Symbolic Testing of OpenCL Code

Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly
{peter.collingbourne03, c.cadar, p.kelly}@imperial.ac.uk

Department of Computing
Imperial College London

Abstract. We present an effective technique for crosschecking a C or C++ program against an accelerated OpenCL version, as well as a technique for detecting data races in OpenCL programs. Our techniques are implemented in KLEE-CL, a symbolic execution engine based on KLEE and KLEE-FP that supports symbolic reasoning on the equivalence between symbolic values.
Our approach is to symbolically model the OpenCL environment using an OpenCL runtime library targeted to symbolic execution. Using this model we are able to run OpenCL programs symbolically, keeping track of memory accesses for the purpose of race detection. We then compare the symbolic result against the plain C or C++ implementation in order to detect mismatches between the two versions.
We applied KLEE-CL to the Parboil benchmark suite, the Bullet physics library and the OP2 library, in which we were able to find a total of seven errors: two mismatches between the OpenCL and C implementations, three memory errors, one OpenCL compiler bug and one race condition.

1 Introduction

The Open Computing Language (OpenCL) [12] is an open standard for parallel computing architectures, such as Graphics Processing Units (GPUs). OpenCL includes a C API which provides the means for a developer to execute computational *kernels* in parallel on an OpenCL compatible device. Kernels are written in a variant of ISO C99 [11] referred to as OpenCL C.

The fundamental unit of execution in OpenCL is the *work-item*, which represents a single invocation of a specified kernel function. A kernel invocation constitutes the parallel execution of a set of work-items, optionally organised into *work-groups*, which can share common resources such as local memory. Each work-item conceptually resides at a point in the kernel invocation's iteration space, referred to as the n-dimensional range, or *NDRange*. Data-level parallelism is achieved by having the kernel function vary the data items accessed depending on the position of the work-item in the iteration space. Figure 1 shows an example of how *work-item functions* can be used for this purpose.

The translation of an existing C or C++ program to OpenCL can be a complex process, especially for those unfamiliar with the concurrency model and the relevant APIs. In the end, the developer has little confidence that their translated OpenCL code is equivalent to the original C or C++ code. Neither can the

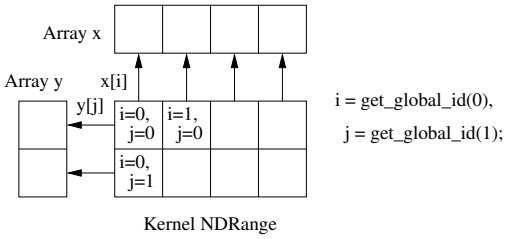

Fig. 1. Using a 2-dimensional NDRange iteration space to vary the data items accessed.

developer easily determine that their code is compliant with the OpenCL specification, because he or she may unknowingly be using undocumented quirks of their particular implementation. For example, memory is generally not required to be consistent across work-items [12, § 3.3.1], and the actual behaviour generally depends on the underlying hardware memory model.

This paper presents a crosschecking and data race detection technique for OpenCL programs. Our approach is based on symbolic execution [13], which provides a systematic way of exploring all feasible paths in a program for inputs up to a certain size. On each explored path, our technique works by building the symbolic expressions associated with the C/C++ and OpenCL versions of the code, and proving their equivalence. During symbolic execution of OpenCL kernels, we also maintain a log of all memory accesses for use in race detection. We build on earlier work [5], in which we extended the KLEE symbolic execution engine with support for crosschecking floating point and SIMD code.

This paper makes the following contributions:

1. We present a symbolic execution based technique for crosschecking OpenCL programs against their original C or C++ implementations.
2. We present a technique for testing for the presence of data races in OpenCL programs using a memory access log.
3. We describe KLEE-CL, an open-source tool that implements our technique by extending KLEE-FP [5] (itself an extension to the open source symbolic execution tool KLEE) with a model of the OpenCL runtime library and our race detection algorithm.
4. We evaluate KLEE-CL by applying it to three Parboil benchmarks, the Bullet physics library and the OP2 library, and show that it can find real bugs, including memory errors, race conditions, and implementation mismatches.

2 Overview

Our approach for testing OpenCL code is illustrated graphically in Figure 2. Given an OpenCL and a C/C++ implementation of a given routine, our technique uses symbolic execution to explore all feasible pairs of paths (or as many as possible in a given time budget) through the given implementations (§3.1).

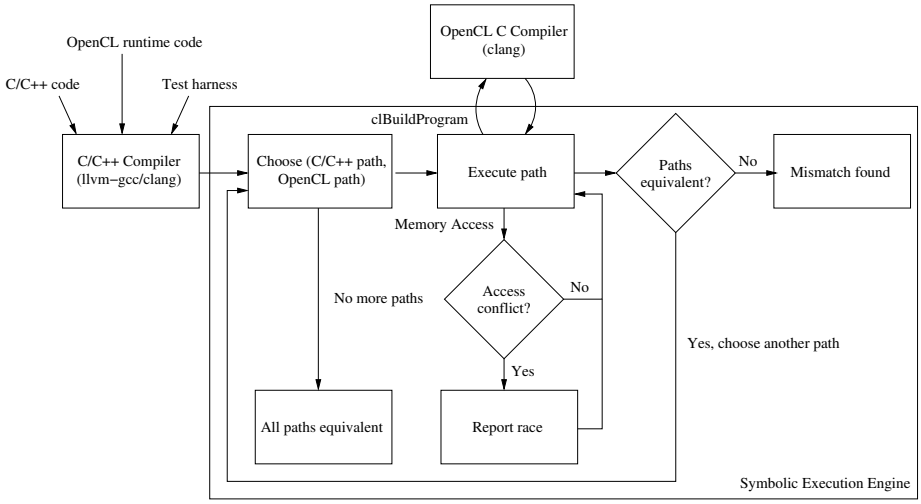

Fig. 2. Architecture diagram for KLEE-CL.

