Automated Testing for GPU Kernels

Chao Peng

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2021

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 widely used in various general-purpose computing domains. However, this has also made testing GPU kernels critical to ensure that their behaviour meets the requirements of the design and specification.

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.

In this thesis, we present novel approaches to measure test effectiveness and generate tests automatically for GPU kernels. To achieve this, we address significant challenges related to the GPU execution and memory model, and the lack of customised thread scheduling and global synchronisation. We make the following contributions:

First, we present a framework, CLTestCheck, for assessing the quality of test suites developed for GPU kernels. The framework can measure code coverage using three different coverage metrics that are inspired by faults found in real kernel code. Fault finding capability of the test suite is also measured by the framework to seed different types of faults in the kernel and reported in the form of mutation score, which is the ratio of the number of uncovered faults to the total number of seeded faults.

Second, with the goal of being fast, effective and scalable, we propose a test generation technique, CLFuzz, for GPU kernels that combines mutation-based fuzzing for fast test generation and selective SMT solving to help cover unreachable branches by fuzzing. 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 and supports GPU-specific data types such as images. When fuzz testing is unable to increase branch coverage with random mutations, we gather path constraints for uncovered branch conditions, build additional constraints to represent the context of GPU execution such as number of threads and work-group size, and invoke the Z3 constraint solver to generate tests for them.

Finally, to help uncover inter work-group data races and replay these bugs with fixed work-group schedules, we present a schedule amplifier, CLSchedule, that simulates multiple work-group schedules, with which to execute each of the generated tests. By reimplementing the OpenCL API, CLSchedule executes the kernel with a fixed work-group schedule rather than the default arbitrary schedule. It also executes the kernel directly, without requiring the developer to manually provide boilerplate host code.

The outcome of our research can be summarised as follows:

1. CLTestCheck is applied to 82 publicly available GPU kernels from industry-standard benchmark suites along with their test suites. The experiment reveals that CLTestCheck is capable of automatically measuring the effectiveness of test suites, in terms of code coverage, faulting finding capability and revealing data races in real OpenCL kernels.

2. CLFuzz can automatically generate tests and achieve close to 100% coverage and mutation score for the majority of the data set of 217 GPU kernels collected from open-source projects and industry-standard benchmarks.

3. CLSchedule is capable of exploring the effect of work-group schedules on the 217 GPU kernels and uncovers data races in 21 of them.

The techniques developed in this thesis demonstrate that we can measure the effectiveness of tests developed for GPU kernels with our coverage criteria and fault seeding methods. The result is useful in highlighting code portions that may need developers' further attention. Our automated test generation and work-group scheduling approaches are also fast, effective and scalable, with small overhead incurred (average of 0.8 seconds) and scalability to large kernels with complex data structures.

Lay Summary

Graphics processing units (GPUs) are equipped with thousands of computing units working in parallel and originally designed to process images and videos for computers. In recent years, due to their power efficiency, massive parallelism and simplified programming model, GPUs are used to accelerate various general-purpose computing domains such as artificial intelligence, scientific simulations and critical systems, including autonomous driving. This has made checking the correctness of GPU kernels critical.

Testing is a solution to address this issue. Software testing is the process of ensuring that a given computer kernel is correct and has the expected behaviour. It involves a repeated execution of the kernel with a set of test inputs, observes that the software behaviour conforms to expectations. Test inputs can be manually created by developers, and automated test input generators also exist for programs written in some particular programming languages. However, general testing techniques are not suitable for GPU kernels as GPU architectures are distinct from CPUs and other devices in the 3-level memory layout, the execution model with thousands of threads clustered in hundreds of work-groups and the programming framework, resulting in the gap in techniques for test quality measurement and test input generation.

In this thesis, we propose the first criteria to judge the effectiveness of tests developed for GPU kernels by taking into account their characteristics of the hardware and programming model. This is useful to highlight portions of the kernel that have not been adequately tested by existing test inputs. An automatic test generation framework is then presented to generate high-quality test inputs with high code coverage and fault finding capabilities. This framework is also equipped with our schedule amplifier - the first simulator that is able to execute GPU kernels with customised work-group schedules. A poorly-developed GPU kernel can be fragile when the schedule changes when it is running on different platforms. In summary, our testing techniques help provide confidence in the correctness of GPU kernels.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor Dr. Ajitha Rajan for her invaluable guidance and inspiration. She supported me not only in my research but in every way possible. When I was depressed or frustrated, she was always available to give me encouragement and advice, especially during the epidemic, to help me get out of the low mood. Without her, it is impossible for me to finish my PhD.

I would also like to thank my second supervisor Dr. Björn Franke and my annual report reviewer Prof. Don Sannella for their insightful comments and advice to my yearly plans and achievements.

I would also like to thank Huawei's 2012 Laboratories and ByteDance's Quality Lab for accepting me to do internships with them. These were valuable experiences for me to develop teamwork skills and gave me opportunities to revise my research ideas in the context of the industrial environment.

Thank you to my fellow colleagues: Panagiotis Stratis, Vanya Yaneva, Sefa Akca and Nick Louloudakis. We were lucky to be supervised by Ajitha and had great fun working together. Thank you to my friends Xiaotong Shen, Yu Gu, Yang Cui and Yangqiuye Gao for our great times in Edinburgh. Thank you to my friends Ziheng Meng, Nan Yue, Jia Ye and Mengjing Yang back in China for your valuable emotional support.

Last but not least, I would like to thank my dad and mom, who have done more than so much for me and our family. I am deeply grateful for having you being in my life.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Chao Peng*)

Table of Contents

List of Publications

Publications related to this thesis:

1. C. Peng, "On the correctness of gpu programs," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 443–447, 2019
2. C. Peng and A. Rajan, "Cltestcheck: Measuring test effectiveness for gpu kernels," in *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pp. 315–331, Springer, 2019
3. C. Peng and A. Rajan, "Automated test generation for opencl kernels using fuzzing and constraint solving," in *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU)*, pp. 61–70, 2020

Other publications during the PhD:

4. C. Peng, S. Akca, and A. Rajan, "Sif: A framework for solidity contract instrumentation and analysis," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 466–473, IEEE, 2019
5. S. Akca, A. Rajan, and C. Peng, "Solanalyser: A framework for analysing and testing smart contracts," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 482–489, IEEE, 2019
6. C. Peng, A. Rajan, and T. Cai, "Cat: Change-focused android gui testing," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021
7. S. Akca, C. Peng, and A. Rajan, "Testing smart contracts: Which technique performs best?," in *2021 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ACM, 2021

List of Figures

List of Tables

Chapter 1

Introduction

Software is playing an integral role in modern life. Nowadays, the growing number of users, the expanded application domains and fast-evolving technologies have raised increasing demand of computational workload [8]. Taking natural language processing as an example, training a widely-used model, BERT, takes around 10^{20} floating-point operations on millions of parameters [9].

Thanks to the improvement of availability and programmability of computing accelerators such as Graphics Processing Units (GPUs), processing data using thousands of threads in parallel has been made easy and convenient [10, 11, 12].

Compared with traditional multi-core CPUs, GPUs have a higher order of magnitudes of threads (hundreds vs. tens) and wider peak memory bandwidth (hundreds of gigabytes vs. tens) [13], which makes them attractive to the growing need of high performance computing. Consequently, general-purpose graphics processing units (GPGPU) are now present everywhere [14]. Scientists can analyse complicated models such as fluid [15], the atmosphere [16] and the universe [17] using GPU clusters. We can intuitively experience this from more accurate weather predictions and smart voice assistants powered by natural language processing and artificial intelligence.

