Automatic Test Case Reduction for OpenCL

Moritz Pflanzer
Imperial College London
moritz.pflanzer14@imperial.ac.uk

Alastair F. Donaldson
Imperial College London
afd@imperial.ac.uk

Andrei Lascu
Imperial College London
andrei.lascu10@imperial.ac.uk

ABSTRACT

We report on an extension to the C-Reduce tool, for automatic reduction of C test cases, to handle OpenCL kernels. This enables an automated method for detecting bugs in OpenCL compilers, by generating large random kernels using the CLsmith generator, identifying kernels that yield result differences across OpenCL platforms and optimisation levels, and using our novel extension to C-Reduce to automatically reduce such kernels to minimal forms that can be filed as bug reports. A major part of our effort involved the design of ShadowKeeper, a new plugin for the Oclgrind simulator that provides accurate detection of accesses to uninitialised data. We present experimental results showing the effectiveness of our method for finding bugs in a number of OpenCL compilers.

1. INTRODUCTION

Reliable OpenCL compilers are important, both to help ensure that OpenCL software operates correctly on a given platform, and also to fulfill the OpenCL promise of portability, whereby an OpenCL application should exhibit functionally equivalent behaviour across a range of OpenCL-conformant platforms. Compiler reliability is a particular challenge in the context of OpenCL because (a) OpenCL compilers must be optimizing (performance is the main reason for using OpenCL in the first place), (b) the OpenCL C kernel programming language is relatively new and evolving, and (c) OpenCL C back-ends are required for many target architectures, many of which are also relatively new and evolving.

Our recent work applying random program generation to test OpenCL implementations [8] identified numerous bugs in OpenCL compilers from several vendors. In our testing campaign, we applied two distinct techniques: *random differential testing* (RDT), a fuzzing method popularised by the Csmith tool [15], and *equivalence modulo inputs testing* [7] (EMI), an instance of metamorphic testing [2]. We focus here on the RDT approach. Our RDT method builds on top of the Csmith technique: in our prior work we built a tool, CLsmith,[1] which generates random OpenCL kernels that are, by construction, free from undefined, implementation-defined and nondeterministic behaviour, avoiding OpenCL C constructs that allow variation in behaviour between implementations.[2] Each generated kernel computes an integer result, and should compute the same result no matter which OpenCL platform is used for execution. A result mismatch between OpenCL platforms, or with respect to the same platform with compiler optimizations enabled vs. disabled, is indicative of a possible compiler bug.

```
1  ulong *l_505[2];
2  int i, j;
3  for (i = 0; i < 2; i++)
4    l_505[i] = &p_1502->g_308[1][3];
5  for (p_1502->g_37 = (-24); (p_1502->g_37 == (-1)); ++p_1502
       ->g_37)
6  {
7    int **l_42 = &l_35[4][5];
8    (*l_42) = l_35[2][5];
9  }
10 barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
11 p_1502->tid = (({ uint ui1 = ((({ uint ui1 =
12   (get_linear_group_id()); uint ui2 = (7) ; (uint)
13   (((unsigned int)(ui1)) * ((unsigned int)(ui2)));}
14   ))); uint ui2 = (permutations[(({ uint ui1 =
15   (func_43(func_45(func_48((l_506 = (({ short si1 =
16   (0L); short si2 = ((((~0x6AF53593L) , func_55(((
17   p_1502->g_138[0] = ((int4)((((int8)(l_57.yyxxxyxy)
18   ).even <= ((int8)(((int2)((~((int4)(({ int4 si1 =
19   (((int4)((-8L), func_58(func_64( ...
```

Figure 1: A small excerpt from a kernel generated by CLsmith

Random differential testing has been shown to be most effective when large test programs are generated [15]. Intuitively this is because a large test program has a higher chance of incorporating combinations of language features that provoke compiler bugs compared with a smaller test program. However, it is practically infeasible to understand the root cause of a compiler bug from a large, randomly generated test program. To illustrate this, look at the code fragment shown in Figure 1. This was taken from an OpenCL kernel generated by CLsmith, comprising more than 1,800 lines of code with a file size of almost 197 kB. The kernel

[1] https://github.com/ChrisLidbury/CLSmith, visited on 09/02/2016.

[2] In particular this eliminates floating-point arithmetic, e.g. because denormal numbers may optionally be flushed to zero [6].

```
1  struct S {
2    int a;
3    long b;
4  };
5
6  void g(struct S *p);
7
8  void h(void);
9
10 void f(struct S *p) {
11   g(p);
12 }
13
14 void g(struct S *p) {
15   p->b = 1;
16   h();
17   barrier(CLK_LOCAL_MEM_FENCE);
18 }
19
20 void h() {
21   barrier(CLK_LOCAL_MEM_FENCE);
22 }
23
24 kernel void entry(global ulong *result) {
25   struct S s = { 5, 0 };
26   f(&s);
27   result[get_global_id(0)] = s.b;
28 }
```

Figure 2: A kernel that triggers an OpenCL compiler bug

was shown to cause a behavioural difference when executed on an Intel Core i7 CPU, using the OpenCL SDK version 5.3 under Windows 10, with driver version 5.2.0.10094. On this platform, the kernel yielded different results with optimizations enabled vs. disabled. It should be apparent that the behaviour of the original kernel (of which just a few lines are shown in Figure 1) is not practically comprehensible to a human. In particular, the fact that the original kernel yields different results for two different OpenCL configurations does not give any immediate information as to which of the OpenCL configurations is buggy, nor any hint related to the root cause of the bug.

Random differential testing for C compilers suffers from precisely the same problem—that large test programs do not directly shed light on compiler defects—and the C-Reduce tool has been developed to aid in automatically reducing a large C program to a smaller C program that still exposes a compiler bug [12].[3]

Our contribution: In this paper we describe how we have extended the C-Reduce tool to the context of OpenCL, providing an automated method for reducing OpenCL test cases that induce compiler bugs. The main challenge in achieving this has been the development of methods for detecting undefined behaviour in an OpenCL kernel. For this purpose we have re-used existing tools, including the Clang Static Analyzer[4] and Oclgrind [11],[5] and have built ShadowKeeper, a new plugin for Oclgrind that detects accesses to uninitialized data with high-precision. The ShadowKeeper plugin can be used independently from the rest of our framework.

[3]In fact, C-Reduce is a general framework for reducing programs with respect to an "interestingness" criterion; in this work we focus on C-Reduce as a method for reducing programs that induce compiler bugs.

[4]http://clang-analyzer.llvm.org , visited on 09/02/2016.

[5]https://github.com/jrprice/Oclgrind, visited on 09/02/2016.

```
1  struct S {
2    ushort a;
3    int **b;
4    ulong c;
5  };
6
7  void f(struct S * p);
8  void g(struct S * p);
9  void h(int * ip, struct S * p);
10
11 void f(struct S * p) {
12   g(p);
13 }
14
15 void g(struct S * p) {
16   barrier(CLK_LOCAL_MEM_FENCE);
17   h(*p->b, p);
18 }
19
20 void h(int * ip, struct S * p) {
21   p->a = p->c - get_group_id(0);
22   barrier(CLK_LOCAL_MEM_FENCE);
23 }
24
25 kernel void entry(global ulong *result) {
26   int t = 0;
27   int * u = &t;
28   volatile int z = 1;
29   struct S s = { 0, &u, z };
30   f(&s);
31   result[get_global_id(0)] = s.a;
32 }
```

Figure 3: Another illustration of a kernel that triggers an OpenCL compiler bug

We hope it will be useful to the OpenCL community as an aid for kernel debugging.