Then, on each explored path it (1) symbolically checks whether the two implementations compute equivalent outputs (§3.2) and (2) checks whether there are any race conditions (§5). In order to be able to reason about OpenCL code, our technique implements a symbolic OpenCL model (§4).

To illustrate the main features of our technique, we walk the reader through a simple example in which we check the equivalence between a C and an OpenCL implementation of a simple routine. The code example shown in Listing 1 contains a function called `cpu_arr_sqrt`, a C implementation of a function that computes the square root of every element of an array `in`, storing it into an array `out`; as well as `gpu_arr_sqrt`, an OpenCL implementation of the same function that makes use of the OpenCL kernel `arr_sqrt_kern`.

Like the C implementation, `gpu_arr_sqrt` takes as arguments the input and output arrays `in` and `out` and their size `size`. However, `gpu_arr_sqrt` receives three additional arguments: a `context`, which is used to execute kernels on one or more devices and to manage objects such as memory and kernel objects; a `command_queue`, which is created on a specific device and is used to enqueue OpenCL commands to be executed by the device; and `kernel`, which represents the function to be executed on the device (in our case `arr_sqrt_kern`).

In order to run the `arr_sqrt_kern` kernel, the code first creates two *memory buffer* objects, `in_buf` and `out_buf`, which represent memory allocated on the device. The memory buffer objects are set up such that OpenCL will copy data between the host and the device when necessary (i.e. `in` will be copied to `in_buf` before kernel execution, and `out_buf` to `out` after execution).

On line 11 the code sets the first kernel function argument to `out_buf`, and on line 12 the second to `in_buf`. Then on line 14 it calls `clEnqueueNDRangeKernel`,

```
1  __kernel void arr_sqrt_kern(__global float *out,
2                              __global const float *in) {
3    size_t i = get_global_id(0);
4    out[i] = sqrt(in[i]);
5  }
6
7  void gpu_arr_sqrt(cl_context context,
8                    cl_command_queue cmd_queue, cl_kernel kernel,
9                    float *out, const float *in, size_t size) {
10   /* Initialisation of in_buf and out_buf: omitted */
11   clSetKernelArg(kernel, 0, sizeof(cl_mem), &out_buf);
12   clSetKernelArg(kernel, 1, sizeof(cl_mem), &in_buf);
13
14   clEnqueueNDRangeKernel(cmd_queue, kernel,
15     /* work_dim */ 1,
16     /* global_work_offset */ NULL,
17     /* global_work_size */ &size,
18     NULL, 0, NULL, NULL);
19
20   clFinish(cmd_queue);
21 }
22
23 void cpu_arr_sqrt(float *out, const float *in, size_t size) {
24   for (size_t i = 0; i != size; ++i)
25     out[i] = sqrt(in[i]);
26 }
27
28 int main(void) {
29   float in[64], cpuout[64], gpuout[64];
30   uint32_t *cpuouti = (uint32_t *) cpuout;
31   uint32_t *gpuouti = (uint32_t *) gpuout;
32   klee_make_symbolic(in, sizeof(in), "in");
33
34   cpu_arr_sqrt(cpuout, in, 64);
35
36   /* Initialisation of context, cq, kernel: omitted */
37   gpu_arr_sqrt(context, cq, kernel, gpuout, in, 64);
38
39   for (size_t i = 0; i != 64; ++i)
40     assert(gpuouti[i] == cpuouti[i]);
41 }
```

Listing 1. A simple test benchmark.

which schedules the execution of `arr_sqrt_kern` on the device. Finally on line 20 it calls `clFinish`, which blocks until kernel execution terminates.

The call to `clEnqueueNDRangeKernel` specifies the bounds of an implicit parallel loop around a call to `arr_sqrt_kern`. In this case, the `work_dim` argument is set to 1, so the loop has one dimension; `global_work_size` is a pointer to `size`, so the loop will have `size` iterations; and `global_work_offset` is `NULL`, so the lower bound of the loop index is 0.

The `arr_sqrt_kern` kernel function implements one iteration of the loop found in `cpu_arr_sqrt`. The `get_global_id(0)` function call on line 3 is used to retrieve the loop index, which indexes the `in` and `out` arrays in the same way as the loop index `i` in `cpu_arr_sqrt`. As with `cpu_arr_sqrt`, the loop index ranges between 0 and `size`-1 due to the loop bounds specified by the arguments to `clEnqueueNDRangeKernel`.

