# Automated Test Generation for OpenCL Kernels using Fuzzing and Constraint Solving

**Citation for published version:**
Peng, C & Rajan, A 2020, Automated Test Generation for OpenCL Kernels using Fuzzing and Constraint Solving. in *GPGPU '20: Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit.* ACM, pp. 61-70, 13th Workshop on General Purpose Processing Using GPU (GPGPU 2020) , San Diego, California, United States, 23/02/20. https://doi.org/10.1145/3366428.3380768

**Digital Object Identifier (DOI):**
10.1145/3366428.3380768

**Link:**
Link to publication record in Edinburgh Research Explorer

**Document Version:**
Peer reviewed version

**Published In:**
GPGPU '20: Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit

# Automated Test Generation for OpenCL Kernels using Fuzzing and Constraint Solving

Chao Peng
School of Informatics
University of Edinburgh, UK
chao.peng@ed.ac.uk

Ajitha Rajan
School of Informatics
University of Edinburgh, UK
arajan@staffmail.ed.ac.uk

## ABSTRACT

Graphics Processing Units (GPUs) are massively parallel processors offering performance acceleration and energy efficiency unmatched by current processors (CPUs) in computers. These advantages along with recent advances in the programmability of GPUs have made them attractive for general-purpose computations. Despite the advances in programmability, GPU kernels are hard to code and analyse due to the high complexity of memory sharing patterns, striding patterns for memory accesses, implicit synchronisation, and combinatorial explosion of thread interleavings. Existing few techniques for testing GPU kernels use symbolic execution for test generation that incur a high overhead, have limited scalability and do not handle all data types.

We propose a test generation technique for OpenCL kernels that combines mutation-based fuzzing and selective constraint solving with the goal of being fast, effective and scalable. Fuzz testing for GPU kernels has not been explored previously. Our approach for fuzz testing randomly mutates input kernel argument values with the goal of increasing branch coverage. When fuzz testing is unable to increase branch coverage with random mutations, we gather path constraints for uncovered branch conditions and invoke the Z3 constraint solver to generate tests for them.

In addition to the test generator, we also present a schedule amplifier that simulates multiple work-group schedules, with which to execute each of the generated tests. The schedule amplifier is designed to help uncover inter work-group data races. We evaluate the effectiveness of the generated tests and schedule amplifier using 217 kernels from open source projects and industry standard benchmark suites measuring branch coverage and fault finding. We find our test generation technique achieves close to 100% coverage and mutation score for majority of the kernels. Overhead incurred in test generation is small (average of 0.8 seconds). We also confirmed our technique scales easily to large kernels, and can support all OpenCL data types, including complex data structures.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Massively parallel systems**.


*GPGPU '20, February 23, 2020, San Diego, CA, USA*

ACM ISBN 978-1-4503-7025-7/20/02…$15.00
https://doi.org/10.1145/3366428.3380768

## KEYWORDS

Software Testing, GPU, OpenCL, Test Case Generation, Fuzz Testing, Constraint Solving, Data Race

**ACM Reference Format:**
Chao Peng and Ajitha Rajan. 2020. Automated Test Generation for OpenCL Kernels using Fuzzing and Constraint Solving. In *General Purpose Processing Using GPU (GPGPU '20), February 23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3366428.3380768

## 1 INTRODUCTION

Recent advances in the programmability of GPUs, accompanied by the advantages of massive parallelism, energy efficiency, low management costs compared to a cluster of CPUs have made them attractive for general-purpose computations across many application domains [20]. However, writing correct GPU programs is a challenge owing to many reasons [13]– a program may spawn tens of thousands of threads, which are clustered in multi-level hierarchies, making it difficult to analyse; programmer assumes responsibility for ensuring concurrently executing threads do not conflict by checking threads access disjoint parts of memory; complex striding patterns of memory accesses are hard to reason about; GPU work-group execution model and thread scheduling vary platform to platform and the assumptions are not explicit. As a consequence of these factors, GPU programs are difficult to analyse and existing approaches [13] for verifying correctness are thwarted by high complexity of sharing patterns, combinatorial explosion of thread interleavings and space of possible data inputs. Existing techniques for testing GPU kernels, GKLEE [15] and KLEE-CL [4], incur a high overhead (from symbolic execution and constraint solving), do not handle all data types and have limited scalability.

There is an urgent need for a *fast*, *effective* and *scalable* technique to check correctness of GPU kernels. We seek to address this need by proposing a testing technique that combines fuzz testing with constraint solving. A fuzz tester (or fuzzer) is a tool that iteratively and randomly generates inputs with which it tests a target program. Fuzz testers were found to be surprisingly effective when compared to more sophisticated tools involving SMT solvers, symbolic execution, and static analysis of security applications. For instance, the popular fuzzer AFL has been used to find hundreds of bugs in popular programs [27][11] and found 76% more bugs when compared to a symbolic executor (angr) in a 24 hour period [23]. Currently, there are no available fuzz testers for GPU kernels. The first contribution in this paper is the development of a fuzz tester for OpenCL kernels. We evaluate the effectiveness of the generated inputs from the fuzzer by measuring branch coverage and fault finding (using seeded mutaitons) over the OpenCL kernels. We use the coverage measurement tool developed by Peng et al. [18] for this assessment.

It is well known that fuzzers, although fast and effective, can struggle with determining specific inputs to pass complex checks within programs. The second contribution in this paper addresses this weakness. When our fuzzer is unable to reach full (or high) coverage of the kernel, we complement it with a constraint solver[1] that is tasked with generating inputs for *uncovered* branches. The test inputs generated by the constraint solver combined with test inputs from the fuzzer form the complete test suite achieving high kernel code coverage.

A final contribution lies in the execution of the generated tests with a work-group *schedule amplifier*. Existing GPU architecture and simulators do not provide a means to control work-group schedules. The OpenCL specification provides no execution model for inter work-group interactions [21]. As a result, the ordering of work-groups when a kernel is launched is non-deterministic and there is, presently, no means for checking the effect of schedules on test execution. We provide this monitoring capability. For a test case $T_i$ in test suite $TS$, instead of simply executing it once with an arbitrary schedule of work-groups, we execute it many times with a different work-group schedule in each execution. We build a simulator that can force work-groups in a kernel execution to execute in a certain order. This is done in an attempt to reveal test executions that produce different outputs for different work-group schedules which inevitably point to problems in inter work-group interactions. Peng et al. [18] produced a partial scheduler by only fixing the first work group in the schedule. Schedules generated by the partial scheduler are not realistic, and resulted in creating deadlocks in some kernel executions, as reported in [18]. In this paper, we provide a simulator that can provide complete orderings over all the work groups, addressing limitations in the partial scheduler.