In addition to daily applications and scientific simulations, GPUs are also used in safety-critical systems, including autonomous driving, for accelerating perception tasks, such as object detection and tracking [18]. In such systems, it is extremely important to ensure the correctness and safety of the GPU implementation.

A survey on deep learning developers [19] shows that 52% participants have encountered faults related to the usage of GPU devices, and 27% of these faults are rated as critical severity. Example faults include failed parallelism, incorrect state sharing between sub-processes and faulty transfer of data to a GPU device.

The research community have devoted efforts to GPU programs verification [20, 21, 22, 23]. However, these techniques are challenged by the combinatorial explosion of thread interleavings and space of possible data inputs. Given these difficulties, it becomes crucial to understand the extent to which a GPU program has been analysed and tested, and the code portions that may need further attention.

1.1 Testing GPU Programs is Different from CPUs

An important practice to ensure the correctness of GPU programs is *software testing* - the process of executing the software under test (SUT) to check if the behaviour is the same as expected [24]. This often involves running the SUT with test input, comparing the actual output with the expected output and catching other exceptional behaviour such as program crashes and infinite loops.

However, the execution model, memory hierarchy and programming framework of GPU programs are different from CPU programs. Therefore, these challenges need to be overcome to test GPU programs effectively.

1.1.1 SIMT Execution Model

Firstly, GPUs have a different parallel model. Modern CPUs commonly have less than 16 cores, and each core can launch a process independently. However, GPUs can spawn hundreds of work-groups, each usually with 32 threads. Within the same work-group, threads follow the SIMT (single instruction, multiple threads) execution model in which all threads execute the same instruction but on different data.

If there exist control flows such as `if statements` and loops, threads within the same work-group may diverge when the condition of the control flow is dependent on the data each thread processes. 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 the execution of the kernel.

Some test effectiveness measurement techniques such as code coverage requires code instrumentation, which insert extra statements to the original program to record runtime information. To minimise the overhead introduced by coverage measurement, the inserted code should take the SIMT execution model into consideration and avoid using control flows and other expensive statements as much as possible. In addition, different threads within the same work-group may enter different control flows, resulting in different per thread code coverage. Therefore, the design of the data structure to record code coverage should be able to gather the results into one place while reducing the overhead introduced by inter-thread communication.

1.1.2 GPU Programming and Memory Management Restrictions

Even if both popular GPU programming models, OpenCL [25] and CUDA [26], are based on C and C++ programming languages, there are still essential but unsupported standard C/C++ features that hinder the test effectiveness measurement for GPU programs. Such features include dynamic memory allocation (OpenCL only), global scope variables and library functions, including standard and file I/O[1]. When testing CPU programs, these features make runtime information recording convenient: 1) as time goes, the size of the data structure for recording execution logs can be extended by dynamic memory allocation. 2) when the program is composed of multiple subroutines, this data structure can be shared by them via a global pointer. 3) recorded runtime information can be stored in files or printed via the standard output.

However, due to restrictions of the GPU architecture, dynamic memory allocation is not allowed. Therefore, the size of the data structure should be fixed before the compilation of the program. In addition, due to the lack of global scope variables, the data structure should only reside in the main function and be shared with others by passing references via function parameters. As the GPU cannot store this data structure to disks, it should transfer it back to the CPU for further analysis, and this procedure introduces extra memory bandwidth consumption and transfer overhead in addition to these resources used by the program data.

1.1.3 Lack of Customised Thread Scheduling and Global Synchronisation

Existing GPU architecture, simulators and programming models do not provide a means to control work-group schedules [27]. In addition, synchronisation is also not possible for threads from different work-groups.

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.

[1]I/O: Input and Output

In addition, this non-determinism of thread scheduling also introduces the randomness to program execution and makes the replay of tests harder.

1.2 Goal

Comprehensive testing for GPU programs is a crucial part of the development process that ensures correct behaviour, reliability and quality of the developed GPU applications. This involves test effectiveness measurement, automated test generation and execution with customisable work-group scheduling.

1.2.1 Test Effectiveness

Test effectiveness measurement indicates how effective a testing method is based on a given criterion [28]. These criteria can be code coverage that reports how many lines, code blocks or other quantities of code are exercised and mutation score that reports how many of the inserted faults are revealed by test cases. Test effectiveness measurement is important for testers and developers to understand the extent to which the subject program has been analysed and tested. It can also highlight code portions that may need further attention.

Test effectiveness measurement is a well-studied field for CPU programs and many tools have been developed for programs written in different programming languages [29]. GPU programs, however, do not have metrics for test effectiveness measurement and a comprehensive framework to perform this task.

Code coverage and mutation analysis have been explored to measure test effectiveness for CPU programs. For code coverage measurement, there exists coverage metrics including control flow and data flow coverage to report the extent to which the test suite explores the structure of the program under test [30]. Mutation analysis is also used to measure test effectiveness [31]. Mutants of the program are generated by inserting bugs into the program and executed with the test suite. A mutation score which is the ratio of detected mutants to the total number of mutants is calculated and used as an indicator of test effectiveness. These methods could be revoluted to measure the quality of test suites developed for GPU programs. In the general case, GPU programs are written in C-like programming models and share a similar code structure with traditional CPU programs. Code analysis and instrumentation framework also exist as a good ground to analyse and generate mutants for GPU programs [32]. With the right tooling and the proper consideration of characteristics of GPUs such as thread scheduling, they could allow us to measure test effectiveness for GPU programs based on code coverage and mutation score.

1.2.2 Test Case Generation

Test case generation for sequential and concurrent CPU programs has been well-studied over the past decades [33, 34, 35, 36]. Researchers have developed random, search-based and model-based test generation for CPU programs [37, 38, 39].

However, considering the challenges of testing GPU programs discussed in Section 1.1 that the programming framework, execution model and memory hierarchy of GPU programs are different from CPU programs, we need to come up with new test generation algorithms.

Due to the lack of global synchronisation of GPU threads, different thread schedules may lead to different program output when synchronisation bugs exist in the program under test. However, when executing the program with the test input and undetermined thread schedule, such bug is not guaranteed to be uncovered. Although existing model checking techniques introduce a 2-threaded model to reduce the complexity of thread interleaving explosion, false positives still present in their verification results [40].

Therefore, thread schedule should also be included as part of testing and a scheduler needs to be implemented for this end.

The goal of this thesis is to achieve these goals by defining coverage criteria suitable for GPU programs and a faulting seeding system to measure the test effectiveness, generating test inputs automatically aiming at high test quality and executing GPU programs with different thread schedules.

1.3 Contributions

This thesis makes three main contributions. Figure 1.1 presents these contributions in a unified workflow to test GPU programs.

1.3.1 Test Effectiveness Measurement

A GPU test effectiveness measurement framework, CLTestCheck, is designed and implemented with the aim to help developers assess how well the GPU program has

Figure 1.1: Thesis contributions

been tested and code regions that require further testing. CLTestCheck defines code coverage metrics considering synchronisation statements, branch conditions and loop boundaries. It is able to automatically instrument the GPU program to record code coverage and generate artificial faults of this program. After code instrumentation, mutant generation, test suites along with the program are executed to report code coverage and mutation score achieved.

1.3.2 Schedule Amplification

CLSchedule, a schedule amplifier for GPU programs is developed to force work-groups to execute in a particular order in a GPU program execution. 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.

1.3.3 Test Case Generation

The final contribution is a tool named CLFuzz, which combines fuzz testing with SMT solving, to generate test inputs for GPU programs. Fuzzing is first performed to generate a number of tests quickly and code coverage achieved by these tests are measured by CLTestCheck. When the fuzzer is unable to reach full (or high) coverage of the GPU program, SMT solving is used to generate inputs for uncovered branches. The test inputs generated by the SMT solver combined with test inputs from the fuzzer form the complete test suite achieving high kernel code coverage.