The `main` function constitutes the test harness, which is similar to the ones used to crosscheck scalar and SIMD implementations in KLEE-FP [5]. In order to use our KLEE-CL tool, developers have to identify the C/C++ and the OpenCL versions of the code being compared, and the inputs and outputs to these routines. In our example, we have one input, namely the array `in`. Thus, the first step is to mark this array as *symbolic*, meaning that its elements could initially have any value (see §3.1 for more details). This is accomplished on line 32 by calling the function `klee_make_symbolic()` provided by KLEE, which takes three arguments: the address of the memory region to be made symbolic, its size in bytes, and a name used for debugging purposes only. Then, on line 34 we call the C version of the code and store the result in `cpuout`, and on line 37 we call the OpenCL version and store the result in `gpuout` (the initialisation of the parameters `context`, `cq` and `kernel` are omitted for brevity). Finally, on lines 39–40 each element of `gpuout` is compared against the corresponding element of `cpuout`. As in KLEE-FP, we use bitcasting to integers via the pointers `gpuouti` and `cpuouti` for a bitwise comparison. This is necessary because in the presence of `NaN` (*Not a Number*) values, the C floating point comparison operator `==` does not always return `true` if its floating-point operands are the same, as distinguished from a bitwise comparison.

3 Crosschecking of OpenCL and C Implementations

Our technique uses symbolic execution to explore multiple paths through the OpenCL and C/C++ implementations being compared, in order to check, on each path, for output equivalence (§3.2) and race conditions (§5).

3.1 Symbolic Execution

At a high level, symbolic execution is a technique that allows the automatic exploration of paths in a program. It works by executing the program on *symbolic* input, which is initially unconstrained. As the programs runs, any operations that depend on the symbolic input add constraints on the input. For example, if the program input is represented by variable `x`, than the statement `y = x+3` would add the constraint that $y = x + 3$. Furthermore, whenever a branch that depends on the symbolic input is reached, the technique first checks if both sides are feasible, and if so, it forks execution and follows each side separately, adding the constraint that the branch condition is true on the true side and false on the other side. For example, given the symbolic input `x`, where `x` is unconstrained, the symbolic execution of the branch `if (x == 3)` would result in two paths being explored, one on which $x = 3$ and one on which $x \neq 3$.

In our work, we use symbolic execution to explore the different paths in the OpenCL and C/C++ implementations being tested, and for each pair of paths, we check whether (1) there are no memory errors (these checks are by default performed by KLEE); (2) the implementations are race free (§5) and (3) the outputs computed by the two implementations are equivalent (§3.2).

One fundamental limitation of symbolic execution is that it only handles objects of fixed size (e.g., each data structure in a program usually has to be assigned a concrete size, as in the normal execution of the program). For our work, this means that we can verify the equivalence of OpenCL and C/C++ programs only up to a certain input size and number of threads. In the rest of the paper, we discuss our experiments solely in terms of input size: this is because in a typical OpenCL program, the number of work-items depends linearly on the size of the input being processed.

3.2 Equivalence Checking

To verify the output equivalence on a pair of paths through the two implementations, our technique first constructs the symbolic expressions corresponding to the output of each implementation, applies a set of canonicalisation rules to bring the two expressions to a canonical form, and then compares the two expressions syntactically. The main advantages of this approach are performance and the ability to deal with floating-point expressions, for which there are no efficient constraint solvers currently available. On the other hand, this approach is prone to false positives, i.e., it can say that two expressions are not equivalent when in fact they are. For more details, we refer the reader to our previous work on KLEE-FP [5].

In addition to the canonicalisation rules already implemented in KLEE-FP, we added a set of new rules, some of which rely on certain *assumptions* about the floating point model. For example, it is generally unsound to simplify $x \times 0$ to 0 because if x is negative or infinite the result is respectively -0 or `NaN`. However, developers are often not interested in such edge cases, and therefore we added the option to enable such assumptions on demand (via command line arguments). We added a total of three assumptions with five associated rules:

- The *positive zero* assumption allows the simplifier to disregard the difference between positive and negative zero, which is usually inconsequential. If this assumption is enabled, $x + 0$ may be simplified to x.
- The *finite* assumption allows the simplifier to assume all results are finite. If this assumption together with the positive zero assumption is enabled, $x \times 0$ and $0 \times x$ may be simplified to 0.
- The *associativity* assumption allows the simplifier to assume that floating point operations are associative. If this assumption is enabled, $+$ and $\times$ operations are rearranged to be left-associative, so $x + (y + z)$ is normalised to $(x + y) + z$ and $x \times (y \times z)$ to $(x \times y) \times z$.

We also added two rules which do not rely on any assumptions being enabled. These rules allow $x \times 1$ and $1 \times x$ to be simplified to x.

4 Modelling the OpenCL Environment

Our OpenCL model presents a single OpenCL compliant device to the program under test. This device presents itself as a CPU-based device with support for the

`cl_khr_fp64` extension, which allows the kernel to use double-precision floating point arithmetic.

The OpenCL model is made up of two distinct parts: the runtime library, which is used by the host to manage the execution of OpenCL kernels, and the OpenCL C environment, which models the execution of a kernel on the device.

