Many-Core Compiler Fuzzing

Christopher Lidbury[1], Andrei Lascu[1], Nathan Chong[2], Alastair F. Donaldson[1]

[1]Imperial College London, UK, [2]UCL, UK

[1]{christopher.lidbury10,andrei.lascu10,alastair.donaldson}@imperial.ac.uk, [2]n.chong@ucl.ac.uk

Abstract

We address the compiler correctness problem for many-core systems through novel applications of fuzz testing to OpenCL compilers. Focusing on two methods from prior work, *random differential testing* and testing via *equivalence modulo inputs* (EMI), we present several strategies for random generation of deterministic, communicating OpenCL kernels, and an injection mechanism that allows EMI testing to be applied to kernels that otherwise exhibit little or no dynamically-dead code. We use these methods to conduct a large, controlled testing campaign with respect to 21 OpenCL (device, compiler) configurations, covering a range of CPU, GPU, accelerator, FPGA and emulator implementations. Our study provides independent validation of claims in prior work related to the effectiveness of random differential testing and EMI testing, proposes novel methods for lifting these techniques to the many-core setting and reveals a significant number of OpenCL compiler bugs in commercial implementations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent programming–parallel programming; D.2.5 [*Software Engineering*]: Testing and Debugging–testing tools; D.3.4 [*Programming Languages*]: Processors–compilers

Keywords Compilers, OpenCL, GPUs, random testing, metamorphic testing, concurrency

1. Introduction

Open Computing Language (OpenCL) [10] is an industry standard model for programming *many-core* computer systems in which parallel processing capabilities are offered by CPUs, GPUs, FPGAs and other accelerators. OpenCL offers a *kernel*-based programming model where the developer factors out data-parallel parts of an application into *kernel functions*. Multiple instances of a kernel function execute in parallel across the processing elements of a many-core device.

A key aim of OpenCL is portability. If an OpenCL kernel conforms to the standard (exhibiting no undefined behaviours), and does not depend on implementation-defined behaviour, then although the kernel may behave nondeterministically (due to concurrency) it should yield a result drawn from a well-defined, implementation-independent set of permitted results, regardless of the device on which it executes.

One of the principal challenges for an OpenCL implementer in achieving this portability guarantee for a given device is the construction of a correct compiler for OpenCL C, the C-like programming language in which kernel functions are written. To be conformant, the compiler must support the full range of OpenCL language constructs, which includes a three-layer memory hierarchy, a rich set of vector data types and operations, read-modify-write atomics, and barrier synchronization. To be practical, the compiler must perform aggressive, device-specific optimizations since performance is the *sole* reason for adoption of OpenCL. OpenCL compiler reliability is especially crucial because, by default, compilation is performed *online*. OpenCL-accelerated applications are written in a device-agnostic manner and the kernels used by an application are compiled at runtime by the drivers of available devices. Online compilation with respect to devices that are unknown during development means that compiler bugs cannot easily be anticipated and circumvented.

In this paper, we investigate many-core compiler *fuzzing* (i.e. testing with respect to randomly generated inputs) in the context of OpenCL. We focus on two recent successful techniques for finding bugs in C compilers: *random*[1] *differential testing* [20] and testing via *equivalence modulo inputs* [12] (henceforth referred to as *EMI testing*). Our work makes four main contributions:

1. We provide a large study independently validating claims made in prior work [12, 20] about the effectiveness of random differential testing and EMI testing, in a new application domain.
2. We lift random differential testing to the many-core setting via three novel methods for generating deterministic, communicating, feature-rich OpenCL kernels.
3. We propose and evaluate injection of *dead-by-construction* code to enable EMI testing in the context of OpenCL.
4. Through a testing campaign using 21 (device, compiler) configurations we have identified and reported more than 50 OpenCL compiler bugs, most in commercial implementations.

Online material We provide our tools, test programs, full result data sets, and details of the bugs we found, online.[2]

2. Overview of Methods and Results

We start with a bird's-eye view of our contribution. Background on OpenCL and compiler fuzzing and full details of our methods and results follow in the subsequent sections.

2.1 The Devices and Compilers We Tested

We conducted testing with respect to 21 distinct OpenCL configurations, summarised in Table 1. A *configuration* refers to an

[1] Throughout the paper we use *random* to mean *pseudo-random*.

[2] `http://multicore.doc.ic.ac.uk/tools/CLsmith/PLDI15`

Table 1. The OpenCL implementations and devices we have tested

Conf.	*SDK*	*Device*	*Driver/compiler*	*OpenCL*	*Operating system*	*Device type*	*Above threshold?*
1	NVIDIA 6.5.19	NVIDIA GeForce GTX Titan	343.22	1.1	Ubuntu 14.04.1 LTS	GPU	✓
2	NVIDIA 6.5.19	NVIDIA GeForce GTX 770	343.22	1.1	Ubuntu 14.04.1 LTS	GPU	✓
3	NVIDIA 7.0.28	NVIDIA Tesla M2050	346.47	1.1	RHEL Server 6.5	GPU	✓
4	NVIDIA 7.0.28	NVIDIA Tesla K40c	346.47	1.1	RHEL Server 6.5	GPU	✓
5	AMD 2.9-1	AMD Radeon HD7970 GHz edition	Catalyst 14.9	1.2	Windows 7 Enterprise	GPU	×
6	AMD 2.9-1	ATI Radeon HD 6570 650MHz	Catalyst 14.9	1.2	Windows 7 Enterprise	GPU	×
7	Intel 4.6	Intel HD Graphics 4600	10.18.10.3960	1.2	Windows 7 Enterprise	GPU	×
8	Intel 4.6	Intel HD Graphics 4000	10.18.10.3412	1.2	Windows 8.1 Pro	GPU	×
9	Anon. SDK 1	Anon. device 1	Anon. driver 1c	1.1	Linux (anon. version)	GPU	✓
10	Anon. SDK 1	Anon. device 1	Anon. driver 1b	1.1	Linux (anon. version)	GPU	×
11	Anon. SDK 1	Anon. device 1	Anon. driver 1a	1.1	Linux (anon. version)	GPU	×
12	Intel 4.6	Intel Core i7-4770 @ 3.40 GHz	4.6.0.92	2.0	Windows 7 Enterprise	CPU	✓
13	Intel 4.6	Intel Core i7-4770 @ 3.40 GHz	4.2.0.76	1.2	Windows 7 Enterprise	CPU	✓
14	Intel 4.6	Intel Core i5-3317U @ 1.70 GHz	3.0.1.10878	1.2	Windows 8.1 Pro	CPU	✓
15	Intel XE 2013 R20	Intel Xeon X5650 @ 2.67GHz	1.2 build 56860	1.2	RHEL Server 6.5	CPU	✓
16	AMD 2.9-1	Intel Xeon E5-2609 v2 @ 2.50GHz	Catalyst 14.9	1.2	Windows 7 Enterprise	CPU	×
17	Anon. SDK 2	Anon. device 2	Anon. driver 2	1.1	Linux (anon. verson)	CPU	×
18	Intel XE 2013 R2	Intel Xeon Phi	5889-14	1.2	RHEL Server 6.5	Accelerator	×
19	Intel 4.6	Oclgrind v14.5	LLVM 3.2, SPIR 1.2	1.2	Ubuntu 14.04	Emulator	✓
20	Altera 14.0	Altera PCIe-385N_D5 (Emulated)	aoc 14.0 build 200	1.0	CentOS 6.5	Emulator	×
21	Altera 14.0	Altera PCIe-385N_D5	aoc 14.0 build 200	1.0	CentOS 6.5	FPGA	×

(OpenCL-capable device, OpenCL device driver) pair, since the OpenCL C compiler for a given device is embedded in the driver software for the device. The table also shows the OpenCL version that is supported, and the operating system used for testing.

GPUs Configurations 1–11 cover nine distinct GPU devices, from NVIDIA (1–4), AMD (5–6) and Intel (7–8), and from a vendors we anonymise due to confidentiality agreements (9–11; these comprise a single device tested with three different driver versions).

CPUs The devices for configurations 12–16 are multi-core Intel CPUs; 12–15 use Intel drivers and 16 uses AMD drivers (so that AMD's OpenCL compiler is under test). The device for configuration 17 is a multi-core CPU from a vendor that we again anonymise.

Misc The remaining configurations consist of the Intel Xeon Phi co-processor (18), Oclgrind[3] [15], an open source platform-independent emulator (19), an emulator for an Altera FPGA (20), and the associated FPGA device (21).

This represents the hardware that was available to us at Imperial College, and spans a wide range of OpenCL-capable devices.

2.2 Lifting Compiler Fuzzing to OpenCL

Many-core random differential testing To lift random differential testing to the many-core setting we have built a tool, CLsmith, for generation of random OpenCL kernels, based on the Csmith C program generator [20]. CLsmith includes six modes. BASIC and VECTOR mode yield "embarrassingly parallel" OpenCL kernels in which threads do not communicate; VECTOR mode exercises the rich set of vector data types and operations available in OpenCL. BARRIER, ATOMIC SECTION and ATOMIC REDUCTION modes use novel strategies for enabling deterministic inter-thread communication. The ALL mode encompasses all of the above. The design of CLsmith and details of these modes is provided in §4.

Many-core EMI testing EMI testing involves fuzzing over statements in a program that are determined, via code-coverage analysis, to be dynamically unreachable for a given input. Finding such dynamically-dead code in OpenCL applications is hindered by (a) the lack of available code-coverage tools for OpenCL, and (b) the fact that dynamically-dead code is rare in practical OpenCL kernels: a recent study of 605 kernels shows that only a handful exhibit input-dependent behaviour [2], a necessary (but not sufficient) condition for dynamically-dead code. Problem (a) could be overcome with non-trivial engineering effort (e.g. we could extend Oclgrind with code-coverage capabilities), but problem (b) is fundamental. To circumvent both problems we investigate injecting *dead-by-construction* code into OpenCL kernels (§5).