We empirically evaluate our test generation technique using 217 GPU kernels from open source projects and industry standard benchmark suites. Tests generated by the fuzzer achieves more than 90% branch coverage and mutation score in 86% and 51% of the subject kernels, respectively. For the 31 kernels with uncovered branches, we augmented tests from the constraint solver to achieve full branch coverage. Average mutation score improved significantly (54% to 73%) with the combined approach for these kernels. We found most of the survivng mutants to be arithmetic operator mutants that are hard to kill with control flow adequate tests. We plan to explore data flow guided test generation to kill such mutations in the future. Our schedule amplifier was able to uncover data races in 21 kernels. Overhead of our test generation technique is very small (average of 0.8 seconds over all kernels). Overall, we find our test generation framework, CLFuzz, provides a fast, effective and scalable means for testing OpenCL kernels.

In summary, the main contributions in this paper are:

(1) Fuzz tester that automatically generates tests using random mutations.
(2) Constraint-solver based test generation to complement the fuzz tester for uncovered kernel code.

[1]We invoke an SMT solver to generate inputs satisfying constraints for uncovered paths. We refer to the SMT solver as a *constraint solver* throughout the paper.

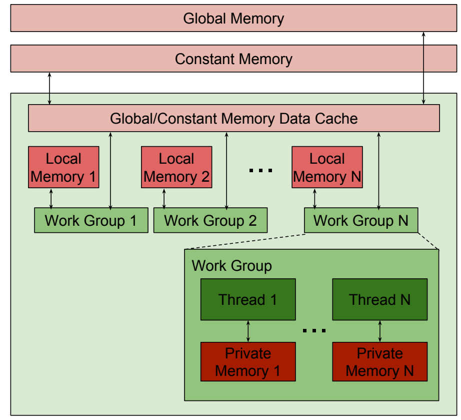


**Figure 1: Memory and thread hierarchy on GPUs**

(3) Schedule amplification to evaluate test executions using different work-group schedules.
(4) Empirical evaluation on a collection of 217 publicly available GPU kernels, examining coverage, fault finding and inter work-group interactions.

## 2 BACKGROUND

The success of GPUs in general purpose applications in recent years has been due to the ease of programming using the CUDA [1] and OpenCL [10] parallel programming models, which abstract away details of the architecture. CUDA and OpenCL use a similar abstraction of the memory and execution model of GPU programs (typically referred to as GPU *kernels*), as shown in Figure 1, and differences between them are mostly syntactic. We choose to focus our test generation approach on OpenCL kernels since it is more widely supported by GPU manufacturers.

OpenCL is a programming framework and standard set from Khronos, for heterogeneous parallel computing on cross-vendor and cross-platform hardware. In the OpenCL architecture, CPU-based *Host* controls multiple *Compute Devices* (for instance CPUs and GPUs are different compute devices). Each of these coarse grained compute devices consists of multiple *Compute Units* which in turn contain one or more *processing elements* (a.k.a *streaming processors*). The processing elements execute groups of individual threads, referred to as work-groups, concurrently. The functions executed by the GPU threads are called *kernels*, parameterised by thread and group id variables. OpenCL has four types of memory regions: global and constant memory shared by all threads in all work-groups, local memory shared by threads within the same work-group and private memory for each thread. Kernels cannot write to the constant memory.

GPUs have SIMT (single instruction, multiple thread) execution model that executes batches of threads (warps) in *lock-step*, i.e all threads in a work-group execute the same instruction but on different data. If the control flow of threads within the same work-group diverges, the different execution paths are scheduled sequentially until the control flows reconverge and lock-step execution resumes. Sequential scheduling caused by divergence results in a performance penalty, slowing down execution of the kernel.

The following listing presents a GPU kernel written using OpenCL that squares the contents of an array and then writes each array element as the sum of neighbouring elements. This example is derived from an application which detects edges in grayscale images. Each Thread in the original kernel computes the summation of values held by its neighbouring threads minus its own value and divides the result by 4. We simplify this kernel into a 1 dimensional version for illustrating purpose.

Listing 1: Example OpenCL Kernel

```
__kernel void foo(__global float *p) {
    int id = get_global_id(0);
    p[id] = p[id] * p[id];
    if (id != 0 && id != get_global_size(0) - 1) {
        barrier();
        float result = (p[id - 1] + p[id + 1] - p[id]) /
             2;
        p[id] = result;
    }
}
```

In this example, the built-in function `get_global_id()` returns thread_id and the `barrier()` function synchronises execution among threads. The example code, however, contains two types of bugs: barrier divergence and data race. We briefly describe the bug types before discussing them in the example. *Inter work-group data race* can occur when a global memory location is written by one or more threads from one work-group and accessed by one or more threads from another work-group. *Intra work-group data race* can occur when a global or local memory location is written by one thread and accessed by another from the same work-group. Barrier is a synchronisation mechanism for threads within a work-group in OpenCL and is used to prevent intra work-group data race errors. *Barrier divergence* occurs if threads in the same group reach different barriers, in which case kernel behaviour is undefined [3] and may lead to intra work-group data race. Data races and barrier divergence, according to Betts et al. [3], make GPU kernels harder for verification than sequential code.

In the example, barrier divergence occurs because the barrier within the `if` statement is not executed by all the threads. Additionally, the purpose of the barrier is to ensure the summation of array elements happens after the squaring operation. However, thread synchronisation is only enforced within a work-group. There is no mechanism provided by the GPU vendor or the OpenCL programming model to synchronise threads from across different work-groups. For instance, if we assume the work-group size to be 32 in this example, there is no way to ensure thread 31 (from a certain work-group) uses the correct p[32] value (after squaring) in the summation operation, as p[32] is controlled by a thread in a different work-group. As a result, inter work-group data races may occur in this kernel.

It is worth noting that there is no means to control or restrict the work-group scheduler in GPUs. The scheduler is free to produce any order of work groups to be executed on the available compute units. For the above example, different work-group schedules can produce different kernel outputs indicating the presence of a data race.