4.1 The OpenCL Runtime

The OpenCL runtime library is specified by two sections of the OpenCL specification: the OpenCL Platform Layer [12, § 4] and the OpenCL Runtime [12, § 5]. The Platform Layer is used to query the set of available OpenCL devices, while the Runtime is used to query and manipulate objects on a specific device or set of devices such as device-side memory buffers and compiled OpenCL programs. In total, our model implements 30 functions specified as part of the Platform Layer and Runtime. For example, the `clEnqueueNDRangeKernel` function discussed in Section 2 is implemented by starting one modelled POSIX thread for each work-item in the iteration space. Each thread sets up the environment appropriately (for example, by initialising thread local variables) and then calls the kernel function. In our implementation, we use the POSIX threading model added to KLEE by Cloud9 [2].

4.2 The OpenCL C Programming Language

OpenCL kernels are written in an extended version of ISO C99 [11] referred to as OpenCL C, which is specified as part of the OpenCL specification [12, § 6]. Among the language extensions provided by OpenCL C are vector data types, specialised memory address spaces and a set of built-in functions.

The vector data types provided by OpenCL are used to exploit the SIMD capabilities common among GPUs. For example, `float4` is the name of a data type referring to a vector of four `float` values. KLEE-FP, which our technique extends, already includes support for vector data types [5].

The four disjoint address spaces provided by OpenCL are named `__global`, `__local`, `__constant` and `__private`. Globally available data resides in `__global`, data local to a work-group in `__local`, read-only data in `__constant` and function arguments and local variables in `__private`.

Three of these address spaces (`__global`, `__constant` and `__private`) can be modelled using the generic address space used by regular CPU implementations. The `__local` address space, however, needs special attention because `__local` data must be shared between work-items in the same work-group, and each work-group must have its own `__local` data. To model `__local`, we added a *group-local* address space, which is an address space shared between user-created thread groups. Each thread belongs to a single thread group. Before beginning kernel execution, we create one thread group for each work-group, and set each thread's group to match its work-group.

Our model implements 18 of the built-in functions specified by the OpenCL specification, which are enough to run our benchmarks. These include work-item

functions, which are used by the kernel to query various properties of the current execution's index space; math functions, which perform various mathematical operations (including vectorised variants); and the `barrier` synchronisation function, which is used to introduce execution barriers into the kernel.

4.3 Runtime Compilation of OpenCL Kernels

Programs that use the OpenCL runtime library (written in languages such as C or C++) are compiled in the usual way, before they are run. By contrast, kernels written in OpenCL C are normally compiled at runtime by passing their source code as a string to the runtime library function `clCreateProgramWithSource`, and later compiling the program using the `clBuildProgram` function. This can pose a challenge for tools such as ours, which necessarily must incorporate a full OpenCL C compiler. Our implementation of `clBuildProgram` invokes a compiler based on the OpenCL C front-end provided by the Clang [3] compiler. Clang is designed to be used as a library, which made it easy to integrate into KLEE-CL. Clang produces an LLVM [14] module representing the compiled program which is then dynamically loaded into the current instance of KLEE-CL.

5 Race Detection

Our model implements race detection capable of detecting, on each path explored, read-write and write-write races across work-items. Note that our analysis is targeted towards detecting races between work-items in the same NDRange. In OpenCL, a command queue may be created in out-of-order mode [12, § 5.11]. By scheduling multiple kernel invocations on an out-of-order command queue, or by scheduling kernel invocations across multiple command queues, a client program may cause kernel NDRanges to run in parallel such that races may occur between NDRanges. In this work, we concern ourselves only with the more common in-order case where only one NDRange is executing at a time.

To detect data races, we keep for each byte in the generic and group-local address spaces a *memory access record (MAR)* of accesses to that byte by a work-item thread. Each item in the MAR consists of the thread identifier of the most recent work-item to access the byte, the work-group identifier of the most recent work-group to access the byte, and four flags indicating whether the byte was (1) written by one or more work-items, (2) read by one or more work-items, (3) read by multiple work-items (*many-read*), and (4) read by multiple work-groups (*wg-many-read*).

The MAR may be stored concretely or symbolically. The concrete representation of the MAR is an array of structs, each holding the MAR for one byte in the array. The symbolic representation of the MAR is a set of 6 symbolic arrays, each as large as the underlying array, and each representing one of the MAR attributes. For efficiency we store the MARs concretely by default, but if a symbolically indexed memory access is performed, the array's MARs are converted to the symbolic representation.

Read

$write[index] \wedge threadId[index] \neq threadId \wedge wgid[index] \neq wgid$

$$
\begin{aligned}
manyRead[index] &\leftarrow manyRead[index] \vee (read[index] \wedge threadId[index] \neq 0 \wedge \\
&\qquad threadId[index] \neq threadId) \\
wgManyRead[index] &\leftarrow wgManyRead[index] \vee (read[index] \wedge wgid[index] \neq wgid) \\
threadId[index] &\leftarrow threadId \qquad\qquad wgid[index] \leftarrow wgid \\
read[index] &\leftarrow \top
\end{aligned}
$$

Write

$manyRead[index] \vee wgManyRead[index] \vee ((read[index] \vee write[index]) \wedge threadId[index] \neq threadId \wedge wgid[index] \neq wgid)$

$$
\begin{aligned}
threadId[index] &\leftarrow threadId \qquad\qquad wgid[index] \leftarrow wgId \\
write[index] &\leftarrow \top
\end{aligned}
$$

Fig. 3. Race condition test and MAR updates.

