KernelInterceptor: automating GPU kernel verification by intercepting kernels and their parameters

Ethel Bardsley
Imperial College London

Alastair F. Donaldson
Imperial College London

John Wickerson
Imperial College London

ABSTRACT

GPUVerify is a static analysis tool for verifying that GPU kernels are free from data races and barrier divergence. It is intended as an automatic tool, but its usability is impaired by the fact that the user must explicitly supply the kernel source code, the number of work items and work groups, and preconditions on key kernel arguments. Extracting this information from non-trivial OpenCL applications is laborious and error-prone.

We describe an extension to GPUVerify, called KernelInterceptor, that automates the extraction of this information from a given OpenCL application. After recompiling the application having included an additional header file, and linking with an additional library, KernelInterceptor is able to detect each dynamic kernel launch and record the values of the various parameters in a series of log files. GPUVerify can then be invoked to examine these log files and verify each kernel instance. We explain how the interception mechanism works, and comment on the extent to which it improves the usability of GPUVerify.

1. INTRODUCTION

GPUVerify is a tool for verifying that GPU kernels, written in either CUDA[1] or OpenCL,[2] are free from data races and barrier divergence [2]. The analysis is performed *statically*; that is, GPUVerify does not actually run the kernel, but merely examines its source code. GPUVerify is useful for discovering defects in kernels, but can also go further than any testing tool can: it is able to certify that a given kernel is free from these classes of defect under *any* execution schedule. GPUVerify has already proved itself to be of practical use when applied to non-trivial OpenCL and CUDA kernels [2]. For instance, it is able to verify, without user intervention, 49 of the 70 kernels in the AMD Accelerated Parallel Processing SDK (version 2.6).[3]

GPUVerify is intended as a completely-automatic tool, requiring minimal expertise and minimal effort from its users. However, assembling all of the necessary inputs to GPUVerify is a significant manual effort. The user must examine the source code of their application, and supply to GPUVerify:

- the source code of each kernel,
- the precise number of work items and work groups that will execute each kernel,
- constraints on the values of selected kernel arguments (where necessary for kernel correctness), and
- barrier invariants [3] and loop invariants (where necessary for successful verification).

In this paper we describe an extension to GPUVerify, called KernelInterceptor, that automates the extraction of the first three items above from a given OpenCL application. The fourth item, invariant discovery, remains a challenging research topic, as discussed in Section 4. Nevertheless, KernelInterceptor marks a significant step toward fully automated verification of GPU kernels.

KernelInterceptor is used as follows.

1. **The user prepares an application for interception.** Small modifications must be made to the source code and build process of the OpenCL application to be analysed.
2. **The user executes the application.** As the application executes, KernelInterceptor intercepts each kernel launch and records the kernel's source code and the parameters passed.
3. **The user executes GPUVerify.** GPUVerify presents a list of intercepted kernels. The user can then ask GPUVerify to try to verify all or some of these kernels.

In the remainder of this paper, we describe how KernelInterceptor is used (Section 2) and how it is implemented (Section 3). Section 4 evaluates KernelInterceptor's limitations and the extent to which it improves the usability of GPUVerify, and also discusses related and future work.

[1] `http://www.nvidia.com/object/cuda_home_new.html`

[2] `http://www.khronos.org/opencl/`

[3] `http://developer.amd.com/sdks/amdappsdk`

IWOCL '14 12–13 May, Bristol, UK

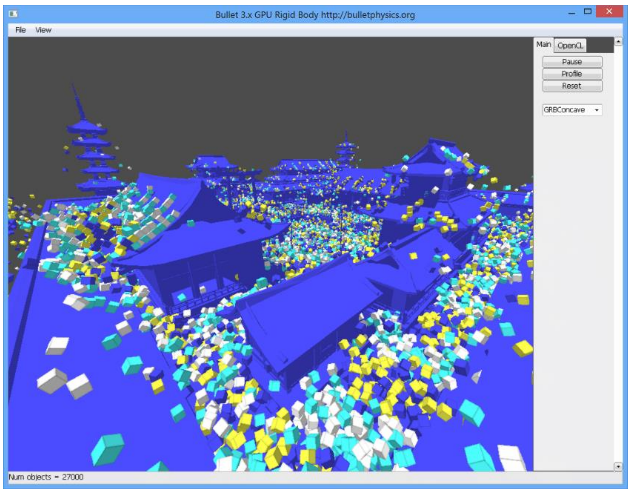

Figure 1: The Bullet rigid body simulator in action, simulating hundreds of thousands of bodies and their collisions, all in real-time. Picture credit: Erwin Coumans [6].

