Specification and Verification of GPGPU Programs using Permission-Based Separation Logic*

Marieke Huisman and Matej Mihelčić

University of Twente, The Netherlands,
{m.huisman,m.mihelcic}@utwente.nl

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 our ideas how to verify GPU programs written in OpenCL, a platform-independent low-level programming language. Our verification approach is modular, based on permission-based separation logic. We first present the main ingredients of our logic, and then illustrate its use on several example kernels. We show in particular how the logic is used to prove data-race-freedom and functional correctness of GPU applications.

1 Introduction

Graphics processing units (GPUs) have initially been designed to support computer graphics. Their specific architecture allows rapid memory manipulation, provides high processing power and massive parallelism, which allows for efficient solving of typical graphics-related tasks. However, such architectures are also suitable for many other programming tasks, leading to the development of the area of GPGPU (General Purpose GPU) programming. Until 2006. GPGPU programming was mainly done in CUDA [9], a proprietary GPU programming language from NVIDIA. However, recently a new platform-independent, low-level programming language standard for GPGPU programming, OpenCL [10], has been developed. As a result, GPUs are now used in many different fields, including media processing [5], medical imaging [15] and eye-tracking [12].

Despite the platform-independence, writing OpenCL programs still requires working on a relatively low level, and optimizing applications for the actual type of device used. Given the importance, range and increasing complexity of GPGPU applications today, formal techniques to reason about correctness of GPGPU programs are thus necessary. This paper presents a verification technique for GPGPU programs based on permission-based separation logic.

Before discussing our verification approach, we first briefly discuss the main characteristics of the GPU architecture (for more details, we refer to the OpenCL

* This work is supported by the EU FP7 STREP project CARP (project nr. 287767).

specification [10]). 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 running on a CPU. Threads are logically grouped into work groups and have local and global identifiers, where global identifier are computed from the local identifier and the work group characteristics. The main means of synchronisation within a kernel are *barriers*: a thread blocks at the barrier until all other threads have also reached this barrier. The memory hierarchy is quite complex. Each thread has access to four distinct memory regions: *global*, *local*, *constant* and *private* memory. Private memory is local to a single thread, local memory is shared between different threads in a work group, and global memory is accessible by all threads in a kernel, and by the host program. Constant memory is a dedicated part of global memory, allocated and initialized by the host, remaining constant throughout the execution of a kernel.

As mentioned above, this paper discusses how we verify GPGPU kernels. Our main inspiration is the use of permission-based separation logic to reason about multithreaded programs [3, 6]. Key ingredient is to annotate a program with read and write permissions. A location can only be accessed or updated if a thread holds the appropriate permission for this location. Program annotations are *framed* by permissions: a functional property can only be specified and verified if a thread holds the appropriate permissions. Permissions can be split and combined, to make them change between read and write permissions. 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 threads. Every time a barrier is reached, a barrier specification specifies how the permissions are redistributed over the threads (similar to the barrier specifications of Hobor et al. [8]). The barrier specification also specifies functional pre- and postconditions for the barrier. Essentially this specifies how knowledge about the global state upon reaching the barrier is spread over the different threads.

The remainder of this paper is organised as follows. Section 2 presents more details about our verification approach; Section 3 presents several verification examples, and Section 4 discusses conclusions, related work and future work.

2 Permission-based Separation Logic for GPGPU Kernels

This section describes our approach to verify GPGPU kernels. We first give a short overview how permission-based separation logic is used to reason about concurrent programs, and then show how we adapt it to OpenCL kernels.

2.1 Permission-based Separation Logic

Separation logic [14] is originally developed as an extension of Hoare logic [7] to enable reasoning about programs with pointers, as it allows to reason explicitly about the heap. Soon it was realised that this setting was also convenient to reason modularly about concurrent programs [13]: if two threads work on disjoint parts of the heap, they do not interfere with each other.

However, classical separation logic is not permissive enough to reason about concurrent programs: it requires use of mutual exclusion mechanisms for all shared locations, and it forbids simultaneous reads to shared locations. To overcome this, Bornat et al. [3] extended separation logic with permissions. Permissions, initially introduced by Boyland [4], contain numerical fractions denoting 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 permission can be joined into a write permission.