Whenever a memory access occurs, the MAR is inspected for any race conditions, and then updated. A race condition can be a read-after-write, a write-after-write or a write-after-read performed by a work-item or work-group other than that identified by the corresponding entry in the MAR, or any write-after-read if either of the *many-read* or *wg-many-read* flags are set.

The race condition test, together with the required MAR updates, are shown in Figure 3. If the MAR is being stored concretely, we perform the test and the MAR updates directly. If the MAR is being stored symbolically, the test is performed by querying the constraint solver as to whether the symbolic expression representing the race condition test is feasible, and the MAR updates are performed by appending an update to the symbolic arrays.

The main intra-work-group synchronisation mechanism provided by OpenCL C is the `barrier` function, which acts as an execution barrier. `barrier` blocks until all work-items in the work-group have reached the call to `barrier`, at which point a memory fence is queued to ensure the correct ordering of memory operations between work-items, and all work-items resume execution.

To simulate this behaviour, when a work-item reaches a barrier we add it to a list of blocked work-items associated with the current work-group. When the size of this list becomes as large as the number of work-items in the work-group, the MAR is *locally reset* and the list emptied, resuming execution. We locally reset the MAR by removing the work-item identifier and clearing the many-read flag of each MAR whose work-group identifier matches the work-group performing the `barrier`. This has the effect of causing no intra-work-group accesses across the reset to be considered a race, while preserving inter-work-group race detection.

At the end of the execution of a kernel, we must perform a *full reset* of the MAR, to prevent access records from one kernel invocation from interfering with accesses from subsequent invocations (since we only support in-order kernel invocation, it is safe to do this). Similar to the case when a barrier is reached, we add the kernel to a list of inactive work-items, which is this time associated with the entire NDRange. When the list size becomes as large as the size of the NDRange, we reset the MAR by removing all identifiers and clearing all flags, and then resume execution.

To illustrate the race detection technique described above, we use the code in Figure 4. This code contains two simple kernels, `avg` and `avg2`, the purpose

```
1 __kernel void avg(__global float *a) {
2   size_t lid = get_local_id (0),
3          lsize = get_local_size (0);
4   float r0 = lid > 0 ? a[lid-1] : 0;
5   float r1 = a[lid];
6   float r2 = lid+1 < lsize ? a[lid+1] : 0;
7   a[lid] = (r0 + r1 + r2) / 3;
8 }
```

	Work-item 1					Work-item 2				
	Tid	Wid	R	W	Con	Tid	Wid	R	W	Con
2						1	1	X	X	
3						1	1	X	X	
4						1	1	X	X	w/r
5	1	1	X			1	1	X	X	
6	1	1	X			1	1	X	X	
7	1	1	X	X		1	1	X	X	

```
1 __kernel void avg2(__global float *a) {
2   size_t lid = get_local_id (0),
3          lsize = get_local_size (0);
4   float r0 = lid > 0 ? a[lid-1] : 0;
5   float r1 = a[lid];
6   float r2 = lid+1 < lsize ? a[lid+1] : 0;
7   barrier(CLK_GLOBAL_MEM_FENCE);
8   a[lid] = (r0 + r1 + r2) / 3;
9 }
```

	Work-item 1					Work-item 2				
	Tid	Wid	R	W	Con	Tid	Wid	R	W	Con
2						1	1	X		
3						1	1	X		
4						1	1	X		
5	1	1	X			1	1	X		
6	1	1	X			1	1	X		
7		1	X			1	1	X	X	
8	1	1	X	X		1	1	X	X	

Fig. 4. Intermediate MARs for the memory location at a[0] during execution of work-items 1 and 2. Column Tid shows the byte's work-item identifier, Wid its work-group identifier, R the read flag, W the write flag, and Con (if present) the nature of the conflict detected at that line. Note that the many-read and wg-many-read flags are not shown here.

of which is to store in each element of array `a` the mean of that element and the two adjacent elements.

The `avg` kernel contains a race condition, while `avg2` uses an execution barrier to avoid the race. For each statement in the kernels, we show alongside it the state of the MAR for the first element of array `a` after execution of that statement. Note that in KLEE-CL we execute each work-item in its entirety until it reaches an execution barrier or terminates; however, our race detection algorithm would work with any other execution schedule. Thus, for `avg` the entirety of work-item 1 is executed before work-item 2, and the MAR persists from the end of execution of work-item 1 to the beginning of execution of work-item 2. For `avg2` the first five lines of work-item 1 are executed (up to the barrier), then the first five lines of work-item 2, the memory access records are locally reset, the last two lines of work-item 1 are executed and finally the last two lines of work-item 2.