2. USAGE

This section explains how KernelInterceptor works from the user's perspective. As a running example, we use an OpenCL application that simulates collisions of rigid bodies [6]. This application is part of the open source Bullet Physics library (version 3)[4] and the code is available online.[5] The capabilities of the simulator are demonstrated in Fig. 1.

2.1 Instrumenting the source code

To use KernelInterceptor, the user must first download GPUVerify, with which the KernelInterceptor header file (`cl_interceptor.h`) and library (`cl_interceptor.cpp`) are shipped.

The line

```
#include "/path/to/cl_interceptor.h"
```

must be added to each `.cpp` file that includes the OpenCL headers (`cl.h` or `opencl.h`). In the case of the Bullet simulator, the only relevant file is `b3OpenCLInclude.h`.

The user must modify their build process so that it compiles `cl_interceptor.cpp` and links it against their application. In the case of the Bullet simulator, it suffices to add `cl_interceptor.o` as a build target in the relevant makefiles.

The application can now be built and run as normal. The interception process records entire kernel texts and writes them to disk on every kernel invocation, which may incur nontrivial runtime overhead. We therefore recommend enabling KernelInterceptor only as part of a debug build.

2.2 Inspecting the intercepted kernels

The user can view information about the intercepted kernels using the command

```
gpuverify --show-intercepted.
```

```
[0] Name: AddOffsetKernel
    File: .gpuverify/AddOffsetKernel001.cl
    local_size=128 global_size=12544
    args=0x7f4b000008000000006300006271
    Built at b3OpenCLUtils.cpp:880
    Run at b3LauncherCL.h:117

[1] Name: AddOffsetKernel
    File: .gpuverify/AddOffsetKernel002.cl
    local_size=128 global_size=12544
    args=0x7f4b000008000000006300006271
    Built at b3OpenCLUtils.cpp:880
    Run at b3LauncherCL.h:117

[2] Name: AddOffsetKernel
    File: .gpuverify/AddOffsetKernel003.cl
    local_size=128 global_size=896
    args=0x7f4b000000800000000800000780
    Built at b3OpenCLUtils.cpp:880
    Run at b3LauncherCL.h:117
...
```

Figure 2: Abridged output obtained from the command `gpuverify --show-intercepted`

After running KernelInterceptor on the Bullet simulator, this command produces the output shown in Fig. 2.

Each kernel instance is identified by a number, which is given in brackets. For each instance, the command reports:

- the name of the kernel;
- the file that contains the kernel's source code;
- the work group size (`local_size`) and the total number of work items (`global_size`);
- the hexadecimal values of the kernel's scalar arguments (see remark below);
- the position in the application's source code where this kernel was compiled; and
- the position in the application's source code where this kernel was invoked.

We remark that KernelInterceptor does not record non-scalar arguments (i.e. array or pointer arguments), since they tend not to affect the correctness of the kernel. Indeed, GPUVerify ignores the values of such arguments as part of its abstraction. Scalar values are stored in hexadecimal format because GPUVerify deals only with untyped bitvectors.

Reporting where each kernel instance was compiled and where it was invoked is valuable to users because tracing the origin of a kernel obtained by KernelInterceptor can be tricky: the kernel's source code may not be simply read from a file, but pieced together from multiple files and string constants at runtime, and possibly configured based on user input.

2.3 Verifying the intercepted kernels

Having inspected the intercepted kernels, the user can now ask GPUVerify to check their correctness.