In this paper, we propose techniques for generating test inputs for OpenCl kernels that are effective in uncovering barrier divergence and inter work group data races.

## 3 RELATED WORK

We discuss related work in the context of work-group synchronisation, verification and testing of GPU kernels.

*Inter Work-group Synchronisation for OpenCL Kernels.* Barrier functions in the OpenCL specification [10] help synchronise threads within the same work-group. As mentioned earlier, there is no mechanism, however, to synchronise threads belonging to different work-groups. Xiao et al. [26] proposed an implementation of inter work-group barrier that relies on information on the number of work-groups. This method is not portable as the number of launched work-groups depends on the device. Sorensen et al. [22] extended it to be portable by discovering work-group occupancy dynamically. Their implementation of inter work-group barrier synchronisation is useful when the developer knows there is interaction between work-groups that needs to be synchronised. Our contribution is in detecting undesired inter work-group interactions, not intended by the developer.

*GPU Kernel Analysis for Data Races.* Verification of GPU kernels to detect data races has been explored in the past. Li et al. [14] introduced a Satisfiability Modulo Theories (SMT) based approach for analysing GPU kernels and developed a tool called Prover of User GPU (PUG). The main drawback of this approach is scalability. With an increasing number of threads, the number of possible thread interleavings grows exponentially, making the analysis infeasible for large number of threads. GRace [28] and GMRace [29] were developed for CUDA programs to detect data races using both static and dynamic analysis. However, they do not support detection of inter work-group data races.

GKLEE [15] and KLEE-CL [4], based on dynamic symbolic execution, provides data race checks for CUDA and OpenCL kernels, respectively. Both tools are restricted by the need to specify a certain number of threads, and the lack of support for custom synchronisation constructs. Scalability and general applicability is a challenge with these tools.

Leung et al. [13] present a flow-based *test amplification* technique for verifying race freedom and determinism of CUDA kernels. For a single test input under a particular thread interleaving, they log the behaviour of the kernel and check the property. They then amplify the result of the test to hold over all the inputs that have the same values for the property integrity-inputs. The test amplification approach in [13] can check the absence of data-races, not the presence. Additionally, their approach amplifies across the space of test inputs, not work-group schedules as done in our schedule amplifier. GPUVerify [3] is a static analysis tool that transforms a parallel GPU kernel into a two-threaded predicated program with lock-step execution and checks data races over this transformed model. The drawback of GPUVerify is that it may report false alarms and has limited support for atomic operations.

*Test Effectiveness Measurement.* Measuring test quality in terms of code coverage and fault finding is common for CPU programs [7, 19]. GKLEE is able to measure code coverage achieved by the tests it generated by translating the GPU code to its sequential version using Perl scripts and applying the Gcov utility, which disregards the GPU programming model. It can also report coverage achieved in the bytecode level as their execution depends on the bytecode virtual machine, but it is hard to map the coverage to the source code level for the developer's reference.

Peng et al. [18] presented the CLTestCheck framework that measures test effectiveness over OpenCL kernels with respect to branch, loop boundary, barrier coverage and mutation coverage using code instrumentation. The framework also provides limited work-group schedule amplification to check for the presence of inter work-group data races. This is done by executing each test with different work group schedules, where each schedule is generated by fixing the first work-group with the remaining work groups in default order. In this paper, we extend the schedule amplification technique by generating schedules that control the order of *all* the work-groups rather than only the first one, thus making the schedules more realistic.

*Test Input Generation for GPUs .* GKLEE [15] and KLEE-CL [4] are the only techniques in literature that provide test generation capability for GPU kernels. Both tools use symbolic constraint solving for test generation. GKLEE has the disadvantage that it does not support floating-point data types which are widely used in scientific computation GPU kernels. Additionally, test inputs generated by both GKLEE and KLEE-CL are in the form of hexadecimal values that are meant to run on KLEE virtual machine. They cannot be used directly to execute the original kernels and are not human readable. Finally, both GKLEE and KLEE-CL suffer from high overhead in test generation as they rely on symbolic execution and constraint solvers. Scalability to large kernels is also an issue because of the high overhead and the path explosion problem associated with symbolic execution. It is worth noting that KLEE-CL is currently not maintained and we could not get it to run on current OpenCL versions and GKLEE is only applicable to CUDA kernels.

*Fuzz Testing.* To mitigate the overhead and scalability problems associated with symbolic execution, we use fuzz testing in our test generation approach. Fuzz testing is based on randomly generating or mutating test inputs and has been shown to be fast and surprisingly effective [6, 24]. However, fuzzing based on random mutations, typically finds it hard to reach program parts protected by complex checks. Other techniques including constraint solving and search-based testing have been proposed to guide fuzzing in finding inputs that are capable of reaching these program parts. The combination of constraint solving and fuzzing has been effective in detecting security bugs in CPU, mobile and web applications [8, 9]. Sapienz [16] utilises search-based exploration and random fuzzing for testing Android applications and uncovered 558 previously unknown crashes in the top 1,000 Google Play apps. A comprehensive overview of fuzz testing techniques over the last decade can be found in [17, 25].

Fuzz testing for GPU kernels has not been explored previously. In the next Section, we discuss how we combine the fast and scalable nature of fuzz testing with the rigor of constraint solving to produce an effective test generator for GPU kernels.

# 4 OUR APPROACH

In this section, we present the CLFuzz framework that provides automated test input generation using 1. *Mutation-based fuzzing* and 2. *Selective constraint solving* for control conditions that remain uncovered with fuzz-based tests. The framework also provides a *Schedule amplifier* that generates several work-group schedules and executes the generated tests with the numerous schedules to detect potential data races. We discuss each of these capabilities in the rest of this section.

## 4.1 Mutation-based Fuzzing

Our technique for mutation-based fuzzing has the following steps,

(1) Generate a random seed with values for each argument (adhering to its data type) of a given kernel.
(2) Execute the seed and record branch coverage achieved over the kernel code. Add the seed to the test suite.
(3) Pick a test from the test suite, generate another test by mutating the value of one of the arguments of the kernel, keeping the other argument values unchanged.
(4) Execute the new test and measure branch coverage achieved.
(5) If the new tests results in additional branches being covered, add it to the test suite and go to Step 3.
(6) If no new branches are covered, discard the test and go to Step 3.