On line 4 of `avg` in work-item 2, we report a read-after-write race. This is due to the earlier write of work-item 1 on line 7 causing the write flag to be set. This race does not exist in `avg2` because on line 4 of `avg2` in work-item 2, line 8 in work-item 1 had not yet been reached, as it had been preempted by the barrier on line 7.

6 Evaluation

We evaluated our technique on a set of benchmarks that compare C/C++ and OpenCL variants of code developed independently by third parties. The codebases that we selected were the Parboil benchmark suite [10], the Bullet physics library [6] and the OP2 [8] library.

6.1 Parboil

Parboil [10] is a popular GPU benchmark suite, which contains C and CUDA [18] implementations of various algorithms. In order to be able to run Parboil benchmarks using KLEE-FP, we used Grewe et al's [9] translation of certain Parboil 1 benchmarks from CUDA to OpenCL. The translation comprised four benchmarks in total, and we tested three of these: `cp` (Coulombic Potential), `mri-q` (Magnetic Resonance Imaging – Q) and `mri-fhd` (Magnetic Resonance Imaging – FHD). We were unable to test the fourth benchmark, `rpes` (Rys Polynomial Equation Solver) because it created a very large number of work-items (> 30000) even for small problems, which KLEE-CL could not execute in a reasonable amount of time.

We modified the code for each benchmark to incorporate the C and OpenCL versions of the benchmarks into the same executable. This allowed us to construct simple test harnesses similar to the one in Listing 1 which invoke both versions of the benchmarks with the same symbolic arguments.

By running these benchmark programs using KLEE-CL, we detected two mismatches between the C and OpenCL implementations of `cp`. We also found three memory errors in `mri-q` and `mri-fhd` as a result of the memory bounds checking performed during symbolic execution.

Mismatches: The `cp` benchmark computes the Coulombic potential for a set of points on a grid. The computation of a Coulombic potential at a grid point involves the calculation of the Euclidean distance of the form $\sqrt{\delta x^2 + \delta y^2 + \delta z^2}$ between an electrically charged particle and that point.

The first mismatch for `cp` is due to an associativity issue. The OpenCL implementation uses an unrolled loop in which a set of adjacent grid points are computed during each iteration. Because only the x coordinate varies during an iteration, the values of δy and δz remain constant, allowing $\delta y^2 + \delta z^2$ to be precomputed at the start of each iteration. So the expression is evaluated as $\sqrt{\delta x^2 + (\delta y^2 + \delta z^2)}$. In the C implementation, the inner expression is left unbracketed and normal C associativity rules apply. Because + is left-associative in C [11], the expression is evaluated as $\sqrt{(\delta x^2 + \delta y^2) + \delta z^2}$. Since + in floating point is not associative, the two expressions do not match.

The second mismatch arises in the context of computing δx in the two implementations. In the C implementation, this is done by subtracting the atom's x coordinate from the grid's x coordinate. In the OpenCL implementation, δx for the iteration's first grid point is computed in the same way. However, for subsequent points in the iteration, δx is computed by adding the grid's spacing to the value of δx for the previous point. Since floating point + and $\times$ are neither associative nor distributive, the expressions do not match.

Whether these mismatches are important or not depends on the specific application. KLEE-CL's job is to flag such mismatches, but it is up to the developer to assess whether strict equivalence should be enforced. Furthermore, developers can use the assumptions discussed in Section 3.2 to ignore the cause of different mismatches. For the current example, developers could add the assumption that floating point operations are associative and rerun KLEE-CL to find other

problems. With this assumption enabled, KLEE-CL verifies a variant of this benchmark in which the second mismatch, but not the first, has been fixed.

Memory errors: A non-obvious memory error was found in `mri-q`. After the OpenCL kernel is invoked, `mri-q` deallocates some OpenCL memory buffers and then copies some data from the GPU to the host. Because OpenCL kernel invocation is asynchronous, the memory buffers may be deallocated by the time that the kernel accesses them. KLEE-CL caught this error as a result of its thread scheduling behaviour—it will defer execution of code running in other threads (i.e. kernel code) until the current thread explicitly yields execution. This means that the deallocations (running in the main thread) were executed before the kernel code. We fixed this error by moving the data copies before the memory deallocations. Since the data copies were synchronous, they caused execution of the main thread to be preempted until after kernel execution.

A memory error found in both `mri-q` and `mri-fhd` was caused by a read beyond the end of a memory buffer used to store (x, y, z) coordinates. This memory buffer was indexed using the work-item identifier, which ranged between 0 and a multiple of the work-group size. This error was never caught, perhaps due to the fact that all benchmark data provided with Parboil had a size that was a multiple of the work-group size. We fixed these errors by enclosing the relevant part of the kernel inside an `if` statement.

A memory error found in `mri-fhd` is related to the use of uninitialised memory. This benchmark allocates a buffer of output data using `memalign`, which was assumed to be zero initialised. Since `memalign` buffers are uninitialised, and KLEE-CL models this, incorrect results were produced. The fix was simply to initialise the buffer using `memset`.

6.2 The Bullet Physics Library

Bullet [6] is a physics library primarily used in gaming and 3D applications. It incorporates a number of physics simulation algorithms, including a soft body simulation. This can be used to simulate objects such as cloths which are freely deformable within the environment. Bullet provides a C++ and an OpenCL implementation of the soft body simulation.

