Many-Core Compiler Fuzzing

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 19 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, reveals a significant number of OpenCL compiler bugs in commercial implementations, and acts as a call to arms for higher quality OpenCL compilers from many-core device vendors.

1. Introduction

Open Computing Language (OpenCL) [Khronos 2009–2014] 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*. Many 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 it should yield a result drawn from a well-defined, implementation-independent set of permitted results, regardless of the devices on which it executes.

The principal challenge 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* [Yang et al. 2011] and testing via *equivalence modulo inputs* [Le et al. 2014] (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 [Yang et al. 2011; Le et al. 2014] 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 controlled testing campaign using 19 (device, compiler) configurations we have identified, reduced and reported 57 distinct OpenCL compiler bugs, 53 in commercial implementations.

We hope our study will serve as a call to arms for improved compiler quality from OpenCL device vendors.

Online material for reproducibility Our tools, test programs and full result data sets are provided online at the following URL, which preserves our anonymity:

`https://sites.google.com/site/manycorecompilerfuzzing/`

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

Conf.	*SDK*	*Device*	*Driver/compiler*	*OpenCL*	*OS*	*Device type*	*Quality*
1	NVIDIA 6.5.19	NVIDIA GeForce GTX Titan	343.22	1.1	Ubuntu 14.04.1 LTS	GPU	Strong
2	NVIDIA 6.5.19	NVIDIA GeForce GTX 770	343.22	1.1	Ubuntu 14.04.1 LTS	GPU	Strong
3	NVIDIA 6.0.1	NVIDIA Tesla M2050	331.75	1.1	Red Hat Enterprise Server 6.5	GPU	Strong
4	NVIDIA 6.0.1	NVIDIA Tesla K40c	331.75	1.1	Red Hat Enterprise Server 6.5	GPU	Strong
5	AMD 2.9-1	AMD Radeon HD7970 GHz edition	Catalyst 14.9	1.2	Windows 7 Enterprise	GPU	Weak
6	AMD 2.9-1	ATI Radeon HD 6570 650MHz	Catalyst 14.9	1.2	Windows 7 Enterprise	GPU	Weak
7	Intel 4.6	Intel HD Graphics 4000	10.18.10.3412	1.2	Windows 8.1 Pro	GPU	Weak
8	Anon. SDK 1	Anon. device 1	Anon. driver 1a	1.1	Linux (anon. version)	GPU	Weak
9	Anon. SDK 1	Anon. device 1	Anon. driver 1b	1.1	Linux (anon. version)	GPU	Weak
10	Intel 4.6	Intel Core i7-4770 @ 3.40 GHz	4.2.0.76	1.2	Windows 7 Enterprise	CPU	Strong
11	Intel 4.6	Intel Core i5-3317U @ 1.70 GHz	4.6.0.92	2.0	Windows 8.1 Pro	CPU	Strong
12	Intel 4.6	Intel Core i5-3317U @ 1.70 GHz	3.0.1.10878	1.2	Windows 8.1 Pro	CPU	Strong
13	Intel XE 2013 R20	Intel Xeon X5650 @ 2.67GHz	1.2 build 56860	1.2	Red Hat Enterprise Server 6.5	CPU	Strong
14	AMD 2.9-1	Intel Xeon E5-2609 v2 @ 2.50GHz	Catalyst 14.9	1.2	Windows 7 Enterprise	CPU	Weak
15	Anon. SDK 3	Anon. device 3	Anon. driver 3	1.1e	Linux (verson anon.)	CPU	Weak
16	Intel XE 2013 R2	Intel Xeon Phi	5889-14	1.2	Red Hat Enterprise Server 6.5	Accelerator	Weak
17	Intel 4.6	Oclgrind v14.5	LLVM 3.2, SPIR 1.2	1.2	Ubuntu 14.04	Emulator	Strong
18	Altera 14.0	Altera PCIe-385N_D5 (Emulated)	aoc 14.0 build 200	1.0	CentOS 6.5	FPGA emulator	Weak
19	Altera 14.0	Altera PCIe-385N_D5	aoc 14.0 build 200	1.0	CentOS 6.5	FPGA	Weak

Table 1. The OpenCL implementations and devices we have tested

2. Overview of our 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 19 distinct OpenCL configurations, summarized in Table 1. A *configuration* refers to an (OpenCL-capable device, OpenCL device driver) pair. The OpenCL C compiler for a given device is embedded in the driver software for the device. Typically a driver is shipped with an SDK from the associated vendor; one exception is configuration 17 which can be embedded in any SDK. The table also shows the OpenCL version supported by the configuration and the OS used for testing.