Assertions in separation logic are expressed as first order logic formulas, extended with three special operators: the points-to predicate, combined with a permission, the separating conjunction ($*$) and the separating implication (or magic wand, $-\!\!*$). The syntax of formulas F is formally defined as follows:

$$lop \in \{*, -\!\!*, \&, |\} \qquad qt \in \{\exists, \forall\} \qquad \pi \in (0, 1]$$
$$F \ ::= \ e \ \mid \ \texttt{PointsTo}(x, \pi, v) \ \mid \ F \ lop \ F \ \mid \ \texttt{(}qt \ T \ \alpha\texttt{)(}F\texttt{)}$$

In classical Hoare logic, assertions are properties over the state. In separation logic, the state is explicitly divided over different heaps (mappings of memory locations to values). Intuitively, an assertion $\texttt{PointsTo}(x, \pi, v)$ (or $\texttt{x} \overset{\pi}{\mapsto} v$ in traditional notation) holds for a thread t if the variable x points to a location that contains the value v, and in addition, the thread t has permission π to access this location. For convenience, we sometimes use $\texttt{Perm}(x, \pi)$ to abbreviate $\exists v.\texttt{PointsTo}(x, \pi, v)$. We denote expressions with e. Expressions are built from variables, values, logical and arithmetical operators applied to variables and values. 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. We use $\mathbin{\raisebox{0.2ex}{$*$}}_{v \in V} F(v)$ as the universal separating conjunction quantifier. A formula $\phi_1 -\!\!* \phi_2$ holds for any heap that has the following property: if the heap is extended with a *disjoint* heap that satisfies ϕ_1, then the combined heap satisfies ϕ_2. The separating implication is sometimes also read as a *trade* operation: the resources specified by ϕ_1 are exchanged for the resources specified by ϕ_2.

2.2 Verification of GPGPU Kernels

To adapt permission-based separation logic for kernels, we need to distinguish explicitly between properties over global and local memory. Thus, every formula over permissions can be separated into formulas over global and local memory. Given a formula over permissions F, we use $F_{|glob}$ and $F_{|loc}$ to denote the subformulae with the permissions over global and local memory, respectively. With

our logic we are able to prove that a kernel does not have data races, and that it respects certain functional correctness properties.

Intended kernel behaviour is specified by the following constructs:

- The *kernel specification* is a triple $(K_{res}, K_{pre}, K_{post})$. Formula K_{res} specifies all resources in global memory that are passed from the host program to the kernel. Further, the kernel specification specifies functional properties as pre- and postcondition, K_{pre} and K_{post}, respectively. A kernel can only be invoked by a host program that transfers the necessary resources and respects the preconditions. Notice that locations defined in the local memory space of a kernel are only valid inside the kernel body and thus the kernel always holds write permissions for these locations.
- Permissions and conditions in the kernel are distributed over the kernel's threads by the *thread specification* $(T_{res}, T_{pre}, T_{post})$. The thread's resource specification T_{res} specifies the resources over global and local memory assigned to the thread. The thread's pre- and postcondition (T_{pre} and T_{post}) specify further properties of these resources. The formulas in the thread specification are expressed in terms of the thread identifier.
- A *barrier specification* $(B_{res}, B_{pre}, B_{post})$ specifies resources, and a pre- and postcondition for each barrier in the kernel. The resources B_{res} specify how permissions are redistributed over the threads (depending on the kind of barrier, these can be only permissions on local memory, only permissions on global memory, 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. 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}, *i.e.*, it may only state something about locations in B_{res}.

Notice that it is sufficient to specify a single permission formula for a kernel; these are the permissions required and returned by the kernel. Within a kernel, the only way to redistribute permissions is at a barrier, the code between barriers holds the same set of permissions at all times. If we only check for data races, the kernel and thread postconditions will typically be true.

Given a fully annotated kernel with body K_{body}, a set of global thread identifiers *Tid*, a set of local thread identifiers *LTid*, and a set of local memory locations *Local*, verification of the kernel behaviour essentially boils down to verification of the following properties.

- The Hoare triple $\{T_{res} * T_{pre}\}\ K_{body}\{T_{post}\}$ is proven correct using standard rules for permission-based separation logic (as in [6]). Each barrier is verified as a method call with precondition B_{pre} and postcondition $B_{res} * B_{post}$[1].
- The kernel resources are shown to be sufficient for the thread specification if the global memory resources for the thread are passed to the kernel, the kernel's precondition implies the thread's precondition, and no two threads

[1] Notice that the properties for locations described in B_{post} can only be used if there is no thread that holds a write permission for those locations.

have write access to the same location (either in global or local memory). This is expressed by the following two formulas:

$$K_{res} * K_{pre} \mathrel{-\!\!*} \mathop{*}_{tid \in Tid} (T_{res|glob} * T_{pre})$$
$$\mathop{*}_{v \in Local} \texttt{Perm}(v, 1) \mathrel{-\!\!*} \mathop{*}_{ltid \in LTid} T_{res|loc}$$

– For each barrier with a memory fence on global memory, we show that it redistributes only the permissions that are available in the kernel, and it does not duplicate write permissions.

$$K_{res} \mathrel{-\!\!*} \mathop{*}_{tid \in Tid} B_{res|glob}$$

– We do a similar check for each barrier with a local memory fence. If there is no control flow divergence between work groups, it is sufficient to show this for an arbitrary work group.

$$\mathop{*}_{v \in Local} \texttt{Perm}(v, 1) \mathrel{-\!\!*} \mathop{*}_{ltid \in LTid} B_{res|loc}$$

– For each barrier with a global memory fence, we show that its postcondition follows from the precondition (over all threads).

$$\mathop{*}_{tid \in Tid} B_{pre} \mathrel{-\!\!*} \mathop{*}_{tid \in Tid} B_{post}|_{RGPerm(tid)}$$

where $B_{post}|_{RGPerm(tid)}$ restricts the formula B_{post} to the set of locations in global memory that can be read by thread *tid*.