To illustrate our contribution, Figure 2 shows a minimal OpenCL kernel, derived from the original large test kernel discussed above by first running our OpenCL extension to C-Reduce, followed by further manual minimisation. It is clear from Figure 2 that each work item executing the kernel should set `s.b` to 1, due to executing `p->b = 1` at line 15, where `p` is a pointer to `s`. However, on the Intel platform described above, when optimizations are disabled, the kernel computes $[1, 0]$ when executed by a single work group of two work items. In this example, running the reduction tool was time-consuming, requiring 26h46m of machine time (see Section 2 for further details); however, no human intervention was required during this process, and machine time is cheap in comparison to engineer time. It took one of the authors 18m to further reduce the machine-reduced kernel down to the minimal form shown in Figure 2.

In contrast, it took the same author 2h20m to reduce the original 200 kB kernel to a minimal form without the aid of C-Reduce. The resulting minimised program is shown in Figure 3, and is slightly harder to understand. A work item produces the final value of `s.a` as its result. At line 21, `p->a = p->c - get_group_id(0)` assigns to `s.a`. At this point, `p->c` is the same as `s.c` and has the value 1, thus for a work item in work group 0 the statement should set `s.a` to 1. On the Intel platform, with optimizations enabled, the kernel produces the expected result when executed by two work items in a single work group: $[1, 1]$. With optimizations disabled, an erroneous result $[1, 0]$ is produced. The structure of the kernels in Figures 2 and 3 is similar, so it would appear that these issues stem from the same root cause.

Terminology: Throughout the paper we shall use CL-Reduce to refer to our OpenCL extension to C-Reduce, to allow us to refer unambiguously to the original C-Reduce tool vs. our extension. However, as we explain in detail below, CL-Reduce is simply the C-Reduce tool equipped with an *interestingness test* specific to OpenCL. Our main contribution has been to engineer this interestingness test (a non-trivial task for OpenCL), after which we have been able to re-use the C-Reduce framework almost unchanged.

Structure: In Section 2 we provide a rough guide to the benefits brought by automatic test case reduction for OpenCL by presenting the experiences of the paper authors reducing OpenCL test cases with and without the help of CL-Reduce. In Section 3 we explain the challenges we faced in extending the C-Reduce method to apply in the context of OpenCL, and in Section 4 we describe how we overcame these challenges in building CL-Reduce. We present a quantitative experimental evaluation in Section 5, demonstrating the effectiveness of CL-Reduce for OpenCL kernel reduction on a variety of platforms. We then provide an overview of related work (Section 6) and conclude (Section 7).

2. CASE STUDY: HUMAN VS. MACHINE REDUCTION FOR OPENCL

The aim of our work is to automate the tedious process of manually reducing a large bug-inducing test case to a small kernel that can be filed as a bug report.

Before discussing the technical details of how we achieve this, we give an indication of the trade-offs associated with manual vs. automatic reduction by reporting on the experiences of two authors of this paper performing test case reduction manually vs. automatically.

Table 1 details two OpenCL configurations used for our case study: an NVIDIA GPU implementation and an Intel CPU implementation, referred to as Intel and NVIDIA henceforth. We detail the device and the host CPU (in the case of Intel these are the same), and provide information about the OpenCL driver version, SDK, and host OS.

2.1 Experimental setup

Each of Donaldson and Pflanzer took one configuration, Intel and NVIDIA, respectively, and used CLsmith [8] to find two kernels for which their OpenCL configuration reported a different result with optimizations enabled vs. disabled. For each test case the author then (a) reduced the test case to a minimal example manually, timing themselves in the process, (b) used CL-Reduce to automatically reduce the kernel, timing how long this process took, and (c) further manually reduced the kernel output by CL-Reduce, to reach a minimal form, timing themselves in the process.

2.2 Results

Table 2 summarises the results of our case study. Each row of the table denotes an instance of our experiment for a particular original test kernel. The author responsible for handling the kernel is listed (Author), with the associated configuration (Config.). Under **Original test** we show the size of the original test, indicating lines of code (*LOC*) and file size in bytes (*bytes*). The size in bytes is arguably more interesting since a single line of code in a randomly-generated kernel can be very large if it involves a complex expression.

Under **Manual** we show the time (*time*) taken for the author to manually reduce the test case to a minimal example, and the size of the resulting minimal test (*LOC* and *bytes*). By *minimal* we mean that the author was satisfied that the correct result computed by the kernel was simple to read from the reduced source code, and that no easy further reduction opportunities would work. It is likely that with continued manual effort the resulting kernels could be made slightly smaller, and their size in bytes could be reduced somewhat by renaming variables.

Under **Automatic** we show the time (*time*) taken by CL-Reduce to automatically reduce the original test kernel, and the size (*LOC* and *bytes*) of the reduced kernel computed by CL-Reduce. The time shown here is wall-clock time, and represents machine time only: no human intervention is required during the reduction process.

Under **Post-automatic** we show the time (*time*) taken for the author to further manually reduce the output generated by CL-Reduce to obtain a minimal test exposing the compiler bug (with same caveats related to *minimal* as in the case of **Manual** above); the size of the resulting minimal test (*LOC* and *bytes*) is also shown.

2.3 Discussion

Our experience is that both manual reduction and CL-Reduce-based reduction followed by manual fine-tuning enabled large test kernels to be reduced to comparably-sized and relatively small examples (in each case smaller than 1 kB). The manual effort associated with fine-tuning (post-automatic reduction) was always significantly lower than the time associated with a fully manual reduction.

On the Linux-based NVIDIA platform, CL-Reduce operated efficiently, taking less than 1.5 hours on each test case. In contrast, the reduction times for the Windows-based Intel platform was higher, with reduction taking more than a day in one case. An analysis of the executions times of the different tasks during the reduction revealed that the process creation time of a few milliseconds on Linux is around one order of magnitude smaller than on Windows. Likely most of the longer runtime can be attributed to this finding because every *interestingness test* (see Section 3.1) itself, and potentially also its sub-tasks, are launched as independent processes. Although automatic reduction can be slow, it can be parallelised: a GPU vendor could use a farm of machines to reduce numerous test cases in parallel, which will clearly be more cost-effective than devoting a team of engineers to manual test case reduction.

A feature of automatic reduction that our case study does not highlight is the problem of duplicate bugs. It is well-known that fuzzing techniques can generate tests that repeatedly trigger the same bug [3], and our experience with manual reduction in our original CLsmith study [8] is that we commonly spent hours reducing a test case by hand only to home in on a minimal test virtually identical to a previously-discovered test. It is clearly advantageous to waste machine time, rather than human time, on such duplicate reduction efforts.

2.4 Threats to validity

Our case study is intended merely to give a rough indication of the benefits and pitfalls associated with manual vs. automatic reduction. Clearly the sample size of test kernels that we reduced is too small to draw general conclusions, and

Short name	Device	Host CPU	Driver version	OpenCL SDK	Operating system
Intel	Core i7-6500U	Core i7-6500U 2.5GHz	5.2.0.10094	Intel OpenCL SDK 5.3	Windows 10
NVIDIA	GTX Titan	Intel Xeon E5-2609 2.4GHz	343.22	CUDA SDK 6.5.12	Ubuntu 14.04.3 LS

Table 1: OpenCL configurations used for our case study

Author	**Config.**	**Original test**		**Manual**			**Automatic**			**Post-automatic**		
		LOC	*bytes*	*time*	*LOC*	*bytes*	*time*	*LOC*	*bytes*	*time*	*LOC*	*bytes*
Donaldson	Intel	1,858	201,338	2h20m	43	571	26h46m	86	1,917	18m	30	388
Donaldson	Intel	979	99,553	1h01m	34	331	7h22m	78	1,977	15m	32	360
Pflanzer	NVIDIA	914	54,290	54m	61	968	1h58m	59	1,218	5m	50	834
Pflanzer	NVIDIA	1,267	135,553	1h16m	45	738	2h58m	72	1,759	32m	51	906

Table 2: Times and sizes associated with manual vs. automated reduction of kernels in our case study

the speed at which the paper authors were able to reduce kernels is related to their past experience doing so, which is more than what an average software developer would have in this task, but is perhaps less than the experience an OpenCL compiler developer would gain if they used random differential testing on a regular basis to strengthen their implementation. The speed of manual reduction is influenced significantly by execution-time properties of the host machine for a particular configuration, and varies considerably between the Windows and Linux operating systems.