1.4 Publications and Research Impact

The ideas, tools and results presented in this thesis are based on three previous peer-reviewed publications. Tools developed for our project have also attracted interest from technological companies from different countries.

1.4.1 Publications

The overall design and research proposal of this thesis was published as a doctoral symposium paper in:

1. C. Peng, "On the correctness of gpu programs," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 443–447, 2019 [1].

Contribution 1.3.1, the CLTestCheck framework for test effectiveness measurement was published as a research paper in:

2. C. Peng and A. Rajan, "Cltestcheck: Measuring test effectiveness for gpu kernels," in *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pp. 315–331, Springer, 2019 [2].

Contribution 1.3.3, automated test case generation was published as a research paper in:

3. C. Peng and A. Rajan, "Automated test generation for opencl kernels using fuzzing and constraint solving," in *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU)*, pp. 61–70, 2020 [3].

In addition, Publications 2 and 3 present the work-group scheduler as well as schedule amplification (Contribution 1.3.2) and discuss how this work is integrated with test effectiveness measurement and test generation, respectively.

1.4.2 Research Impact

CodePlay is a company based in Edinburg that works on open programming standards, especially for complex processors and heterogeneous systems. They develop self-driven car systems running on GPUs and expressed interest in using our tools to test their critical systems. The tool integration has not been commenced at the time of writing.

Huawei Munich Research Centre works on the implementation of OpenCL on their mobile platforms. They have drafted a proposal in applying CLFuzz to test OpenCL programs on Android.

1.5 Organisation

The rest of the thesis is organised as follows:

Chapter 2 discusses background information on the GPU architecture, programming model and potential bugs. Concepts and techniques of software testing that are relevant to this thesis are also introduced in this chapter.

In Chapter 3 we provide a literature review on existing research that is related to all the contributions of this thesis: static and dynamic analysis of CPU and GPU program verification, test amplification for GPU programs, global work-group scheduling and GPU test case generation.

The definition of the coverage metrics for GPU programs is presented in Chapter 4. The automated test effectiveness measurement framework, CLTestCheck, with code coverage, fault seeding and mutation score capabilities is then discussed in this chapter.

Chapter 5 addresses the needs of test case generation aiming at high test quality. Random fuzzing and SMT solving are adapted to testing GPU programs and evaluated using 217 industry-standard GPU programs. The work-group scheduler, CLSchedule, is also presented in this chapter. The scheduler allows the GPU program to be executed with the generated tests in different work-group schedules.

Finally, this thesis concludes in Chapter 6 with a summary of contributions and main findings of this thesis. This chapter also provides a critical review and discusses potential future work.

Chapter 2

Background

In this chapter, we provide necessary background information on various aspects and concepts of software testing and general-purpose GPU programming that are relevant to the scope of this thesis.

Section 2.1 illustrates the foundations of general-purpose computing on GPUs (GPGPUs) including hardware architecture (Section 2.1), execution model (Section 2.1.2) and memory hierarchy (Section 2.1.3). Section 2.2 describes common GPGPU programming models. A simple toy program is built in this section and used as a motivating example to better describe potential bugs in GPGPU programming. In Section 2.3, we describe the software testing workflow and concepts involved in the testing process.

2.1 General-purpose Computing on GPUs

GPUs were originally designed to rapidly process images for output to a display device. Nowadays, they are widely used in general-purpose computing due to their highly parallel structure [41]. GPUs do not work alone but is controlled by CPUs and they together form a heterogeneous system.

2.1.1 GPU Architecture

Figure 2.1 shows the a typical general-purpose-computing-capable GPU architecture. It is organised into threaded streaming multiprocessors (also known as compute units) which in turn contain one or more streaming processors (also known as processing elements). A threaded streaming multiprocessors serves as a core in the GPU, which reads and executes program instructions. A Streaming processor is an ALU (Arith-

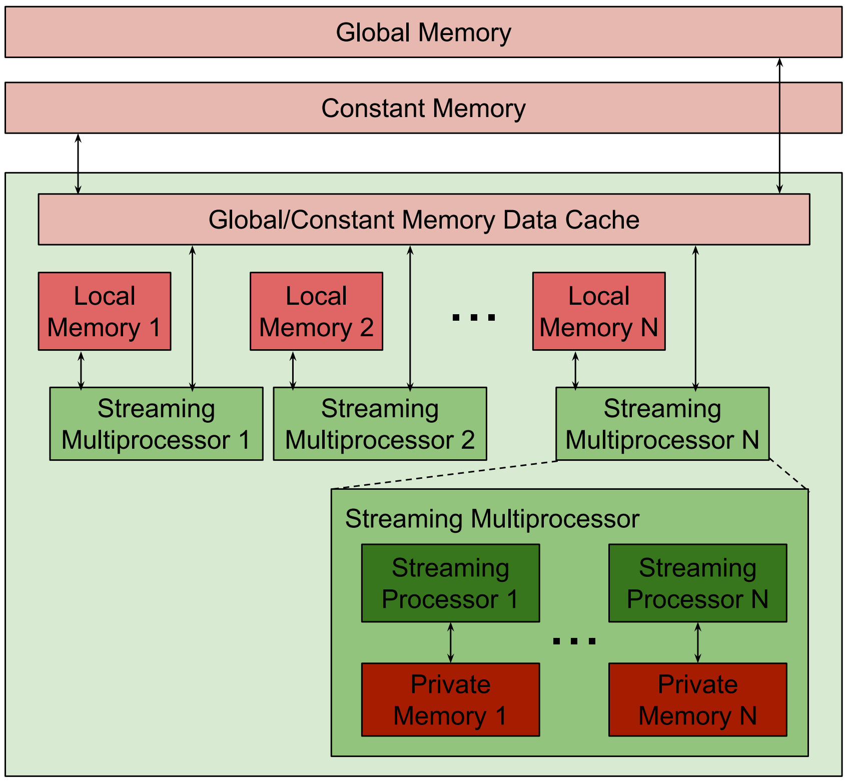


Figure 2.1: GPU Architecture

metic Logic Unit) which is the most basic processing unit in the GPU and executes individual GPU threads. A GPU thread is the sequence of kernel instructions that can be executed concurrently by these streaming processors.

2.1.2 SIMT Execution Model

GPUs follow the SIMT (single instruction, multiple threads) execution model. Streaming processors within the same unit share one instruction counter, and instructions are fetched and executed in lock-step, i.e. all streaming processors execute the same instruction but on different data.

SIMT is intended to save the overhead of fetching and decoding instructions from the memory and benefits high performance execution despite considerable latency in memory access operations [42]. However, if the control flow executed by the threads within the same work-group diverges, different execution paths are scheduled sequentially until the control flow reconverge and lock-step execution resumes. For instance, when the condition of an if-statement is to determine whether the thread ID is 0, Thread 0 and other threads will reconverge at the end of this if-statement and continue together when all these threads reach this point. Thus, sequential scheduling caused by divergence results in a performance penalty, slowing down the kernel execution.

2.1.3 GPU Memory Hierarchy

GPU has four types of memory regions: global and constant memory shared by all processors across compute units, local memory shared by processors within the same compute unit and private memory for each processor.

- *Global memory* is large but slow. Performance can be improved by coalesced accesses during execution, i.e. all processors access consecutive addresses in global memory.
- *Constant memory* is a read-only portion of global memory. Processors only have read access to this region. It has a special cache that enables faster read accesses.
- *Local memory* is shared within the compute unit it belongs to.
- *Private memory* is only accessible to individual processors.