Our approach for mutation-based fuzzing supports all data types in OpenCL, as seen in Table 1. We use CLTestCheck [18] to measure branch coverage of test executions (used in Steps 2 and 4 above). We enhanced the CLTestCheck framework to check if tests cover additional branches (Steps 5 and 6 above).

*Fuzzer Limitation.* Since our mutation-based fuzzer randomly mutates inputs, albeit with the goal of increasing branch coverage, the generation of a "specific" input required to pass complex checks in the kernel (i.e., condition checks that require inputs to have a particular value or very few values) is extremely unlikely. Consider the example kernel code snippet in the listing below.

Listing 2: Example OpenCL Kernel

```
__kernel void complexCheck(__global int x) {
 ...
    if (x == -2987) {
        specialCalc();
    }
 ...
}
```

The above kernel function checks if the kernel argument $x$ matches $-2987$. If a match occurs then a special calculation is done. However, due to the nature of fuzzing, it is extremely unlikely that a fuzzer will ever satisfy the predicate. The mutation-based fuzzing technique will cover the false predicate easily and apply random mutations on the existing path with a very small chance of setting $x$ to the specific value of $-2987$ (likelihood of 1 out of $2^{32}$).

## 4.2 Selective Constraint Solving

We address the limitation of mutation-based fuzzing in determining specific inputs to pass complex checks using selective constraint solving. When the fuzzer is unable to increase branch coverage after

| Category | OpenCL API Type | Description |
|---|---|---|
| Scalar | cl_char | Signed, 8-bit |
| | cl_uchar | Unsigned, 8-bit |
| | cl_short | Signed, 16-bit |
| | cl_ushort | Unsigned, 16-bit |
| | cl_int | Signed, 32-bit |
| | cl_uint | Unsigned, 32-bit |
| | cl_long | Signed, 64-bit |
| | cl_ulong | Unsigned, 64-bit |
| | cl_float | Floating point, 32-bit |
| | cl_double | Floating point, 64-bit |
| | cl_half | Floating point, 16-bit |
| Vector | scalarn | A vector of n scalar values, e.g., int2, float16 |
| Struct | struct | A struct comprised of scalar and vector values |
| Image | imagend_t | An nD image, e.g., image2d_t |

**Table 1: Summary of kernel argument data types**

going through a predetermined amount of mutations (proportional to number of kernel arguments), we consider the fuzzer to have reached its limit. We then invoke selective constraint solving to generate tests for uncovered branches in the kernel.

For each uncovered branch, selective constraint solving first gathers path constraints to reach the uncovered code location. This is done with the aid of a control flow graph built from the abstract syntax tree (AST) of the kernel code [2]. We traverse the CFG, recording path constraints, until the uncovered branch condition is reached. We then feed the path constraints to the Z3 [5] constraint solver to determine an input that will satisfy the constraints. If such an input can be found then it is added to the test suite as a test exercising the uncovered branch. We repeat this for all uncovered branches.

Constraint solvers incur high overhead, proportional to the number and complexity of path constraints. The advantage with selective constraint solving is that we keep the number of path constraints given to the constraint solver to a reasonable number. Mutation-based fuzzing complemented by selective constraint solving, thus, helps achieve fast, effective and scalable test generation. This is in contrast to existing approaches like, GKLEE and KLEE-CL, that use symbolic constraint solving to generate all the tests, incurring high overhead with limited scalability.

## 4.3 Schedule Amplification

As mentioned earlier in Section 2, no mechanism is provided by GPU vendors to manipulate and set work-group schedules. Work-group schedule used in kernel executions is non-deterministic and can cause data races. To allow monitoring for such data races, the *schedule amplifier* provides the following two capabilities, 1. Generates multiple work-group schedules, and 2. Executes kernels with different work group schedules and checks for discrepanies in outputs. We built the schedule amplifier as an extra layer over the standard OpenCL built-in functions.

To better understand our approach for generating multiple work-group schedules, we first present how work-groups are typically launched on GPUs. Consider the example in Figure 2 that illustrates 8 work groups required for the execution of a kernel. Assume there are only 4 available physical processing elements on the GPU. As a result, at any given time, at most 4 work groups can be running in parallel. The default schedule will pick four work-groups to execute on the 4 processing elements. We assume the default schedule chooses work-groups 0 to 3 to go first. Once one of them finishes, it will launch the next work-group. This is repeated until all the work-groups finish execution. When a work-group is running, threads in this work-group acquire thread IDs and the work-group ID by calling built-in functions get_global_id() and get_group_id(). These IDs are then typically used by the threads to locate the region of input data to process.

To generate different work-group schedules, the schedule amplifier manipulates the values returned by the built-in functions. To do this, we maintain an array *new_id* storing a sequence of numbers from 0 to $the\ number\ of\ groups\ -\ 1$ in a shuffled order. When the kernel function asks for its work-group ID, the modified function gives the value of new_id[global_id] rather than the global_id. The modification of global_id does not affect the semantics of the kernel code and is used solely to launch work-groups in different orders on the compute units. An example of shuffled work-group order is shown in blue in Figure 2 where work-groups 3, 5, 2, 6 are launched first, followed by 4, 1, 0, 7. Although the example shows a 1-dimenional work-group schedule execution model, our schedule amplifier is capable of supporting multi-dimensional work-group schedules.

Kernels usually launch hundereds of work-groups, this makes it impractical to generate all possible work-group schedules. In this paper, we randomly generate 10 different work-group schedules for every test execution over every kernel in our experiment. Our schedule amplifier allows the user to specify the number of work-group schedules to be generated.

The schedule amplifier launches every kernel execution with each of the generated work-group schedules and checks if there are any discrepancies in kernel output. Differences in kernel output indicate problems in inter work-group interactions. Thus, with little extra cost, we are able to check significantly more number of

[2]Our implementation of CFGs for OpenCL kernels is available at https://github.com/chao-peng/CLFuzz

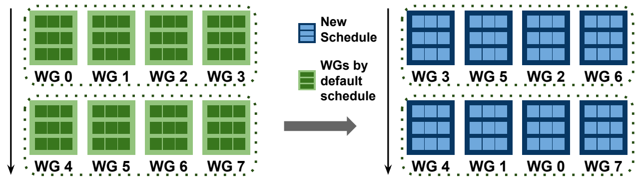


**Figure 2: Schedule controlled by the enhanced scheduler**

schedules than is currently possible, achieving better coverage of the work-group schedule space.