The command

```
gpuverify --check-all-intercepted
```

[4] `http://bulletphysics.org`

[5] `https://github.com/erwincoumans/bullet3`

```
GPUVerify kernel analyser checked 37 kernels.
Successfully verified 35 kernels.
Failed to verify 2 kernels.

Successes:
[0]  Verification of AddOffsetKernel
     (.gpuverify/AddOffsetKernel001.cl) succeeded with:
     local_size=128 global_size=12544 args=3
...
Failures:
[13] Verification of scatterKernel
     (.gpuverify/scatterKernel003.cl) failed with:
     local_size=12 global_size=256 args=14,8
[27] Verification of SubtractKernel
     (.gpuverify/SubtractKernel020.cl) failed with:
     local_size=12 global_size=24 args=7
Run ‘gpuverify --check-intercepted=<number>’ for
more details.
```

Figure 3: Abridged output obtained from the command `gpuverify --check-all-intercepted`

instructs GPUVerify to attempt to verify all of the kernel instances. In an effort to maintain readability when there are many kernel instances, the output from GPUVerify is abbreviated, so as to identify only those kernels that failed to verify. These kernels can then be examined and re-verified individually. An illustrative output is shown in Fig. 3.

The command

```
gpuverify --check-intercepted=2
```

instructs GPUVerify to attempt to verify the kernel instance identified as number 2. In this case, GPUVerify outputs a message that it has verified the kernel, which implies that there are no data races and no instances of barrier divergence. Had GPUVerify detected the potential for any of these defects, it would have directed the user to the relevant line(s) in the `AddOffsetKernel003.cl` file. The third possible result from running GPUVerify is a timeout, which occurs when GPUVerify is unable to prove or to disprove the kernel's correctness.

3. IMPLEMENTATION

We now discuss some of the technical details of the implementation of KernelInterceptor. We continue to use the Bullet simulator as a running example.

3.1 Intercepting kernel launches

Relevant OpenCL host functions, such as `clSetKernelArg` or `clBuildProgram`, are intercepted at the source level, such that, for example, a call to `clSetKernelArg` in the host code actually calls our wrapper function, `clSetKernelArg_hook`. The wrapper functions log the relevant information and then pass the parameters to the original functions, as normal.

3.2 Logging kernel parameters

Each time a kernel is invoked, KernelInterceptor creates a file, whose name is formed from the name of the kernel, followed by a unique identifier to avoid name clashes. These files are stored in a `.gpuverify` directory, which KernelInterceptor creates in either the application's main directory, or in a directory specified by the environment variable `GPUV_KI_DIR`. In the case of the Bullet simulator, when executed for a few seconds on several of the standard demonstrations, over a thousand such files were created, corresponding to the invocations of 44 different kernels.

```
// --local_size=128 --global_size=896 ↵
   --kernel-args=AddOffsetKernel,↵
     0x00007f4b000000080000000080000780
// Built at ../../src/Bullet3OpenCL/↵
   Initialize/b3OpenCLUtils.cpp:880
// Run at ../../src/Bullet3OpenCL/↵
   ParallelPrimitives/b3LauncherCL.h:117
...
__kernel
void AddOffsetKernel(__global u32 *dst,↵
  __global u32 *blockSum, uint4 cb)
{
   ...
}
```

Figure 4: Data logged in `AddOffsetKernel003.cl` **for the third instance of the** `AddOffsetKernel` **kernel**

Let us now consider one of these files, `AddOffsetKernel003.cl`, which is created when KernelInterceptor intercepts the third launch of the kernel called `AddOffsetKernel`. Its contents is shown in Fig. 4. The file contains the kernel's source code, preceded by three commented lines. The first of these records the work group size and total number of work items, plus the hexadecimal value of `AddOffsetKernel`'s sole scalar argument (which is named `cb`). The second and third lines record the positions in the source code where the kernel was compiled and invoked, respectively.

3.3 Passing kernel arguments to GPUVerify

We have extended GPUVerify to accept a `--kernel-args` flag through which values for the arguments of a given kernel can be provided.

If K is the name of a kernel, and K's scalar arguments are x_1, ..., x_n, then

`--kernel-args=`K`,`v_1`,...,`v_n

instructs GPUVerify to assume the precondition

`__requires(`x_i`==`v_i`)`

for each $0 < i \leqslant n$, when verifying the kernel K. The order of the values provided to `--kernel-args` matches the order in which K's scalar arguments are declared.