GPU processors do not have direct access to the hard disk or CPU memory. Therefore, before the program is executed, the CPU is needed to read input data from the hard disk or other storage devices and move them to the GPU memory. For the same reason, output data is also moved from the GPU memory back to the CPU memory after kernel execution.

2.2 Programming GPUs with OpenCL

The success of GPUs in the past few years has also been due to the improved programmability brought by the CUDA [26] and OpenCL [25] parallel programming models, which abstract away architecture details. In these programming models, the developer uses a C-like programming language to implement algorithms. The parallelism in those algorithms has to be exposed explicitly: both OpenCL and CUDA provides standard interface functions for the developer to query device configurations such as number of available threads, handle resource and task allocation in the algorithm based on this information.

OpenCL is used by the thesis for three reasons.

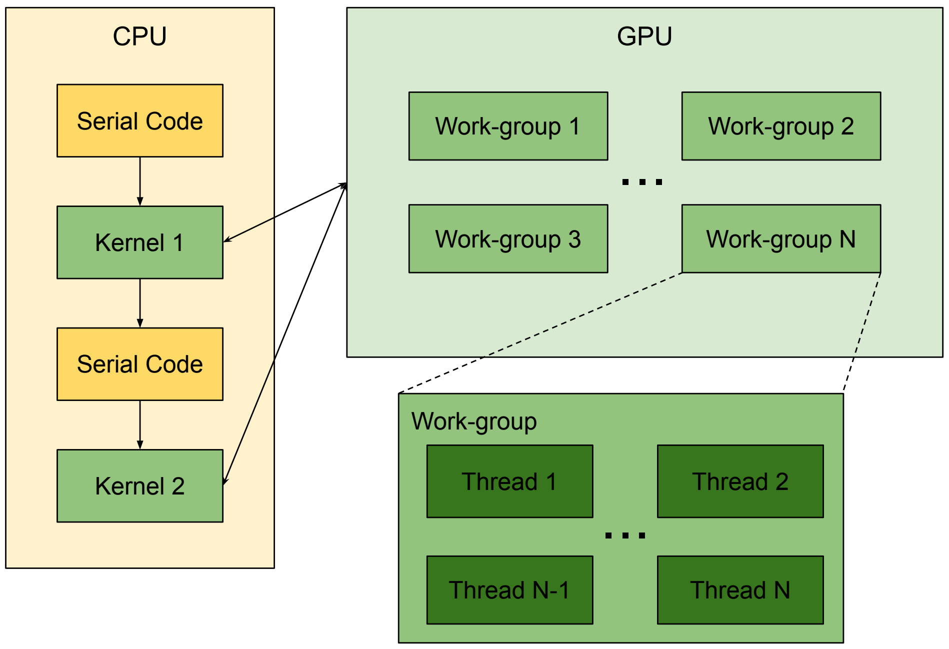


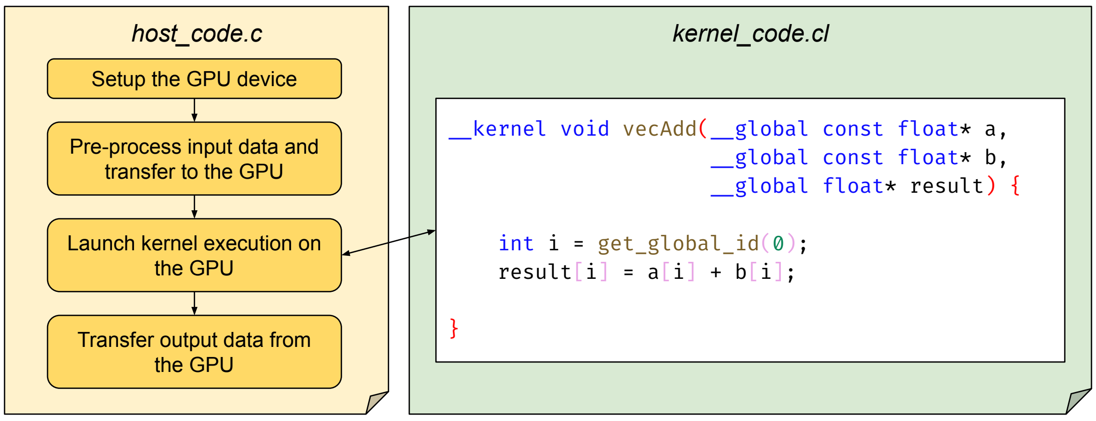
Figure 2.2: The OpenCL Abstraction of GPU Architecture

1. OpenCL is an open and non-profit standard maintained by Khronos Group [43] and is supported by most mainstream GPU vendors [44] including NVIDIA, AMD, Intel, Qualcomm, Apple, etc. By contrast, CUDA is developed by NVIDIA and is only applicable to NVIDIA GPU devices.
2. OpenCL uses a similar abstraction of GPUs with CUDA and mitigating CUDA programs to OpenCL can be automated [45]. This is the major reason why we select OpenCL to illustrate our testing techniques for GPU programs.
3. OpenCL is not only designed for GPUs but can also be used for multicore CPUs and other accelerators such as FPGAs [46], which potentially enlarges the applicable domains of our contributions.

2.2.1 OpenCL Overview

OpenCL is a programming framework and standard set from Khronos, for heterogeneous parallel computing on cross-vendor and cross-platform hardware.

The OpenCL abstraction of GPUs is illustrated in Figure 2.2. In the OpenCL architecture, the 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 work-groups. Each work-group contains one or more


host_code.c
Setup the GPU device
Pre-process input data and transfer to the GPU
Launch kernel execution on the GPU
Transfer output data from the GPU
kernel_code.cl

```
__kernel void vecAdd(__global const float* a,
                     __global const float* b,
                     __global float* result) {

    int i = get_global_id(0);
    result[i] = a[i] + b[i];

}
```

Figure 2.3: OpenCL Program Structure

threads.

The functions executed by the GPU threads are called ***kernels*** with the file extension *.cl*. Kernels are launched by the CPU and executed on the GPU.

The basic OpenCL program structure is illustrated in Figure 2.3 with an example OpenCL kernel for vector addition. The host and device code are executed by the CPU and GPU respectively. The host code written in C or C++ contains the OpenCL ”boilerplate” code necessary to setup the GPU device, pre-process the input data, transfer them to the GPU memory, compile and launch the OpenCL kernel and read the output produced by the kernel back. These operations are defined as standard functions in the OpenCL API and called by the host code. The kernel part is written in OpenCL. During execution, each thread executes an instance of the kernel using the data located in the global memory transferred from the host device.

Considering a function that takes two arrays as input, computes the summation of their elements at the same location and writes the result to another array, the C implementation shown in Listing 2.1 uses a loop to iterate through input arrays. The execution time is linear to the size of input arrays.

Figure 2.3 shows the corresponding implementation in OpenCL. The number of launched threads is the same as the number of array elements and is specified by the host code. At the kernel size, each thread retrieves its unique id and performs the addition operation on the array element it corresponds to.

Listing 2.1: vecAdd implemented in C

```
void vecAdd(int arraySize, const float* a,
            const float* b, float* result) {
    for (int i = 0; i < arraySize; ++i) {
        result[i] = a[i] + b[i];
    }
}
```

When the number of requested threads is greater than the number of physically available threads, OpenCL launches, in turn, available threads to perform part of the job until all requested threads are finished. In this vector addition example, if the number of available threads is 128 and this kernel is used for arrays of 1024 elements, these 128 threads may first calculate for elements 0 to 127, then 128 to 255, etc. However, the order is not guaranteed to be from lower ids to higher ids. The OpenCL specification and GPU vendors do not have any explicit rule on this [27]. Therefore, it is possible for OpenCL to calculate elements, for example, 876 to 1023 in the beginning and 0 to 127 in the very end. This may introduce synchronisation bugs when the computation in one work-group relies on elements assigned to different work-groups. We will look into this further in the following sections.