– Similarly, for each barrier with a local memory fence, we show that its postcondition follows from the precondition (over all threads).

$$\mathop{*}_{ltid \in LTid} B_{pre} \mathrel{-\!\!*} \mathop{*}_{ltid \in LTid} B_{post}|_{RLPerm(ltid)}$$

where $B_{post}|_{RLPerm(ltid)}$ restricts the formula B_{post} to the set of locations in local memory that can be read by thread *ltid*.

– Finally we show that the universal quantification of all threads' postconditions imply the kernel's postcondition.

$$\mathop{*}_{tid \in Tid} T_{post} \mathrel{-\!\!*} K_{post}$$

Finally, when verifying the host code, it is verified that sufficient permissions are given to the kernel, and that the precondition is established. Thus, kernels with an incorrect resource specification, *e.g.*, requiring write permissions on the same location twice, cannot be invoked, because the call from the host to the kernel cannot be verified.

3 Examples

This section discusses several example kernels to illustrate how they are verified using our approach. For convenience we use the following shorthands in

Kernel spec: $(\ast_{i\in[0\ldots size-1]} \texttt{Perm}(a[i],1) \ast \texttt{Perm}(b[i],1), size = n \wedge num\_threads = n, \texttt{true})$
Thread spec: $(\texttt{Perm}(a[tid],1) \ast \texttt{Perm}(b[tid],1), \texttt{true}, \texttt{true})$

```
__kernel void example(__global int *a, __global int *b) {
    int tid = get_global_id(0);
    a[tid]=tid;
    b[tid] = a[(tid+1)%size]; }
```

Fig. 1. Fields access with insufficient permissions

```
__kernel void example(__global int *a, __global int *b) {
    int tid = get_global_id(0);
    a[tid]=tid;
    barrier(CLK_GLOBAL_MEM_FENCE); //B
    b[tid] = a[(tid+1)%size]; }
```

Fig. 2. Corrected kernel code with barrier

specifications: *tid* denotes the result of the function call `get_global_id(0)`, and *num_threads* denotes the the number of threads executing the kernel. This information is defined in the host code before executing the kernel.

First we start with some examples where we only consider data race properties. Consider the simple example kernel in Figure 1. It is obvious to see that this kernel has a data race, because for example the thread with id 1 could be writing its id in location $a[1]$, at the same time as the thread with id 0 is reading this location. This problem is detected when verifying the kernel body: the assignment $b[tid] = a[(tid+1)\%size]$ cannot be verified because the thread does not have sufficient permissions to access location $a[(tid+1)\%size]$. Suppose we try to solve this by changing the thread specification as follows:

$$(\texttt{Perm}(a[tid],1) \ast \texttt{Perm}(a[(tid+1)\%size],\frac{1}{2}) \ast \texttt{Perm}(b[tid],1), \texttt{true}, \texttt{true})$$

With this specification, the kernel body itself can be verified. However the condition $K_{res} \mathrel{-\!\!\ast} \ast_{tid\in Tid}\ T_{res|glob}$ cannot be verified, because together all threads running the kernel require more permissions on the elements of `a` than are available in the kernel. Instead, the correct way to solve this is to insert a global memory fence barrier, as in Figure 2. Of course, this also requires a barrier specification. With the following barrier specification for B, the kernel (as specified in Figure 1) can indeed be verified.

Barrier spec(B) : $(\texttt{Perm}(a[(tid+1)\%size],\pi) \ast \texttt{Perm}(b[tid],1), \texttt{true}, \texttt{true})$

However, for a different barrier specification, verification might fail. Consider for example the following barrier specification, which redistributes more resources than the kernel possesses, *i.e.*, it violates the condition $K_{res} \mathrel{-\!\!\ast} \ast_{tid\in Tid} B_{res|glob}$.