We implemented two benchmark programs which create a simulation with two soft body objects, each containing three vertices connected by three edges. The coordinates of the vertices are concrete values, but all other simulation parameters are symbolic. The program runs a single simulation step using both the C++ and the OpenCL implementations, and compares the results.

The first of our benchmarks (`softbody`) tests the soft body simulation in isolation, while the second benchmark (`dynworld`) tests the simulation using a soft rigid dynamics world, which exercises more of the soft body code.

For the `softbody` benchmark, KLEE-CL verified that the C++ and OpenCL code produce the same results. For `dynworld`, KLEE-CL was able to verify equivalence under the assumption that $x \times 0 = 0$ in floating point.

One important caveat is that we do not model inaccurate floating point operations, such as the single precision division operation in OpenCL (which

need only be accurate to 2.5 ulp [12, § 7.4]), because the LLVM IR generated by the Clang compiler does not provide the accuracy of each individual operation. In fact, while running a test using real GPU hardware (an NVIDIA Tesla C1060), we found discrepancies between the C++ and OpenCL results, which were due to a single precision floating point division operation, caused by the incorrect modelling discussed above. We attempted to rectify this by casting the operands of the division operator to double precision ([12, § 9.3.9] requires double precision division to be correctly rounded).

OpenCL compiler bug: Of course, these equivalence results hold under the additional assumption that all the components involved in running the code—from compilers to hardware—are correct. The bug discussed below illustrates this point.

After fixing the single precision issue mentioned above, we were surprised to see that the test run on real GPU hardware still showed discrepancies between the OpenCL and C++ implementations, despite the fact that we were able to verify their equivalence. After further investigation, we found that the PTX assembly code produced by NVIDIA's OpenCL compiler continued to use a single precision division instruction (`div.full.f32`), despite the cast to double precision. If we disabled compiler optimisations, using the `-cl-opt-disable` flag to the OpenCL compiler, the double precision division instruction (`div.rn.f64`) was used. This suggested that the problem may lie in the optimiser.

We worked around this issue by postprocessing the PTX code to replace `div.full.f32` with `div.rn.f64` together with appropriate conversions, similar to the unoptimised code. After doing this, the results obtained were identical.

We reported the issue to NVIDIA who confirmed our bug report, and as of this writing had fixed the bug, but had not yet released a version of their OpenCL implementation with the fix.

6.3 OP2

OP2 [8] is a library for generating parallel executables of applications using unstructured grids. OP2 enables users to write a single program targeting multiple platforms. OP2 has four implementations: a serial reference (library) implementation and source-to-source transformations to CUDA, OpenCL and OpenMP.

Among the operations offered by OP2 is the *global reduction* operation, which is used to reduce a set of results computed across a set of grid nodes to a single result. We used KLEE-CL to test the correctness of the OpenCL implementation of the global reduction operation by extracting the relevant kernel from the OP2 source code and constructing a benchmark program which uses this kernel to perform a global reduction on an array of symbolic data.

KLEE-CL detected a race condition in this kernel, and the problematic code is shown in Listing 2. Each iteration of the `for` loop on lines 4–9 uses a result computed in an earlier iteration by another work-item (specifically, work-item `tid` uses a result computed by work-item `tid+d`) without using an execution barrier beforehand. Because of the lack of synchronisation, the behaviour of the kernel is undefined by the OpenCL specification.

```
1  int tid = get_local_id( 0 ), d = get_local_size( 0 )>>1;
2  __local volatile float *vtemp = temp;
3  ...
4  for ( ; d>0; d>>=1 ) {   /* d is at most 16 here */
5    if ( tid<d )
6      ...
7      vtemp[tid] = vtemp[tid] + vtemp[tid+d];
8      ...
9  }
```

Listing 2. OP2's unsynchronised loop (slightly modified for formatting purposes).

To understand why this loop was written in this way, one must consider the history of the code. The OpenCL implementation was heavily based on the CUDA implementation and was in many places developed by replacing CUDA constructs with the relevant OpenCL constructs. In CUDA (and the NVIDIA GPU architecture), each group of 32 work-items within a work-group (referred to as a *warp*) is executed in lockstep with implicit synchronisation between work-items [18]. However, no such feature is present in OpenCL, and OpenCL code relying on warps has implementation-defined behaviour. In the case of the NVIDIA implementation of OpenCL this happens to function correctly, however there is no requirement that it do so on other architectures.

We modified the kernel to introduce a local execution barrier using the `barrier` function before each iteration of the loop (between lines 4 and 5). With this modification in place, KLEE-CL does not report a race condition.

7 Related Work

Despite the growing popularity of GPU languages, there has been relatively little work on testing and verification techniques for code written in these languages. While we are not aware of any work directly targeting OpenCL, several relevant testing techniques exist for checking CUDA code.

Race detection. Most previous work in this space has focused on race detection [1, 15, 24, 25]. Li and Gopalakrishnan [15] and Tripakis et al. [24] propose two static race detection techniques based on translating CUDA code into SMT constraints. The main advantage of a static analysis approach is coverage: our dynamic approach depends on the number of paths explored by symbolic execution in a given time budget and can only reason about objects with concrete bounds. On the other hand, static analysis suffers from false positives, due to various over-approximations resulting from, e.g., analysing kernels in isolation and loop unrolling.