2.2.2 Data races, Barriers and Barrier Divergence

2.2.2.1 Data Races

A *data race* occurs when two or more threads access the same memory location at the same time and at least one of these threads performs a write operation [23].

Data races can be classified into inter and intra work-group data races according to the memory region in which the data race takes place.

1. *Inter work-group data race* is referred to as 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.

2. *Intra work-group data race* is referred to as a global or local memory location is written by one thread and accessed by another from the same work-group.

Figure 2.4 shows an example parallel algorithm to compute partial summations for an input array. In this example, the input array is divided into small parts and each part is handled by work-groups of threads. Data elements are first copied to local memory regions of the work-groups and the reduction summation is calculated by threads within each work-group.

Listing 2.2 shows an OpenCL kernel implementation of this algorithm. However, this example exposes both intra and inter work-group data races.

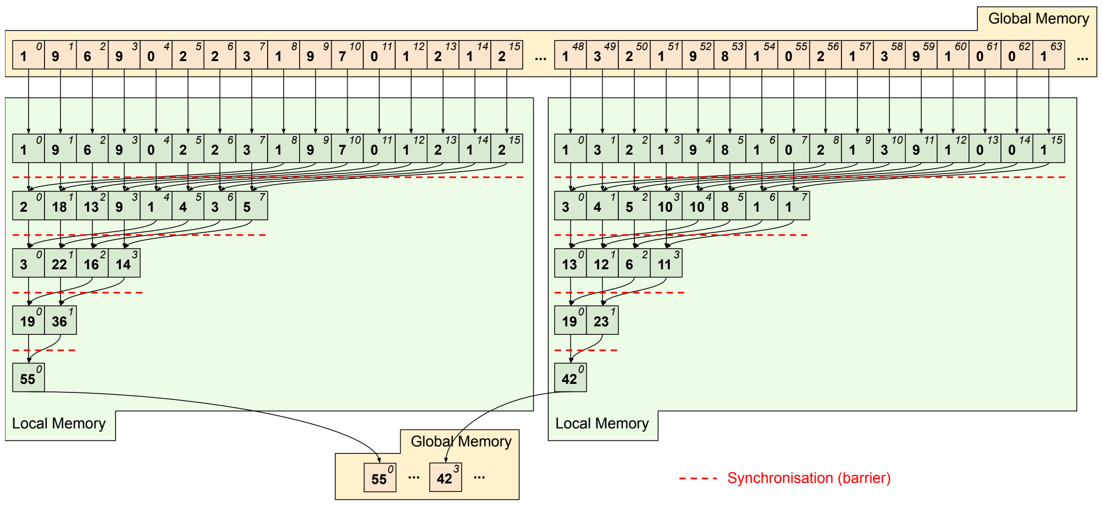


Figure 2.4: Parallel Partial Sum Algorithm

Listing 2.2: partialSum implemented in OpenCL with data races

```
 1 __kernel void sum(__global const int *array,
 2                   __local int *localSums) {
 3     int globalID = get_global_id(0);
 4     int localID = get_local_id(0);
 5     int groupSize = get_local_size(0);
 6     localSums[localID] = array[globalID];
 7 
 8     for (int stride = groupSize / 2; stride > 0; stride /= 2) {
 9         if (localID < stride) {
10             localSums[localID] += localSums[localID + stride];
11         }
12     }
13 
14     if (localID == 0) {
15         array[get_group_id(0)] = localSums[0];
16     }
17 }
```

At Line 6, threads of the work-group copy elements from the input array to their shared local memory and at Line 10, these threads add two elements in the shared memory using proper stride based on the input size. An intra work-group data race is possible to happen at Line 10, if the element one thread refers to is not yet copied from the global memory by another thread. On the other hand, at Lines 14 - 16, the thread with local ID 0 is used to copy the result from its work-group to global memory reusing the global array to save space. An inter work-group data race is possible to happen, if work-group 1 finishes execution before threads from work-group 0 starts copying their elements (at Line 6). In this case, thread 1 from work-group 0 tries to read global data element 1 and the value of this element is the result stored by work-group 1 instead of the initial input value.

2.2.2.2 Barriers

To prevent intra work-group data races, the *barrier* function can be used as a synchronisation mechanism for threads within a work-group. All threads from the same work-group must execute the barrier before any are allowed to continue execution beyond the barrier [25].

With barriers, the example OpenCL kernel is improved as shown in Listing 2.3. The barrier at Line 9 ensures threads do not proceed until all values are updated and the parameter of this barrier function is used to queue a memory fence to ensure correct ordering of memory operations in local memory.

The barrier at Line 15, however, cannot guarantee that the memory operation is free of data races. Firstly, the last iteration of the previous loop is always done by thread 0 and the following write access to the global memory region is also performed by this thread. Therefore, the local memory fence is not needed. Secondly, barriers can only be used to synchronise threads within the same work-group. The OpenCL specification and GPU vendors do not provide a means to synchronise threads from different work-groups [27]. To solve this problem, we need to introduce a separate global array to store results.

Listing 2.3: partialSum implemented in OpenCL with barriers

```
1  __kernel void sum(__global const int *array,
2                    __local int *localSums) {
3      int globalID = get_global_id(0);
4      int localID = get_local_id(0);
5      int groupSize = get_local_size(0);
6      localSums[localID] = array[globalID];
7
8      for (int stride = groupSize / 2; stride > 0; stride /= 2) ↩
          {
9          barrier(CLK_LOCAL_MEM_FENCE);
10         if (localID < stride) {
11             localSums[localID] += localSums[localID + stride];
```

```
            }
        }

        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
        if (localID == 0) {
            array[get_group_id(0)] = localSums[0];
        }
    }
```

2.2.2.3 Barrier Divergence

When the barrier is placed inside a control flow of a conditional statement of a loop, *barrier divergence* could occur if threads in the same group reach different barriers, in which case kernel behaviour is undefined [23] and may lead to intra work-group data race or indefinite loops.

In the previous example, if the barrier is placed in the if statement, a barrier divergence is likely to happen at runtime as in each work-group, the number of threads enabled to enter the branch is halved after each iteration, resulting in some of them executing the barrier while the rest not.

2.3 Software Testing

Software testing is a process, or a series of processes, designed to make sure computer code does what it is designed to do and that it does not do anything unintended [47]. The goal of testing is to demonstrate that the behaviour of the software is as intended. This process usually involves executing the software with a set of test inputs and checking whether the actual outputs are the same as the expected output.

2.3.1 Terminology and Software Testing Workflow

Identical ideas may have different names or be described using informal words in the literature. To avoid confusion and be consistent, we provide in this section a set of definitions of terms used in this thesis.

- **software/system under test (SUT)**: software or system that is tested.
- **test oracle**: a mechanism for determining whether a test has passed or failed.

