A certification-oriented OpenCL subset

Jean-Louis Dufour

Safran Electronics & Defense
October 4, 2021

Abstract

Manycores are emerging in embedded systems, but only in non-critical usages. Indeed, the correctness justifications required by a certification process are beyond the reach of industrial practice, and at the frontier of the academic state of the Art. We focus on a subset of the problem, the correctness of the parallel code. We emphasize that it has nothing to do with the corresponding problem on multicores, and propose a justification strategy based on a subset of the OpenCL language and a simple formal specification and proof.

Keywords: Manycore, GPU, OpenCL, safety, certification, deterministic execution, formal method

1 Introduction

Manycores (i.e. mainly GPUs programmed in CUDA or OpenCL) are emerging in embedded systems, but only in non-critical usages. In safety-critical usages, of course they will not ensure alone safety: they will be just a part of a redundant and diverse architecture. But this will not suppress the expection for correctness justification: it will only be lower. Indeed, the justifications required by a certification process are beyond the reach of industrial practice, and at the frontier of the academic state of the Art. The problem can be decomposed as follows:

Hardware is a complex black box, where critical mechanisms like the network-on-chip linking cores to memory banks must be guessed from the patents [1]; certification-friendly suppliers are a rare exception [4].

Software / Compiler-and-Scheduler is also a complex black box, with very few justifications of the compliance w.r.t. the OpenCL specification (beyond passing the OpenCL conformance test suite; from now on, we focus on OpenCL [9]),

Software / Application (a.k.a. *(compute) kernel*) consists of tens, hundreds or thousands of lightweight threads sharing two levels of memory ('local' and 'global' in OpenCL). This kind of software is the favorite playground of *heisenbugs*, hence test-based methodologies are notoriously ineffective, whether to debug or to verify.

These three aspects are equally important, but the first two are more 'industrial/business' than technical, because of the competitive nature of the main market of GPUs : video games. We focus here on the third aspect, which is mainly technical, but has also a non-negligeable 'industrial/human' facet.

Our proposal can be simply summed up :

1. we demonstrate that the kernel is *deterministic* : it performs betwen inputs and outputs a transfer *function* and not a transfer *relation*. Due to the astronomical number of possible interleavings between the threads (the 'work-items' in OpenCL), the only way to do this is to do it *formally*. The state of the Art doesn't allow to automate this in the general case, but we claim here that a special case can be defined, whose semi-automation is *industrially* achievable. This 'industrial/human' aspect is a key point, and relates both to the engineers but also to the certifiers: the mastery of a formal language is not needed, and as often as possible, only simple annotations are to be written. It is the subject of this paper.

2. we then demonstrate that the (parallel) kernel has an equivalent sequential version : thanks to the previous property, it can convincingly be done by test. This is not covered by this paper.

3. finally, it only remains to validate the sequential version functionally: business as usual (and usually by test); again not covered by this paper.

Heisenbugs are detected by the first step, and after only 'standard' bugs remain. A formal method is mandatory only for this first step, and we will do our best to not use it for more than that : functional correctness is not in its scope, and even low-level correctness (like array indexes in their range) as such is not in its scope (but see at the end).

We cheat a little bit when we talk about an 'OpenCL subset' for which semi-automation is possible. This subset is

1. a syntactic subset of the OpenCL language (device side),
2. a subset of the possible codes written in the former syntactic subset : let's call them the *'almost-embarrassingly-parallel' codes*, this will be explained later.

The usefulness of this subset will be demonstrated on a 'real-life' set of OpenCL kernels : the OpenCV library. In particular the process will be illustrated on a representative kernel: the histogram equalization function 'equalizeHist'.

Lastly, this process raises an interesting question: will many-core be the Trojan horse of formal methods to penetrate the impassable enclosure of safety-critical software development ?

2 Multicores and manycores pose different certification problems

Multicores and manycores look the same from an hardware point of view : cores sharing memories. Therefore both undergo the same problems w.r.t. critical applications:

- variability in access times to shared memory, which induces variability in execution times.
- race conditions: [10]

> when two or more threads access a common resource, e.g., a variable in shared memory, and the order of the accesses depends on the timing, i.e., the progress of individual threads

For multicores the timing variability is significant [6], even with long pipelines and out-of-order execution. But with only slight exaggeration, this is their only problem with regard to certification.

For manycores, this variability is also a active research subject [7], but it may be less important in practice, because the hardware architecture is completely 'data-oriented', and especially because the first implementations will be careful not to execute different kernels in parallel. But let's say it is as important: it's still not the most feared phenomenon. The problem comes from the particular kind of supported algorithm, which induces a particular kind of sharing between threads :

- on a multicore, *in an embedded use*, the design aims to safely assess the timing variability, for example with a synchronous approach. In this case, each task has exclusivity on its own data, and data exchanged between tasks are carefully read and written at the start and end of the tasks, in such a way to avoid simultaneous accesses. In other words the few data sharing which occurs is under time-control, and there is mainly a non-functional bus/memory sharing: there is no fundamental difficulty in obtaining a deterministic functional behavior.
- on a manycore, hundreds of work-items read and write simultaneously the same data arrays. Work-items share not only organs, but also data, leading if we are not careful to race conditions. This is a potential source of non-determinism, which is usually a show-stopper for certification.