GPUs Configurations 1–9 cover nine distinct GPU devices (one tested with two different drivers), from NVIDIA, AMD and Intel (1–7), and from vendors that we anonymize due to confidentiality agreements (8–9).

CPUs The devices for configurations 10–15 are multi-core Intel CPUs; configurations 10–13 use Intel drivers and configuration 14 uses AMD drivers (so that AMD's OpenCL compiler is under test). The device for configuration 15 is a multi-core CPU from a vendor that we again anonymize.

Misc The remaining configurations consist of the Intel Xeon Phi co-processor (16), oclgrind,[2] an open source platform-independent emulator (17), an emulator for an Altera FPGA (18), and the associated FPGA device (19).

This selection represents the hardware available at our institution 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. CLsmith is based on the Csmith random generator for C programs [Yang et al. 2011], and 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 a full description 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 difficult because (a) there is no straightforward method for performing code-coverage on closed-source OpenCL implementations, and (b) dynamically dead code is rare in practical OpenCL kernels: a recent study [Bardsley et al. 2014] of 605 kernels from nine sources found that just 17 kernels (3%) exhibited input-dependent behaviour. To overcome both problems we investigate injecting *dead-by-construction* code into OpenCL kernels (§5).

2.3 Our experimental method

Classifying configurations We classified each configuration (Table 1) as *strong* or *weak* depending on how they fared on a set of 600 CLsmith-generated programs. We dubbed a configuration *weak* if there was a practical issue blocking intensive testing (e.g., unpredictable machine crashes in the case of configurations 5 and 6, and prohibitively slow compilation for configuration 16), or if testing results fell below a quality threshold which we detail in §6.1.

EMI testing using standard OpenCL benchmarks The *weak* configurations suffer from serious defects typically related to the use of complex structs (see §6.1 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 even the

[2] Oclgrind: `https://github.com/jrprice/Oclgrind/wiki`

weak configurations should 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 [Stratton et al. 2012] and Rodinia [Che et al. 2009] suites.

Intensive CLsmith-based testing Focusing on the *strong* configurations, we conducted CLsmith-based testing at a larger scale (§6.3), testing 5000 random kernels generated by each of the six modes provided by CLsmith. In all this led to a set of 30,000 test programs executed with and without optimizations on 9 strong configurations, totalling 540,000 test executions. The purpose of this experiment was (1) to validate prior work [Yang et al. 2011] 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 *strong* configurations using CLsmith-generated kernels with injected dead-by-construction code (§6.4). The purpose of this experiment was (1) to validate the claim of prior work [Le et al. 2014] that mutating dynamically dead code is an effective mechanism for finding compiler bugs in a new domain, and (2) to compare the effectiveness of the EMI testing with random differential testing in terms of bug-finding ability.

2.4 Summary of our findings

Many OpenCL implementations are low quality Many of the configurations we tested exhibit fundamental compiler bugs (see Figure 1); miscompilations caused machine crashes for some configurations; applying EMI testing to standard benchmarks revealed that some configurations do not work at all with certain benchmarks, while other benchmarks readily revealed 25 compiler bugs after injection of dead code. These issues undermine the portability aim of the OpenCL effort. We hope our study will serve as a call to arms for better OpenCL implementations from vendors.

Fuzz testing is effective in the OpenCL domain Both random differential and EMI testing reveal significant numbers of defects in OpenCL implementations. It is hard to compare the techniques directly, but our test results suggest that random differential testing using a diverse set of kernels across multiple platforms is most effective at identifying bugs.

We did not find communication-related bugs Our novel methods for generating deterministic, communicating kernels led to the discovery of compiler bugs that only manifest in the presence of barrier synchronization. However, once reduced, none of these bugs were inherently communication-related and we did not find obtain bug-inducing test cases that involved 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. We reported these defects to the Parboil and Rodinia developers (the latter have confirmed the bug). This hammers home the point that compiler fuzzing requires deterministic, well-defined programs; real-world examples often do not have this property.

Full details of the bugs we identified and reported are available at our companion website (see the Introduction).