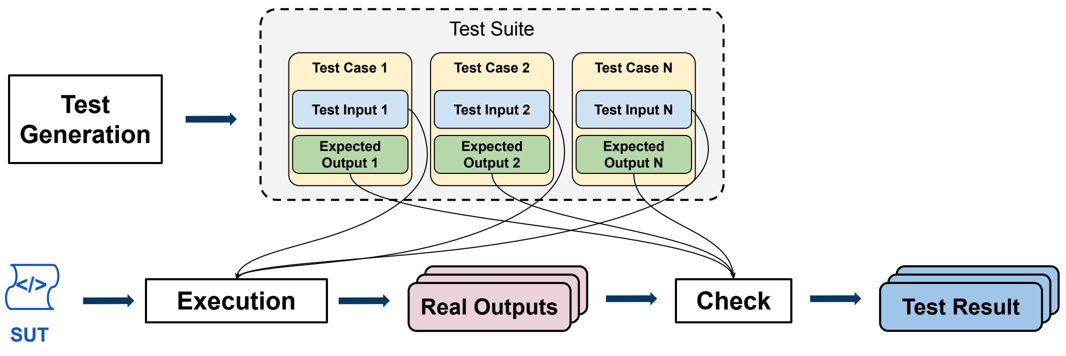


Figure 2.5: Software Testing Workflow

- **software specification**: a description of the intended behaviour of the SUT and the foundation of the test oracle.
- **test input**: the input data used to execute the SUT.
- **test output**: the real output data associated with a specific test input generated by the SUT after execution.
- **expected output**: the expected output data of a specific test input.
- **test case**: the test input and its expected output.
- **test suite**: a set of test cases.
- **test failure**: the real output is not always the same as the expected output. Test failure can also be used to describe when the SUT encounters indefinite loops or other exceptions.
- **test result**: indication of whether a test is passed or failed.

Figure 2.5 illustrates the software testing workflow. Firstly, test generation is performed to produce the test suite. The SUT is then executed repeatedly, once with each test input from the test suite and produces a set of real outputs. After the execution, the real outputs are compared with the expected output and test results are reported for each test case.

For example, when we test the partial sum program presented in Listing 2.3, this program is referred to as the SUT. The array of numbers shown in Figure 2.4 is one possible test input and the resulting array is the expected output of this test input. When barriers are used properly, the real output might be different from the expected output due to data races. In this case, the test result is said to be a failure.

2.3.2 Example: Partial Sum

To better illustrate these concepts involved in software testing, we take the partial sum program presented in Listing 2.3 as the software under test and the software specification is the following:

1. Should accept an array of integer values, the size of partitions and the size of the array as input.

2. The number of working items should be the size of the array and the size of work-groups should be the same as the size of partitions.

3. Partitions are consecutive sub arrays of the input array and a partial sum is calculated by adding all the elements within this sub array.

4. Should return calculated partial sums as first elements in the input array.

Test ID	**Test Input**			**Expected Output**
	array	array size	partition size	
1	{1, 2, 3, 4, 5, 6}	6	3	{6, 15, 3, 4, 5, 6}
2	{1, 2, 3, 4, 5, 6}	6	2	{3, 7, 11, 4, 5, 6}

Table 2.1: Example tests for the partial sum OpenCL program.

Table 2.1 gives an example test suite with two test cases for the partial sum OpenCL program. The software specification requires an array of integers, the size of the array and the size of partitions as input. Therefore, the test input of each test case contains one integer array and two integer scalar values. The expected output for this program is based on the specification and manually calculated by the developer to check the program correctness.

2.3.3 Test Effectiveness

To understand to what extent the SUT is tested, we can measure the the test suite effectiveness in aspects of code coverage achieved and fault finding capabilities.

2.3.3.1 Code Coverage

Code coverage measures the degree to which the source code of the SUT is exercised by the test input. It can be measured using different coverage criteria with different interests [48].

- **Statement coverage** is the most basic coverage criterion that reports how many of the statements are executed by the test suite.
- **Branch coverage** measures the ratio of covered branches to all the branches. An if statement has two branches - true and false while a switch statement has multiple branches, each introduced by one case keyword.
- **Loop coverage** is an indication of how loops are exercised. Loops can be completely skipped by unsatisfying the loop condition, executed exactly once, executed multiple times or jumped out if there is a break statement.

2.3.3.1.1 Mutation Testing Mutation testing is a type of testing method that injects errors by modifying the source code of the SUT, executes the SUT with the test suite and checks if the test cases are able to reveal these injected errors [49].

Each modified copy of the SUT is called a mutant. Possible operations to generate a mutant include:

- Statement deletion - delete one statement.
- Statement duplication - duplicate a statement and add its copy after it.
- Constant replacement - replace a constant value with another value, such as navigating the number, replacing 0 by 1 or true by false.
- Variable replacement - replace a variable with another variable from the same scope.
- Operator replacement - replace an operator with another one that is applicable to this place.

The following listing gives an example of a statement with two mutable operators $<$ and $++$:

Listing 2.4: Original Statement

```
    if (id < size) i++;
```

By applying operator replacement, four mutants can be generated as shown in Listing 2.5 - 2.12. The first five mutants are generated by changing the $<$ to $<=$, $>$, $>=$, $!=$ and $==$.

Listing 2.5: Generated Mutant 1

```
  if (id <= size) i++;  /* Mutant 1 changes < to <=  */
```

Listing 2.6: Generated Mutant 2

```
  if (id > size) i++;   /* Mutant 2 changes < to >   */
```

Listing 2.7: Generated Mutant 3

```
  if (id >= size) i++;  /* Mutant 3 changes < to >=  */
```

Listing 2.8: Generated Mutant 4

```
  if (id != size) i++;  /* Mutant 4 changes < to !=  */
```

Listing 2.9: Generated Mutant 5

```
  if (id == size) i++;  /* Mutant 5 changes < to ==  */
```

The remaining changes the operator $++$ to $--$ and the place of these operators for variable i.

Listing 2.10: Generated Mutant 6

```
  if (id < size) i--;   /* Mutant 6 changes i++ to i-- */
```

Listing 2.11: Generated Mutant 7

```
  if (id < size) --i   /* Mutant 7 changes i++ to --i */
```

Listing 2.12: Generated Mutant 8

```
  if (id < size) ++i;   /* Mutant 8 changes i++ to ++i */
```

It is worth noticing that in this scenario, the original program and Mutant 8 are semantically the same. These types of mutants are called equivalent mutants.

When a number of mutants are generated, the test suite is executed with each of them. If one mutant is failed with the test suite, this mutant is said to be killed. The percentage of the killed mutants is the mutation score of the test suite, which has the positive correlation with the quality of the test input set [49].

2.4 Test Generation

Test generation is the process of creating a set of test cases for a software system. As creating test cases are very labour intensive, if this process could be automated, we can significantly reduce the cost of the process of software development [50]. Various algorithms exist to guide the creation of test cases and arch across from one end of black-box random testing to the other end of white-box symbolic execution [33].

2.4.1 Random and Fuzz Testing

The simplest method of test case generation should be selecting test inputs fully at random [51]. The selection can be optimised by introducing some probability distributions or guided by some bias chosen by the developer that lead to program-specific corner cases. **Fuzz testing** or simply **Fuzzing** is an evolved technique that aims at generating random data as test inputs to uncover exceptions such as crashes and memory leaks [52].

Depending on how the initial inputs are generated, fuzz testing can be classified to **generation-based fuzzing** and **mutation-based fuzzing** [53]. Generation-based fuzzing generates inputs from scratch based on the input model provided by the user. For instance, a compiler fuzzer takes the syntax of the target programming language as the input model and generate random syntactically correct programs to test the compiler. A mutation-based fuzzer, on the other hand, takes existing test input as seed and generate new inputs by mutating them. For example, an image viewer fuzzer takes a valid image file as a seed and modifies metadata or pixels of the original image and tests the image viewer.

Fuzzers are now widely used in various application domains and have been shown to be fast and surprisingly effective [54, 55, 56, 57]. For example, libFuzzer is provided along with the Clang C/C++ compiler to test programs written in these languages. For Java and .Net programs, Randoop can be used to generate random test inputs and automatically store them as unit tests.

