Specification and verification of GPGPU programs

Stefan Blom, Marieke Huisman and Matej Mihelčić

University of Twente, Enschede, The Netherlands
{*s.c.c.blom,m.huisman*}*@utwente.nl*

November 8, 2013

Abstract

Graphics Processing Units (GPUs) are increasingly used for general-purpose applications because of their low price, energy efficiency and enormous computing power. Considering the importance of GPU applications, it is vital that the behaviour of GPU programs can be specified and proven correct formally. This paper presents a logic to verify GPU kernels written in OpenCL, a platform-independent low-level programming language. The logic can be used to prove both data-race-freedom and functional correctness of kernels. The verification is modular, based on ideas from permission-based separation logic. We present the logic and its soundness proof, and then discuss tool support and illustrate its use on a complex example kernel.

1. Introduction

Graphics processing units (GPUs) originally have been designed to support computer graphics. Their architecture supports fast memory manipulation, and a high processing power by using massive parallelism, making them suitable to efficiently solve typical graphics-related tasks. However, this architecture is also suitable for many other programming tasks, leading to the emergence of the area of *General Purpose GPU* (GPGPU) programming. Initially, this was mainly done in CUDA [1], a proprietary GPU programming language from NVIDIA. However, from 2006 onwards, OpenCL [2] has become more and more popular as a new platform-independent, low-level programming language standard for GPGPU programming. Nowadays, GPUs are used in many different fields, *e.g.*, media processing [3], medical imaging [4], and eye-tracking [5].

Despite the platform-independence, OpenCL programs are still developed at a relatively low level, and in particular, applications have to be optimised for the actual device used. Given the importance, range and increasing complexity of GPGPU applications, formal techniques to reason about their correctness are necessary. This paper presents a verification technique for GPGPU programs based on permission-based separation logic.

Before presenting our verification technique, we first briefly discuss the main characteristics of the GPU architecture (for more details, see the OpenCL specification [2]). A GPU runs hundreds of threads simultaneously. All threads within the same *kernel* execute the same instruction, but on different data: the *Single Instruction Multiple Data (SIMD)* execution model. GPU kernels are invoked by a *host* program, typically running on a CPU. Threads are grouped

into *work groups*. GPUs have three different memory regions: *global*, *local*, and *private* memory. Private memory is local to a single thread, local memory is shared between threads within a work group, and global memory is accessible to all threads in a kernel, and to the host program. Threads within a single work group can synchronise by using a *barrier*: all threads blocks at the barrier until all other threads have also reached this barrier. A barrier instruction comes with a flag to indicate whether it synchronises global or local memory, or both. Notice that threads within different work groups cannot synchronise.

The main inspiration for our verification approach is the use of permission-based separation logic to reason about multithreaded programs [6, 7, 8]. 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. [9]). 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 paper is organised as follows. Section 2 outlines our verification approach; Section 3 formally defines the kernel programming language, and its semantics; Section 4 presents the logic and its soundness proof. Section 5 discusses tool support for the logic, and Section 6 presents several verification examples. Finally, Section 7 discusses related work, while Section 8 presents conclusions and future work. This paper extends the short paper presented at Bytecode 2013 with a formal semantics, verification rules, a soundness proof, a tool description, and a more involved example.

2. 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.

2.1. Permission-based Separation Logic

Separation logic [10] was originally developed as an extension of Hoare logic [11] 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, while in separation logic, the state is explicitly divided in the heap and a store, related to the current method call. Separation logic is also suited to reason modularly about concurrent programs [12]: 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. [6] extended separation logic with fractional permissions. Permissions, originally introduced by Boyland [13], 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):

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

Note that our logic is restricted to a positive fragment by omitting disjunction, and using implication from booleans to resources and conjunction of resources; 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, $\mathop{\Large *}_{v:e} F(v)$ 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 4.1.

2.2. 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

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

not ordered by an appropriate barrier, and (*ii*) parallel threads within different work groups can access the same locations in global memory. 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:

- 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 4. 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.

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

Figure 1: Basic example kernel

- 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.
- 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 1. *Consider the kernel in Figure 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. 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 6 below, for the tool implementation it is sufficient to provide the thread and the barrier specifications.