3. Background

3.1 The OpenCL programming model

OpenCL [Khronos 2009–2014] allows an application running on a *host*, e.g. the CPU of a regular PC, to offload computation to one or more parallel *devices* (see e.g. the devices summarised in Table 1). Offloading is achieved by expressing the computation to be accelerated as a *kernel*: a function parameterised by a thread identifier to 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. The host also copies data to/from devices and launches kernels on devices.

Threads and groups An OpenCL kernel is executed by an *ND-range* (*N-Dimensional range*) of *work-items*, which we henceforth refer to as *threads* for brevity. We have not encountered 4D or higher-dimensional kernels in practice, so 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, t_{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 in the set $\{2, 4, 8, 16\}$.[3] The C arithmetic and logical operations all lift to apply component-wise to vectors. A rich set of additional built-in vector operations is also provided.

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, compare-and-exchange, plus arithmetic and bitwise operations.

A *data race* occurs between two distinct threads if the threads access a common memory location, at least one thread modifies the location and either: (a) the threads are in different groups, or (b) the threads are in the same group, at least one of the accesses is non-atomic, and no barrier synchronization operation occurs in-between the accesses.

Barrier divergence occurs if two threads in the same group arrive at syntactically 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 [ISO 1999]. Data races and barrier divergence are considered undefined behaviours, and some of the vector operations specify new undefined behaviours. There are fewer sources of implementation-defined behaviour: in particular, the widths of primitive data types are fixed in OpenCL (e.g. **int** always denotes a 32-bit integer). Notably, whether irreducible control flow [Aho et al. 2006] is allowed 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 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. This essence of the requirement on such a program is captured by the following definition, intended as a guideline (the definition is necessarily imprecise because it does not refer to a specific programming language):

DEFINITION 1. *[Program with deterministic output] A program P produces* deterministic output *with respect to an input I if, when executed on I, P exhibits no undefined or implementation-defined behaviour, and terminates printing a string s that is uniquely determined by I.*

Random differential testing Csmith [Yang et al. 2011] represents the current state-of-the-art using random differential testing [Sheridan 2007; McKeeman 1998]. 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, 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 The main limitation of random differential testing is that it requires multiple compilers. Compiler testing via *equivalence modulo inputs* (EMI testing) [Le et al. 2014] avoids this by testing a single compiler against multiple programs that all produce the same deterministic output with respect to 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—$P_1, \ldots, P_N$ say—by substituting I-dead statements with alternative statements. Compile $P, P_1, \ldots, P_N$ 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 [Le et al. 2014]. The authors argue that the method

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

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 [Yang et al. 2011] 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 in which every thread computes a numeric result by executing an adapted Csmith-generated program. These computations are independent, and each thread writes its result to index t_{linear} of a designated global memory array, `out`. A host application, the *launcher*, allocates memory for the `out` array on the device under test, compiles and invokes the kernel, and prints the elements of `out` 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, initialise an instance of this struct on kernel entry and pass the struct by reference to every function invoked by the kernel. 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 §6.1 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 extend Csmith with type-sensitive vector expression generation. To avoid undefined behaviours arising from vector computations we avoided potentially unsafe operations such as `/` and `%`, and restricted operations that might overflow to operate only on unsigned vectors. In future work we could enrich the supported vector operations further by designing a set of "safe math" vector macros, following the approach used by Csmith for scalar operations [Yang et al. 2011].

Randomizing grid and group dimensions To test a diverse range of thread arrangements, CLsmith randomly selects a total thread count in the range $[0, 9999]$ and then chooses random divisors of this thread count to select appropriate values for $\vec{N}$ and $\vec{W}$ (see §3.1). Kernels using a dimension count lower than three are in effect still be generated in this manner if size 1 is selected for one or more dimensions.

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. The OpenCL 1.x specifications are ambiguous as to whether inter-group communication is legal: atomic operations and memory fences that would appear to support such communication are provided [Khronos 2012, pp. 277,281–283], but are undermined by the statement *"there are no guarantees of memory consistency between different groups executing a kernel"* [Khronos 2012, p. 29]. We cautiously limit our methods to consider only intra-group communication.

In what follows we use rnd and rnd_i ($i \in \mathbb{N}$) to denote constants chosen randomly during program generation.