However, random tests are not guaranteed to explore all branches of the program, especially when the program under test has some deeper branches or some branches with strict conditions, such as requiring a specific value. One complementing strategy is symbolic execution.

2.4.2 Symbolic Execution

Symbolic execution [58] analyses the program by representing computations as functions of symbolic values and conditional expressions as constraints of these symbolic values. It then builds a tree with nodes of these constraints. Possible inputs to enter a certain branch can be then generated by solving all the constraints leading to that branch.

Consider the following C program that determines if the given year is a prime year.

Listing 2.13: A C program to check if the given year is a leap year

```
bool isLeapYear(int year) {
    if (year % 400 == 0) {
        return true;
    } else if (year % 100 == 0) {
        return false;
    } else if (year % 4 == 0) {
        return true;
    } else {
        return false;
    }
}
```

During symbolic execution, the input value `year` is treated as a symbolic value. When the first condition is reached, the path constraint to reach the *then* branch is `year % 400 == 0` and `year % 400 != 0` for the *else* branch. When the second condition is reached, it will evaluate `year % 400 != 0 && year % 100 == 0` and `year % 400 != 0 && year % 100 != 0`. Similarly, for the last condition, the symbolic execution will evaluate `year % 400 != 0 && year % 100 != 0 && year % 4 == 0` and `year % 400 != 0 && year % 100 != 0 && year % 4 != 0`.

A satisfiability modulo theory solver (SMT Solver) is then used to solve these constraints and generate proper values for each condition. For this example, the Z3 SMT solver gives 0, -300, 104 and 1 to enter these four branches.

Limitations of symbolic executions include path explosion, memory aliasing, arrays, etc.

1. **Path explosion.** The number of paths in a program grows exponentially with the programs introduces new embedded control flows. The overhead of solving constraints of large programs can be significant.

2. **Memory aliasing.** Symbolic execution is not feasible when some memory locations are pointed using different names, especially when dynamic pointers are used. For instance, if the collected path constraints contain values with aliasing relations (one value is the alias of another) and are sent to the solver, the solver does not have the information of aliasing relations, leading to inaccurate results. Therefore, extra pointer analysis is needed in this case.

3. **Arrays** are collections of a number of values. The symbolic analyser must choose to either treat each element as a separate value, or the entire array as one value. For the first choice, when the array index in the program is dynamically assigned, it is hard to determine which value it refers to. The second case, however, requires a symbolic representation for arrays, which is not easy to do so as arrays accesses are represented using symbolic indices which can generate more paths to be explored [59].

2.4.3 White-box Fuzzing with SMT Solving

A white-box fuzzer leverages program analysis to systematically increase code coverage or to reach specific critical program locations. In the above example, we ran an experiment to generate 100 random integer numbers as test inputs and got Branches 1, 3 and 4 covered. We can then apply the SMT solver to solve the second constraint only, which solves the overall overhead.

2.5 Program Instrumentation and Clang LibTooling

Program instrumentation is a technique to measure performance, diagnose errors and collect trace information by inserting extra code to the program under test [60]. The inserted code is harmless to the original functionality of the program.

Some representative examples of program instrumentation include:

1. **Profiling** [61] measures dynamic program behaviours by running the program. This can help the developer analyse some information that cannot be measured by static analysis and improve the program before launching them.

2. **Timing and performance estimation** [62] inserts timers to code fragments that are computation-extensive or performs major tasks of the program. This infor-

mation can be used to report overhead and reveal the time-consuming part of the program.

3. **Logging execution information** [63] includes recording code coverage, crashes and other major events. This type of instrumentation can help report test effectiveness achieved by executing the program with a given test suite.

In this thesis, program instrumentation is needed to add code blocks to record coverage information at runtime and inject faults to original GPU kernels to check the fault finding capabilities of test suites. We discuss details of our approach in the next chapters.

There exist a vast variety of tools designed to instrument programs written in different languages [64, 65, 66]. The LibTooling framework [1] is an outstanding tool to instrument C, C++, OpenCL and CUDA programs. The most significant advantage of this tool is that it supports all syntactic structures and provides semantic analysis as it is built on top of the Clang compiler. LibTooling is equipped with a rich set of functions to query information from the compiler instance before instrumenting the program. After the compiler instance is setup, LibTooling traverses the abstract syntax tree (AST) of the program. For each AST node, there is a corresponding data structure containing all the information for that node. For instance, a node of a binary expression contains the operator type and two pointers to other nodes that acts as its operands. This information is used to determine if the currently visited node is of interest and for all the interested nodes, the developer can write code by calling LibTooling APIs to insert, modify or delete AST nodes.

2.6 Summary

In this chapter, we introduce background information related to the problem and our approach presented in this thesis. The introduction to the GPU architecture, SIMT execution model, memory hierarchy and the OpenCL framework is essential for understanding the challenge and our solutions in this thesis. Information on the software testing process and test generation demonstrates the importance of testing GPU programs and gives an overview of how programs are tested. Finally, the sections on program instrumentation discusses basic techniques used by our approach.

[1] https://clang.llvm.org/docs/LibTooling.html

Chapter 3

Related Work

In this chapter, we present a literature review of existing work related to each contribution of this thesis. We begin in Section 3.1 with the state-of-the-art research on concurrency bug detection for multithreaded CPU programs. GPU kernel verification and analysis is then presented in Section 3.2. In Section 3.3 , we discuss existing work on testing GPU programs in the context of test effectiveness measurement and test case generation for GPU kernels in Sections 3.3.1 and 3.3.2 respectively. Finally, in Section 3.4 , we present studies on inter work-group synchronisation techniques.

3.1 Why CPU Concurrency Bug Detection Is Inadequate for GPU Kernels?

Detecting concurrency bugs is a long-standing problem since the appearance of concurrent programs and many techniques have been proposed by the research and industry community [67, 68].

3.1.1 Lockset Based Approaches

Static lockset based concurrency bug detection technique employs race-violation rules and checks whether locks are held correctly for all shared variable accesses. This technique has been applied to multithreaded Java [69, 70, 71, 72, 73, 74] and C programs [75]. To locate the faults in concurrent programs, Savage et al. [76] present Eraser based on a dynamic lockset algorithm that reports the file and line number when a data race is detected. Park et al. [77] propose a lockset based dynamic analysis approach to detect data races for distributed memory parallel CPU programs. This approach first executes the program and finds sets of statement pairs that could potentially lead to data races and confirms the data race by fixing the thread schedule and execute the marked statements.

However, GPU kernels use barriers for synchronisation instead of locks. In addition, threads from the same work-group follow the SIMT execute model and cannot be rescheduled, making this lockset based approach inappropriate for testing GPU kernels.

3.1.2 Happens-before Based Approaches

Lamport [78] proposes a happens-before based approach that records read and write accesses to shared variables by tracking synchronisation events. If to the same shared variable, there exist two accesses in an undetermined order, a data race error is reported. Smaragdakis et al. [79] introduce a new generalisation of Lamport's happens-before algorithm, named causally precedes, which is able to observe more races without losing a lot of accuracy and completeness. To improve accuracy. Hybrid lockset and happens-before based approaches are also explored by existing literature. Choi et al. [80] use a hybrid lockset and happens-before based approach to detect data races and propose a weaker-than relation to minimise the number of accesses that need to be considered, which reduces the overhead. RaceTrack [81] is another low overhead hybrid tool to detect data races using an adaptive approach that directs the analysis to more suspicious areas.

Unlike concurrent CPU programs, GPU kernels usually spawn thousands of threads for execution, making recording memory accesses to this amount of threads infeasible.