In our experimental design we had each author first manually reduce the kernel, then subsequently perform post-automatic reduction on the output of CL-Reduce. As a result, the authors had some advantage in performing post-automatic reduction, knowing some reduction steps that were more likely to work well than others. We could have accounted for this by having authors swap configurations before undertaking post-automatic reduction; one reason we did not do this is that each author has particular editing tools they prefer to use to reduce kernels effectively, that are not portable across platforms. For example, Donaldson uses Visual Studio for this purpose (due to its brace matching functionality, with which he is fluent), and would be slower at reducing a kernel on a Linux setup using a different editor.

3. BARRIERS TO USING C-REDUCE FOR REDUCTION OF OPENCL KERNELS

We explain how the test case reduction process implemented by C-Reduce works, in the context of C programs (Section 3.1), and then outline the specific challenges that had to be faced in lifting this method to the context of OpenCL (Section 3.2).

3.1 Background on C-Reduce

The C-Reduce tool [12] implements a generalised notion of *delta debugging* [16], whereby a program is reduced through application of a variety of transformations. The reduction process is specialised for reduction of C programs. Reduction is guided by a configurable *interestingness test*. For example, to reduce a test case that causes a compiler internal error message, the interestingness test would be configured to check that the compiler does indeed exit with the particular error message. To reduce a test case that causes compiled code to yield different results when a program is compiled with vs. without optimisations, the interestingness test would check that compilation succeeds in both optimisation modes, and that the results output by the compiled binaries differ.

Program transformations for reduction. C-Reduce uses three classes of transformations to reduce programs.

The most basic transformations operate at the level of source text, ignoring program structure. Some transformations modify contiguous regions of the source program, for example changing the value of integer literals or removing parts of arithmetic expressions or text segments within surrounding brackets. Other transformations make non-contiguous changes like removing pairs of brackets without deleting their contents.

The second class comprises a set of semantics-aware source-to-source transformations based on the abstract syntax tree of the program. They cover a wide range of language-specific changes, ranging from the removal of unused or statically dead expressions and functions, through alteration of types, to complex code refactorings. These transformations are carefully designed to ensure that they do not introduce undefined or unspecified behaviour. All these transformations are bundled in the *clang_delta* helper tool (a component of C-Reduce), which uses the Clang AST parser.

The third class of transformations works at the level of source code tokens, whereby tokens are deleted or modified according to specific patterns. Examples include deleting every other token, or one token in three, or reversing all tokens in a specified range, enabling e.g. the expression `a < b` to be transformed to `b < a`. To achieve these transformations, C-Reduce incorporates *clex*, a custom lexer for C.

Transformation cycle. The reduction process of C-Reduce works as follows. The size of the initial program is logged. C-Reduce then iterates over all the available program transformations. A given program transformation is repeatedly applied until the transformation either leads to an error (due to the transformation tool crashing), or until the space of reductions for that transformation is exhausted. As an example transformation space: line-level removal uses a form of binary search, first trying to remove the upper half of the program, then (if this does not succeed) the lower half; if this in turn fails then the process continues, halving the number of lines for which removal is attempted in one transformation step. Once iteration over all the program transformations has completed, the size of the program is compared with the size that was logged initially. If the sizes are the same, C-Reduce concludes that it will not be able to reduce the program further, and exits. Otherwise, the reduced program is treated as the initial program, and the process repeats.

Interestingness tests. Each time the source file is altered during the reduction process, an *interestingness test* is applied. If the interestingness test reports that the modified code is interesting, the modified code is used as a basis for further transformations. Otherwise the modified code is discarded and reduction continues from the state of the code before the latest reduction step was applied.

When reducing a program that causes a wrong code compiler bug, it is important that the interestingness test checks that undefined behaviours have not been introduced by the reduction process, as well as checking that the compiled binaries yield different results when executed. This is because it is legitimate for a compiler at multiple optimization levels, or for multiple distinct compilers, to produce binaries whose execution yields different results when applied to a program that exhibits undefined behaviour.

We now discuss the design of C-Reduce's interestingness test for wrong code bugs. The interestingness of a transformation is checked by first passing the modified code through one or more compiler front-ends, using flags that generate warnings about possible undefined behaviours. The purpose of this is to allow for a fast fail if the program is syntactically invalid, or if the compilers warn about undefined behaviour. Next, the *Clang Static Analyzer*[6] and *Frama-C*[7] tools are applied to identify deeper semantic issues that may indicate undefined behaviours. If no such issues are identified, the code is checked for dynamically invalid behaviour using *KCC*,[8] an interpreter based on formal semantics, and the *Valgrind*[9] instrumentation framework. Only if all these undefined behaviour checks are passed is the modified program checked for a result mismatch with respect to the compiler(s) under test. The importance of KCC and Valgrind is highlighted by experimental results showing that, without dynamic checks, C-Reduce converges on a program that exhibits undefined behaviour in 29% of cases [12].

The cost of applying the interestingness test, and the relatively high probability that a given transformation will introduce undefined behaviour (leading to an interestingness test failure), can lead to a slow reduction process. To mitigate this, multiple program transformations and associated interestingness tests can be executed in parallel. When an interestingness test succeeds, concurrent transformations under consideration can be killed, and a set of fresh transformations can be kicked off with respect to the reduced program. As long as one parallel transformation and interestingness test does not affect the performance of another (due to resource contention), this parallel approach should in the worst case be no slower than a serial reduction involving the same series of program transformations.

3.2 OpenCL-specific challenges

Our aim was to extend C-Reduce to handle OpenCL kernels, to enable reduction of large random kernels generated by the CLsmith tool [8]. This presented two immediate challenges. First, a number of the tools that C-Reduce uses cannot be applied to OpenCL; specifically GCC, Frama-C, KCC and Valgrind. In contrast, we could re-use the Clang-based tools—the Clang compiler itself, the Clang Static Analyzer and clang_delta—because Clang provides a mature OpenCL front-end; we could use all the language agnostic components directly, and because of the relationship between C99 and OpenCL we could also reuse the *clex* component. Among the tools that do not support OpenCL, the most problematic cases were GCC and Valgrind.

A number of diagnostic warnings that GCC generates, to indicate possible undefined behaviours, are not reported by Clang. In particular, GCC, but not Clang, warns about reads from possibly uninitialised structs. For CLsmith generated programs this was particularly problematic: as detailed in [8], CLsmith emits kernels that use a struct with many fields to model the globally-scoped variables that Csmith would generate for a C program (OpenCL does not support globally-scoped variables prior to OpenCL 2.0). We found that without knowledge of such uninitialised accesses, our first version of CL-Reduce would produce small kernels with undefined behaviour: these kernels would simply declare a struct, not initialise the fields of the struct, and then produce a result obtained from one of the uninitialised fields.