An argument can be left unconstrained by inserting an asterisk. For instance, if K accepts three scalar arguments, `a`, `b` and `c`, then the flag

```
--kernel-args=binning_kernel,*,0x42,*
```

will insert the single precondition

```
__requires(b==0x42).
```

It is allowable to pass several `--kernel-args` flags to GPUVerify, each providing arguments for a different kernel in the same `.cl` file. By default, GPUVerify seeks to verify all the kernels in a given file, but we arrange that when one or more `--kernel-args` flags are provided, GPUVerify only checks those kernels that are named in those flags. A `.cl` file may contain a large number of kernels, only some of which are used by an application; our arrangement ensures that GPUVerify seeks to verify only those kernels that are actually invoked.

3.4 Caching verification results

When multiple kernel instances share the same source code, launch parameters and kernel arguments, the results of attempting to verify them will be the same. To avoid redundant calls to GPUVerify, we arrange that the results of successful verification attempts are written to a cache file, whose path is specified using the command-line flag

```
    --cache=<path>.
```

The cache file is consulted before each verification attempt, and if there is a match, the cached result is displayed. Failed verification attempts are not cached, since such attempts might become successful when a more capable version of GPUVerify becomes available.

4. DISCUSSION

In this section, we comment on the usability of our tool, discuss related work, consider some limitations of our tool, and suggest some future lines of enquiry.

4.1 Usability of KernelInterceptor

The GPUVerify team used KernelInterceptor to assist with the verification of the Parboil benchmark suite [12]. This suite consists of 12 programs and 25 unique kernels, some programmatically generated.

KernelInterceptor accelerated the process of extracting kernel source, compiler options, and valid local and global sizes. We observe that some kernels, such as those in the `stencil` benchmark, are only race free when given certain arguments; this would have been difficult to infer without the data provided by KernelInterceptor.

Using KernelInterceptor required adding just a handful of lines to the benchmark source and makefiles. It removed a significant amount of labour in the preparation of a recent conference paper [1].

4.2 Limitations

Discovery of invariants.

Although this work increases the degree of automation in GPU kernel verification, we should point out that *completely automatic* verification requires significant further research, due to the problem of discovering invariants for verifying barrier statements [3] and loop statements. Many kernels cannot be verified without these invariants, and although much progress has been made in using heuristics to infer these automatically, the task of supplying them often falls back to the user.

Dependence on particular kernel parameters.

Note that because the parameters are extracted from a *particular execution* of the OpenCL application, we cannot claim every kernel to be 'fully verified': the kernel may not be correct when launched with different parameters. What we *can* claim is that with these parameters, the kernel is correct under any execution schedule.

4.3 Future directions

Generalising parameters.

As noted above, a successfully verified kernel is only guaranteed to be defect-free when launched with specific parameters. In future work, we plan to investigate how to generalise these parameters, in order to strengthen the verification result.

Consider `SubtractKernel`, one of the kernels from the Bullet simulator. Starting from a successful verification with parameters

```
--local_size=64 --global_size=256 ↵
--kernel-args=SubtractKernel,0x000065f4,0x00000100
```

one could greedily unconstrain values, by setting them to "`*`", until a minimal set of constraints is obtained. We find that the correctness of this particular kernel does not depend on the kernel arguments, so the constraints

```
--local_size=64 --global_size=256 ↵
--kernel-args=SubtractKernel,*,*
```

are sufficient.

When there are many kernel instances to check, this parameter generalisation technique may lead to fewer calls to GPUVerify being required. For instance, all instances of `SubtractKernel` where `local_size` is `64` and `global_size` is `256` can now be considered verified, regardless of the other parameters, since the stronger result has already been proven.

We also plan to investigate other ways to unconstrain kernel parameters. Constraints such as 'this parameter must be a power of 2' or 'that parameter must not exceed 1024' could reasonably be conjectured by a tool such as Daikon [7], and then checked.

Run-time instrumentation.

We are considering implementing an alternative mechanism that operates solely at run-time. This would be even less intrusive to the user than the current mechanism, because no recompilation would be necessary. However, it would require additional work on our part to ensure compatibility with all platforms and drivers.

In the case of a Linux environment, we would make use of the `LD_PRELOAD` environment variable. This identifies a directory of libraries that should, at run-time, be linked before any other. By pointing this variable to our library of wrappers for the relevant OpenCL host functions, we can attain run-time interception.