[3] `https://github.com/jrprice/Oclgrind`

2.3 Experimental Method

Classifying configurations We assessed each configuration using a set of 600 test kernels randomly generated by CLsmith, with respect to a *reliability threshold*. We detail this threshold in full in §7.1, but in summary: for a configuration to lie above the threshold we required that no more than 25% of the 600 tests should fail (either with a build failure, runtime crash or wrong code result) when executed on the configuration. The classification for each configuration is given by the final column of Table 1.

EMI testing using standard OpenCL benchmarks The configurations below our reliability threshold suffer from defects typically related to the use of complex structs (see §7.1 for a full discussion). As structs are fundamental to the CLsmith approach (see §4.1) it was clear we would not gain deeper insights into these configurations through more intensive CLsmith-based testing. However, our assumption was that these configurations should still be capable of correctly compiling standard OpenCL applications and variations thereof. We thus applied EMI testing with injection of dead-by-construction code to all configurations using 10 benchmarks from the widely-used Parboil [18] and Rodinia [3] suites (§7.2).

Intensive CLsmith-based testing Focusing on the configurations lying above the reliability threshold, we conducted CLsmith-based testing at a larger scale (§7.3), testing large sets of random kernels generated by each of the six modes provided by CLsmith. The purpose of this experiment was (1) to validate prior work [20] by assessing the effectiveness of random differential testing in a new domain, and (2) to evaluate the effectiveness, in terms of bug-finding ability, of our CLsmith modes.

Intensive EMI testing with random programs We also performed large-scale EMI testing on the configurations lying above the reliability threshold, using CLsmith-generated kernels with injected dead-by-construction code (§7.4). The purpose of this experiment was (1) to validate in a new domain the claim of prior work [12] that mutating dynamically-dead code is an effective mechanism for finding compiler bugs, and (2) to compare the effectiveness of the EMI testing with random differential testing in terms of bug-finding ability.

2.4 Summary of Findings

Many OpenCL implementations are not robust Many of the configurations we tested exhibit basic compiler bugs (see Figure 1), miscompilations caused machine crashes for some configurations, and we found that some configurations are incapable of running standard OpenCL benchmarks. These issues undermine the portability aim of the OpenCL effort.

Fuzz testing is effective in the OpenCL domain Both random differential and EMI testing revealed significant numbers of defects in the OpenCL implementations we tested. In §6 we show and discuss a selection of these bugs.

We did not find communication-related bugs Our methods for generating deterministic, communicating kernels led to the discovery of compiler bugs that only manifest in the presence of barrier synchronization (see e.g. Figures 1(f), 1(d), 2(c) and 2(d)). However, none of these bugs were inherently communication-related. We did not find bug-inducing tests that relied on atomic operations.

EMI testing with existing code can be challenging We wasted significant effort trying to reduce kernels from two standard benchmarks (Parboil spmv and Rodinia myocyte) until we found that result differences were arising due to previously unidentified data races. This emphasizes the point that compiler fuzzing requires deterministic, well-defined programs; real-world examples often do not have this property. We reported these bugs to the Parboil and Rodinia developers, and both have been confirmed.

3. Background

3.1 The OpenCL Programming Model

OpenCL [10] allows an application running on a *host* to offload computation to one or more parallel *devices*. Offloading is achieved by expressing the computation to be accelerated as a *kernel*: a function, parameterised by a thread identifier, that will be executed simultaneously across the processing elements of a device. Kernels are written in OpenCL C, a restricted version of C99 equipped with a variety of extensions, some of which are summarised below. At runtime, the host application uses an API to identify the set of available devices and compiles a given kernel for a selected device. Thus compilation occurs *online*: the OpenCL driver for a device includes an OpenCL C compiler for that device.

Threads and groups An OpenCL kernel is executed by an *NDRange* (*N-Dimensional range*) of *work-items*, which we henceforth refer to as *threads* for brevity. The standard requires a minimum of three dimensions to be supported, and most devices do not support 4D or higher-dimensional kernels in practice, so we assume hereafter that all kernels are 3D (1D and 2D kernels can be viewed as degenerate 3D kernels). We use $\vec{v}$ to denote a 3D vector (v_x, v_y, v_z). Letting $\vec{N}$ denote the kernel dimensions, each thread has a distinct 3D global id $\vec{t}$ ($0 \leq t_i < N_i, i \in \{x, y, z\}$). The threads are organised into a 3D grid of *work-groups* (henceforth referred to as *groups* for brevity) with dimension $\vec{W}$, where $\vec{W}$ divides $\vec{N}$ component-wise. A thread may access the id $\vec{g}$ of the group to which it belongs ($0 \leq g_i < N_i/W_i, i \in \{x, y, z\}$), and a *local id* $\vec{l}$ within its group ($0 \leq l_i < W_i, i \in \{x, y, z\}$). Global, group and local ids are related: $\vec{t} = \vec{g} \cdot \vec{W} + \vec{l}$.

The global id $\vec{t}$ can be linearized to give a *global linear id* $t_{linear} = (t_z \cdot N_y + t_y) \cdot N_x + t_x$. Linear group and local ids, g_{linear} and l_{linear}, are defined similarly. The linear size of a group is defined as $W_{linear} = W_x \cdot W_y \cdot W_z$, and the linear total number of threads as $N_{linear} = N_x \cdot N_y \cdot N_z$.

Memory spaces Data in an OpenCL kernel resides in one of four *memory spaces*: *global* and *constant* memory are shared among all threads, with constant memory being read-only; each group has access to a separate *local* memory shared between all threads in the group; every thread has a separate *private* memory. The **`global`**, **`constant`**, **`local`** and **`private`** qualifiers are used to specify memory spaces on data, with **`private`** being the default.

When we say that a location is in *shared* memory we mean that the location is either in local or global memory.

Vector types and operators OpenCL provides a rich set of vector types and operations on integer data. Signed and unsigned **`char`**, **`short`**, **`int`** and **`long`** vectors can be declared with lengths 2, 4, 8 and 16.[4] The C arithmetic and logical operations all apply component-wise to vectors. A rich set of additional vector operations is also provided. We discuss two built-in vector operations, `clamp` and `rotate`, that we shall refer to later. Applied to scalar integers x, min, max, with $min \leq max$, `clamp(`x`,` min`,` max`)` returns min if $x < min$, max if $x > max$, and x otherwise. Applied to scalar integers x and y, `rotate(`x`,` y`)` returns the integer obtained by left-shifting the bits of x by y places; bits shifted off the left of x are shifted back in from the right. Both operations are lifted component-wise to vectors.

Barriers, atomic operations and data races OpenCL 1.x offers no mechanism for synchronization between threads in different groups during the execution of a kernel. Threads within a work group can synchronize by executing a collective *barrier* operation: on reaching a barrier a thread must wait until all threads in the group arrive at the same syntactic barrier, after which the group can proceed beyond the barrier. A barrier accepts a *fence* argument specifying the consistency guarantee that should be provided on leaving the barrier: consistency over global, local, or both global and local memory spaces can be requested.

Atomic read-modify-write operations also allow intra-group communication. These include *exchange* and *compare-and-exchange*, plus a number of arithmetic and bitwise operations.

A *data race* occurs between two distinct threads that access a common memory location if at least one thread modifies the location, and either: the threads are in different groups, or the threads are in the same group, at least one of the accesses is non-atomic, and the accesses are not separated by a barrier synchronization. *Barrier divergence* occurs if two threads in the same group arrive at distinct barriers, or arrive at a barrier inside a loop nest having executed different numbers of iterations of the enclosing loops.

Undefined and implementation-defined behaviour A large set of undefined behaviours are inherited in OpenCL from C99 [9]. Additionally, data races and barrier divergence are considered undefined behaviours, and some of the vector operations specify new undefined behaviours. For example, `clamp(`x`,` min`,` max`)` yields undefined behaviour if some component of min is larger than the corresponding component of max. There are fewer sources of implementation-defined behaviour: in particular, the widths of primitive data types are fixed and a two's complement representation is mandated for signed integers, so that, for example, **`int`** denotes a 32-bit two's complement signed integer. A two's complement representation means that bit-level operations, such as the `rotate` function discussed above, are well-defined on signed data.

Notably, whether irreducible control flow [1] is allowed in OpenCL is implementation-defined. This means that kernels that exhibit irreducible control flow are not portable.

3.2 Compiler Fuzzing

Compiler testing is hindered by the *oracle* problem: determining whether a compiler correctly processes an input program requires knowledge of how the input program should behave. The methods

[4] OpenCL 1.1 and higher also support vectors of length 3.

we study here, random differential testing and EMI testing, use majority voting to circumvent the oracle problem by exploiting the fact that a deterministic program should always yield a unique, well-defined result. The following definition captures this requirement (it is intended as a guideline and is necessarily imprecise because it does not refer to a specific programming language):

DEFINITION 1 (Program with deterministic output). *Program P produces* deterministic output *for input I if, when executed on I, P exhibits no undefined or implementation-defined behaviour, terminates, and prints a string s that is uniquely determined by I.*

Random differential testing Csmith [20] represents the current state-of-the-art using random differential testing [13, 17]. Csmith generates random C programs that take no input and are guaranteed to produce deterministic output (Definition 1), except that Csmith can be configured to allow implementation-defined behaviour. The oracle problem is pragmatically circumvented by comparing the results obtained for a program using multiple compilers, or the same compiler configured with different optimization settings, assuming that the majority result (if one exists) is the correct one. Deviations from the majority likely indicate miscompilations which can be investigated to identify compiler bugs.

Equivalence modulo inputs testing A limitation of random differential testing is that it requires multiple compilers, or at least multiple optimization settings for a single compiler. The OpenCL standard exposes a single such setting: optimizations can be on (default) or off. Instead of avoiding the oracle problem through multiple compilers, *equivalence modulo inputs* (EMI testing) [12] employs *metamorphic* testing [4], whereby defects are identified by observing that a system behaves differently on test inputs that should guarantee identical outputs. In particular, one compiler is tested against multiple programs that should all produce the same deterministic output for a particular input. Deviations between programs compiled with a *single* compiler indicate miscompilations.