The partial schedule generator in CLtestCheck [18] generates work-group schedules by only manipulating and fixing the first work-group while using the default schedule (set by the GPU) for the remaining work-groups. This technique for partial scheduling is not effective as it results in unrealistic schedules that causes deadlocks from launching work group ids that exceed the number of available compute units. We avoid this problem in our approach with complete work-group scheduling that ensures work groups launched match available compute units and produces valid schedules with no deadlocks for the subject kernels in our experiment. The kernel developer also has better control over work-group schedules with our schedule amplifier.

## 4.4 Host Code Generation

GPU kernels are only responsible for computation over input data residing in the GPU memory. Other tasks, such as reading data from a file, transferring data to and from the GPU memory, executing the kernel and validating the output are implemented in the host code that runs on a CPU by the developer. Writing host code to do these tasks can be laborious and time consuming. To ease the burden on the developer, we automatically generate host code by analysing the kernel interface and allocating GPU memory as needed.

Generated test inputs and work group schedules are stored in a file adhering to a pre-defined format. The host code generator then reads the input data from this file, allocates GPU memory and sends the data to the allocated memory according to data types and sizes, compiles and executes the kernel, reads the output from the GPU and stores the output in another file.

## 4.5 Implementation of the Framework

The CLFuzz framework is implemented using Clang LibTooling [2]. Building a CFG from AST, gathering path constraints, extracting kernel interface are all implemented within this framework. The kernel interface comprising kernel arguments, their data type and scope is stored on a data file, shown in Table 2. The developer can modify this data file to specify attributes, such as desired size of arrays and if the argument is an input or output parameter. The framework is written in Python and uses PyOpenCL API [12] for kernel execution and PyZ3 [5] for constraint solving. We use the CLCov and CLMT tools from the CLTestCheck framework [18] to measure code coverage and fault finding achieved by the generated tests. The implementation of our CLFuzz framework is available at https://github.com/chao-peng/CLFuzz.

| Property of an argument | Description |
| --- | --- |
| cl_scope | The address apace qualifier of parameters which can be global, local, private, and constant. |
| cl_type | Data type of the parameter. |
| pointer | True if the parameter is an array. |
| size | Desired size of an array. |
| fuzzing | True by default indicating it needs random input. |
| init_file<br>initial_value | The user can specify an initial value or provide a file to initialise the parameter if needed. |
| result | True if it is used to store the output of the kernel. |
| pos | The position of the parameter in the interface. . |

**Table 2: Kernel interface information**

# 5 EXPERIMENT

In our experiment, we evaluate the feasibility and effectiveness of mutation-based fuzz testing, selective constraint solving and schedule amplification proposed in Section 4 using 217 OpenCL kernels from open source projects and industry standard benchmark suites. We investigate the following questions:

**Q1. Effectiveness of Fuzz Testing:** *What is the branch coverage and fault finding achieved by test inputs generated by the fuzzer?* We measure branch coverage using the coverage measurement tool, CLTestCheck [18]. For fault finding, we generate mutants by analysing the kernel source code and applying mutation operators provided by CLTestCheck to eligible locations. We then assess number of mutants killed by the generated tests for each benchmark. To check if a mutant is killed, we compared execution results between the original kernel and mutant.

**Q2. Effectiveness of Selective Constraint Solving:** *Can selective constraint solving generate tests that enhance coverage and fault finding achieved by fuzz tests?* For kernels over which fuzz tests do not achieve 100% branch coverage, we augment the test suite with tests from selective constraint solving and check if there is an improvement in coverage and/or mutation score.

**Q3. Effectiveness of Schedule Amplification:** *Is the schedule amplifier able to detect inter work-group data races ?* Inter work-group data races occur when test executions produce different outputs for different work-group schedules. For each test, we generate 10 different work-group schedules with the schedule amplifier. The kernel is then executed with the test using each of the 10 different work-group schedules, and we check if the outputs from the executions differ.

*Subject Kernels.* We use 217 kernels collected from the following benchmark suites for our experiments:

- 24 kernels collected from open source projects bilateral, cl-practical, DeepCL and gaussian-blur,
- 38 kernels from the OpenDwarfs benchmark suite,

- 25 kernels from the Parboil benchmark suite,
- 47 kernels from the Polybench benchmark suite,
- 45 kernels from the Rodinia benchmark suite and,
- 38 kernels from the SHOC benchmark suite.

Our subject kernels span a wide range of application domains including scientific computing, image processing, biomolecular simulation, linear algebra, data mining, heterogeneous computing, stencil computations, among others. The large and diverse set of subject kernels varying in size and complexity, ranging from 12 to 2235 lines of code, containing all OpenCL supported data types, allows us to evaluate feasibility, overhead and scalability of our test generation approach.

Our experiments are performed on a desktop with an Intel Core i5-6500 3.2GHz quad-core CPU and an Intel HD Graphics 530 GPU using the Intel OpenCL SDK 2.1.

## 6 RESULTS AND ANALYSIS

For each of the 217 subject kernels in our experiment, we generated test inputs using CLFuzz and report results in terms of coverage achieved, fault finding and overhead incurred. We executed the test suites 20 times for each measurement. Our results in the context of the questions in Section 5 is presented below.

### 6.1 Effectiveness of Fuzz Testing

Figure 3 presents a frequency plot with number of kernels for which fuzz testing achieved branch coverage and mutation score in the ranges specified on the x-axis. We stop fuzz testing when branch coverage achieved does not increase after 50 mutation attempts. The average number of generated tests across all subject kernels is 45 and median is 29. The kernel with most generated tests is `MD` from SHOC with 143 tests. The maximum time taken for generating tests is 2 seconds (for the `MD kernel`). On average, test inputs generated by the random fuzzer achieved 91.5% branch coverage and 74.9% mutation score across all subject kernels.

As seen in Figure 3, fuzz testing is able to achieve full coverage for 186 out of 217 kernels, and 92% for one of the other kernels. With respect to mutation score, fuzz testing achieves 100% for 70 kernels and over 90% for 40 kernels. In the subsequent paragraphs, we analyse why fuzz testing was not as effective in achieving high branch coverage and mutation scores for some of the other kernels.

*Branch coverage.* For 31 out of 217 kernels, fuzz testing does not achieve full branch coverage with the generated test inputs. Upon investigation, we found the following main reasons for uncovered branches with the fuzzer.