3. 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 2: Syntax for Kernel Programming Language

3.1. Syntax

Our language is based on the Kernel Programming Language (KPL) of Betts *et al.* [14]. 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. 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 [14]). 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 $\mathcal{P}_{\mathsf{LS}}$ be the set of *lock-step-safe* private variables $\mathcal{P}$ 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 [14] can be adapted to implement a semantic check for barrier divergence and lock-step-safeness of expressions, rather than our syntactic check.

3.2. 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 $\mathsf{KernelState}$ consists of the global memory, and all its group states. The state of each group $\mathsf{GroupState}$ consists of local memory, and all its thread states. Finally, the state of a thread $\mathsf{ThreadState}$ 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 $b \in B$:

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

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

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

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

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

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

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

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

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

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

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

Figure 3: Small step operational semantics rules

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:

$$\begin{array}{rcl} \rightarrow_{\mathcal{K}} & \subseteq & (\mathsf{KernelState})^2 \\ \rightarrow_{\mathcal{G},gid} & \subseteq & (\mathsf{GlobalMem} \times \mathsf{GroupState})^2 \\ \rightarrow_{\mathcal{T},tid} & \subseteq & (\mathsf{GlobalMem} \times \mathsf{LocalMem} \times \mathsf{ThreadState})^2 \end{array}$$

Fig. 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 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 $[\![e]\!]_{\gamma}^{tid}$; 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.

$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(c) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(v) = (\emptyset,\emptyset) \quad \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(f(x_1,\cdots,x_n)) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(x_1) \cup \cdots \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(x_n)$$

$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{rdglob}(E)) = (\{[\![E]\!]^{tid}_{(\sigma,\delta,\gamma)}\},\emptyset) \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E) \quad \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{rdloc}(E)) = (\emptyset,\{[\![E]\!]^{tid}_{(\sigma,\delta,\gamma)}\}) \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E)$$

$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{true}) = (\emptyset,\emptyset) \quad \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R_1 \star R_2) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R_1) \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R_2)$$

$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{GPerm}(E,p)) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{LPerm}(E,p)) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E) \cup \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(p)$$

$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E \Rightarrow R) = \mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E) \cup ([\![E]\!]^{tid}_{(\sigma,\delta,\gamma)})?(\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R)) : ((\emptyset,\emptyset))$$

$$\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(\mathop{\Large *}_{v:E(v)} R(v)) = (\bigcup\{\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(E(v)) \mid v \in \mathbb{Z}\}) \cup (\bigcup\{\mathsf{foot}^{tid}_{(\sigma,\delta,\gamma)}(R(v)) \mid [\![E(v)]\!]^{tid}_{(\sigma,\delta,\gamma)}, v \in \mathbb{Z}\})$$

$$\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{true}) = (\emptyset,\emptyset) \quad \mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R_1 \star R_2) = \mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R_1) \cup \mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R_2)$$

$$\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(E \Rightarrow R) = ([\![E]\!]^{tid}_{(\sigma,\delta,\gamma)})?(\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R)) : ((\emptyset,\emptyset))$$

$$\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{GPerm}(E,p)) = (\{[\![E]\!]^{tid}_{(\sigma,\delta,\gamma)}\},\emptyset) \quad \mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(\mathsf{LPerm}(E,p)) = (\emptyset,\{[\![E]\!]^{tid}_{(\sigma,\delta,\gamma)}\})$$

$$\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(\mathop{\Large *}_{v:E(v)} R(v)) = \bigcup\{\mathsf{prov}^{tid}_{(\sigma,\delta,\gamma)}(R(v)) \mid [\![E(v)]\!]^{tid}_{(\sigma,\delta,\gamma)}, v \in \mathbb{Z}\}$$

Figure 4: Definition of footprint and provided resources

4. 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.

4.1. Syntax of Resource Formulas

Section 2.1 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.

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

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 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.

4.2. 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 $\mathsf{Perm} = \{\bot, \mathsf{rd}, \mathsf{rw}\}$:

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

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 $\mathcal{R}$, defined as:

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

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

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.

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

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

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

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

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

Figure 5: Validity of Resource Formulas

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

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

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

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

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

Figure 6: Hoare logic rules

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

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* $\mathcal{R}$, and a thread's private memory γ. Fig. 5 defines validity of the forcing relation $\Gamma \vdash \mathcal{R}; \gamma \models R$ by induction on the structure of the resource formula.