To summarize, race conditions and non-determinism are not a problem for embedded multicores, but are THE problem for manycores (embedded or not).

3 The OpenCL subset

Due to the complexity of the possible interleavings between the work-items, the elimination of race conditions is a challenging aspect of parallel programming : a standard name for this is *'Data Race Freedom'* ('DRF'; here a data race will be defined as a race condition on the simplest object : the memory cell). The DRF property is an active research topic of parallel programming, and several tools have been developped for tracking race conditions on manycores (among other things), among them PUG [8], GPUVerify [2], VerCors [3].

Now, the real property we are looking for is not DRF but *determinism* : every possible execution of the kernel gives the same outputs (starting from given inputs). Determinism is also a hot subject for parallel

programming [5], both properties are related but not in an obvious way. We will set a stronger objective which we call the *'almost-embarrassingly-parallel'* property, which implies both DRF and determinism.

To define it, we must first recall what is a *barrier interval* : it is the kernel code between two consecutive barriers. The start and the end of a kernel are implicit barriers, so a barrier-free kernel has a single barrier interval (which is the kernel itself). For this notion of 'consecutive barrier' to be meaningful, we have to restrict the placement of barriers : typically, a barrier will be forbidden in a conditional statement. The chosen restriction will also ensure statically that *barrier divergence* will not occur: a kernel can now be seen as a predictable sequence of barrier intervals, the same for all work-items.

We restrict OpenCL not only syntactically, but also semantically: *each barrier interval must be embarrassingly parallel*. We formalize this fuzzy notion in the following way : consider any barrier interval, then any pair of work-items will work on disjoint subsets of each shared array. These partitions of the shared arrays will vary from barrier interval to barrier interval, that's what makes the difference between *'almost-embarrassingly'* and *'embarrassingly'*.

We don't even try to infer these disjoint subsets : they are the rationale for the design, so they have to be explicitely stated by the designer. They are in fact a simplified version of the 'separation logic' used in VerCors [3]. Typically, for each shared array, the subset is an interval parameterized by the work-item id.

They give rise to two kind of proof obligations (for any barrier interval):

disjointness for any pair of work-items, for any shared array, the subsets are disjoint,

correctness for any work-item, for any shared array, for any access to this array, the access is in the corresponding subset.

Let's state the restrictions (the first three define the syntactic subset, the fourth is the semantic restriction):

1. the kernel execution involves a unique work-group,
2. the only synchronization mechanism is the barrier (no atomics),
3. barriers occur either at toplevel in the kernel, or at toplevel in a toplevel 'for' loop (containing neither 'break' nor 'continue') whose iteration values (start, stop, step) depend only on the scalar inputs of the kernel (not on the arrays, not on the work-item ids),
4. each barrier interval is embarrassingly parallel.

4 Issues and discussion

There are three main issues: on the principle itself, on the subset and on the proof obligations.

The fact that *'almost-embarrassingly-parallel'* implies *DRF* seems (at least to us) obvious, but the implication towards *deterministic* is not so obvious, and the informal justification we will present would have been advantageously replaced by a more formal proof.

The subset is very restrictive, that's why we are testing it on the OpenCV kernels.

The two kinds of proof obligations differ in terms of complexity:

- the disjointness needs few context and is within the reach of the best SMT-solvers. Of course, in absolute terms the problem is undecidable, that's why a 5th optional restriction is that this proof obligation belongs to Presburger arithmetic.
- the correctness needs more context and, as its name implies, is in fact the typical proof obligation associated with an assertion in a Hoare-logic framework like Frama-C. It is a bit harder than proving indexing correctness (alluded to in the introduction), because the subset of indexes is strictly smaller than the full range of the array. In particular, if the access is located after a loop, a loop invariant may be necessary.

5 Related works

As already mentioned, this work is strongly inspired by PUG [8], GPUVerify [2] and VerCors [3]. The motivation is to make these technologies accessible to engineers. For this, it is necessary to significantly reduce the complexity, hence the new concept of *'almost-embarrassingly-parallel'*.

6 Conclusions

The three issues mentioned constitute the work plan for the next few months.

References

[1] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture*, 13(2):1–140, 2018.

[2] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: a verifier for gpu kernels. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 113–132, 2012.

[3] Stefan Blom and Marieke Huisman. The vercors tool for verification of concurrent programs. In *International Symposium on Formal Methods*, pages 127–131. Springer, 2014.

[4] Marc Boyer, Benoît Dupont de Dinechin, Amaury Graillat, and Lionel Havet. Computing routes and delay bounds for the network-on-chip of the kalray mppa2 processor. In *ERTS 2018-9th European Congress on Embedded Real Time Software and Systems*, 2018.

[5] Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 3–12, 2009.

[6] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. Predictability considerations in the design of multi-core embedded systems. *Proceedings of Embedded Real Time Software and Systems*, 36:42, 2010.

[7] Maximilien Dupont de Dinechin, Matheus Schuh, Matthieu Moy, and Claire Maiza. Scaling up the memory interference analysis for hard real-time many-core systems. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 330–333. IEEE, 2020.

[8] Guodong Li and Ganesh Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, 2010.

[9] Aaftab Munshi. The opencl specification version: 1.2 document revision: 15. *Khronos Group*, 2011.

[10] David Padua. *Encyclopedia of parallel computing*. Springer Science & Business Media, 2011.