**1. Requiring a specific value for one or some of the array elements** is the main limitation for fuzzers, and appears in 17 kernels. As inputs are generated and mutated randomly, if a branch condition requires a specific input value for its satisfaction, there is a very low likelihood of the fuzzer being able to satisfy such a condition. The following listing illustrates a code snippet from a subject kernel (`Hidden Markov Model`) from the OpenDwarfs benchmark suite to exemplify this issue.

Listing 3: The acc_b_dec kernel from bwa_hmm_opencl_11

```
__kernel void acc_b_dev( int obs_t, //current observation
    other arguments...) {
    unsigned int idx = get_group_id(0) * get_local_size
        (0) + get_local_id(0);
    unsigned int idy = get_group_id(1) * get_local_size
        (1) + get_local_id(1);

    if (other conditions && obs_t == idy) {
        //Computations using idx and idy
    }
}
```

In Listing 3, variable *idy* represents the ID in the y-axis of the current thread whose maximum value is the number of threads (1024 in our experiment). When generating a value for integer variable *obs_t* which is used in the branch condition, the probability that it will match the exact value of *idy* is very low.

**2. Boundary check before accessing elements of input arrays** results in low coverage among 6 subject kernels (kmean_2 in the OpenDwarfs benchmark, sad2, convs, tpacf, bfs in Parboil, srad_6, leukocyte_find_ellipse_kernel_1 and leukocyte_find_ellipse_kernel_2 in Rodinia). The following code listing extracted from the bfs kernel illustrates this issue.

Listing 4: Example boundary check of OpenCL kernel

```
__kernel void BFS_kernel( int no_of_nodes, //number of
     array elements
    other arguments...) {
    int tid = get_global_id(0);

    if (tid < no_of_nodes) {
        int pid = q1[tid]; //the current frontier node
        //Computations on the frontier node
    }
}
```

For this kernel, the fuzzer generates values for *no_of_nodes* that sets the branch condition to true. However, it is unable to find values that can set the condition to false, leaving the false branch uncovered.

**3. Nested control flows with strict conditions** is challenging for mutation-based fuzz tests to satisfy and this issue appears in 6 kernels. This is because the likelihood of random inputs satisfying conditions with specific value checks further reduces when the checks are nested.

*Fault Finding.* for the subject kernels is assessed with the help of the mutant generation component in the CLTestCheck framework [18]. The framework produces kernel mutants by mutating arithmetic, relational, logical, bitwise, assignment operators and barriers. The mutation score, percentage of mutants killed, is used to estimate fault finding capability of test inputs with the subject kernels. Each kernel is run 20 times to determine the killed mutants. A mutant is considered killed if the test suite generates different outputs on the mutant and original kernel on all 20 repeated runs of the test suite.

In general, we find that fuzz-based test suites achieving high branch coverage also achieve high mutation score. For 110 kernels, tests suites achieving full branch coverage achieved more than 90% mutation score. However, for 66 kernels with full branch coverage, mutation score achieved is not very high, between 60% and 89%. This is because control flow adequate tests are not designed to be effective in killing mutations that do not affect the control flow,

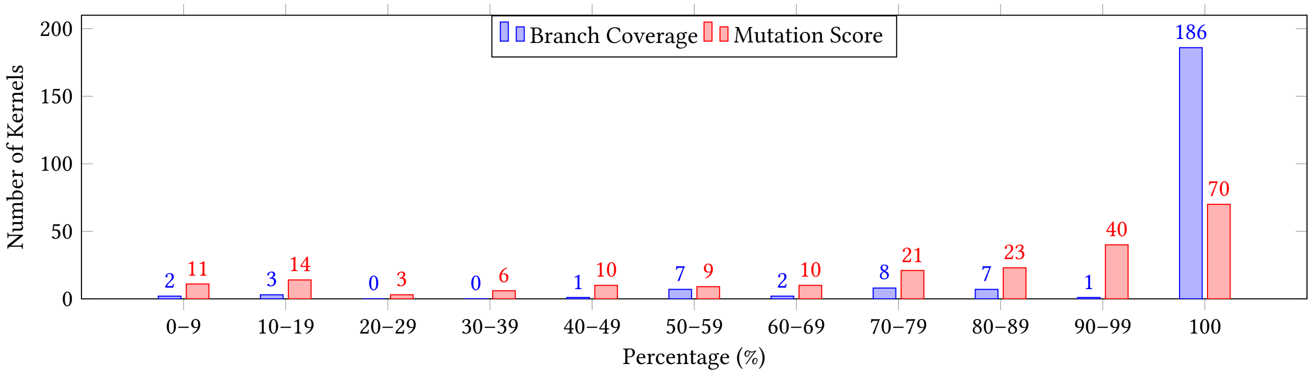


**Figure 3: Frequency of branch coverage and mutation score across subject kernels**

like many of the arithmetic operation mutants. Relational operator mutations also survive in our evaluation. Most of the surviving relational operator mutations made slight changes to operators, such as < to <=, or > to >= and vice versa. The test inputs generated by the fuzzer missed these boundary mutations. Data flow coverage adequate tests may be better suited at killing such mutations.

A smaller fraction of kernels (44 out of 217) have low mutation scores, less than 50%. Many of the surviving mutants are arithmetic operator mutants and boundary mutations. It is worth noting that 31 kernels in our evaluation do not have full branch coverage with fuzzer tests. In the next section we check if increasing branch coverage with tests generated by constraint solving helps increase the mutation score for these kernels.

## 6.2 Effectiveness of Constraint Solving

For the 31 kernels that do not have full branch coverage with fuzz tests, we run the constraint solver to generate tests for uncovered branches. We find all the path constraints generated for uncovered branches could be satisfied for all 31 kernels. As a result, fuzz tests combined with the tests generated by the Z3 solver achieved 100% branch coverage over all 31 kernels. We did not use the constraint solver over the remaining kernels as they were fully covered using just fuzz tests.

More specifically, the 17+6 kernels involving uncovered branches with conditions checking a specific value or boundary before accessing elements of arrays, as described in Section 6.1 are easily covered by the constraint solver by assigning the desired value indicated in the constraint. The overhead in solving such constraints is small, and only takes 0.2 seconds. With the constraint solver tests, the average mutation score of these kernels increases from 60.9% to 77.3%.