Suppose a program P produces deterministic output (Definition 1) on input I and further that P exhibits no internal nondeterminism when executed on I. Suppose a (possibly compound) statement s is found to be unreachable when P is executed on I; s is said to be *I-dead*. Let $Q = P[s'/s]$ be the program obtained by replacing s with a different (possibly compound, and possibly empty) statement s'. If Q type-checks then clearly Q should produce the same deterministic output as P when executed on I; P and Q are said to be *equivalent modulo the input I*.

This leads to the following strategy for compiler testing, which we refer to as *EMI testing*: given a program P with a test input I, profile P on I to identify I-dead statements. Then, for some $N > 0$, derive N variants of P—$Q_1, \ldots, Q_N$ say—by substituting I-dead statements with alternative statements. Compile $P, Q_1, \ldots, Q_N$ and execute each program on I; result mismatches indicate miscompilations. The Orion tool implements EMI testing and has uncovered numerous bugs in GCC and LLVM using regression tests and Csmith-generated programs as source programs [12]. The authors argue that the method is effective because it induces subtle changes to the control flow graph of a program that can trip up incorrectly implemented optimizations or incompatible optimization passes.

4. Random Differential Testing for OpenCL

We have built a tool, CLsmith, which builds on Csmith [20] to generate random OpenCL kernels that produce deterministic output. We first explain how the Csmith approach can be lifted to OpenCL to yield "embarrassingly parallel" kernels where threads do not communicate (§4.1). We then discuss the design of three strategies for enabling deterministic communication between threads (§4.2).

4.1 Embarrassingly Parallel Random Kernels

BASIC *mode: lifting Csmith to OpenCL* In BASIC mode our CLsmith tool generates an OpenCL kernel whose body is an adapted Csmith-generated program. Every thread executes this program to compute a numeric result. These computations are independent, and each thread writes its result to index t_{linear} of a designated global memory array, `out`. The final values of `out` are subsequently printed as a comma-separated list.

Because OpenCL 1.x does not support variables declared at global program scope we had to modify Csmith to declare a struct with one field for each would-be-global variable, initialize an instance of this struct on kernel entry and pass the struct by reference to every function. A consequence of this *globals struct* is that CLsmith-generated programs depend critically on accurate compilation of structs, and are thus biased towards identifying struct-related miscompilations. As we discuss in §7.1 and illustrate in Figure 1, we found fundamental problems related to the compilation of structs in several of the configurations we tested (Table 1).

When leveraging Csmith, CLsmith disables bit-fields which are illegal in OpenCL. We also ascertained that programs generated by Csmith have reducible control flow graphs; whether irreducibility is supported is implementation-defined in OpenCL (see §3.1).

VECTOR *mode: supporting OpenCL vectors and built-ins* CLsmith extends Csmith with the capability of generating variables and expressions with vector types, exercising the rich set of vector operations available in OpenCL (see §3.1). This extension was non-trivial to implement because the standard Csmith tool exploits the fact that the C type system allows arbitrary coercions between integer data types. This is not the case for OpenCL vectors, for instance it is not possible to cast an **`int4`** (4D **`int`** vector) to a **`short4`** or even a **`uint4`**. Thus we had to provide support for type-sensitive vector expression generation. To avoid undefined behaviours arising from vector computations we designed a set of "safe math" vector macros, following the approach used by Csmith for scalar operations [20]. For instance, instead of directly issuing a `clamp` operation, CLsmith instead generates an invocation of our `safe_clamp` macro, where `safe_clamp(`x`,` min`,` max`)` expands to `(`min `>` max `? `x` : clamp(`x`,` min`,` max`))`.

Randomizing grid and group dimensions To test a diverse range of thread arrangements, CLsmith randomly selects a total thread count in the range $[100, 10000)$ and then chooses random divisors of this thread count to select appropriate values for $\vec{N}$ and $\vec{W}$ (see §3.1). Kernels with dimension 1 and 2 are in effect generated if size 1 is selected for one or more dimensions. The maximum work-group size supported by all the configurations of Table 1 is 256, thus we constrained $\vec{W}$ such that $W_x \cdot W_y \cdot W_z \leq 256$.

4.2 Deterministic, Communicating Random Kernels

We present three methods for generating random OpenCL kernels that exhibit deterministic *intra*-group communication using barriers and atomic operations. We restrict to intra-group communication because the OpenCL 1.x specifications provide no inter-group memory consistency guarantees [11, p. 29].

BARRIER *mode* In this mode, threads in a group communicate via shared arrays, synchronizing using barriers to avoid data races. A kernel is equipped with a 2D array `permutations` of size $d \times W_{linear}$, for some small d, such that `permutations[`i`]` is a randomly selected permutation of the set $\{0, \ldots, W_{linear}-1\}$ for each $0 \leq i < d$; we use $d = 10$ in practice. Each group is also equipped with a shared memory array `A` of type `uint` and length W_{linear}, initialized with a uniform value (we use the value 1 in practice). The array is allocated either in local or global memory; the choice is random. Each thread has a private variable `A_offset`, of type

uint, initialized to permutations[rnd][l_{linear}], where rnd is a literal value selected randomly during program generation, with $0 \leq rnd < d$ (rnd is uniform across threads). Thus A_offset initially provides each thread with a distinct offset into A. At random points in the kernel the threads synchronize using a barrier and then reset A_offset using a randomly chosen permutation. The generated code for the ith such synchronization point to be generated has the form:

```
barrier(FENCE);
A_offset = permutations[rnd_i][l_linear];
```

where rnd_i is a literal value selected randomly during program generation, with $0 \leq rnd_i < d$, and FENCE is a global or local memory fence depending on the memory space in which A is allocated. The index rnd_i is uniform across threads so that every thread accesses the same permutation. This randomly re-distributes ownership of elements of A among the threads.

This allows CLsmith to randomly emit reads from and writes to A[A_offset] in the kernel; the use of a barrier before ownership re-distribution ensures these accesses will not lead to data races. Because only barriers are used for synchronization, race-freedom ensures that this communication mechanism yields deterministic results (see [6] for a proof of this general result).

ATOMIC SECTION *mode* In this mode, CLsmith generates *atomic sections*; the ith generated section has the following form:

```
if(atomic_inc(c) == rnd_i) {
  /* statements */
  atomic_add(s, hash);
}
```

where rnd_i is a literal value, uniform across threads, chosen randomly during program generation, c points to a volatile uint *counter* in shared memory, and s points to a volatile uint *special value* associated with the counter, also in shared memory. Each group has a separate copy of c and s so that there is no interaction between groups. The idea is that *only* the rnd_i-th thread to increment c enters the conditional; which thread this is (if any) depends on the order in which threads are scheduled. If a thread does enter the conditional the thread executes statements and then increments s by a hash of the results of this computation (indicated by hash). The hash is computed by summing the values of all variables declared immediately inside the atomic section. At the end of kernel execution the thread with $l_{linear} = 0$ incorporates the value of special value s into its final result, on behalf of the group.

To ensure determinism, assignments in statements are restricted to only modify data declared *inside* the atomic section. This ensures that the local state of the thread that executes an atomic section is the same on exit from the section as it was on entry to the section. Similarly, an atomic section should not contain jumps (via **return**, **break**, **continue** and **goto**) that allow execution to leave the section; these would cause the thread that executed the section to deviate from the behaviour of other threads by following a different control path. As well as leading to nondeterministic result differences between threads, this issue could lead to barrier divergence if atomic sections are combined with other modes that issue barriers. To aid in ensuring these guarantees, atomic sections are restricted so that they do not issue function calls, though in principle they could issue calls to functions that have been constructed to satisfy the guarantees required by atomic sections. Our hypothesis was that atomic sections might provoke compiler bugs that break the determinism guarantee CLsmith attempts to enforce.

In practice each group is equipped with arrays containing a randomly chosen number of counters and special values (between 1 and 99 in practice). Each atomic section uses a randomly selected (counter, special value) pair.

ATOMIC REDUCTION *mode* In this mode, threads within a group randomly perform a reduction into a designated volatile shared memory location, r, of type uint, using one of the commutative and associative arithmetic and bitwise atomic operations provided by OpenCL: add, min, max, or, and and xor. After the reduction the threads synchronize via a barrier, the thread with $l_{linear} = 0$ adds the result of the reduction to a running total, and the threads synchronize again so that the location r can be re-used in further reductions without inducing data races. If p is a pointer to the location r then the form of the ith atomic reduction to be generated is:

```
atomic_op_i(p, expr_i);
barrier(FENCE);
if(l_linear == 0) { total += *p; }
barrier(FENCE);
```

Here op_i is one of the above operators and $expr_i$ is a randomly generated expression. Because each of the atomic operations we consider are commutative and associative, the order in which threads participate in the reduction does not affect the result, thus determinism is guaranteed.

Avoiding barrier divergence The BARRIER and ATOMIC REDUCTION modes both generate barrier synchronization operations. To avoid *barrier divergence* (see §3.1) it is essential that barriers do not appear in a context where threads in the same group may follow divergent control flow paths. We ensure this by universally prohibiting CLsmith from generating thread global or local ids, t_i, l_i, for $i \in \{x, y, z, linear\}$ in expressions, and by initializing the array A uniformly with a single value. These restrictions make it impossible for the identity of a thread to influence the control flow taken by the thread during execution.