4.3. 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.

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

Fig. 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. [15]):

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

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.

4.4. Soundness

Finally, we can prove soundness of our verification technique.

Theorem 2. *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.,

$$K_{res} \circledast \mathop{\Large *}_{gid\in\mathsf{Gid}} G_{res}(gid) \qquad \forall gid \in \mathsf{Gid}.G_{res}(gid) \circledast \mathop{\Large *}_{tid\in\mathsf{Tid}(gid)} T^{res}_{pre}(tid)$$

$$\forall gid \in \mathsf{Gid}, bid \in \mathsf{Bid} : G_{res}(gid) \circledast \mathop{\Large *}_{tid\in\mathsf{Tid}(gid)} B_{res}(bid, tid)$$

$$K_{pre} \Rightarrow (\forall gid \in \mathsf{Gid} G_{pre}(gid)) \qquad \forall gid.(G_{pre}(gid) \Rightarrow \forall tid \in \mathsf{Tid}(gid).T_{pre}(tid)$$

$$(\forall gid \in \mathsf{Gid}.G_{post}(gid)) \Rightarrow K_{post} \quad \forall gid.((\forall tid \in \mathsf{Tid}(gid).T_{post}(tid)) \Rightarrow G_{post})$$

$$\forall gid.(G_{pre}(gid) \Rightarrow \forall tid \in \mathsf{Tid}(gid).T_{pre}(tid)$$

$$\forall gid.((\forall tid \in \mathsf{Tid}(gid).T_{post}(tid)) \Rightarrow G_{post})$$

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. □

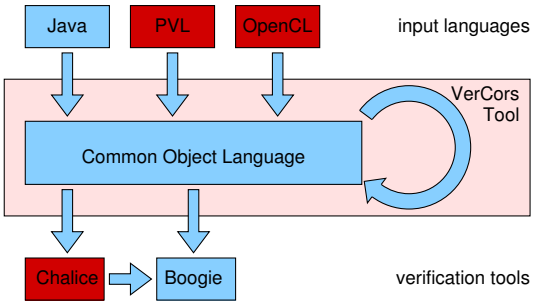

Figure 7: Overall architecture VerCors tool set

5. 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 programs in several program transformation steps into Chalice [16]. 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 [17], 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 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 3. *Consider again the kernel in Example 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 (\forall* int tid;0<=tid&&tid<gsize;Perm(this.a[tid],100));
requires (\forall* int tid;0<=tid&&tid<gsize;Perm(this.b[tid],100));
ensures  (\forall* int tid;0<=tid&&tid<gsize;Perm(this.a[((tid+1)%gsize)],10));
ensures  (\forall* int tid;0<=tid&&tid<gsize;Perm(this.b[tid],100));
void main_resources_of_1(int tcount,int gsize,int gid){}
```

```
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 8: Tool input for the running example.

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 knowledge about unfenced 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

```
kernel binomial {
  global int[gsize] bin;
  local  int[gsize] tmp;

  requires gsize > 1 * perm(bin[tid],100) * perm(tmp[tid],100);
  ensures  perm(bin[tid],100) * bin[tid]=binom(gsize-1,tid);
  void main(){
    int temp;
    int N:=1;
    bin[tid]:=1;
    invariant perm(ar[tid],100) * perm(tmp[tid],100);
    invariant tid<N ? ar[tid]=binom(N,tid) : ar[tid]=1;
    while(N<gsize-1){
      tmp[tid]:=ar[tid];
      barrier(1,{local}){
        ensures perm(ar[tid],100) * perm(tmp[(tid-1) mod gsize],10);
        ensures 0<tid & tid<=N -> tmp[(tid-1) mod gsize]=binom(N,tid-1);
      }
      N := N+1;
      if(0<tid & tid<N){
        temp:=tmp[(tid-1) mod gsize];
        ar[tid]:=temp+ar[tid];
      }
      barrier(2,{}){
        ensures perm(ar[tid],100) * perm(tmp[tid],100);
      }
    }
  }
}
```

Figure 9: Kernel program for binomial coefficients

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

Example 4. *Figure 8 gives the running example in the PVL language used by the tool. All other specifications are generated by the tool.*

6. 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. 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.