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 an array `permutations` of length $d \cdot W_{linear}$ containing d random permutations of the set $\{0, \ldots, W_{linear} - 1\}$, for some small d; we use $d = 10$ in practice. The permutations are allocated in constant memory. Each group is also equipped with a shared memory array `A_offset` of type `uint` and length W_{linear}, initialised 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`, initialised to `permutations[`rnd`][`l_{linear}`]`, for $0 \leq rnd < d$. 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, like so:

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

where `FENCE` is a global or local memory fence depending on the memory space in which `A` is allocated. This randomly re-distributes ownership of elements of `A` among the threads.

This allows CLsmith to generate random 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 commu-

nication mechanism yields deterministic results (see [Chong et al. 2014] for a proof of this general result).

ATOMIC SECTION *mode* In this mode, CLsmith generates *atomic sections* of the following form:

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

where `c` points to a volatile `uint` *counter* in shared memory, and `s` is a `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-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.

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.

Our hypothesis was that atomic sections might identify compiler bugs that break the determinism guarantee CLsmith attempts to enforce.

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 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 an atomic reduction is:

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

Here op is one of the above operators and $expr$ is a randomly generated expression. Because each of the atomic operations we consider are commutative, 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 initialising the array `A` uniformly with a single value. These restrictions make it impossible for the identity of a thread to influence the control flow the thread takes during execution.

5. EMI testing for OpenCL

As discussed in §2.2, direct EMI testing (as per [Le et al. 2014]) for OpenCL is hindered by the difficulty of performing code coverage and the scarcity of I-dead code (see §3.2).

We overcome this by injecting *dead-by-construction* code. We equip a kernel with an additional array parameter, `dead` of length d (for some small d) and randomly insert into the kernel blocks of the following form:

```
if(dead[rnd_1] < dead[rnd_2]) {
  /* statements */
}
```

where $0 \leq rnd_2 < rnd_1 < d$. We call such a conditional statement an *EMI block*. The OpenCL compiler knows nothing about the runtime values of elements of `dead`. We also modify the host application to initialise `dead` so that dead[i] $= i$ ($0 \leq i < d$). This means that, by construction, the statements inside the EMI block are dynamically unreachable. As long as the original kernel produces deterministic output, so should any variation of the kernel injected with any EMI block.

Dead-by-construction code in CLsmith-generated kernels We extended CLsmith with an option to equip the generated kernel with a `dead` array and then produce random EMI blocks. Variants of the program are then produced by *pruning* the EMI blocks according to a set of configurable probabilities. 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 (e.g. **`if`** and **`for`** statements) are *branch* nodes, and at each node a series of prunings are considered. We reproduce two pruning strategies from [Le et al. 2014], *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. Because *compound* and *lift* are not independent (they can both remove branch nodes) and are applied in the order (*compound*, *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 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 an EMI block with substitutions would be more effective in provoking compiler bugs because operations inside and outside the block on common data would give the compiler the opportunity to optimize across the block boundary. We evaluate this hypothesis in §6.2.

6. Our testing campaign

We now describe our testing process and results in detail, which uncovered bugs in all configurations. The bugs we discuss have been reported to the associated vendors.

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.

6.1 Initial testing to classify configurations

We tested every configuration of Table 1, with and without optimizations, on a set of 600 kernels generated by CLsmith which we call the *initial* kernels, using a timeout of 60 seconds for compilation and execution (but excluding the time taken for kernel generation). This set consisted of 100 kernels generated using each of the six modes supported by the generator: from BASIC through to ALL. We then examined mismatches between configurations arising from this testing.

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 are in discussions with NVIDIA about the correct operator precedence rules here (we do not find the OpenCL specification sufficiently clear). We find it interesting that our accidental generation of ambiguous, possibly erroneous, expressions led to the identification of compiler front-end mismatches.

Many tests initially failed on the Altera configurations (18 and 19) due to the front-end rejecting logical operations on vectors, which *are* supported in OpenCL. To work around this 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.

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

The AMD GPU and CPU configurations (5, 6, 14) produce wrong results (regardless of thread count) with optimizations for the trivial kernel of Figure 1(a); more generally these configurations appear 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; both were confirmed and the padding bug (but not the bug of Figure 1(a)) is fixed in the Catalyst 14.9 drivers.

The anonymized GPU configurations (8, 9) 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.

Kernels that use vectors in structs (Figure 1(c)) produce LLVM IR generation errors when compiled by the Altera configurations (18, 19).

Compilation for the Xeon Phi co-processor (configuration 16) is prohibitively slow when relatively large structs are used optimizations enabled, which we regard as a bug. The host for our Xeon Phi card (a 2.0 GHz Intel Xeon CPU) takes more than 20 seconds to compile a kernel summarized in Figure 1(c) 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.