5. EMI Testing for OpenCL

As discussed in §2.2, direct EMI testing (as per [12]) for OpenCL is hindered principally by the scarcity of dynamically-dead code in practical kernels. We overcome this by injecting *dead-by-construction* code into kernels. We equip a kernel with an additional array parameter, dead of length d (for some small d) and randomly insert into the kernel a number of *EMI blocks*, where the ith EMI block to be generated has the following form:

```
if(dead[rnd_i,1] < dead[rnd_i,2]) {
  /* statements */
}
```

where $rnd_{i,1}$ and $rnd_{i,2}$ are literal values randomly selected during program generation such that $0 \leq rnd_{i,2} < rnd_{i,1} < d$. The OpenCL compiler knows nothing about the runtime values of elements of dead. We also modify the host application to initialize dead so that dead[j] $= j$ ($0 \leq j < d$). This means that, by construction, the statements inside the EMI block are *dynamically unreachable*. If the original kernel produces deterministic output, any variation of the kernel injected with an EMI block should produce the same deterministic output.

Dead-by-construction code in CLsmith We extended CLsmith with an option to equip the generated kernel with a dead array and a number of randomly generated and randomly placed EMI blocks. Variants of the kernel are then produced by *pruning* the EMI blocks according to a set of configurable probabilities. This strategy is the same as the one employed by the original EMI work [12]. We consider each EMI block as an abstract syntax tree (AST), such that non-compound statements (e.g. assignment and **break** statements) are *leaf* nodes and compound statements (**if** and **for** statements) are *branch* nodes. At each node a series of prunings

```
struct S { char a; short b; };

kernel void k(global ulong *out) {
  struct S s = { 1, 1 }; out[t_linear] = s.a + s.b;
}
```

(a) Configs. 5+, 6+, 16+ yield result 1 (expected: 2)

```
typedef struct {
  short a; int b; volatile char c;
  int d; int e; short f[10];
} S;

kernel void k(global ulong *out) {
  S s; S* p = &s;
  S t = {0,0,0,0,0, {0,0,0,0,0,0,0,0,1,0,0}};
  s = t; out[t_linear] = p->f[7];
}
```

(b) Configs. 10−, 11− yield result 0 (expected: 1)

```
kernel void k() {
  struct S { int4 x; };
  struct S s = { (int4)((int2)(1, 1), 1, 1) };
}
```

(c) Configs. 20±, 21± yield internal errors when vectors appear in structs

```
typedef struct { int x; int y; } S;
void f(S *p) { p->x = 2; }

kernel void k(global ulong *out) {
  S s = { 1, 1 }; barrier();
  f(&s); out[t_linear] = s.x + s.y;
}
```

(d) Configs. 17± yield result 2 (expected result: 3)

```
kernel void k(global int *p) {
  for(int i=0; i < 197; i++) if(*p) while(1) { }
}
```

(e) Configs. 8±, 7± enter an infinite loop during compilation of this kernel

```
typedef struct { int a; int *b; ulong c[9][9][3]; } S;

kernel void k(global ulong *out) {
  S s; S* p = &s; S t = { 0, &p->a, { 0 } }; s = t;
  barrier();
  out[t_linear] = p->c[0][0][1];
}
```

(f) Config. 18+ takes more than 20s to compile this kernel

Figure 1. Kernels illustrating compiler bugs for the configurations that lie below the reliability threshold used for our study

are considered. We reproduce two pruning strategies from [12], *leaf* and *compound*, and propose a further strategy, *lift*. The *leaf* pruning deletes a leaf node with probability p_{leaf}; the *compound* pruning deletes a branch node with probability $p_{compound}$.

Our new *lift* pruning has associated probability p_{lift}, and promotes the children of a branch node to become children of the parent of the branch node, after which the branch node is removed. Lifting transforms a conditional with *then* block S and *else* block T into the sequence $S; T$, and a for loop with initializer S and body T into the sequence $S; T'$, where T' is identical to T except that outermost **break** and **continue** statements are removed (this ensures that the code is syntactically valid after lifting).

Because *compound* and *lift* are not independent (they can both remove branch nodes), and because *compound* is applied before *lift*, the actual probability of lifting will be $(1 - p_{compound}) \cdot p_{lift}$, therefore we perform lifting with the adjusted probability $p'_{lift} = p_{lift}/(1 - p_{compound})$; this necessitates enforcing $p_{compound} + p_{lift} \leq 1$ to ensure $p'_{lift} \leq 1$.

Injecting into real-world kernels To inject EMI blocks into existing OpenCL kernels we use CLsmith to generate a selection of EMI block variants using the generation and pruning strategies described above. To place such a block into an existing kernel we need to account for *free* variables that are used inside the EMI block. The free variables can either be defined at the start of the block, or can be renamed via a *substitution* to take the names of variables in the original kernel (using the **#define** construct). Our hypothesis was that using substitutions would make EMI blocks more effective at provoking compiler bugs: they mean that computations described inside and outside the block operate on common data, giving the compiler the opportunity to optimize (possibly erroneously) across the block boundary. We evaluate this hypothesis in §7.2.

6. Example Bugs

We present details of various compiler bugs we discovered through our study. A selection of bugs related to the configurations that fell below and above our reliability threshold are summarised in Figures 1 and 2, respectively, and discussed below. For brevity, we write **barrier**() for a barrier equipped with a local memory fence, omitting the CLK_LOCAL_MEM_FENCE flag.

OpenCL kernels are compiled with optimizations enabled by default, and a -cl-opt-disable flag may be passed to turn optimizations off. Throughout the discussion, if i is a configuration id we use $i+$ and $i-$ to denote the configuration with optimizations enabled and disabled, respectively, and $i\pm$ to denote both cases.

Front-end issues An early version of CLsmith generated unintentionally ambiguous vector expressions. One example was the expression (**int2**)(1,2).y, the intent of which was to access the y component of a 2D vector. Some compilers accepted this expression, interpreting the expression as ((**int2**)(1,2)).y (which is what we intended, and what CLsmith will now produce); other compilers rejected the expression, interpreting it as (**int2**)((1,2).y) which would clearly be wrong. We do not find the description of operator precedence rules in the OpenCL specification clear in this situation. We find it interesting that our accidental generation of ambiguous, possibly erroneous, expressions led to the identification of compiler front-end mismatches; this suggests that it might be interesting to investigate fuzzing techniques that specifically target grey areas in a language specification.

Many tests initially failed on the Altera configurations (20 and 21) due to the front-end rejecting logical operations on vectors; conformant OpenCL implementations *are* required to support these operations. To work around this, to enable deeper testing, we adapted CLsmith to wrap vector logical operations in macros, provided Altera-specific implementations of these macros to avoid the bug, and repeated *initial* testing.

Machine crashes We had difficulty testing with our AMD and Intel GPUs (configurations 5, 6, 7, 8) because kernel execution would occasionally, and unpredictably, crash the OS of the host machine. We were able to mitigate this to some extent (but not entirely) for the AMD GPUs by ensuring that the GPU under test was not being used simultaneously for graphics processing. This was not possible for the Intel GPUs: we found that disabling their use for graphics processing also made them inaccessible via the OpenCL API. Unpredictable machine crashes make batch testing, and thus intensive fuzz testing, infeasible. It is also interesting and potentially worrying that erroneously-compiled OpenCL kernels can bring down a system.

Problems with structs The *initial* testing revealed severe bugs related to struct compilation for several configurations.

The AMD GPU and CPU configurations (5, 6, 16) produce wrong results (regardless of thread count) with optimizations for the trivial kernel of Figure 1(a); more generally these configurations appear to miscompile any struct that starts with **char** followed by a larger member. We reported this, and a similar bug related to struct padding, to AMD in May 2014 and both were confirmed. The padding bug, but not the bug of Figure 1(a), was **fixed**

```
struct S { short c; long d; };
union  U { uint  a; struct S b; };
struct T { union U u[1]; ulong x; ulong y; };

kernel void k(global ulong *out, global int *in) {
  struct T c;
  struct T t = { {{1}}, in[t_x], in[t_y] };
  c = t;
  ulong total = 0;
  for(int i = 0; i < 1; i++) total += c.u[i].a;
  out[t_linear] = total;
}
```

(a) Configs. 1−, 2−, 3−, 4− yield 0xffff0001 due to incorrect union initialization (expected: 1)

```
kernel void k(global ulong *out) {
  out[t_linear] = rotate((uint2)(1, 1), (uint2)(0, 0)).x;
}
```

(b) Config. 14± yields result 0xffffffff (expected: 1)

```
int f();
void k(int *p) { barrier(); *p = f(); }
void h(int *p) { k(p); }
int f() { barrier(); return 1; }

kernel void k(global ulong *out) {
  int x = 0; h(&x); out[t_linear] = x;
}
```

(c) Configs. 12−, 13− yield result [1,0] when executed by two threads in the same group (expected result: [1,1]). Configs. 14−,15− crash with a segmentation fault.

```
typedef struct { int a; int * volatile * b; int c; } S;

void f(S *s) {
  for (s->a = 0; (s->a > 0); s->a = 0) {
    int x = 1; int *p = &s->c; // loop body is
    barrier();                 // unreachable
    // complex expression over x, p and s (not shown)
} }

kernel void k(global ulong *out) {
  S s = { 1, 0, 0 }; f(&s); out[t_linear] = s.a;
}
```

(d) Configs. 14−, 15− yield result [0,1] when executed by two threads in the same group (expected result: [0,0])

```
void f(int *p) {
  if((((((*p - g_x) != 1) >> *p) < 2) >= *p)) { *p = 1; }
}

kernel void k(global ulong *out) {
  int x = 0; f(&x); out[t_linear] = x;
}
```

(e) Config. 9+ yields result 0 (expected: 1)

```
kernel void k(global ulong *out) {
  short x = 1; uint y;
  for(y = -1; y >= 1; ++y) { if(x , 1) break; }
  out[t_linear] = y;
}
```

(f) Config. 19± yields result 0 (expected: 0xffffffff)

Figure 2. Kernels illustrating compiler bugs for the configurations that lie above the reliability threshold used for our study

in the Catalyst 14.9 drivers we used for testing. However, the bug of Figure 1(a) has since been **fixed** in the Catalyst 14.12 drivers.