As discussed in Section 3.1, the dynamic checks provided by Valgrind were found by the C-Reduce authors to be useful in reducing the rate at which C-Reduce produced programs exhibiting undefined behaviour; OpenCL is similar to C in terms of operations that can exhibit undefined behaviour.

The second problem is that OpenCL offers a new set of undefined behaviours that are not relevant to sequential C programs: data races between work items, and barrier divergence (where work items in the same work group do not reach the same barrier statement). By construction, a CLsmith-generated kernel is free from both of these undefined behaviours. However, they can be introduced by the reduction process and need to be checked for as part of the interestingness test when reducing a wrong code bug.

To overcome the lack of diagnostic checks for OpenCL that GCC and Valgrind provide in the case of C, and to provide OpenCL specific diagnostic checks, we turned to Oclgrind, a simulator for OpenCL applications [11]. Initially, we thought that Oclgrind would provide the necessary checks out-of-the-box, as the tool already provided a number of memory analysis capabilities. However, we had to refine the capabilities of Oclgrind and add a new, state-of-the-art plugin to detect access to uninitialised memory, in order to make automated reduction of OpenCL kernels feasible.

We next discuss this process of lifting C-Reduce to the context of OpenCL.

4. LIFTING C-REDUCE TO OPENCL

Recall that, for purposes of disambiguation, we use CL-Reduce to refer to our OpenCL-aware extension to C-Reduce. We first describe the design of an interestingness test suitable for minimising test cases that expose OpenCL wrong code compiler bugs (Section 4.1). We then discuss various tooling issues that had to be solved to realise this interestingness test (Section 4.2), and describe in detail a new plugin for Oclgrind that we implemented to provide precise warnings related to accessing uninitialized data (Section 4.3). During the process of engineering our extension we made various improvements to the C-Reduce infrastructure, which we briefly outline (Section 4.4).

4.1 The OpenCL interestingness test

Structurally, the OpenCL interestingness test is the same

[6]http://clang-analyzer.llvm.org, visited on 12/02/2016.
[7]http://frama-c.com, visited on 12/02/2016.
[8]https://github.com/kframework/c-semantics, visited on 12/02/2016.
[9]http://valgrind.org, visited on 12/02/2016.

as that used for reduction of C programs [15]. Lightweight tools are first used to quickly detect undefined behaviour introduced by a reduction step. Slower, more thorough tools are then applied, and execution of the test case on the device under test is postponed until last.

The objectives for an interestingness test are two-fold. A test must be precise, reliably detecting undefined behaviour. For OpenCL, the main issue here is to detect accesses to undefined values, and data races between concurrently executing work items. A test must also be fast, to make the time for automatic reduction practical. For instance, during a reduction of an average-sized program (>100 kB), the interestingness test typically has to be invoked around 30,000 times. To reduce the average runtime of the interestingness test, all relevant tools are executed in increasing order of expected runtime, with the test aborting as soon as a tool marks the reduced program as invalid. This fast-failure principle ensures that only *successful* reduction steps require that all tools have been run.

Our OpenCL interestingness test first performs some hard-coded checks, specialised towards test cases generated by CLsmith. These check the structural integrity of test cases. For instance, a CLsmith-generated kernel must include an initial comment line that contains metadata on which the host program that runs CLsmith-generated kernels depends; if this line is removed we immediately reject the reduced kernel as not interesting. For reduction of kernels not generated by CLsmith these checks can be deactivated. Next, the Clang compiler is used to filter out syntactically incorrect kernels. The warning messages of Clang, and of the Clang Static Analyzer, are checked for signs of undefined behaviour. Both tools act as a replacement for the GCC compiler that is used by C-Reduce; GCC empirically shows a better detection rate for illegal uses of undefined values but (as discussed in Section 3.2) does not include an OpenCL front-end. In addition to indicators of undefined behaviour, the static warnings are also searched for situations that might lead to dynamic failures, such as assigning a non-zero integer value to a pointer, and situations that may introduce non-determinism, e.g. comparing a pointer to a non-zero integer value. This is important because (a) nondeterminism can lead to result differences between OpenCL configurations applied to a given kernel, deviating from the aim of identifying result differences that arise due to a compiler bug, and (b) nondeterminism may prevent a dynamic analysis tool from detecting an undefined behaviour if the nondeterministic choices resolve in a manner that leads to an execution path that does not trigger any undefined behaviour, even though such behaviours are possible along other program paths.

Limitations of the static tools mean that they may let through kernels that contain undefined behaviour; for example, a variable that is initialised only conditionally may not be flagged by static analysis as possibly uninitialised. The three most common sorts of undefined behaviour that often go undetected by static analysis are: usage of undefined values (e.g. non-initialised values) and invalid pointers (e.g. dereferencing a null-pointer), as well as array out-of-bounds accesses. To identify these issues, we use Oclgrind for dynamic analysis, as an alternative to the Valgrind dynamic analysis framework that is used for reduction of C programs but is not compatible with OpenCL (Section 3.2). Oclgrind also checks for data races and barrier divergence during kernel execution. To make the interestingness test as fast as possible, Oclgrind stops as soon as a warning of undefined behaviour is reported.

The drawback of using dynamic tools is that they actually have to execute the program. Moreover, the injection of a custom memory management system (to enable memory-aware analysis) adds additional overhead. This slows down the validation process significantly, especially for large and high-dimensional OpenCL kernels. Nevertheless, Oclgrind is an essential component of CL-Reduce due to the limitations of static tools.

A kernel that passes all static checks and is deemed valid by Oclgrind is executed on the two OpenCL configurations that are being compared. In the experiments of this paper we always executed a kernel using one OpenCL implementation, comparing results with and without optimisations, but the same concept applies to comparing results across two different OpenCL platforms.[10] The interestingness test succeeds if the kernel executes successfully on both configurations and different results are generated. If the kernel fails on either platform (e.g. due to a compiler crash or a runtime exception) we deem the reduction attempt not interesting. This is because our aim is to detect bugs where the compiler silently produces wrong code, rather than arguably less serious cases where the compiler crashes or generates code that leads to a crash. Of course, the interestingness test can be configured to focus on such cases if desired.

Another important aspect of the interestingness tests is that all tools are equipped with a fixed time-limit. This helps to prevent situations in which the reduction process would get stuck if one of the programs freezes. For the dynamic executions of the kernels, the time-limit is vital to cope with infinite loops that can be created by reduction transformations.

4.2 Interestingness test tooling issues

We had to solve a number of immediate issues before we could apply CL-Reduce to our CLsmith-generated kernels.

Clang invalid shufflevector operands. While analyzing the logs of an Oclgrind crash, we observed that the root cause lay within Clang (specifically, running the kernel without optimisations through Clang 3.6 on a 64-bit system). Reducing the test kernel used to trigger the crash led to the example of Figure 4, allowing us to pinpoint the root cause of the crash: a specific code pattern where an existing vector is reused rather than building a new one. More precisely, a combination of an `extractelement` instruction with an index of 64-bits is generated alongside a `shufflevector` instruction applied on a vector containing 32-bit `undef` values. This mismatch triggers the bug, which is caught when building Clang with assertions enabled. We submitted a patch to fix this issue, which was applied in the Clang 3.8 release and back-ported to Clang 3.7.