Among the 6 kernels with nested control flows and strict conditions, the `hotspot` kernel from Rodinia takes least time (1 second) for generating tests satisfying the path constraints for an uncovered branch condition that requires a variable to be within range for entering the true branch and out of range for entering the false branch. It is also worth noticing that the `nqueens1` kernel from the OpenDwarf benchmark has a deep nested control flow of 5 levels but only takes 1.3 second for the constraint solver to reach the deepest path. This is because the constraints for the control flow conditions were quite simple, only requiring an array element to be within different ranges.

The most time-consuming constraint solving happens in the `mergesort` kernel from Rodinia that takes 1 minute. The implementation of the mergesort algorithm handles 4 different possibilities that check the presence of elements in array A and array B. When both arrays have elements, an additional branch compares the elements and stores them in the correct location within the result array. Solving these different possibilities along with values for all array elements and their data dependencies takes longer than other kernels with simpler path constraints. Mutation score of this kernel is increased from 53% to 95% with the constraint solver tests.

Figure 4 illustrates a comparison of mutation score achieved by fuzz tests versus fuzz + constraint solver tests for the 31 kernels that used the constraint solver. The average mutation score for the 31 kernels increases from 54.3% to 73.3% with the combined approach. We find including tests from the constraint solver improves the mutation score significantly for 14 out of the 31 kernels. Among the remaining kernels with unchanged mutation score, for 8 of them the constraint solver generates tests to cover false branches of conditions. There is no kernel code within the false branch. As a result, no new mutations are exercised. Surviving mutants in all 17 kernels, as with fuzz tests, are either arithmetic operator mutations or boundary mutations. We will explore augmenting our test generation technique with data flow coverage to kill these mutant types.

## 6.3 Effectiveness of Schedule Amplification

Each test generated by the fuzzer and selective constraint solver is executed with 10 different schedules generated by the schedule amplifier. It is worth noting that the schedules generated by our schedule amplifier were all valid – there were no instances of kernel deadlock resulting from launching work group ids that exceed the number of available compute units. On the other hand, work-group schedules generated by the partial scheduler in CLTestCheck had several instances of kernel deadlock due to unrealistic work-group schedules. [3]

[3] Our experiment includes all 82 kernels used in the evaluation of CLTestCheck and also 135 additional ones.

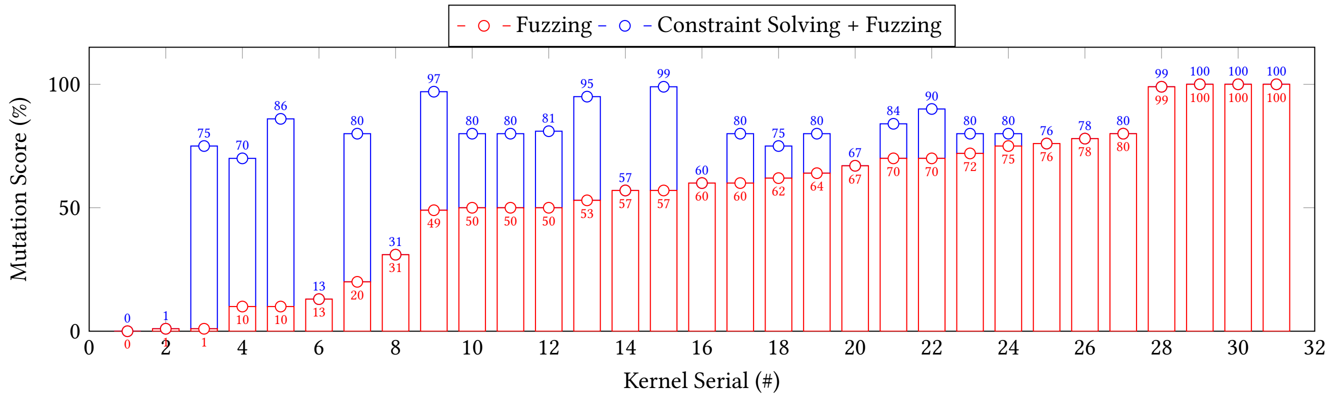


**Figure 4: Mutation score achieved over the 31 kernels using "Only Fuzzing" versus "Fuzzing + constraint solving".**

*Data Races.* Our schedule amplification technique was able to detect data races in 21 kernels, as shown in Table 3. Tests executed on these kernels produced different outputs on at least 2 out of 10 different work-group schedules. Each work-group schedule is additionally executed 20 times.

19 out of the 21 kernels with data races, produce the same output when the kernel is repeatedly executed with a fixed work-group schedule and test but the outputs across different work-group schedules is inconsistent. This observations confirms inter work-group data race that occurs when threads from different work-groups make read/write or write/write access to the same global memory location.

2 out of the 21 kernels (`cdf_1` and `swat_2` from the OpenDwarf benchmark suite) produce inconsistent output across repeated executions with a fixed work-group schedule and test. This indicates intra work-group data race that occurs when threads within the same work-group have memory access conflicts.

## 6.4 Scalability and Overhead

The largest kernel in our data set is the `heartwall` kernel from the Rodinia benchmark suite with 2235 Lines of Code. Our technique for test generation easily scales to this kernel, only taking 51 seconds to achieve full branch coverage. The kernels in our data set cover all the basic data types supported by OpenCL and include complex data structures. We were able to verify that CLFuzz was able to generate tests efficiently for all the kernels supporting all data types and constructs.

Time consumed by fuzz testing ranges from 0.01 seconds (`reduce_1 kernel` from SHOC with 1 test) to 2 seconds (`MD kernel` from SHOC with 143 tests) for the 217 kernels. Factors affecting the overhead of the fuzzer are the number of tests generated and the data structure of the kernel input. The `MD kernel` uses a OpenCL-specific data type, *double4*, which is a vector of 4 double values. Additional time is needed by CLFuzz for converting and storing the test inputs for such special types.

Constraint solving takes between 0.2 seconds to 1 minute across the 31 kernels to generate tests for uncovered branches. Since we only use the constraint solver selectively, for uncovered branches, the overhead incurred is not considerable.

The schedule amplifier does not introduce noticeable overhead as our framework for manipulating schedules is implemented at the OpenCL interface level. The schedule amplifier ensures there is no idle computing resource when executing the generated work-group schedules. In contrast, the partial schedules generated by Peng et al. [18] does not make efficient use of the compute units. When the first work-group is executing, their approach requires other work-groups to wait till execution of the first one is finished, making kernel execution slower.