Barrier spec(B) : $(\texttt{Perm}(a[tid],1) \ast \texttt{Perm}(a[(tid+1)\%size],\frac{1}{2}) \ast \texttt{Perm}(b[tid],1), \texttt{true}, \texttt{true})$

Finally, to illustrate verification of functional correctness properties, consider again the kernel in Figure 2. For this kernel, we can show for example that the following specifications are respected:

Kernel spec: $(\ast_{i\in[0\ldots size-1]}\ \texttt{Perm}(a[i],1)\ast\texttt{Perm}(b[i],1),$
$size = n \wedge num\_threads = n, \forall_{i\in[0\ldots size-1]} b[i] = (i+1)\%size)$
Thread spec: $(\texttt{Perm}(a[tid],1), \texttt{true}, b[tid] = (tid+1)\%size)$
Barrier spec(B) : $(\texttt{Perm}(a[(tid+1)\%size], \frac{1}{2})\ast\texttt{Perm}(b[tid],1),$
$a[tid] = tid, a[(tid+1)\%size] = (tid+1)\%size)$

4 Conclusions, Related 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 we specify all permissions that are necessary to execute the kernel, (ii) the permissions in the kernel are distributed over the threads, and (iii) at each barrier the permissions are redistributed over the threads. Verification of individual threads uses standard program verification techniques, while additional verification conditions check consistency of the specifications.

Related Work There already exists some work on the verification of GPU kernels. However, their focus is on the verification of the interleaving of two arbitrary threads, whereas we verify an arbitrary single thread. We believe this makes our approach more suitable to verify also functional correctness properties.

Guodong and Gopalakrishnan [11] verify CUDA programs by symbolically encoding thread interleavings. They were the first to observe that 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 can detect possible data races.

Betts et al. [2] verify GPU programs based on a novel operational semantics called *synchronous, delayed visibility*. They log writes and reads to shared variables made by two arbitrary threads and assert at each barrier that there has not been any data races between those reads and writes. The two threads, with assertions are encoded into BoogiePL, and then verified using standard verification condition generation.

The main synchronization mechanism in GPGPU programs are barriers. We tailored the approach of Hobor et al. [8] 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 constrast, Hobor et al. need a complete state machine to specify the barrier behaviour.

Future Work We are currently at the first steps of our project, and there is still much work to be done. First of all, a detailed formalisation of the logic and its soundness proof has to be worked out. To support the logic, we will

develop tool support as an extension of the VerCors tool [1], a tool to reason about multithreaded Java programs using permission-based separation logic. To further enhance tool support, we will study automatic generation of permission specifications, which would alleviate the need for user-written specifications.

The main focus of our work on the verification of GPGPU programs will be the verification of functional properties. In this paper, we have illustrated this on an example, but we need to study more examples. We will also look at other typical GPGPU correctness properties, such as the absence of barrier divergence[2] and study how these can be verified in our approach. Finally, we will investigate how to reason about the host program. This will allow verification of multi-kernel applications running in a heterogeneous setting.

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

References

1. A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski, *The VerCors project: Setting up basecamp*, in PLPV 2012, 2012, pp. 71–82.
2. 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.
3. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, *Permission accounting in separation logic*, in POPL '05, ACM, 2005, pp. 259–270.
4. J. Boyland, *Checking interference with fractional permissions*, in SAS'03, Springer-Verlag, 2003, pp. 55–72.
5. B. Cowan and B. Kapralos, *GPU-based acoustical occlusion modeling with acoustical texture maps*, in AM '11, ACM, 2011, pp. 55–61.
6. C. Haack, M. Huisman, and C. Hurlin, *Reasoning about Java's reentrant locks*, in APLAS'09, G. Ramalingam, ed., vol. 5356 of LNCS, Springer, 2008, pp. 171–187.
7. C. A. R. Hoare, *An axiomatic basis for computer programming*, Commun. ACM, 26 (1983), pp. 53–56.
8. A. Hobor and C. Gherghina, *Barriers in concurrent separation logic*, in ESOP 2011, Springer, 2011, pp. 276–296.
9. E. K. Jason Sanders, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 2010.
10. Khronos Group, *The OpenCL 1.2 Specification*, 2011.
11. G. Li and G. Gopalakrishnan, *Scalable SMT-based verification of GPU kernel functions*, in SIGSOFT FSE 2010, Santa Fe, NM, USA, ACM, 2010, pp. 187–196.
12. J. B. Mulligan, *A GPU-accelerated software eye tracking system*, in ETRA '12, ACM, 2012, pp. 265–268.
13. P. W. O'Hearn, *Resources, concurrency, and local reasoning*, Theoretical Computer Science, 375 (2007), pp. 271–307.
14. J. Reynolds, *Separation logic: A logic for shared mutable data structures*, in Logic in Computer Science, IEEE Computer Society, 2002, pp. 55–74.
15. S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, Z.-P. Liang, and B. P. Sutton, *Accelerating advanced MRI reconstructions on GPUs*, in CF '08, ACM, 2008, pp. 261–272.

[2] All threads in the same work group must reach the same barrier, otherwise the program behaviour is undefined.