Two of the anonymized GPU configurations (10, 11) miscompile the kernel of Figure 1(b) when optimizations are *disabled* and, curiously, only if $N_x = 1$. This shows the value of randomizing group dimensions. We reported this bug to the vendor who confirmed they can reproduce it on their trunk build; they noted that they rarely test their implementation with optimizations disabled. Configuration 9 includes a fix for this, among other bugs we reported; these bug fixes bring configuration 9 above our reliability threshold, compared with configurations 10 and 11.

Kernels that use vectors in structs (such as in Figure 1(c)) produce LLVM IR generation errors when compiled by the Altera configurations (20, 21); we reported this issue to Altera. Figure 1(d) illustrates a struct-related miscompilation for configuration 17 (confirmed by the anonymized vendor); the **barrier** is required for the bug to manifest. We also identified struct-related miscompilations for the Intel HD Graphics 4600 and 4000 configurations (7,8).

Compilation for the Xeon Phi co-processor (configuration 18) is prohibitively slow when relatively large structs are used with optimizations enabled. The host for our Xeon Phi card (a 2.0 GHz Intel Xeon CPU) takes more than 20 seconds to compile a kernel summarised in Figure 1(f) when targeting the Xeon Phi with optimizations; in contrast the kernel is compiled in less than 0.5 seconds when targeting the CPU. We do not observe this problem for kernels without structs or with small structs. Compilation speed becomes regular if the **barrier** is removed. From a usability perspective, we regard the prohibitively slow, struct- and barrier-dependent compilation that we have observed, as a bug.

Figure 2(a) is a struct-related bug associated with the NVIDIA configurations, which lie above our reliability threshold. Without optimizations, configurations 1−, 2−, 3− and 4− incorrectly initialize the `u` field of the `T` struct in the declaration of `t`. The value 1 provided for the single-element array `u` should initialize the first field, `a`, a 4-byte unsigned integer. Inspecting the PTX code produced for configuration 1− we find that only the first two bytes of the union are initialized (corresponding to the `c` field of the `S` struct, which is the `b` field of the union). The other two bytes are left uninitialized, so the expression `c.u[i].a` reads garbage when computing `total`. NVIDIA confirmed our report of this bug.

A vector-related bug Intel configuration 14± miscompiles the example of Figure 2(b). Rotating the vector $(1, 1)$ by the vector $(0, 0)$ should have no effect, yielding the vector $(1, 1)$ whose x component is 1 (see §3.1 for a description of the `rotate` vector builtin). Inspecting the assembly code generated by the compiler for configuration 14, we find that the value of this x component has been incorrectly constant-folded to 0xffffffff. The bug is not present in the more recent drivers associated with configurations 12 and 13, nor in the older driver associated with configuration 15.

Bugs related to barriers The struct-related compiler issues of Figures 1(f) and 1(d), discussed above, both require the presence of barriers to manifest. We discuss two further miscompilations that involve barriers and arise in relation to Intel configurations.

The example of Figure 2(c) is miscompiled by Intel configurations 12− and 13−; two threads in the same group expose the issue. The barriers play no role in protecting shared data, yet their presence is necessary for miscompilation to occur. The forward-declaration of `f` is also necessary, and we inline any function in the source code the example is no longer miscompiled. Enabling optimizations (which perhaps forces inlining) also yields the correct result. The kernel leads to segmentation faults with configurations 14− and 15−. We have reported these issues to Intel.

The kernel of Figure 2(d) is miscompiled by Intel configurations 14− and 15−. The kernel involves a complex expression that we do not show here, but which is available as part of our online material. Notice that the body of the loop in `f` is not reachable. Nevertheless, removing the barrier from the loop body causes the kernel to yield the correct result. Moving the declarations at the start of the loop body past the barrier causes the kernel to crash at runtime, as does simplification of the complex expression after the barrier. Enabling optimizations causes the correct result to be produced, presumably because the optimizer identifies the body of the loop as dead code. The bug is not present in the more recent drivers associated with configurations 12 and 13.

Other basic wrong code bugs The kernel of Figure 2(e) is miscompiled by configuration 9+. When executed by a single group consisting of a single thread, careful inspection of the conditional guard shows that it evaluates to *true*, because `*p` and g_x (the first dimension of group id) are both 0, hence the value 1 should be produced. It is interesting that this bug requires the presence of the global id g_x; if the literal 0 is used explicitly instead the problem does not manifest. The vendor associated with configuration 9 have confirmed our report of this bug.

Figure 2(f) shows a kernel for which Oclgrind (configuration 19) yields wrong results (with and without optimizations). On reporting this to the Oclgrind developers they confirmed that it was due to mis-handling of the comma operator. The latest Oclgrind release **fixes** this bug, as well as two other bugs we reported and two further bugs we independently discovered but did not report.

Build failures Our testing revealed a number of build failures associated with NVIDIA configurations 1 and 2, yielding LLVM-related internal error messages such as *"Wrong type for attribute zeroext"*, *"Wrong type for attribute signext"* and *"Attributes after last parameter!"*. NVIDIA have **fixed** all the build failure issues we reported in version 346.47 of their OpenCL driver (which is used by configurations 3 and 4).

Testing the Intel CPU configurations led to failures in Intel-specific LLVM optimization passes, including passes named *"Intel OpenCL Barrier"* and *"Intel OpenCL Vectorizer"*, and several other LLVM-related build failures; sample messages associated with configuration 12 include: *"Both operands to ICmp instruction are not of the same type!"*, *"Call parameter type does not match function signature!"* and *"Instruction does not dominate all uses!"*. We have reported these build failures to Intel.

We encountered a very high rate of build failures for configuration 15, because it rejects legal arithmetic expressions that mix the **int** and **size_t** types with certain operators. For example, the code fragment: **int** `x; x |=` g_x `;` (where g_x denotes the x component of a group's id, which has type **size_t**) results in *"error: invalid operands to binary expression ('int' and 'size_t')"*.

Intel GPU configurations 7 and 8 appear to get stuck in an infinite loop when *compiling* the kernel of Figure 1(e). If the **for** loop bound of 197 is reduced to 196 then compilation immediately fails with an internal error.

The AMD GPU configurations, with optimizations enabled (5+, 6+), complain about unsupported irreducible control flow for an example kernel that do not use **goto** or **switch** statements (the only possible sources of irreducibility at the kernel source level). The problem only occurs with optimizations, so we speculate that irreducibility is being introduced during optimization. We reported this issue to AMD. It persists in the Catalyst 14.12 drivers.

For the majority of the tests we tried, the non-emulated Altera FPGA configuration (21) either crashed or emitted an internal error.

7. Our Testing Campaign

We now describe our testing process and results in detail, which uncovered bugs in all configurations. We continue to use the notation $i-$ and $i+$ to denote the configuration i with optimizations disabled and enabled, respectively.

7.1 Initial Testing to Classify Configurations

We tested every configuration of Table 1, with and without optimizations, on a set of 600 kernels randomly generated by CLsmith which we call the *initial* kernels. This set consisted of 100 kernels randomly generated using each of the six modes supported by the generator: from BASIC through to ALL. We used a timeout of 60 seconds for compilation and execution (but excluding the time taken to use CLsmith to generate the kernel). For Oclgrind (configuration 19) we increased the timeout to 300 seconds as we knew the emulator to be slow compared with the other configurations. For the Altera configurations (20, 21) we did not count the lengthy time required for offline compilation in the timeout limit. We then examined mismatches between configurations arising from this testing.

Table 2. OpenCL benchmarks studied using EMI testing

Suite	*Benchmark*	*Description*	*No. of Kernels*	*Lines of Code*	*Uses FP?*
Parboil	bfs	Graph breadth-first search	1	65	×
	cutcp	Molecular modeling simulation	1	98	✓
	lbm	Fluid dynamics simulation	1	139	✓
	sad	Video processing	3	134	×
	spmv	Linear algebra	1	32	✓
	tpacf	Nbody method	1	129	✓
Rodinia	heartwall	Medical imaging	1	1060	✓
	hotspot	Thermal physics simulation	1	89	✓
	myocyte	Medical simulation	1	1050	✓
	pathfinder	Dynamic programming	1	102	×

We set our reliability threshold as follows: for a configuration to lie above the threshold, no more than 25% of the *initial* test results for the configuration (considering results with and without optimizations) should be build failures, runtime crashes or wrong code results (where we judged the latter based on disagreement with the majority result). We also considered the Intel Xeon Phi (configuration 18) to be below the reliability threshold due to the issue of prohibitively slow compilation of structs (see Figure 1(f) and the discussion in §6), which made intensive fuzz testing impractical for this configuration.

7.2 EMI Testing Over the Rodinia and Parboil Suites

Table 2 summarises 10 benchmarks, drawn from the Parboil v2.5 [18] and Rodinia v2.8 [3] suites, that we evaluated using EMI testing. The Parboil and Rodinia suites are mature and widely-used, and each benchmark ships with tests and input data. This benchmark suite is comparable in size to the 11 real-world benchmarks (9 SPECINT, Mathomatic and tcc) used to evaluate EMI testing for C compilers [12]. Table 2 summarises each benchmark, indicating the number of kernels, total lines of kernel code[5] and whether the kernels use floating-point arithmetic. We selected these benchmarks by initially favouring the three benchmarks that do not use floating-point, selecting further benchmarks in decreasing order of kernel code size. We preferred to avoid floating-point kernels because of the possibility that the configurations under test may employ so-called "fast math" optimizations (optimizations that exploit laws of the real numbers that do not hold for floating-point numbers, such as associativity). Such optimizations can change the results computed by a floating-point kernel, and we feared that the triggering of these optimizations might be sensitive to EMI injection. However, we did not find floating point error to be the cause of result mismatches when we manually investigated possible wrong code bugs induced by our EMI injection process.

