makes a finite number of steps $N_{gid,ltid}$, where atomic blocks and barriers count as one step. Because a Hoare logic proof of the thread exists, we can find formulas $P^0_{gid,ltid}, \cdots, P^{N_{gid,ltid}}_{gid,ltid}$ that are valid before, between and after these steps, where $P^0_{gid,ltid}$ is the precondition of the thread and $P^{N_{gid,ltid}}_{gid,ltid}$ is its postcondition.

All states $\sigma_0, \cdots, \sigma_N$ in the finite global trace of N steps can be described by a function f that maps each global trace position to the positions in the local threads. We do not know in which order the steps of the threads are executed, but we know they all start in position 0, so $f(0, gid, ltid) = 0$. We also know they end in their last state, so: $f(N, gid, tid) = N_{gid,ltid}$.

We claim that before and after every step in the trace the state satisfies a specific separation logic formula.

$$\forall i = 0, \cdots, N : \ \sigma_i \models K_{rinv} \star \bigstar_{gid \in [0..ks)} \left(G_{rinv}(gid) \star \bigstar_{ltid \in [0..gs)} P_{gid,ltid}^{f(i,gid,ltid)} \right)$$

This claim is proven by induction on i. For $i = 0$ this is precisely the given precondition. Assuming that the claim is correct for $0 \leq i < N$, then there are three cases. If the step is a plain step or an atomic step, by correctness of the standard CSL Hoare triple used to prove that step, the validity for $i+1$ follows.

The interesting case is the barrier step, in which all threads of a group are involved. The Hoare triple for each thread is valid so each thread starts knowing $P(t) \star R(t)$ and ends knowing $P(t) \star E(t)$. Because of the correctness of the standard CSL Hoare triple for the barrier statement S, the change to the state is from $\bigstar_{t \in [0..gs)} R(t) \star G_{rinv}(gid)$ to $\bigstar_{t \in [0..gs)} E(t) \star G_{rinv}(gid)$, which is precisely the change in the formulas, so $i+1$ is established.

The last statement is precisely the kernel postcondition which proves that the end state satisfies the kernel postcondition.

A data race happens if: there is an access to a location l in step i_1 by thread t_1, followed by an access to the same location in step i_2 by thread t_2, there is no memory fence in between these accesses, and one of these accesses is a write. Suppose that t_1 used fraction p_1 for the access and thread t_2 used fraction p_2. Because one of the accesses is a write, $p_1 + p_2 > 1$. Because there is no memory fence, that is no barrier or atomic in between, at time i_1 thread t_2 must have already owned fraction p_2. Thus at time i_1, fraction $p_1 + p_2$ permission for location l existed, which leads to a contradiction. □

4.4 Tool Support

We have implemented tool support for the verification of kernels in the VerCors tool set [4], whose stable version can be tried online[8]. The VerCors tool set compiles programs that are specified in a complex specification language, such as kernels, into much simpler specified programs and then verifies the latter to prove that the former are correct. The main compilation target used for kernel programs is Silver, the intermediate language of the Viper framework [9]. Silver is a specification language designed along the lines of Implicit Dynamic Frames [14]. We can then verify these Silver programs with the Silicon tool that is part of the framework.

[8] See http://www.utwente.nl/vercors/.

For the verification of kernels with atomics, two transformation passes have been added to the VerCors tool set. The first pass transforms a kernel into an intermediate form that uses the same barrier and atomic constructs as used in the kernel programming language used in this section. The second pass replaces those atomic and barrier constructs with code that mimics the conclusion of the corresponding proof rules (see Figure 2) and adds code that encodes that the premisses of the rule is valid. The replacement ensures that when using a barrier or atomic proof rule the program is correct. The added code verifies that the rule is used correctly.

5 Related Work

There is very little related work in this area, as reasoning techniques for GPU kernels are still relatively fresh. Bardsley et al. propose additional support in GPUVerify for reasoning about GPU kernels where warps and atomic operations are used for synchronisation [2]. In GPUVerify the user does not need to add specifications manually, because the tool internally speculates and refines kernel specifications [3]. However, GPUVerify is not able to reason about the functional properties of kernels, it can only prove absence of data races. As future work, we would like to investigate if GPUVerify could be used to infer some of the annotations that we need.