Oclgrind index out-of-bounds check. We observed a specific instance where Oclgrind would not generate a warning for an invalid memory read. By declaring a struct with elements of varying sizes (illustrated in Figure 5; `ulong` and `uint` have sizes 8 and 4 bytes, respectively, in OpenCL), accessing the array field `a` at index 1 would not be flagged as in issue by Oclgrind. The root cause was that the mismatch

[10] The latter case is complicated somewhat if the two OpenCL platforms under test are on different machines.

```
1 typedef unsigned int uint2 __attribute((ext_vector_type(2)))
      ;
2
3 void test1(void) {
4     (uint2)(((uint2)0).s0, 0);
5 }
```

Figure 4: Reduced OpenCL kernel used to trigger an Oclgrind crash with the warning "Invalid shufflevector operands!". The root cause was due to a bug with the `shufflevector` instruction in Clang.

```
1 struct S0 {
2     ulong f;
3     uint a[1];
4 };
```

Figure 5: Struct declaration that could trick Oclgrind by reading uninitialized memory due to the difference in size between the elements of the struct.

between type sizes led to four padding bytes being added to the struct; Oclgrind erroneously regarded these bytes as part of the struct, in turn regarding an access to `a[1]` as in-bounds. We submitted a patch for this issue, which was accepted, but later had to be adapted by us to cater for the fact that it is legitimate in C for a pointer to point one element past the end of an array, as long as the pointer is not dereferenced.

Oclgrind custom warning messages. We found the warning messages generated by Oclgrind insufficiently expressive for our needs. We reworked Oclgrind's diagnostics system by extending the existing types of diagnostics with targeted ones for specific issues (e.g. array out-of-bounds errors, uninitialized value warning, etc.). As such, we can filter the diagnostics as per the requirements of the analysis. In addition, Oclgrind does not stop the execution even if an error has been detected. In order to accelerate our interestingness test, we added a new option to stop execution after a user-defined number of errors have been produced.

4.3 The ShadowKeeper plugin

As mentioned in Section 4.1, in CL-Reduce we had to replace Valgrind with Oclgrind, the latter being especially developed for the runtime instrumentation of OpenCL programs.

In contrast to Valgrind, Oclgrind originally included only limited capabilities for detecting undefined behaviour arising from accesses to uninitialised values. Initially, Oclgrind would monitor all interactions with the memory system and emit a warning when an undefined value was involved in a memory operation. This caused false positives in cases where uninitialised values were copied around without subsequently influencing program behaviour. In particular, we found that when copying between struct data, Oclgrind would warn about uninitialised accesses to padding bytes (which, by definition, cannot be initialised by assigning to fields of a struct).

To obtain reliable analyses for the interestingness tests it was necessary to extend Oclgrind with a more precise plugin. The resulting *ShadowKeeper* plugin was a key piece of our CL-Reduce framework, while improving the general functionality of Oclgrind by helping developers to write valid or debug invalid OpenCL programs. As a result, it has been merged into the main project and replaces the old plugin.

The internal mechanics of ShadowKeeper are derived from Valgrind's Memcheck plugin [13] and Clang's MemorySanitizer [14]. Both tools use the term *shadow* as a metaphor for the validity state of a memory location or register during program execution. The authors of Valgrind's Memcheck plugin identify three general requirements that have to be met in order to fully support shadow values [9]. The three categories are described as: monitoring the current state of a program (registers and memory), instrumenting instructions that read or write memory and instrumenting instructions that allocate or deallocate memory. A precise detection of undefined values is only possible if all three categories are covered. While the effects of a less precise handling of shadow registers and memory are well-defined, the consequences of only a partial coverage of all operations are hard to confine. For instance, if shadow values were only accurate up to the scale of bytes it would be clear that every operation involving bit operations could lead to wrong detection results. In contrast, any instruction, system call or external function has the potential to perform memory operations. As long as the operation is not covered with a corresponding operation on the shadow memory both states will diverge and result in an incorrect analysis.

We now summarise the principles behind Valgrind's Memcheck and Clang's MemorySanitizer, after which we present the design of ShadowKeeper in detail.

4.3.1 Valgrind's Memcheck

Valgrind's Memcheck plugin is able to detect "invalid" usages of undefined values such as uninitialised variables with bit-level precision by instrumenting programs according to the disassemble-and-resynthesise (D&R) paradigm. This includes partially initialised bytes, such as bit fields inside structs. To achieve this precision, every bit of data has to be shadowed with an additional definedness bit. Internally, these shadow values are referred to as *V bits* (validity bits). A V bit is set to zero if and only if the corresponding data bit is considered to be defined.

Data registers are shadowed through a simple one-to-one mapping to shadow registers. Both data registers and shadow registers can be modified through the instructions provided by the intermediate representation. This makes it easy to keep the actual data synchronised with the shadow values. Additionally, a large amount of memory has to be monitored. To be able to do this efficiently a two-level mapping scheme is used.[11] The first level table (PM) divides the 32 bit address space into 65,536 blocks. Every entry in the table points to a secondary table (SM) containing 65,536 entries to shadow 64 kB of memory. Space in the second level table is allocated on demand (copy-on-write) and deallocated together with the data values. Therefore, not all entries in the first level table are set. At program startup, everything, aside from literals, and read-only and mapped memory, is considered as undefined and the corresponding shadows are thus all set to 1.

In addition, Valgrind's Memcheck is able to achieve a total coverage of dynamically linked libraries. For each instruction a trade-off between accuracy of the shadow propaga-

[11] The actual implementation uses an additional compression scheme to save further memory.

tion and the performance of the shadow operation has to be found. For most operations, the plugin puts emphasis on the accuracy and sacrifices efficiency. For example, arithmetic operations shadow the effects of carry chains, as opposed to the faster approach of invalidating the complete results as soon as one operand is (partially) undefined.

Another design decision is when to emit warning messages about invalid uses of undefined values. Most of the time the Memcheck plugin propagates undefined values lazily through the shadow operations producing warnings only at a few critical check points. An operation is considered as critical if it alters the observable behaviour of the program. In [13], four distinct groups of such operations are defined. The first two groups include operations that change the control flow, e.g. branch instructions and conditional moves. The third group comprises memory operations where the address operand might be undefined. Finally, system calls form the fourth group. After a warning has been generated, the associated undefined values are explicitly set as defined to prevent chains of warnings from the same source.

4.3.2 Clang's MemorySanitizer

The main difference to Valgrind's Memcheck plugin is that Clang's MemorySanitizer uses static compile time instrumentation instead of the D&R paradigm. If a program is compiled with MemorySanitizer support, the resulting executable contains all necessary instructions to detect undefined values. Because shadow operations are directly injected into the LLVM IR of the application, it is unnecessary for MemorySanitizer to shadow registers explicitly. The instructions themselves represent the temporary values and are automatically assigned a unique identifier.

Similar to Memcheck, memory is shadowed with bit-level accuracy, but a simpler one-to-one mapping between application data and shadow bits is applied [14]. This comes at the cost of higher memory consumption, but simplifies the address computation. In fact, the shadow address is derived by flipping one bit in the original address such that all shadow addresses are projected into a commonly unused address space. Due to preferring speed over accuracy, most operations that perform shadow propagation are less precise than the corresponding functions in Memcheck. Moreover, some of the operations even allow false negatives to occur if this has the potential to greatly improve performance. For instance, an addition is approximated through a simple bitwise OR instruction of both operands. This only guarantees that (a) the result is defined if both operands are defined and (b) that the result cannot be valid if either operand is (partially) undefined. However, the actual bits that are undefined after the addition might not be correctly shadowed, as carry propagation is not modelled.

MemorySanitizer is generally unable to achieve full coverage of all functions if external libraries are involved; this would require all libraries to be instrumented by MemorySanitizer. To mitigate this issue, MemorySanitizer provides wrapper functions for some widely used and hard-to-compile libraries (e.g. libc) which define the side effects of each function and change the shadow values accordingly.