A dynamic race detection approach similar to our technique is introduced by Boyer et al. in the context of CUDA programs [1]. A more recent technique from Zheng et al. [25] combines dynamic race detection with a static analysis pass that removes accesses that can be statically proven to be safe or unsafe, resulting in a system with a relatively small runtime overhead. The main weakness of these techniques is that they depend on the concrete inputs with which the program

is run. Instead, our approach can check for symbolic race conditions on all the different paths explored via symbolic execution.

Our approach is also similar in spirit to previous dynamic race detection approaches for CPU code [7, 19, 21], although the barrier-based synchronisation model used in OpenCL allows for a simpler algorithm than in the case of traditional synchronisation primitives such as locks and semaphores.

Concurrently and independently with our work, GKLEE [16] has extended KLEE with the ability to find several categories of errors in CUDA programs, including race conditions and deadlocks caused by execution barrier divergence.

Equivalence checking. As far as we know, this is the first technique that focuses on checking the equivalence between an OpenCL and a C or C++ implementation. Our work builds up on previous research on KLEE-FP [5], in which we have applied a similar approach to crosscheck SIMD and scalar implementations. To make this general crosschecking approach effective to OpenCL code, we had to construct an OpenCL model, and add support for concurrency, race detection and several additional rules and assumptions. In addition to our work on KLEE-FP, this approach has been successfully used in the past to verify code equivalence in other contexts, such as hardware verification [4], compiler optimisations [17], block cipher implementations [23] and parallel numerical programs [22]. The main advantages of normalizing symbolic expressions and then comparing them syntactically are that (1) the technique is lightweight compared to more precise symbolic analyses such as [20], and (2) it can deal with floating-point constraints, for which there are no efficient constraint solvers currently available. On the other hand, this approach is prone to false positives, i.e., it can say that two expressions are not equivalent when in fact they are.

8 Conclusion

We presented an effective technique for crosschecking OpenCL and C/C++ programs and for detecting race conditions in OpenCL code. We implemented our approach in the KLEE-CL tool, and applied it to three real OpenCL code bases, in which it found seven previously unknown errors: two mismatches between the OpenCL and C implementations, three memory errors, one OpenCL compiler bug and one race condition. The KLEE-CL tool is freely available from our website at `http://www.pcc.me.uk/~peter/klee-cl/`.

Acknowledgements

We would like to thank Stefan Bucur and the authors of Cloud9 [2] for providing the POSIX threading model; Lee Howes for assistance with the Bullet Physics library; Daniel Dunbar for helpful discussions; Guy Benyei, Tanya Lattner, Anton Lokhmotov and Alberto Magni for their contributions to the Clang OpenCL front-end; and Paul Marinescu for valuable comments on the text. This work was partially funded by an EPSRC DTA studentship and the EPSRC Platform Grant EP/I012036/1.

References

[1] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of CUDA programs. In *STMCS'08*, Apr. 2008.

[2] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys'11*, Apr. 2011.

[3] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[4] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *ASP-DAC'03*, Jan. 2003.

[5] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *EuroSys'11*, Apr. 2011.

[6] E. Coumans et al. Bullet continuous collision detection and physics library. `http://bulletphysics.org/`.

[7] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *PLDI'09*, June 2009.

[8] M. B. Giles, G. R. Mudalige, Z. Sharif, G. R. Markall, and P. H. J. Kelly. Performance analysis of the OP2 framework on many-core architectures. *SIGMETRICS Performance Evaluation Review*, 38(4):9–15, 2011.

[9] D. Grewe and M. F. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *CC'11*, Mar.-Apr. 2011.

[10] IMPACT Research Group, UIUC. Parboil benchmark suite. `http://impact.crhc.illinois.edu/parboil.php`.

[11] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, Dec. 1999.

[12] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1, revision 36*, Sept. 2010.

[13] J. C. King. A new approach to program testing. In *ICRS'75*, Apr. 1975.

[14] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*, Mar. 2004.

[15] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *FSE'10*, Nov. 2010.

[16] G. Li, P. Li, G. Sawaya, and I. Ghosh. GKLEE: Concolic verification and test generation for GPUs. In *PPoPP'12*, Feb. 2012.

[17] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI'00*, May 2000.

[18] NVIDIA. *NVIDIA CUDA Programming Guide, Version 3.0*, Feb. 2010.

[19] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP'03*, June 2003.

[20] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE'08*, Nov. 2008.

[21] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *SOSP'97*, Oct. 1997.

[22] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *ISSTA'06*, July 2006.

[23] E. W. Smith and D. L. Dill. Automatic formal verification of block cipher implementations. In *FMCAD'08*, Nov. 2008.

[24] S. Tripakis, C. Stergiou, and R. Lublinerman. Checking non-interference in SPMD programs. In *HotPar'10*, June 2010.

[25] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. GRace: A low-overhead mechanism for detecting data races in gpu programs. In *PPoPP'11*, Feb. 2011.