Support for other kernel programming languages.

We plan to extend our kernel interception technique to support kernels that have been pre-compiled to the SPIR[6] intermediate representation. GPUVerify has direct support for the LLVM[7] intermediate representation [5], of which SPIR is a dialect, so this should prove quite straightforward. We plan also to support kernels written in CUDA, but we note that the run-time linking trick described above would not work in a CUDA setting, where host programs are typically linked statically.

[6] `http://www.khronos.org/spir`

[7] `http://llvm.org/`

Static analysis.

We plan to investigate the use of static analysis on the host program as an alternative way to discover kernel parameters. This would mean that the OpenCL application would not need to be executed at all; our tool would simply examine the application's source code. An advantage of an approach based on static analysis is that the correctness of the kernel can be guaranteed for *all possible* executions of the application, rather than just a particular execution. A disadvantage, however, is that the kernel verification is more likely to fail. It may, for instance, be understood that the application is only to be provided with positive inputs, but unless this requirement is codified as an explicit precondition in the source code, the static analysis will be ignorant of this and report that the kernel is incorrect in general.

4.4 Related work

There has been significant interest recently in methods for analysing and verifying GPU kernels.

Li and Gopalakrishnan's PUG analyser shares the problem of requiring the user to supply kernel arguments and the number of work items manually [10]. Our technique for addressing this problem only applies to OpenCL kernels, and hence is not directly applicable to PUG, which analyses only CUDA kernels.

The GKLEE [11] and KLEE-CL [4] tools, which are based on dynamic symbolic execution, do not have this problem because they execute symbolically both host and device code. However, although these tools seek to *discover* data races, they do not attempt to verify their *absence* as GPUVerify does.

The technique of Leung et al. [9] for verifying race-freedom of CUDA kernels is based on dynamic analysis and thus already exploits information about thread configurations and kernel arguments.

Huisman and Mihelčić have developed a technique to allow functional verification of GPU kernels without the need to fix the number of work items [8]. We observe that many kernels require some constraints on the number of work items (such as 'must be a power of 2' or 'must not exceed 1024') in order to be correct. The KernelInterceptor concept could therefore prove useful in this setting.

Acknowledgements.

This work was supported by EPSRC grant EP/K011499/1, the EU FP7 STREP project CARP (project number 287767), a gift from Intel Corporation, and the Imperial College Undergraduate Research Opportunities Programme (UROP). We thank the members of the Multicore Programming Group at Imperial for their feedback and encouragement.

5. REFERENCES

[1] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, and S. Qadeer. Engineering a static verification tool for GPU kernels. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV '14)*, volume 8559 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2014.

[2] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: A verifier for GPU kernels. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*, pages 113–132. ACM, 2012.

[3] N. Chong, A. F. Donaldson, P. H. Kelly, J. Ketema, and S. Qadeer. Barrier invariants: A shared state abstraction for the analysis of data-dependent GPU kernels. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*, pages 605–622. ACM, 2013.

[4] P. Collingbourne, C. Cadar, and P. Kelly. Symbolic crosschecking of data-parallel floating-point code. *IEEE Transactions on Software Engineering*, 40(7):710–737, July 2014.

[5] P. Collingbourne, A. F. Donaldson, J. Ketema, and S. Qadeer. Interleaving and lock-step semantics for analysis and verification of GPU kernels. In *Proceedings of the 22nd European Symposium on Programming (ESOP '13)*, volume 7792 of *Lecture Notes in Computer Science*, pages 270–289. Springer, 2013.

[6] E. Coumans. GPU rigid body simulation using OpenCL. In Multithreading and VFX*, a tutorial at* SIGGRAPH 2013, 2013. `http://www.multithreadingandvfx.org/course_notes/GPU_rigidbody_using_OpenCL.pdf`.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[8] M. Huisman and M. Mihelčić. Specification and verification of GPGPU programs using permission-based separation logic. In *The 8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '13)*, 2013.

[9] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner. Verifying GPU kernels by test amplification. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*, pages 383–394. ACM, 2012.

[10] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*, pages 187–196. ACM, 2010.

[11] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: concolic verification and test generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '12)*, pages 215–224. ACM, 2012.

[12] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Mar. 2012.