4.3.3 ShadowKeeper

Conceptually, ShadowKeeper lies between Memcheck and MemorySanitizer. MemorySanitizer is entirely trimmed for efficiency, accepting even loss in accuracy for better performance. Valgrind's Memcheck aims to be as precise as possible and optimises only for efficiency whenever this does not have a major impact on accuracy. The goal of ShadowKeeper is to make the detection of undefined values as complete as possible, but (to achieve fast runtime) not necessarily at a high level of precision. This allows the propagation of undefined values to be approximated in order to improve the performance, as long as the approximations are conservative such they do not decrease the "undefinedness" of any value. For instance, the shadow operation for the `add` instruction validates only the state of the operands and conservatively assumes that all bits of the result are undefined if any bit of either operand is undefined.[12]

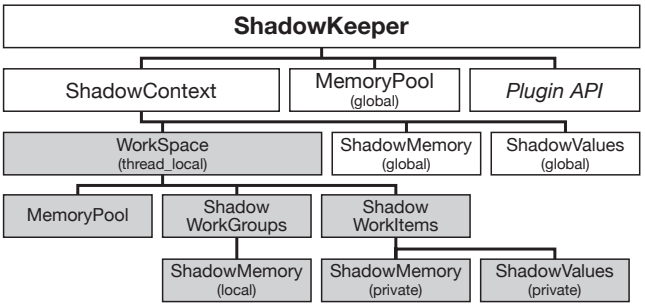

Figure 6: The hierarchical design of ShadowKeeper

The plugin follows the spirit of Oclgrind, having been designed with maintainability and modularity in mind, even though this introduces some overhead in terms of performance. The existing Oclgrind plugin system provided an excellent staring point to integrate the new ShadowKeeper plugin into the existing framework. Figure 6 provides a high-level overview over the different components.

The design is closely tied to the architecture of Oclgrind, as every memory location and intermediate value has to be shadowed to track its definedness. As with every Oclgrind plugin, the main class acts as controller that intercepts action callbacks (e.g. `workItemBegin`), before, during and after the kernel execution. It further sets up various data structures that help to maintain and track the definedness of all values. Non-global resources are defined as thread local to make lock free accesses possible. Solely global resources have to be guarded by locks.

As with the other tools, the abstract concept of shadow data is divided into two categories. Shadow memory mirrors the address space of the original kernel and shadow values represent the validity of global and private variables in the kernel. ShadowKeeper creates a simple one-to-one mapping with bit-level accuracy between the address spaces of the kernel and the shadow address spaces. The only exception is the constant address space, which is currently not mapped. The reason behind this decision is that in OpenCL kernels all constants have to be statically initialised, hence they cannot contain undefined values. Therefore it is faster to generate a new clean shadow for each access to constant memory instead of performing a costly lookup. Further, generating the shadow values on demand saves space. Each of the other three address spaces (global, local, private) is rep-

[12] A bit-level accurate shadow computation for arithmetic instructions can be extreme costly because of the effects of carry bits.

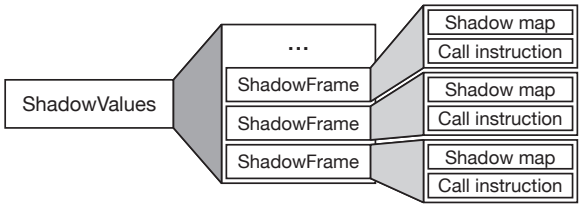

Figure 7: The structure of shadow values

resented by a separate map which uses the original address of the memory access as key. Using separate maps has the advantage that the lookup of a particular value is likely to be faster, and prevents clashes between addresses of different address spaces.

To reduce the memory overhead, the implementation uses sparse maps into which memory buffer objects are inserted every time an allocation is performed. The use of a copy-on-write mechanism similar to the one in Valgrind's Memcheck plugin turned out to be inefficient in combination with sparse maps. In contrast, deallocations are handled lazily and are only performed on demand. The main reason is that tracking deallocations precisely would introduce a huge amount of extra work because there is no explicit instruction for deallocations in the LLVM IR. An object is simply considered as deallocated if its lifetime ends (e.g. the scope of the allocation is exited).

In general all allocated memory is assumed to be undefined and the shadow memory in consequently filled with special "poisoned" values. However, since ShadowKeeper cannot determine the validity of the data that the host program writes to the memory, the default is to mark it as defined. Therefore the shadow memory for buffers that are mapped for writing and the regions that are written by the host program are filled with clean shadow values. This bears the risk of missing some undefined values, but the alternative (marking global memory as poisoned after every interaction with the host application) would lead to false alarms that would render the plugin unusable in practice.

Intermediate results computed during the execution of an OpenCL program are also shadowed with bit-level precision and stored in a sparse map. Instead of the address, the signature of the original LLVM IR instruction is used as key. The handling of variables in global scope, i.e. constants and pointers to the local address space, is straightforward. They are stored before the actual execution of the kernel is started, exist throughout the entire execution of the kernel and can be accessed from every point in the plugin. On the other hand, the management of variables in private scope requires more effort than a single map. The lifetime of the temporary variables is important, as shadow values from different function scopes might interfere with each other. The solution implemented in ShadowKeeper includes an explicit construction of a new "ShadowFrame" each time a function is called and a return to the old frame when the control flow leaves the function (see Figure 7).

In addition to the storage of the shadow values, a shadow operation had to be defined for each of the 48 LLVM IR instructions and 94 built-in functions that Oclgrind currently supports. Instead of computing the result shadow with bit-level precision, only the shadows of all operands are checked and if either is (partially) poisoned the entire result is considered poisoned. This has the great advantage of being faster than an exact computation of the result shadow. Furthermore, it cannot make a value "less" poisoned and does not sacrifice much accuracy.[13]

Warnings about undefined values are only emitted if the values would change the observable behaviour of the program. That is, if undefined values are written to memory of a non-private address space or if the control flow depends on undefined values. Further, warnings are generated if undefined values are passed as arguments to external functions and if undefined values are used as index operand in instructions. Lastly, the addresses of all memory loads and stores are directly checked for their definedness. Without these address checks, undefined values could remain unnoticed (for example, a load from a poisoned access luckily hits a valid address and a clean value is loaded). The objective to emit warnings only when strictly necessary has been set to leverage the reduction processed by allowing undefined intermediate states.

Another important aspect while designing ShadowKeeper has been to guarantee thread-safety. Only then is Oclgrind able to simulate multiple work items at the same time by running them in different threads. Serialised execution would have rendered ShadowKeeper useless for large kernels. A major concern in terms of thread safety are data structures used to store the shadow data. First is the map storing all global shadow values. The only writing accesses to this map are before the actual execution. During execution, only reading accesses can happen which do not introduce data races. Moreover, for the shadow memory of non-local address spaces, although they are accessible from concurrent threads, no locking mechanism is needed. Any concurrent access to the same address would correspond to a data race on the actual values because the shadow operations are always synchronised to the real operations. Since data races are forbidden by the OpenCL standard, ShadowKeeper does not make any guarantees for kernels that exhibit this kind of undefined behaviour.[14] The concurrent access to different addresses in the same map is enabled by storing the shadow memory in a buffer separate from the actual map and inserting only a pointer to the buffer into the map. The only exception from the lock-free implementation are atomic operations that affect global data structures. Here it is not the responsibility of the kernel to prevent data races but the implementation of the atomic operations must handle concurrent accesses without introducing undefined behaviour. Every atomic shadow operation that writes to the global address space acquires a lock prior to reading the old shadow value and releases it after writing the new shadow value. Additionally, to reduce contention, multiple locks are provided such that each operation locks only a part of the address space.