Preparing benchmarks We wrote a script that processes the kernels of a benchmark and (i) equips the kernel with an additional array parameter `dead` and (ii) randomly chooses one or two EMI *injection points* (we decided on a per-benchmark basis whether to use one or two injection points). For each injection we generated 125 possible EMI block variants using CLsmith, using a combination of the *leaf*, *compound* and *lift* pruning strategies with various probabilities. As described in §5, we must cater for free variables (variables not defined in the EMI block). The script automatically generates a header that either declares the variables locally within the EMI block (substitutions off) or aliases free variables with randomly chosen variables appearing in the original kernel

[5] Using cloc 1.62, `http://cloc.sourceforge.net`

Table 3. Results for EMI testing using the Parboil and Rodinia benchmarks of Table 2 (except myocyte and spmv)

Configuration; IDs as per Table 1; configurations 20 and 21 are excluded due to their reliance on offline compilation

Benchmarks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
bfs	$\mathbf{w}^{\star}$	$\mathbf{w}^{\star}$	$\mathbf{w}^{\star}$	$\mathbf{w}^{\star}$	$\mathbf{c}^{\star}$	**to**	**to**	**to**	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	✓	✓	✓	✓	✓	✓
cutcp	$\mathbf{w}^{e}$	✓	**ng**	$\mathbf{w}^{\star}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{e}$	$\mathbf{w}^{d}$	$\mathbf{w}^{\star}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	$\mathbf{w}^{e}$	$\mathbf{w}^{e}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	**ng**	✓	$\mathbf{c}^{\star}$	**ng**
lbm	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{d}$	$\mathbf{c}^{d}$	**ng**	**ng**	✓	$\mathbf{c}^{e}$	**ng**	**ng**	**ng**	✓	✓	✓	✓	**ng**	**ng**	**to**	**ng**
sad	✓	**ng**	✓	✓	**ng**	**ng**	**ng**	**ng**	**ng**	**ng**	**ng**	✓	✓	✓	✓	**ng**	**ng**	**to**	**ng**
tpacf	✓	✓	✓	✓	$\mathbf{c}^{\star}$	$\mathbf{w}^{d}$	✓	✓	**ng**	**ng**	**ng**	**ng**	**ng**	**ng**	$\mathbf{w}^{e}$	**ng**	**ng**	$\mathbf{c}^{\star}$	**ng**
heartwall	$\mathbf{w}^{d}$	$\mathbf{w}^{d}$	$\mathbf{w}^{d}$	$\mathbf{w}^{d}$	$\mathbf{w}^{\star}$	**ng**	✓	✓	**ng**	**ng**	**ng**	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	**ng**	✓	$\mathbf{c}^{\star}$	**ng**
hotspot	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	✓	✓	$\mathbf{w}^{e}$	$\mathbf{w}^{e}$	$\mathbf{w}^{e}$	$\mathbf{w}^{e}$	**ng**	**ng**	**ng**	$\mathbf{w}^{e}$	$\mathbf{w}^{e}$	$\mathbf{w}^{e}$	$\mathbf{w}^{e}$	$\mathbf{w}^{e}$	✓	$\mathbf{w}^{e}$	**ng**
pathfinder	✓	✓	$\mathbf{c}^{d}$	$\mathbf{c}^{d}$	$\mathbf{c}^{\star}$	$\mathbf{c}^{\star}$	**to**	$\mathbf{c}^{d}$	✓	$\mathbf{c}^{d}$	$\mathbf{c}^{d}$	$\mathbf{c}^{d}$	$\mathbf{c}^{d}$	$\mathbf{c}^{d}$	$\mathbf{c}^{d}$	**ng**	$\mathbf{c}^{d}$	$\mathbf{w}^{e}$	**ng**

using `#define` macros (substitutions on). Some manual tweaking was necessary to ensure well-typed substitutions. We edited the host code of each benchmark to allocate and initialize the `dead` array, and added command line arguments to configure which EMI to be used and to toggle substitutions. It took one of the authors around half a day to prepare each benchmark. Significant effort was also required to enable execution on our Windows platforms.

Results For each benchmark, we considered 500 tests (125 EMI blocks, substitutions on/off and optimizations on/off), each executed with a 100s timeout. We compared the output of each test against the expected output for the benchmark, which we generated by running the benchmark with an empty EMI block. Table 3 summarises the results for 19 of our 21 configurations; we exclude the Altera configurations (20 and 21) because they require offline compilation which was non-trivial to integrate with the Parboil and Rodinia benchmarks. The table excludes myocyte and spmv: as discussed in §2.4 we found data races in these benchmarks. For each configuration we give a single result for each benchmark that summarises the worst outcome observed over its 500 tests (in decreasing order): (**w**) at least one test generated the wrong result without crashing; (**c**) at least one test crashed;[6] (**to**) at least one test timed out (but a timeout did not occur during generation with an empty EMI block); (**ng**) generation with an empty EMI block failed; (✓) all tests ran successfully with no observed mismatches. Although **ng** may seem severe—it indicates that the configuration cannot run the benchmark at all—from a developer's perspective this is at least easy to observe; the other defects are induced by EMI variants and are indicative of more subtle problems.

The superscript for each **w** and **c** denotes whether substitutions were necessary for provoking the issue. We use $\mathbf{w}^{e}$ (respectively, $\mathbf{w}^{d}$) to indicate whether it was necessary to enable (respectively, disable) substitutions to provoke a wrong code bug, $\mathbf{w}^{\star}$ indicates that a wrong code bug could be observed both with and without substitutions. We use $\mathbf{c}^{e}$, $\mathbf{c}^{d}$and $\mathbf{c}^{\star}$ analogously for crashes.

Testing with the Intel HD Graphics 4000 GPU (configuration 8) led to a large number of timeouts, the cause of which was the compiler bug of Figure 1(e), discussed in §6. As a work-around we removed `while(1)` loops from EMI blocks for this configuration.

The table shows that problems were identified with all configurations. Configurations 11, 10, 16 and 19 were unable to generate the expected output (with an empty EMI block) for five or more benchmarks (**ng**), showing that these configurations are not robust with respect to standard OpenCL benchmarks. Three configurations suffered problems due to timeout. Turning to the 28 wrong code outcomes: enabling substitutions was necessary in 15 cases; disabling substitutions was necessary in 6 cases; in the remaining 7 cases wrong code bugs could be observed both with and without substitutions. This indicates that it is worth testing both with and without substitutions, but that overall substitutions were effective.

[6] In this context *crash* encapsulates both compiler errors and runtime errors because compilation occurs online; differentiating between these outcomes would have required extra manual work per benchmark.

7.3 Testing Using CLsmith

Table 4 summarises the results of applying the configurations lying above the reliability threshold to large batches of kernels, each generated using a different CLsmith mode. A 60-second timeout was used for each test case for all configurations except Oclgrind (configuration 19) for which a 300-second timeout was used.

The number of kernels used to test each mode is shown in the left column of the figure. We generated 10,000 kernels per mode, but had to discard 1563 ATOMIC SECTION mode and 1622 ALL mode kernels due to a bug in the implementation of atomic sections that we discovered close to publication. There was not time to generate replacement tests and re-test them across our configurations, but it was straightforward to syntactically identify and remove the affected tests. We used configuration 1+ (NVIDIA GTX Titan) to generate the tests, discarding tests that failed to compile or that timed out in order to ensure a reasonable number of terminating tests.

For each mode and configuration, we indicate the number of *wrong code* results observed (**w**), the number of build failures (**bf**), the number of cases where the OpenCL application crashed (**c**), the number of timeouts (**to**), and the number of tests that terminated producing a result that we did not deem to be wrong (✓).

We say that a configuration produces a *wrong code* result for a kernel at a given optimization level if, among all the results computed for the kernel, there is a majority of at least 3 among the non-{**bf**,**c**,**to**} results for the kernel, and the configuration yields a non-{**bf**,**c**,**to**} result that *disagrees* with the majority. Of course, it is possible that in some instances the majority result is not the correct result, though we have never found this to be the case when investigating specific result mismatches. We also show, for each mode and configuration, **w%**, the percentage of results non-{**bf**,**c**,**to**} results that are wrong code results; i.e. the percentage of **w** and ✓results combined that fall into the **w** category. We refer to this metric as the wrong code percentage. This is useful to take into account when comparing modes because the modes are evaluated using varying numbers of tests.

It is also useful to consider **w%** when comparing configurations. For example, looking at the results for the BARRIER tests, configuration 3− yields 15 wrong code results compared with 13 wrong code results for configuration 15−. At first glance this seems like a small difference. However, on closer inspection, 3− produced results (without failing during compilation, crashing at runtime or timing out) for 8995 BARRIER tests, compared with just 4592 tests for 15−. This means that 3− had significantly more opportunity to produce wrong code results compared with 15−, but did not do so. The **w%** figures of 0.2% and 0.3% for 3− and 15− take this into account.