7. Related Work

There already exists some work on the verification of GPU kernels. However, these approaches mainly focus on the verification of data race freedom of the interleaving of two arbitrary threads, whereas we verify an arbitrary single thread, and also consider functional correctness.

Guodong and Gopalakrishnan [18] 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 access, and where in the code they occur. By analysing this array, they detect possible data races.

Betts et al. [14] verify GPU programs based on a novel operational semantics called *synchronous, delayed visibility*, which tracks reads and writes in shadow memory, and synchronises this when reaching a barrier. The changes to shadow memory are then used to identify possible data races. This semantics is encoded in BoogiePL.

The main synchronization mechanism in GPGPU programs are barriers. We tailored the approach of Hobor et al. [9] for Pthreads-style barriers to OpenCL barriers. Since OpenCL barriers are simpler, our specifications also are much

simpler. For each barrier it is sufficient if we specify how permissions are redistributed over threads, with associated functional properties. In contrast, Hobor et al. need a complete state machine to specify the barrier behaviour.

8. Conclusions and Future Work

This paper 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.* [19]). 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 [7]. 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 [20]) 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.

Acknowledgements. We are very grateful to Christian Haack, who helped clarifying many of the formal details of the logic. We acknowledge support by the EU STREP project 287767 CARP (Huisman, Mihelcic), and the ERC project 258405 VerCors (Blom, Huisman).

References

[1] E. K. Jason Sanders, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.
[2] OpenCL, The opencl 1.2 specification, 2011.
[3] B. Cowan, B. Kapralos, GPU-based acoustical occlusion modeling with acoustical texture maps, in: AM '11, ACM, 2011, pp. 55–61.

[4] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, Z.-P. Liang, B. P. Sutton, Accelerating advanced MRI reconstructions on GPUs, in: CF '08, ACM, 2008, pp. 261–272.

[5] J. B. Mulligan, A GPU-accelerated software eye tracking system, in: ETRA '12, ACM, 2012, pp. 265–268.

[6] R. Bornat, C. Calcagno, P. O'Hearn, M. Parkinson, Permission accounting in separation logic, in: J. Palsberg, M. Abadi (Eds.), POPL, ACM, 2005, pp. 259–270.

[7] C. Haack, C. Hurlin, Separation logic contracts for a Java-like language with fork/join, in: J. Meseguer, G. Rosu (Eds.), Algebraic Methodology and Software Technology, volume 5140 of *LNCS*, Springer-Verlag, 2008, pp. 199–215.

[8] C. Haack, M. Huisman, C. Hurlin, Reasoning about Java's reentrant locks, in: G. Ramalingam (Ed.), Asian Programming Languages and Systems Symposium, volume 5356 of *LNCS*, Springer-Verlag, 2008, pp. 171–187.

[9] A. Hobor, C. Gherghina, Barriers in concurrent separation logic, in: 20th European Symposium of Programming (ESOP 2011), LNCS, Springer-Verlag, 2011, pp. 276–296.

[10] J. Reynolds, Separation logic: A logic for shared mutable data structures, in: Logic in Computer Science, IEEE Computer Society, 2002, pp. 55–74. doi:10.1109/LICS.2002.1029817.

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

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

[13] J. Boyland, Checking interference with fractional permissions, in: R. Cousot (Ed.), Static Analysis Symposium, volume 2694 of *LNCS*, Springer-Verlag, 2003, pp. 55–72.

[14] A. Betts, N. Chong, A. Donaldson, S. Qadeer, P. Thomson, GPUVerify: a verifier for GPU kernels, in: OOPSLA'12, ACM, 2012, pp. 113–132.

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

[16] K. Leino, P. Müller, J. Smans, Verification of concurrent programs with Chalice, in: Lecture notes of FOSAD, volume 5705 of *LNCS*, Springer-Verlag, 2009, pp. 195–222.

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

[18] G. Li, G. Gopalakrishnan, Scalable SMT-based verification of GPU kernel functions, in: SIGSOFT FSE 2010, Santa Fe, NM, USA, ACM, 2010, pp. 187–196.

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

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