Concerning verification of GPU kernels, we should also mention the work of Li and Gopalakrishnan [10]. They verify CUDA programs by symbolically encoding thread interleavings. They were the first to observe that to ensure data race freedom it was sufficient to verify the interleavings of two arbitrary threads. For each shared variable they use an array to keep track of read and write accesses, and where in the code they occur. By analysing this array, they detect possible data races. However, they do not consider atomic operations.

In the verification of (general) concurrent programs synchronized with barriers, Hobor *et al.* [8] propose a sound extension of CSL for Pthreads-style barriers. The simplicity of the OpenCL barriers makes our specification simpler. Additionally, we support barriers in the presence of atomic operations.

6 Conclusion

This paper presented an approach to specify and verify GPGPU programs in the presence of atomic operations and barriers. The main characteristics of the approach are that it can be used to prove both data race freedom and functional correctness. To specify the shared memory accesses, the notion of resource invariant from CSL is lifted to the GPU memory model, distinguishing between kernel and group resource invariants. An appropriate Hoare logic is proposed and proven sound to reason about GPGPU programs using atomic operations and barriers. The approach is illustrated on some examples, and supported by an implementation in the VerCors tool set.

At the moment, the user still has to write quite a substantial amount of annotations to make verification work. We will investigate how to make use of inference techniques for program annotations to reduce this annotation burden.

Acknowledgement This work is supported by the ERC 258405 VerCors project and by the EU FP7 STREP 287767 project CARP.

References

1. K. R. Apt, *Ten years of Hoare's logic: A survey − Part I*, ACM Trans. Program. Lang. Syst., 3 (1981), pp. 431–483.
2. E. Bardsley and A. Donaldson, *Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels*, in NASA Formal Methods, vol. 8430 of LNCS, Springer, 2014, pp. 230–245.
3. A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, *GPUVerify: a verifier for GPU kernels*, in OOPSLA'12, ACM, 2012, pp. 113–132.
4. S. Blom and M. Huisman, *The VerCors Tool for verification of concurrent programs*, in FM 2014, C. Jones, P. Pihlajasaari, and J. Sun, eds., vol. 8442 of LNCS, Springer, 2014, pp. 127–131.
5. S. Blom, M. Huisman, and M. Mihelčić, *Specification and verification of GPGPU programs*, Science of Computer Programming, 95(3) (2014), pp. 376 – 388.
6. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, *Permission accounting in separation logic*, in POPL '05, ACM, 2005, pp. 259–270.
7. J. Boyland, *Checking interference with fractional permissions*, in SAS'03, Springer-Verlag, 2003, pp. 55–72.
8. A. Hobor and C. Gherghina, *Barriers in concurrent separation logic*, in ESOP 2011, LNCS, Springer, 2011, pp. 276–296.
9. U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers, *Viper: A verification infrastructure for permission-based reasoning*, tech. rep., ETH Zurich, 2014.
10. G. Li and G. Gopalakrishnan, *Scalable SMT-based verification of GPU kernel functions*, in SIGSOFT FSE 2010, ACM, 2010, pp. 187–196.
11. NVIDIA Corporation, *CUDA C programming guide, version 5.5*, 2013.
12. P. W. O'Hearn, *Resources, concurrency and local reasoning*, Theoretical Computer Science, 375 (2007), pp. 271–307.
13. J. Reynolds, *Separation logic: A logic for shared mutable data structures*, in Logic in Computer Science, IEEE Computer Society, 2002, pp. 55–74.
14. J. Smans, B. Jacobs, and F. Piessens, *Implicit dynamic frames*, ACM Trans. Program. Lang. Syst., 34(1) (2012), pp. 2:1–2:58.
15. J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu, *Parboil: A revised benchmark suite for scientific and commercial throughput computing*, Center for Reliable and High-Performance Computing, (2012).
16. V. Vafeiadis, *Concurrent separation logic and operational semantics*, Electr. Notes Theor. Comput. Sci., 276 (2011), pp. 335–351.