Figure 1(e) illustrates a struct-related miscompilation for configuration 15 (confirmed by the anonymized vendor and reproduced in their trunk build); the `barrier` is required for the bug to manifest.

We also identified struct-related miscompilations for the Intel HD Graphics 4000 configuration (7).

Machine crashes We had difficulty testing with our AMD and Intel GPUs (configurations 5, 6, 7) because kernel execution would occasionally, and unpredictably, crash the OS of the host machine. We were able to mitigate this to some extent for the AMD GPUs by that the GPU under test was not being used simultaneously for graphics processing. Unpredictable machine crashes make batch testing, and thus in-

```
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) Expected result: 2. Result from configurations 5+, 6+ and 14+: 1

```
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,
         {0,0,0,0,0,0,0,1,0,0}};
  s = t;
  out[t_linear] = p->f[7];
}
```
(b) Expected result: 1. Result from configurations 9− and 8−: 0

```
kernel void k() {
  struct S1 { int4 x; } s =
    { (int4)((int2)(1,1),1,1) };
}
```
(c) Vectors in structs cause IR generation errors with configurations 18± and 19±

```
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,
            {{{0,0,0},...,{0,0,0}},
               ...
            {{0,0,0},...,{0,0,0}}}
  };
  s = t;
  barrier(CLK_LOCAL_MEM_FENCE);
  out[t_linear] = p->c[0][0][1];
}
```
(d) Compilation for configuration 16+ takes more than 20s on a state-of-the-art host

```
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(CLK_LOCAL_MEM_FENCE);
  f(&s);
  out[t_linear] = s.x + s.y;
}
```
(e) Expected result: 3. Result from configuration 15±: 2