4.4 Improvements to C-Reduce

To prevent undefined behaviour in OpenCL kernels, a strict compliance with the C99 standard is necessary. This revealed some edge cases in which C-Reduce's transformations produced non strictly conforming results. Independent of the incorrect transformations, though also based on gen-

[13]Actually, we have not encountered a single false positive warning since the old plugin has been exchanged with ShadowKeeper.

[14]Oclgrind includes a separate plugin for data race detection.

erated compiler warnings in C99 mode, a few other transformations have been optimised to produce smaller test cases. Further, we addressed some minor technical issues with the processing of OpenCL source files. As an example, we explain how we solved some issues related to the `empty-struct-to-int` transformation. For full details of this, as well as changes to the `remove-unused-function` and `remove-unused-field` transformations, and the solution to handling of OpenCL source files, see [10].

The `empty-struct-to-int` transformation removes struct declarations and replaces all usages of the struct type with type `int`. Despite its name, the transformation does not only remove structs without any member but also structs with at most one unreferenced member. This is important as the behaviour for structs without any named member is undefined:[15] test cases with empty structs would be rejected by the interestingness test and the struct declaration could only be removed by the simple delta-reduction steps.

Completely empty structs cannot have an initialiser list when a variable of the struct type is defined. In this case the transformation is easy as it is possible to just replace every occurrence of the struct type with an integral type. However, structs with at least one named member can have an initialiser list even if the member itself is unreferenced. Moreover, initialiser lists can be nested for members and variables of array or struct type (Figure 8). The existing implementation of C-Reduce did not consider these cases and left the initialiser list unmodified. But again such a transformation result would have to be rejected in the interestingness test since according to the C99 standard "[n]o initializer shall attempt to provide a value for an object not contained within the entity being initialized" [5].

In order to solve the problem all structs are visited recursively and all usages of the rewritten struct are collected in a new tree-like data structure (Figure 9). Each leaf node represents an instance of the struct itself while the inner nodes represent structs for which one of its (recursive) members is of the changed struct type. This information is essential as also the initialiser lists of these "surrounding" structs have to be altered. Once all occurrences have been determined the specific initialiser lists—their position within the surrounding struct is denoted by the numbers on the edges—are simply replaced with a zero value.

5. EXPERIMENTAL EVALUATION

We evaluate two properties of our CL-reduce implementation: efficiency and robustness. We evaluate robustness by whether a completed reduction is free from undefined behaviour and still triggers a bug. Note that it might be the case that a bug observed in a reduced kernel is not the same as the bug in the original version. For efficiency, we consider the primary metric being the runtime per reduction. Aside from this, we also look at the final size of the reduced program compared to the original, as well as the success rate of individual reduction passes. Finally, we assess the runtime overhead of our new ShadowKeeper plugin, and the effectiveness of the plugin for detecting uninitialised values.

5.1 Experimental setup

The experimental campaign was carried out on 5 different OpenCL devices (configurations) hosted in 3 different machines. We use a–e to refer anonymously to the names of the devices, as we are not allowed to disclose results for some of the devices. These devices are disjoint from those used in Section 2. From an initial pool of 35,750 CLsmith-generated kernels, we selected 127 that yielded different results between an optimised and an unoptimised execution for some configuration. For these selected kernels, we performed a total of 272 automatic reductions, because some of the kernels presented wrong-code bugs on multiple configurations. Table 3 summarises the results, categorising tests according to the device they executed on (denoted "dev-x"); the CLsmith option that that was used to generate the tests, one of *basic*, *vectors*, *atomic reductions* (ar), *inter-thread communication* (itc), *atomics*, and *divergence* (see the CLsmith documentation for details of these modes), and the number x of parallel interestingness tests used for reduction (dnoted "nx"). For the kernels in *basic* and *vectors* modes, we manually reduced the number of executing work items to 1 to accelerate reduction. The $*$ indicates that the Clang Static Analyzer was not used in the interestingness test. A number of the kernels were found to contain undefined behaviours ("Failure" column) or data races ("Race" column). We have removed them from further evaluation. In addition, we found one test caused a compiler crash ("Crash" column) after reduction. The "Success" column shows the number of reductions performed that ended with a kernel free of undefined behaviours.

5.2 Reduction results

Of the 272 automatic reductions that we have attempted, 189 have been successful—a 69% success rate. We observed an average reduction in size of 99.2%, with a size of 844 bytes on average after automatic reduction (varying from 387 bytes to roughly 2000 bytes). The practical question is whether this is a simple enough test case to report to compiler engineers. While this is a difficult question to answer in general, the developers of GCC recommend reporting test cases of under 30 lines of code.[16] We found that this is generally the case for our reduced kernels, but we can further minimize the tests manually with little effort. In general, we recommend applying a final manual reduction pass, which make the final test case simpler to understand and more readable. Our experience shows that 15 minutes are enough to apply some complex, hard to automate reductions.

The average runtime of the reductions was roughly 6 hours when running sequential interestingness tests, going down to an average of 1.7 hours when executing 4 interestingness tests in parallel. An overview of the reduction times for the sequential case is given by the cumulative distribution function (CDF) plot of Figure 10. To gather these results in a timely fashion only the kernels for which we manually adjusted the runtime parameters so that only a single work item executes are included in the distribution. A point with coordinates (x, y) indicates that a fraction of y of the reduction attempts took x hours or fewer to complete. Our result for executing 4 interestingness tests in parallel is similar to that obtained in [15]. In order to further reduce the runtime, we attempted multiple powers of two for the number of parallel interestingness tests and found that running 4 in parallel is ideal, having greatly diminished runtime improvements if going over this value (see Figure 11).

[15]Empty structs are a GNU C extension of the C99 standard.

[16]See https://gcc.gnu.org/bugs/minimize.html, visited on 29/08/2015.

```
1 struct S0 {
2    int a[5];
3 };
4
5 struct S0 s = {{1,2,3,4,5}};
6 struct S0 as[2] = {{1}, {6,7}};
```

(a) Test case

```
1
2
3
4
5 int s = 0;
6 int as[2] = {0, 0};
```

(b) Correct transform

```
1
2
3
4
5 int s = {{1,2,3,4,5}};
6 int as[2] = {{1}, {6,7}};
```

(c) Actual transform

Figure 8: Example of the `empty-struct-to-int` transformation

```
1 struct S0 {
2    int a[3];
3 };
4
5 struct S1 {
6    int b;
7    struct S0 s0;
8 }
9
10 struct S0 s0 = {1,2,3};
11 struct S1 s1 = {0, {1,2,3}}
```

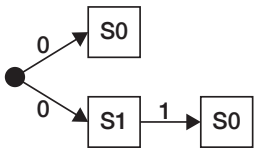

Figure 9: Example of the collected type usage information

Test category	Success	Crash	Race	Failure
dev-a_basic_n1	24	0	—	12
dev-a_basic_n4	20	0	—	9
dev-a_basic_n8	20	0	—	13
dev-b_vectors_n1	5	0	—	1
dev-b_ar_n1	15	1	0	4
dev-b_itc_n1	18	0	0	5
dev-b_atomics_n1	8	0	5	4
dev-b_divergence_n1	20	0	—	5
dev-c_basic_n8	6	0	—	0
dev-d_basic_n8	28	0	—	14
dev-e_basic_n1	3	0	—	2
dev-e_itc_n1	7	0	2	4
dev-a_basic_n1*	15	0	—	2

Table 3: Overview of the experimental results over 5 different OpenCL devices

Regarding the reduced kernels we deemed as failing due to presenting undefined behaviours, upon manual inspection, we observed that the interestingness tests were not checking for the occurrence of those particular undefined behaviours. This shows that writing the interestingness tests is not a trivial process, due to the multitude of possible undefined behaviours that have to be taken into consideration and checked for. In addition, checking for more undefined behaviours reduces the number of possible reductions that can be applied. Thus, we opted for a demand-driven approach to implementing undefined behaviour checks. In addition, there was an instance of a wrong-code test being reduced to a program triggering a compiler crash; our interestingness test should not allow this to occur (because we aim to distinguish between wrong-code bugs and arguably less interesting compiler crash bugs); we have not yet investigated the reason for this instance of *bug slippage* [3].