# 7 CONCLUSION

We present a test generation technique for OpenCL kernels that combines mutation-based fuzzing and selective constraint solving aimed at achieving high branch coverage. Our mutation-based fuzzer generates tests by randomly mutating kernel argument values with the goal of increasing branch coverage. Our fuzzer supports all OpenCL data types. When the fuzzer is unable to increase coverage, we gather path constraints for uncovered branches and use the Z3 constraint solver to generate tests for them. We also provide a schedule amplifier, that generates multiple work-group schedules with which to execute each of the generated tests. The schedule amplifier helps uncover inter work-group data races.

We evaluated our test generation and schedule amplification technique using 217 OpenCL kernels, varying in size and complexity. We find mutation-based fuzzing on its own produces 100% branch coverage for 186 of the 217 kernels. For 31 kernels that did not have full coverage, we augmented the fuzz tests with tests generated by the constraint solver to achieve 100% branch coverage. Fault finding for 110 (out of 217) kernels with mutation-based fuzzing was > 90%. Average fault finding achieved with fuzz-based tests across all kernels was 74.9%. For the 31 kernels that were augmented with constraint solver tests, the average mutation score increased from 61% to 77%. Mutations that were not killed by the generated tests were primarily arithmetic operator mutations and boundary value mutations. Control-flow adequate tests are not effective in catching such mutations. In the future, we will explore test generation techniques that also target data flow in kernels. We were able to uncover data races in 21 kernels with our schedule

| Benchmark | Kernel | Type of Data Race |
|---|---|---|
| SHOC | MD, rdwdot, reduce1, spmv3 | Inter work-group |
| Rodinia | cdf_4 | Inter work-group |
| Polybench | 2mm_1, 3mm_1, 3mm_2, 3mm_3, adi_2, covariance_1, mat_2, syrk, syr2k | Inter work-group |
| Parboil | sgemm | Inter work-group |
| OpenDwarf | cfd_1, swat_2 | Intra work-group |
| | hmm_3, hmm_15, sad_1 | Inter work-group |
| Open Source Project | deepcl_forward_1 | Inter work-group |

**Table 3: Kernels with data races**

amplifier. The overhead of our test generation technique was negligible (average 0.8 second). In summary, we find our test generation technique combining fuzzing with constraint solving, and schedule amplification is fast, effective and scalable.

## REFERENCES

[1] Cuda zone (Sep 2017), https://developer.nvidia.com/cuda-zone
[2] Clang: a c language family frontend for llvm (March 2018), http://clang.llvm.org/
[3] Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: Gpuverify: a verifier for gpu kernels. In: ACM SIGPLAN Notices. vol. 47, pp. 113–132. ACM (2012)
[4] Collingbourne, P., Cadar, C., Kelly, P.H.: Symbolic testing of opencl code. In: Haifa Verification Conference. pp. 203–218. Springer (2011)
[5] De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
[6] Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: Proceedings of the 31st International Conference on Software Engineering. pp. 474–484. IEEE Computer Society (2009)
[7] Gay, G., Rajan, A., Staats, M., Whalen, M., Heimdahl, M.P.: The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. ACM Transactions on Software Engineering and Methodology (TOSEM) **25**(3), 25 (2016)
[8] Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: ACM Sigplan Notices. vol. 43, pp. 206–215. ACM (2008)
[9] Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. Citeseer
[10] Group, K.O.W.: The opencl specification version 2.2 (May 2017)
[11] Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 2123–2138. ACM (2018)
[12] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. Parallel Computing **38**(3), 157–174 (2012). https://doi.org/10.1016/j.parco.2011.09.001
[13] Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying gpu kernels by test amplification. In: ACM SIGPLAN Notices. vol. 47, pp. 383–394. ACM (2012)
[14] Li, G., Gopalakrishnan, G.: Scalable smt-based verification of gpu kernel functions. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. pp. 187–196. ACM (2010)
[15] Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: Gklee: concolic verification and test generation for gpus. In: ACM SIGPLAN Notices. vol. 47, pp. 215–224. ACM (2012)
[16] Mao, K., Harman, M., Jia, Y.: Sapienz: Multi-objective automated testing for android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. pp. 94–105. ACM (2016)
[17] Miller, C., Peterson, Z.N., et al.: Analysis of mutation and generation-based fuzzing. Independent Security Evaluators, Tech. Rep **56**, 127–135 (2007)
[18] Peng, C., Rajan, A.: Cltestcheck: Measuring test effectiveness for gpu kernels. In: International Conference on Fundamental Approaches to Software Engineering. pp. 315–331. Springer (2019)
[19] Rajan, A., Heimdahl, M.P.: Coverage metrics for requirements-based testing. University of Minnesota (2009)
[20] Rajan, A., Sharma, S., Schrammel, P., Kroening, D.: Accelerated test execution using gpus. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 97–102. ACM (2014)
[21] Sorensen, T., Donaldson, A.F.: The hitchhiker's guide to cross-platform opencl application development. In: Proceedings of the 4th International Workshop on OpenCL. p. 2. ACM (2016)
[22] Sorensen, T., Donaldson, A.F., Batty, M., Gopalakrishnan, G., Rakamarić, Z.: Portable inter-workgroup barrier synchronisation for gpus. In: ACM SIGPLAN Notices. vol. 51, pp. 39–58. ACM (2016)
[23] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. vol. 16, pp. 1–16 (2016)
[24] Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Pearson Education (2007)
[25] Takanen, A., Demott, J.D., Miller, C., Kettunen, A.: Fuzzing for software security testing and quality assurance. Artech House (2018)
[26] Xiao, S., Feng, W.c.: Inter-block gpu communication via fast barrier synchronization. In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. pp. 1–12. IEEE (2010)
[27] Zalewski, M.: American fuzzy lop (afl) fuzzer (2017)
[28] Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: Grace: a low-overhead mechanism for detecting data races in gpu programs. In: ACM SIGPLAN Notices. vol. 46, pp. 135–146. ACM (2011)
[29] Zheng, M., Ravi, V.T., Qin, F., Agrawal, G.: Gmrace: Detecting data races in gpu programs via a low-overhead scheme. IEEE Transactions on Parallel and Distributed Systems **25**(1), 104–115 (2014)