```
kernel void f(global int *p)
{
  for(int i=0; i < 197; i++)
    if(*p) while(1) { }
}
```
(f) Configuration 7± hangs during compilation (it appears that the compiler gets stuck in an infinite loop)

Figure 1. Kernels illustrating compiler bugs for the *weak* configurations

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	×

Table 2. OpenCL benchmarks studied using EMI testing

tensive fuzz testing, infeasible. It is also interesting and potentially worrying that erroneously-compiled OpenCL kernels can bring down a system.

Other notable bugs For the majority of our *initial* tests, the Altera FPGA configuration (19) either crashed or emitted an internal compiler error: *"Residual unsplit local memory space. Internal error while updating IR. Please report this compiler bug."* The AMD Radeon HD 6570 configuration (6) complained about unsupported irreducible control flow for example kernels that do not use **goto** or **switch** statements (the only possible sources of irreducibility at the kernel source level).

***Dubbing configurations as* weak** A configuration is deemed *weak*, and thus not worthy of further fuzz testing efforts (as argued in §2.3) if more than 25% of our *initial* tests led to compiler crashes, runtime crashes or miscompilations (judged by disagreement with the majority result). This was the case for all the *weak* configurations shown in Table 1 except for the Intel Xeon Phi (configuration 16) which we dubbed *weak* due to the issue of prohibitively slow compilation of structs which made intensive fuzz testing impractical.

6.2 EMI testing over the Rodinia and Parboil suites

Table 2 summarises 10 benchmarks, drawn from the Parboil v2.5 [Stratton et al. 2012] and Rodinia v2.8 [Che et al. 2009] 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 [Le et al. 2014]. Table 2 summarises each benchmark, indicating the number of kernels, total lines of kernel code[4] and whether the kernels use floating-point arithmetic. We selected these benchmarks by initially favouring the three benchmarks that do not use floating-point (to avoid precision issues), selecting further benchmarks in decreasing order of kernel code size.

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 1 or 2 EMI *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 variables appearing in the original kernel using **#define** macros (substitutions on). Some manual tweaking was necessary to ensure well-typed substitutions. Finally we edited the host code of each benchmark to allocate and initialise the `dead` array, and added command line arguments to configure which EMI to be used and to toggle substitutions. Using this method, it

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

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

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

took one of the authors approximately half a day to prepare a single benchmark.

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 compare the output of a given test against the expected output of the benchmark, which we generated by running the benchmark with an empty EMI block. Table 3 summarises the results for 17 of our 19 configurations; we exclude the Altera configurations (18 and 19) because they require offline compilation which was non-trivial to integrate with the Parboil and Rodinia 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;[5] (**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** denotes whether substitutions were necessary for provoking the wrong code: in cases where enabling (respectively, disabling) substitutions was necessary to provoke the wrong code we denote this as $\mathbf{w}^{e}$ (respectively, $\mathbf{w}^{d}$); otherwise a wrong code bug could be observed both with and without substitutions (denoted as $\mathbf{w}^{\star}$).

The table excludes myocyte and spmv: as discussed in §2.4 we found data races in these benchmarks.

Testing with the Intel HD Graphics 4000 GPU (configuration 7) initially revealed a large number of timeouts, the cause of which was a compiler bug illustrated in Figure 1(f). The compiler appears to get stuck in an infinite loop compiling this kernel. 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 8, 9, 14 and 17 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 25 wrong code outcomes: enabling substitutions was necessary in 13 cases; disabling substitutions in 6 cases; in the remaining 6 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.

We reduced several result mismatches arising from EMI testing over these benchmarks to small test cases that induce bugs; these are available from our companion website.

6.3 Testing *strong* configurations with CLsmith

Table 4 summarises the results of applying the *strong* configurations to six sets of 5000 kernels, each generated using a different CLsmith mode. For each program we obtain 18 results: a result for each of the 9 configurations with and without optimizations, each with a 60 second timeout. Thus we obtained 540,000 test results in all. The **Total** rows sum the results across all configurations.

For each mode and each configuration we indicate the number of wrong code miscompilation bugs observed (**w**), the number of compiler crashes ($\mathbf{c}_{\mathrm{c}}$), and the number of cases where the OpenCL application exhibited a runtime crash ($\mathbf{c}_{\mathrm{r}}$). We say that a configuration produces a wrong code result for a kernel if there is a majority of at least 3 among the non-crash results for the kernel, and if the configuration yields a non-crash result which disagrees with this majority.

The results show that CLsmith identifies a significant number of possible compiler defects for all the configurations we tested. We manually reduced a number of tests for all configurations, focusing primarily on narrowing down wrong code bugs. A selection of these bugs are presented and discussed in our online companion material.