Discussion Recall that a prerequisite for the generated tests was that they successfully compiled and did not time out on the GTX Titan with optimizations (configuration 1+). As a result, the **bf** entries are all 0 for 1+, and the **to** entries are close to 0 for this configuration (there are some timeouts because we recorded full result

Table 4. Applying the configurations above our reliability threshold to batches of CLsmith-generated tests

Configuration; IDs as per Table 1; only configurations above reliability threshold are considered; ± denotes opimizations off (−) vs. on (+)

		1−	1+	2−	2+	3−	3+	4−	4+	9−	9+	12−	12+	13−	13+	14−	14+	15−	15+	19−	19+	*Total*
BASIC (10000)	**w**	9	32	9	32	9	31	9	32	122	144	20	5	20	4	7	5	4	2	626	625	1747
	bf	396	0	397	0	397	0	396	0	0	0	0	33	0	33	0	70	1720	1720	0	0	5162
	c	352	523	365	537	553	530	539	520	289	185	831	630	831	622	51	226	18	155	4	4	7765
	to	184	0	185	1	0	0	0	0	1857	1350	272	1724	289	1721	279	422	402	1042	1759	1760	13247
	✓	9059	9445	9044	9430	9041	9439	9056	9448	7732	8321	8877	7608	8860	7620	9663	9277	7856	7081	7611	7611	172079
	w%	0.1	0.3	0.1	0.3	0.1	0.3	0.1	0.3	1.6	1.7	0.2	0.1	0.2	0.1	0.1	0.1	0.1	0.0	7.6	7.6	2.1
VECTORS (10000)	**w**	9	17	9	24	9	25	9	17	176	160	31	12	31	12	29	102	11	59	915	916	2573
	bf	366	0	366	0	366	0	366	0	0	0	7	55	7	56	6	78	1348	1348	0	0	4369
	c	427	582	438	578	652	578	630	571	343	273	831	645	831	629	75	245	1836	540	163	162	11029
	to	200	3	194	0	0	0	2	0	1276	951	277	1297	282	1396	318	461	254	945	1687	1708	11251
	✓	8998	9398	8993	9398	8973	9397	8993	9412	8205	8616	8854	7991	8849	7907	9572	9114	6551	7108	7235	7214	170778
	w%	0.1	0.2	0.1	0.3	0.1	0.3	0.1	0.2	2.1	1.8	0.3	0.1	0.3	0.2	0.3	1.1	0.2	0.8	11.2	11.3	2.9
BARRIER (10000)	**w**	16	23	11	25	15	33	16	25	152	158	186	21	186	21	23	104	13	69	968	965	3030
	bf	380	0	380	0	380	0	380	0	0	0	11	50	11	52	202	78	1331	1331	0	0	4586
	c	415	554	420	560	625	541	607	544	466	317	905	713	904	713	3696	384	3988	812	143	141	17448
	to	186	1	188	0	0	0	1	3	1279	930	236	871	236	842	238	518	89	885	1649	1677	9829
	✓	9003	9422	9001	9415	8980	9426	8996	9428	8103	8595	8662	8345	8663	8372	5841	8916	4579	6903	7240	7217	165107
	w%	0.2	0.2	0.1	0.3	0.2	0.3	0.2	0.3	1.8	1.8	2.1	0.3	2.1	0.3	0.4	1.2	0.3	1.0	11.8	11.8	3.7
ATOMIC SEC. (8437)	**w**	13	17	12	21	13	23	12	18	158	127	40	19	39	19	24	83	12	51	845	846	2392
	bf	225	0	225	0	226	0	225	0	0	0	2	25	2	25	4	60	883	883	0	0	2785
	c	246	242	251	247	353	236	305	255	321	216	436	418	436	416	42	83	920	314	91	91	5919
	to	54	1	57	0	0	0	1	0	445	293	64	193	67	195	92	151	59	272	740	725	3409
	✓	7899	8177	7892	8169	7845	8178	7894	8164	7513	7801	7895	7782	7893	7782	8275	8060	6563	6917	6761	6775	154235
	w%	0.2	0.2	0.2	0.3	0.2	0.3	0.2	0.2	2.1	1.6	0.5	0.2	0.5	0.2	0.3	1.0	0.2	0.7	11.1	11.1	3.0
ATOMIC RED. (10000)	**w**	5	13	5	28	5	29	5	13	190	190	187	14	188	15	15	101	10	61	962	964	3000
	bf	387	0	387	0	387	0	387	0	0	0	9	75	9	76	217	94	1410	1410	0	0	4848
	c	439	570	450	579	659	580	638	577	421	273	934	697	935	688	3873	518	4093	862	172	172	18130
	to	193	6	187	0	0	0	3	0	1257	933	260	1051	260	1114	210	409	78	844	1683	1662	10150
	✓	8976	9411	8971	9393	8949	9391	8967	9410	8132	8604	8610	8163	8608	8107	5685	8878	4409	6823	7183	7202	163872
	w%	0.1	0.1	0.1	0.3	0.1	0.3	0.1	0.1	2.3	2.2	2.1	0.2	2.1	0.2	0.3	1.1	0.2	0.9	11.8	11.8	3.6
ALL (8378)	**w**	13	25	12	28	15	34	15	26	161	127	250	30	249	28	11	109	3	72	857	857	2922
	bf	214	0	214	0	211	0	214	0	0	0	6	21	6	21	143	48	836	836	0	0	2770
	c	233	226	236	230	337	214	296	230	413	267	532	430	530	431	3604	375	3340	641	75	75	12715
	to	56	1	58	1	0	0	0	2	368	234	68	106	65	108	21	89	18	205	633	638	2671
	✓	7862	8126	7858	8119	7815	8130	7853	8120	7436	7750	7522	7791	7528	7790	4599	7757	4181	6624	6813	6808	146482
	w%	0.2	0.3	0.2	0.3	0.2	0.4	0.2	0.3	2.1	1.6	3.2	0.4	3.2	0.4	0.2	1.4	0.1	1.1	11.2	11.2	3.9

data for 1+ *after* completing the test generation process, using a separate test run during which some of the generated tests did time out). The other NVIDIA configurations (2, 3, 4) show similarly low build failure and timeout numbers without optimizations, as might be expected since they have similar performance characteristics and likely share compilation infrastructure between driver versions.

The lack of build failures, and the low number of timeouts, means that most tests executed with a non-crash result on the NVIDIA configurations. Our tests are thus biased *towards* discovering wrong code bugs in the NVIDIA configurations. With this in mind, the percentage of wrong code bugs over computed results, **w%**, is notably low for these configurations with optimizations enabled. With optimizations disabled the wrong code percentage increases considerably. Our modes do not appear to greatly affect the wrong code percentage for NVIDIA configurations.

The wrong code percentage associated with anonymous GPU configuration 9 is high, with **w%** consistently in the 1.5%-2.3% range. This metric does not vary much across modes.

The Intel i7 CPU configurations (12, 13) exhibit a notably higher wrong code percentage when optimizations are disabled. In contrast, the Intel i5 and Xeon CPU configurations (14, 15) exhibit more wrong code bugs when optimizations are enabled. VECTOR mode appears to slightly increase the percentage of wrong code bugs across the Intel CPU configurations. The BARRIER and ATOMIC REDUCTION tests cause a dramatic increase in the wrong code percentage for configurations 12 and 13. We did not find any bugs related explicitly to the use of atomic operations, but we did find bugs related to the use of barriers, and both the BARRIER and ATOMIC REDUCTION modes make liberal use of barriers. The wrong code percentage for these configurations is similarly high for the ALL tests, which also feature barriers heavily due to incorporating the features of the BARRIER and ATOMIC REDUCTION modes.

Oclgrind (configuration 19) exhibits a very high wrong code percentage due to a small number of bugs that have a high chance of affecting randomly-generated kernels. One example is the bug of Figure 2(f) related to handling of the comma operator. We reduced several tests for which Oclgrind produced a minority result, and inevitably ended up with a reduced program affected by one of these known bugs. As discussed in §6, these issues have since been **fixed** in the latest version of the tool. It is notable that the wrong code percentage associated with Oclgrind is considerably lower for the BASIC tests, that do not use vectors, compared with the other test sets, all of which do use vectors. However, we did not find a reduced test exhibiting a vector-related Oclgrind bug.

As discussed above, the NVIDIA configurations show an artificially low number of build failure since we required our generated tests to build successfully on configuration 1+. Nevertheless, the results show that CLsmith is able to provoke build failures in the NVIDIA configurations when optimizations are disabled.

We did not provoke any build failures for the anonymous GPU configuration (9). The associated vendor told us that they already employed fuzzing in-house to identify build failures; our results indicate that their efforts to minimise build failure issues have paid off. Oclgrind also reported zero build failures, which may be because it is based on the mature Clang/LLVM framework and because it does not attempt to optimize kernels (observe that the data for 19− and 19+ are practically identical, with the only differences arising from timeout fluctuations).

Intel CPU configurations 12, 13 and 14 generally show a high number of build failures with optimizations on compared with

optimizations off. An exception is that 14− has a high number of build failures for the BARRIER, ATOMIC REDUCTION and ALL tests; we attribute this to problems compiling kernels that make extensive use of barriers, the feature that is common to these modes.

The Intel Xeon CPU configuration (15) has a very high number of build failure due to the issue discussed in §6 where legal code that mixes **`int`** and **`size_t`** data is rejected. We also note that the build failure rates for 15− and 15+ are identical.

We focused our efforts primarily on understanding the causes of wrong code bugs, and secondarily on build failures. We did not invest much time investigating runtime crashes. The number of runtime crashes are somewhat even across Table 4, except that Intel CPU configurations 14 and 15 show a dramatic increase in the number of runtime crashes for the BARRIER, ATOMIC REDUCTION and ALL modes. Again, use of barriers is a common factor that may be causing this. Recall that the example of Figure 2(c), which uses barriers, causes 14− and 15− to crash with segmentation faults.

7.4 Testing Using CLsmith+EMI

To assess the effectiveness of EMI testing using CLsmith-generated kernels, we generated a set of *base* kernels using the ALL mode, each containing a random number of EMI blocks in the range 1–5. CLsmith inherits from Csmith the property that large portions of a generated program are dead code. We did not expect it would be fruitful to inject dead-by-construction code exclusively into code that is *already* dead. To avoid this, for each candidate base kernel we compared the results obtained running the kernel using the GTX Titan (configuration 1) with the `dead` array initialized to ensure the *dead-by-construction* property (§5), and with the *dead* array inverted to remove this property. If inversion did not affect the computed result, we assumed that all EMI blocks were placed at dead code points, and discarded the candidate program.

We used this procedure to generated 250 base kernels. From each base, we generated 40 EMI variants by applying the pruning strategies of §5, with every combination of p_{leaf}, $p_{compound}$, p_{lift} ranging over the set $\{0, 0.3, 0.6, 1\}$ and satisfying the constraint $p_{compound} + p_{lift} \leq 1$ (§5), yielding 10,000 kernels total. We subsequently had to discard 2800 of these kernels, as we discovered that 70 of the base programs suffered from the bug in our implementation of atomic sections (see §7.3). We report results for the 7200 kernels derived from the remaining 180 bases.