5.3 Evaluation of uninitialised value detection

Our ShadowKeeper plugin for Oclgrind offers better precision at the cost of performance. We evaluated the effectiveness of the plugin, both terms of runtime and precision.

For runtime, we compare executing Oclgrind with our ShadowKeeper plugin against default Oclgrind, as a baseline, and against Oclgrind with the previous uninitialised value plugin enabled. We run 1,000 kernels on each of the three different machines mentioned in Section 5.1, setting a hard timeout of 120 seconds. Using this limit, default Oclgrind times out in 35% of the cases, compared to 65% with either plugin enabled. We observe an expected increase in runtime when using ShadowKeeper, with an average slowdown of 4.6× compared to the baseline and 1.3× compared to the old plugin, a slight increase in runtime compared to the old plugin. However, compared to the tools that have inspired ShadowKeeper, Valgrind's Memcheck incurs a larger slowdown (typically a 20× slowdown compared to original runtime), while Clang's MemorySanitizer is slightly faster (typically a 3× slowdown compared to original runtime). Considering that MemorySanitizer executes the programs directly through the operating system, as opposed to Oclgrind having to simulate them, we consider ShadowKeeper competitive in terms of runtime.

The main purpose of developing the plugin was to improve upon the high rate of false positives the previous plugin presented. During our tests, we have not observed any instance of ShadowKeeper reporting a false positive bug. However, we have observed situations where ShadowKeeper failed to identify uninitialised values, such as a load from an undefined address. While this particular issue has been fixed in a later version of Oclgrind, we believe that even a perfect plugin would find itself in situations where it could not identify certain uninitialised values. This is because Oclgrind uses Clang to compile an OpenCL kernel into LLVM IR. If the original kernel exhibits undefined behaviour, it is not necessarily the case that the resulting LLVM IR also exhibits undefined behaviour, because Clang is free to fix on specific semantics for the otherwise undefined behaviour during translation. As ShadowKeeper analyses the LLVM IR, not the original OpenCL kernel, undefined behaviours that are eliminated during translation to LLVM IR will be missed.

6. RELATED WORK

The CLsmith tool that we use to generate random kernels [8] is built on top of the Csmith generator for C programs [15]. In addition, in prior work we also investigated the use of *equivalence modulo inputs* testing [7], a metamorphic testing technique [2], for finding OpenCL compiler bugs. The EMI method we proposed in [8] involved injecting dead-by-construction code into an OpenCL kernel, using an opaque predicate to ensure that the compiler cannot deduce that the code is dead; we have also applied this idea to the testing of compilers for the OpenGL shading language (GLSL) [4]. This method yields a simple approach

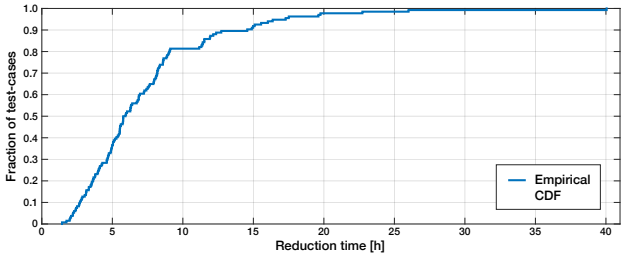

Figure 10: CDF plot of the runtime for test cases reduced with our CL-Reduce framework.

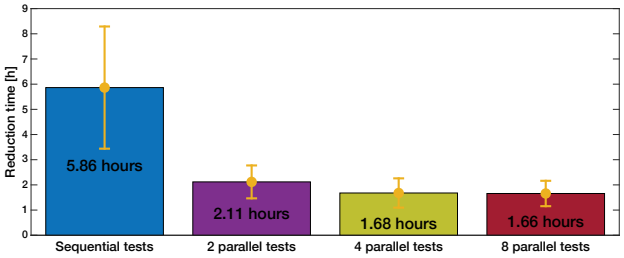

Figure 11: Average runtime of a reduction plotted against number of interestingness tests in parallel.

to test case reduction, because injected dead code can be eliminated without concern for undefined behaviour (being dead, the code in question cannot trigger undefined behaviour at runtime). An open problem for future investigation is whether random kernels generated by CLsmith and reduced by CL-reduce, or real-world kernels injected with dead-by-construction code and reduced as described above, yield more useful bug reports for compiler developers.

Oclgrind provides a form of dynamic analysis for OpenCL kernels via simulation. In contrast, the GPUVerify tool provides static warnings about data races and barrier divergence [1]. In principle we could incorporate GPUVerify into the interestingness test of CL-Reduce, but have found that GPUVerify does not yet scale well to kernels of the size produced by CLsmith.

For C compilers, the Csmith and C-Reduce tools provide a way to generate a large set of small programs that trigger compiler bugs. A remaining problem is that of *duplicate bugs*, where many reduced programs trigger bugs arising from a common root cause. Methods for automatically ranking reduced test cases in an attempt to prioritise programs that trigger a diverse range of bugs have been proposed [3]. Our OpenCL extension to C-Reduce paves the way for investigating these methods in the context of OpenCL.

7. CONCLUSIONS AND FUTURE WORK

We have presented the design and implementation of a novel extension to the C-Reduce test case reduction framework that handles OpenCL kernels. A by-product of this work is ShadowKeeper, a new plugin for the Oclgrind simulator that detects uninitialised memory accesses for OpenCL kernels. Our experiments and case study using CL-Reduce to automatically reduce large OpenCL test cases show that the method can be useful as an aid to finding small kernels that trigger compiler bugs. Open avenues for future work include adding further plugins to Oclgrind to detecting additional kinds of undefined behaviour, and investigating methods for ranking reduced bugs in order of priority.

Acknowledgements

We are grateful to James Price for support related to Oclgrind, to the developers of C-Reduce for accepting our contributions to the C-Reduce framework, and to the Clang developers for reviewing a submitted patch arising from this project. This work was supported by the EPSRC-funded HiPEDS CDT, a gift from Intel Corporation, and an equipment grant from GCHQ.

8. REFERENCES

[1] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Trans. Program. Lang. Syst.*, 37(3):10, 2015.

[2] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

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

[4] A. F. Donaldson and A. Lascu. Metamorphic testing for (graphics) compilers. In *MET*, 2016.

[5] International Organization for Standardization. ISO/IEC 9899:1999, Programming languages – C, 1999.

[6] Khronos Group. The OpenCL C specification, v2.0.

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

[8] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *PLDI*, 2015.

[9] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE*, 2007.

[10] M. Pflanzer. Automatic test case reduction of randomly generated OpenCL kernels. Master's thesis, Imperial College London, 2015. http://www.doc.ic.ac.uk/~afd/homepages/papers/pdfs/2015/PflanzerThesis.pdf.

[11] J. Price and S. McIntosh-Smith. Oclgrind: An extensible opencl device simulator. In *IWOCL*, 2015.

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

[13] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, 2005.

[14] E. Stepanov and K. Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *CGO*, 2015.

[15] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[16] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.