Effectiveness of our generation modes We cannot directly compare across modes because the 5000 programs generated for two different modes are different; however we believe that 5000 programs ensures a sufficiently diverse range of kernels so that a comparison between the overall effectiveness of the modes is meaningful. Comparing the **Total** rows for BASIC and ALL suggests that the use of vectors and communication devices provokes a large number of wrong code bugs, both with and without optimizations, and a large num-

[5] 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.

	BASIC			VECTOR			BARRIER			ATOMIC SECTION			ATOMIC REDUCTION			ALL		
Conf.	**w**	$\mathbf{c_c}$	$\mathbf{c_r}$	**w**	$\mathbf{c_c}$	$\mathbf{c_r}$	**w**	$\mathbf{c_c}$	$\mathbf{c_r}$	**w**	$\mathbf{c_c}$	$\mathbf{c_r}$	**w**	$\mathbf{c_c}$	$\mathbf{c_r}$	**w**	$\mathbf{c_c}$	$\mathbf{c_r}$
1−	4	189	156	12	206	202	12	210	211	27	161	147	10	201	216	25	155	153
1+	23	0	228	18	18	279	24	47	297	34	16	148	25	16	293	39	42	153
2−	4	190	156	11	206	204	13	210	216	27	161	149	10	201	219	18	155	153
2+	23	0	230	18	18	280	26	47	300	35	16	151	25	16	296	32	42	153
3−	74	0	297	104	0	428	90	40	412	124	3	286	113	0	413	104	27	284
3+	14	0	159	28	0	266	22	40	267	27	3	166	41	0	269	49	26	168
4−	76	0	284	95	0	431	88	40	407	117	3	281	102	0	412	102	27	284
4+	14	0	157	18	0	274	16	40	272	28	3	166	31	0	276	37	26	174
10−	150	0	362	243	17	396	292	7	454	409	16	294	502	15	470	656	23	351
10+	147	0	299	222	17	362	216	7	385	395	16	296	439	15	369	591	23	296
11−	150	0	358	242	17	392	290	7	448	407	16	291	503	15	463	654	23	350
11+	148	0	344	230	17	396	223	7	397	399	16	300	455	15	404	598	23	307
12−	163	0	15	212	18	43	67	47	1897	394	16	39	56	16	1987	91	42	2407
12+	157	0	140	226	18	165	194	47	265	417	16	96	420	16	328	544	42	331
13−	7	785	7	21	716	894	10	666	1915	27	580	593	11	696	2010	16	597	2137
13+	3	785	84	39	716	295	40	666	390	62	580	171	48	696	445	78	597	424
17−	278	0	0	492	16	72	476	5	65	485	13	44	507	13	76	527	16	52
17+	292	0	0	476	16	71	476	5	65	492	13	47	514	13	76	527	16	52
Total−	906	1164	1635	1432	1196	3062	1338	1232	6025	2017	969	2124	1814	1157	6266	2193	1065	6171
Total+	821	785	1641	1275	820	2388	1237	906	2638	1889	679	1541	1998	787	2756	2495	837	2058

Table 4. Applying the *strong* configurations to 5k batches of CLsmith-generated tests

ber of runtime crashes without optimizations. Rates for the other bug categories increase less dramatically.

The **Total** results suggest that ATOMIC SECTION and ATOMIC REDUCTION mode induce more wrong code bugs than BARRIER mode. However, we did *not* manage to reduce any failing tests to minimal examples that require atomic operations whereas we reduced several tests, for Intel configurations, to minimal kernels that required barriers to demonstrate a compiler bug. We note that ATOMIC REDUCTION mode does generate barrier statements.

The results show that there is little difference between the NVIDIA GTX Titan and GTX 770 configurations (1, 2), and little difference again between the NVIDIA Tesla M2050 and Tesla K40c configurations (3, 4), but significant differences between the two families. The rate of compiler crashes is also constant across certain configurations, and has the same value for some configuration from different vendors, e.g. configurations 1–4 (NVIDIA), 10–13 (Intel) and 17 (oclgrind) all give 295 compiler crashes in BASIC mode with optimizations; we conjecture that these implementations are based on the same or similar versions of LLVM (oclgrind is based on LLVM 3.2).

6.4 Testing *strong* configurations with CLsmith+EMI

To assess the effectiveness of EMI testing using CLsmith-generated kernels we generated 100 kernels using the ALL mode, each containing between 1 and 5 EMI blocks. CLsmith inherits from Csmith the property that large portions of a generated program are already dynamically unreachable; this property was of benefit to the original EMI approach which discovers such dead code by profiling [Le et al. 2014]. In our context we did not expect it would be fruitful to consider injecting dead-by-construction code at locations that are already dynamically unreachable. To avoid this we compared the results of each 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 reversed to remove this property; if inversion did not affect the computed result we assumed that all EMI blocks were at already dead positions and discarded the kernel.

This process left 59 kernels containing EMI blocks. For each 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). This yielded 2360 kernels in total which we tested on the *strong* configurations with a timeout of 60 seconds per kernel.

We refer to 885 of these kernels as *leaf+compound* because they are obtained by pruning exclusively using the *leaf* and *compound* strategies from prior work [Le et al. 2014] (i.e. with $p_{lift} = 0$), and to 177 kernels as *lift* because they are obtained by pruning exclusively using our novel *lift* strategy (i.e. with $p_{leaf} = p_{compound} = 0$). We refer to the remaining 1298 kernels, which use all strategies, as *combined*. The *combined* kernels are pruned by both *compound* and *lift*, which each remove branch nodes, thus the associated EMI blocks contain little conditional code.

Table 5 shows the extent to which these classes of programs revealed compiler bugs with respect to the strong configurations. If a configuration at a given optimization setting yields a majority result over the EMI variants for an original kernel, and the results for k variants disagree with the majority, we add k to the wrong code (**w**) total for the configuration at this optimization setting.