Table 5 summarises the results obtained from applying the configurations lying above our reliability threshold to these EMI variants. If all 40 variants associated with a base program lead to a build failure, runtime crash or timeout result for a configuration, i.e. no variant terminates with a computed value, we call the base program a *bad base* for this configuration, and do not consider the results from the EMI variants any further. The **base fails** row of the table records the number of base failures for each configuration.

Otherwise, we say that a base program induces a wrong code result for a configuration if there exist two variants for the base that terminate yielding differing values. The **w** row of Table 5 records the number of base programs that induced a wrong code result for each configuration. We say that a base program induces a build failure, runtime crash or timeout if at least one variant led to a build failure, runtime crash or timeout result, respectively. The **bf**, **c** and **to** rows of Table 5 record the number of (non-bad) base programs that induced these observations for each configuration. If all 40 variants for a base terminate, computing a uniform value, for a configuration, we say that the base is *stable* for the configuration; the **stable** row captures this. The numbers for a configuration sum to at least 180, the number of base programs, and sometimes exceeds 180 because variants of a single base program might induce multiple observations. We *do not* compare results between configurations nor across optimization levels, because one of the claimed benefits of EMI testing is that it does not require multiple configurations or optimization levels: discrimination is provided by the EMI variants derived from a base program. There are no bad bases for configuration 1+ because we used this configuration to generate the base programs; likely as a knock-on effect, there are no bad bases for other NVIDIA configurations with optimizations enabled.

Discussion We focus our discussion on wrong code results, drawing a comparison with the results obtained through pure CLsmith-based testing (§7.3).

The results indicate that EMI testing is effective at exposing wrong code bugs in the NVIDIA configurations (1–4) when compared with pure CLsmith-based testing: CLsmith using the ALL mode (Table 4) showed a low wrong code rate for these configurations, despite a large set of distinct tests and the availability of multiple configurations for cross-checking. We thus regard the number of bases that induce wrong code bugs for these configurations in Table 5 as high. In extreme contrast, EMI testing is totally ineffective at exposing wrong code bugs for Oclgrind (configuration 19), while CLsmith-based testing showed a high rate of miscompilation. This is because, as discussed in §7.3, the wrong code rate associated with Oclgrind arises from a small number of basic issues (c.f. Figure 2(f)), rather than from optimization-related bugs. Between these extremes, pure CLsmith-based testing readily exposed many wrong code bugs for anonymous GPU configuration 9, but Table 5 shows that EMI variants induce mismatches for only three base programs with this configuration.

For Intel CPU configurations 12 and 13, EMI testing induces numerous wrong code results; Table 4 shows that the wrong code rate for pure CLsmith-based testing in ALL mode is also high for these configurations. EMI testing for Intel CPU configurations 14 and 15 yields less meaningful results due to the high rate of bad base programs, which are proportional to the high rate of build failures and runtime crashes observed for these platforms during CLsmith-based testing in ALL mode.

It is clear from Table 5 that CLsmith+EMI testing is capable of inducing a significant number of build failures and runtime crashes. Timeouts are induced in a small number of cases, but as we did not record run-times for tests we do not know whether these are due to fluctuations causing long-running tests to time out, or due to deeper compiler-related issues.

On this set of base programs and configurations, we found our novel *lift* pruning strategy (§5) to be slightly *less* effective overall than the existing *leaf* and *compound* strategies in its ability to induce defects, and to slightly reduce the effectiveness of the other strategies when combined with them.

8. Related Work

Random testing The use of random testing to complement manual compiler test suites is well-established [5]. The majority of this work has focused on sequential programs, e.g. in C [20], C++ [21], JavaScript and PHP [8]. Two exceptions investigate compilation of *volatiles* [7] and C and C++ atomics [14]. Random differential testing to detect volatile miscompilations by Eide and Regehr [7] is based the idea that an execution of a program that uses volatiles has an associated *access summary*) that should be invariant across *all* compilers. This hinges on the fact that compilers are restricted in the transformations that they can apply to volatile accesses. The access summary metric is a count of the total number of loads and stores to each volatile variable of the program; differences between access summaries flag up possible miscompilations of volatiles.

Differential testing of volatiles has been extended to C++11 atomics [14] via generation, using a modified version of Csmith, of deterministic C programs that use pthread mutexes and atomic accesses. In this case a more complex metric comparing the *traces*

Table 5. CLsmith +EMI results for the configurations lying above our reliability threshold

Configuration; IDs as per Table 1; only configurations above reliability threshold are considered; ± denotes opimizations off (−) vs. on (+)

	1−	1+	2−	2+	3−	3+	4−	4+	9−	9+	12−	12+	13−	13+	14−	14+	15−	15+	19−	19+	*Total*
base fails	6	0	6	0	13	0	13	0	30	21	15	15	14	15	123	27	129	55	33	33	548
w	10	11	9	11	10	6	9	7	3	3	20	16	21	16	1	11	1	10	0	0	175
bf	4	0	4	0	4	0	4	0	0	0	2	10	2	10	3	6	4	6	0	0	59
c	0	0	0	0	6	0	4	0	11	6	2	2	4	2	31	24	27	23	0	0	142
to	0	0	0	0	0	0	0	0	1	0	0	3	0	2	0	0	1	3	1	2	13
stable	160	169	161	169	147	174	150	173	136	150	141	137	141	138	25	117	21	88	146	145	2688

of memory accesses is required. These techniques are more sophisticated than necessary for OpenCL 1.x concurrency, but could be brought to bear in future work testing OpenCL 2.0 kernels.

EMI testing EMI testing [12] is a form of *metamorphic* testing [4], which modifies a program to produce variants whose outputs can be predicted. For example, the Mettoc tool [19] uses semantics-preserving transformations to yield program variations whose output should match the original program. Hence, EMI testing can be viewed as a type of metamorphic testing based on the transformation of dynamically unreachable code. We are not aware of any prior work that uses our *dead-by-construction* method for manufacturing metamorphic compiler test cases.

Test case reduction and ranking Our experience confirms that manual reduction of randomly generated programs to isolate compiler bugs is time-consuming. The C-Reduce tool [16] automates this process for C programs, using static analysis to avoid introduction of undefined behaviours. A reducer for OpenCL would require a concurrency-aware static analysis to avoid introducing data races. The problem of ranking test cases in an order that promotes diversity has also been investigated [5]; one successful ranking metric—code coverage during compilation—may be difficult to apply to proprietary OpenCL compilers that are invoked at runtime.

9. Generality, Limitations and Future Work

Our study has been in the context of OpenCL, but many of the ideas we present could be more generally applied. EMI testing with *dead-by-construction* injection is a general concept that could be applied in other compiler fuzzing contexts. The methods we propose for generating deterministic, communicating OpenCL kernels using barriers and atomics are transferable to other multi-core, many-core and distributed programming models that have these operations, including CUDA, OpenMP and MPI. Like Csmith, CLsmith does not generate test floating point programs. We view this as an exciting open challenge: floating point imprecision is tolerated in the accelerator programming domain, but the fuzzing methods we study demand precise results. Our method is also limited to generation of concurrency primitives used in OpenCL 1.x; OpenCL 2.0 offers relaxed atomics that could enable richer communicating kernels.

Acknowledgments

Special thanks to: Anton Lokhmotov (dividiti) for suggesting fuzz testing of OpenCL compilers as a research direction; Jeroen Ketema, Gordon Inggs, George Constantinides, Lloyd Kamara, Geoff Bruce and Imperial's High Performance Computing Service for providing us with access to and support for numerous OpenCL implementations; James Price and Simon McIntosh-Smith for support related to Oclgrind; Vinod Grover and Yang Chen (NVIDIA), Laurent Morichetti (AMD) and Ajit Dingankar (Intel) for liaising with us regarding bug reports; Cristian Cadar, Yang Chen, Jeroen Ketema, James Price, John Wickerson and the PLDI reviewers for feedback on earlier drafts of this work; the PLDI Artifact Evaluation Committee for their efforts evaluating our tools.

This work was supported by the EU FP7 CARP project, Imperial College's UROP programme, an EPSRC Pathways to Impact award, EPSRC project EP/K039431/1, and a GCHQ studentship.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson Education, 2nd edition, 2006.

[2] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, and S. Qadeer. Engineering a static verification tool for GPU kernels. In *CAV*, pages 226–242, 2014.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.

[4] T. Chen, T. Tse, and Z. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1–9, 2003.

[5] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *PLDI*, pages 197–208, 2013.

[6] N. Chong, A. F. Donaldson, and J. Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *POPL*, pages 397–410, 2014.

[7] E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *EMSOFT*, pages 255–264, 2008.

[8] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, pages 445–458, 2012.

[9] ISO. ISO/IEC 9899:TC2: Programming Languages–C, 1999.

[10] Khronos. The OpenCL specification, versions 1.0, 1.1, 1.2 and 2.0, 2009–2014.

[11] Khronos. The OpenCL specification, version 1.2, 2012. Document revision 19.

[12] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014. Article 25.

[13] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[14] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, pages 187–196, 2013.

[15] J. Price and S. McIntosh-Smith. Oclgrind: An extensible OpenCL device simulator. In *IWOCL*, 2015. To appear.

[16] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *PLDI*, pages 335–346, 2012.

[17] F. Sheridan. Practical testing of a C99 compiler using output comparison. *Software — Practice and Experience*, 2007.

[18] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, UIUC, March 2012.

[19] Q. Tao, W. Wu, C. Zhao, and W. Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *APSEC*, pages 270–279, 2010.

[20] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294, 2011.

[21] C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang. Automated test program generation for an industrial optimizing compiler. In *AST*, pages 36–43, 2009.