It is hard to compare the EMI testing results of Table 5 head-to-head with the ALL mode CLsmith results of Table 4 because there are fewer kernels in the EMI case, and because these kernels are drawn from a relatively small number of source programs. However, we can observe from the **Total** rows and **w** columns of Table 5 that EMI testing over 2360 kernels originating from ALL mode programs revealed *no* result mismatches with optimizations enabled, and 63 result

	leaf + compound			*lift*			*combined*		
Conf.	**w**	$\mathbf{c}_c$	$\mathbf{c}_r$	**w**	$\mathbf{c}_c$	$\mathbf{c}_r$	**w**	$\mathbf{c}_c$	$\mathbf{c}_r$
1−	0	62	0	0	73	0	0	11	0
1+	0	0	0	0	0	0	0	0	0
2−	0	62	0	0	73	0	0	11	0
2+	0	0	0	0	0	0	0	0	0
3−	0	0	73	0	0	91	0	0	12
3+	0	0	48	0	0	43	0	0	7
4−	2	0	48	4	0	60	2	0	8
4+	0	0	48	0	0	43	0	0	7
10−	9	0	147	9	0	182	2	0	25
10+	0	0	183	0	0	224	0	0	32
11−	8	0	135	7	0	172	1	0	23
11+	0	0	182	0	0	224	0	0	30
12−	6	0	183	13	0	202	0	0	35
12+	0	0	762	0	0	894	0	0	127
13−	0	200	594	0	232	703	0	29	103
13+	0	200	214	0	232	244	0	29	41
17−	0	0	24	0	0	28	0	0	4
17+	0	0	8	0	0	7	0	0	1
Total−	25	324	1204	33	378	1438	5	51	210
Total+	0	200	1445	0	232	1679	0	29	245

Table 5. CLsmith+EMI results for the *strong* configurations

mismatches with optimizations disabled. In contrast the **Total** rows and ALL-**w** column of Table 4 show that 5000 distinct ALL mode kernels identify 2174 and 2422 result mismatches across configurations with optimizations disabled and enabled, respectively. This suggests that diversity across both programs and configurations is more effective at identifying miscompilations than diversity across variants of a pool of source programs on an individual configuration.

Comparing the pruning strategies, Table 5 shows that our *lift* strategy identifies more issues in total than the *leaf+compound* methods from prior work despite the fact that the *lift* set of kernels is significantly smaller than the *leaf+compound* set. The *combined* approach works surprisingly badly: we attribute this to the lack of conditional code in the associated EMI variants as discussed above.

We manually reduced a number of kernels that produced mismatches and crashes, but have not yet discovered bugs distinct from those uncovered by the CLsmith method.

7. Related work

Random testing The use of random testing to complement manual compiler test suites is well-established [Chen et al. 2013]. The majority of this work has focused on sequential programs, e.g. in C [Yang et al. 2011], C++ [Zhao et al. 2009], JavaScript and PHP [Holler et al. 2012].

Two exceptions investigate compilation of *volatiles* Eide and Regehr [2008] and C and C++ atomics Morisset et al. [2013]. Random differential testing to detect volatile miscompilations Eide and Regehr [2008] 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.

The idea of differential testing of volatiles has been extended to C++11 atomics [Morisset et al. 2013] 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* 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. The challenges here would include recording memory trace across heterogeneous devices, and scaling the tracing method to thousands of threads.

EMI testing EMI testing [Le et al. 2014] is a form of *metamorphic* testing [Chen et al. 2003], which modifies a program to produce variants whose outputs can be predicted. For example, the Mettoc tool [Tao et al. 2010] 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 work that has applied EMI testing to the compilation of parallel programs.

Test case reduction and ranking Our experience with this work confirms that manual reduction of randomly generated programs to isolate compiler bugs is time-consuming. The C-Reduce tool [Regehr et al. 2012] 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 [Chen et al. 2013]; one successful ranking metric—code coverage during compilation—may be difficult to apply to proprietary OpenCL compilers that are invoked at runtime.

8. Generality, limitations and future work

Our study of many-core compiler fuzzing 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 clearly 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.

References

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

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 *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2014.

S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, pages 44–54. IEEE, 2009.

T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-Based Testing Without the Need of Oracles. *Information and Software Technology*, 45 (1):1–9, 2003.

Y. Chen, A. Groce, C. Zhang, W. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 197–208. ACM, 2013.

N. Chong, A. F. Donaldson, and J. Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 397–410. ACM, 2014.

E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, 2008.

C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362831`.

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

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

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

V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 25. ACM, 2014.

W. M. McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1):100–107, 1998.

R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346. ACM, 2012.

F. Sheridan. Practical Testing of a C99 Compiler Using Output Comparison. *Software — Practice and Experience*, 2007.

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, University of Illinois, at Urbana-Champaign, March 2012.

Q. Tao, W. Wu, C. Zhao, and W. Shen. An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique. In *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, APSEC '10, 2010.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.

C. Zhao, Y. Xue, Q. Tao, L. Guo, and Z. Wang. Automated test program generation for an industrial optimizing compiler. In *ICSE Workshop on Automation of Software Test*, pages 36–43, May 2009. doi: 10.1109/IWAST.2009.5069039.