3.1.3 Model Checking and Symbolic Execution

To detect concurrent bugs for CPU programs, mathematics based techniques, model checking and symbolic execution, are also applied to verify software and hardware systems.

Model checking. Qadeer et al. [82] propose a context-bounded model for concurrent bug detection. This technique can find bugs within a bounded number of context switches where a context means an uninterrupted sequence of actions of a single thread. The context-bound model has been improved since then in supporting different program abstractions and eliminating model complexity [83, 84, 85, 86, 87, 88, 89]. In addition to the context-bounded model, partial order techniques are also applied to reduce equivalent thread interleavings and reduce the search space [90, 91, 92, 93, 94, 95, 96]. However, computing the precise dependence relation may be as hard as verification itself [93].

Symbolic execution. Symbolic execution is also useful in verify concurrent CPU programs. Some predictive bug detection methods based on satisfiability modulo theory (SMT) solvers have been presented to build a symbolic predictive model by tracing the execution and use SMT solvers to check the potential interleavings for concurrent bugs [97, 98]. However, symbolic execution has the limitation of state explosion problem when the number of threads grows, while having thousands of threads is common in GPU kernels.

In summary, these concurrency bug detection techniques that are designed for CPU programs are not suitable for GPU kernels for the following reasons:

1. **Number of threads.** GPUs usually spawn thousands of threads while concurrent CPU programs only use tens of threads in most cases. For this reason, it is not practical to record all memory accesses and check data races using recorded conflicts.

2. **Memory hierarchy.** There are two types of shared memory regions in GPUs, global memory and local memory, shared by threads from different work-groups and threads within the same work-group respectively. Barriers can only fence memory accesses to the local memory while global memory accesses cannot be synchronised. However, multithreaded CPUs does not have this issue.

3. **SIMT execution model.** GPUs follow the SIMT execution model in which threads within the same work-group follow the lock-step execution i.e. executing the same instruction at any given time. This makes lock-based analysis not suitable to check data races for GPU kernels.

3.2 GPU Kernel Verification and Analysis

Verifying and analysing GPU kernels is a critical problem, which has received a lot of attention from academia and industry, having both static, dynamic and hybrid techniques explored [40].

Static analysis is the analysis of a program on the source code or object code level without executing it. Dynamic analysis, by contrast, is performed by tracing the execution of the program and analysing information gathered at runtime. Both techniques

Tool	Bug Types Supported	False Positives	False Negatives	User Participation Required	Other Limitations
PUG [20]	Data race Barrier divergence	Y		Y	Scalability issue.
GPUVerify [23]	Data race Barrier divergence	Y			No full programming feature support. No inter work-group data race detection.
GRace [21] GMRace [22]	Data race		Y		
LDetector [99]	Data race		Y		No intra work-group data race detection.
Test Amplification [100]	Data race		Y		
KLEE-based [101, 102, 103]	Data race Barrier divergence				No full programming feature support. No inter work-group data race detection. Scalability issue.
ESBMC-GPU [87, 86]	Data race	Y	Y	Y	
Hardware-accelerated [104]	Only a prototype and not currently available. No barrie divergence detection				

Table 3.1: Summary of Existing GPU Kernel Verification and Analysis Techniques

have been explored in GPU kernel verification and analysis for data race and barrier divergence detection. In this section, we present existing work using static, dynamic and hybrid techniques for GPU kernel verification and analysis. The summary of these techniques is presented in Table 3.1.

PUG. Li et al. [20] introduce a scalable and comprehensive symbolic verifier for GPU kernels written in CUDA and implement an automated tool named Prover of User GPU (PUG). PUG employs a C frontend based on the LLNL Rose framework [105] with customised extensions to support CUDA features. To verify a CUDA kernel, the user needs to specify the number of threads and PUG captures all possible interleavings between CUDA threads as compact SMT formulae. During verification, PUG avoids modelling all captured interleavings to improve efficiency based on the fact that in many cases, a data race between one pair of variables can be predicated on the existence of data races between other variables. PUG also proposes a model of the semantics of barriers and generates SMT formulae to help verify that all barriers are well synchronised.

Limitations. The main drawback of PUG is scalability. With an increasing number of threads, the number of possible thread interleavings grows exponentially, making the analysis infeasible for a large number of threads. In addition, PUG may report false alarms when it fails to parse complicated loops and in this case, the user is required to provide more information as a config to the parser.

GPUVerify. To avoid reasoning about thread interleavings, GPUVerify [23] aims at data races and barrier divergence detection by 2-threaded symbolic static analysis. This approach is based on formal operational semantics for GPU programming, synchronous delayed visibility (SDV) semantics. According to the SDV semantics, group execution is synchronous, allowing precise intra work-group divergence detection. Each access to the shared memory is logged and the visibility of write accesses by one thread to the group is delayed until an intra work-group synchronisation is reached. This logging enables race detection when threads synchronise at the barrier and the delaying ensures that the threads do not see the memory synchronised before the barrier, considering the fact that the group execution is not always fully synchronous. GPUVerify is built on top of the Boogie verification system [106]. To verify a GPU kernel, GPUVerify first translates it into a sequential Boogie program that models the lock-step execution of two threads based on the SDV semantics and the correctness of this Boogie program implies the absence of barrier divergence and data races.
Limitations. The main drawback is that GPUVerify does not support full kernel programming features such as atomic operations. In addition, false positives are not avoidable for static analysis based approaches. More importantly, GPUVerify's 1-group 2-threaded model does not support the detection of inter work-group data races.

GRace and GMRace. Zheng et al. present GRace [21] and GMRace [22] to detect data races based on static and dynamic analysis. GRace first uses static analysis on statements with read and write accesses to the shared memory. When the accessed address can be proved to be identical using static approaches, they are reported as data races. For the remaining statements with shared memory accesses to addresses that cannot be determined by static analysis, GRace executes the kernel, monitors the exact memory index and checks for data races. With the runtime information, GRace eliminates the problem of false positives brought by static analysis. GMRace, on the other hand, reduces the overhead of GRace by refining its engineering details.
Limitations. GRace and GMRace are limited to detect only data races and neither of them supports the detection of barrier divergence. In the dynamic analysis stage, both tools can only detect data races in the covered branches and do not generate new inputs as an effort to exercise code blocks that are not exercised. As a result, these tools may miss some data races in GPU kernels.

LDetector. Instead of using static analysis to reduce the number of memory accesses to be recorded as introduced in GRace and GMRace, Li et al. [99] present LDetector which redirects global memory accesses to a work-group local copy which is not visible to other work-groups. With this copy, LDetector executes the kernel twice. In the first run, LDetector detects writes of a work-group by comparing stale values with the local copy and these writes of all work-groups are then compared to find write-write conflicts. In the second run, LDetector returns a work-group with new values from other work-groups and compares new results with the previous one to report read-write conflicts.
Limitations. The design of LDetector does not take intra work-group data races and barrier divergence into consideration, leaving these two types of bugs unsupported. In addition, data races are detected by value checking, which makes this method incomplete and may lead to false negatives.

Test amplification. Leung et al. [100] propose a flow-based test amplification based on hybrid static and dynamic analysis to verify CUDA kernels. This approach first logs memory accesses by executing the kernel with some fixed test inputs and a particular thread interleaving. If no data race is detected by dynamic race checking, static information flow is built based on pointer analysis and taint tracking. It can be determined that the control flow taken during execution is independent of the fixed input data, the kernel is considered as free of data races. Otherwise, this approach cannot approve the presence of data races.
Limitations. However, this technique focuses on data races only and have no support for barrier divergence, which is also a common bug for GPU kernels. In addition, this approach only amplifies the space of test inputs rather than thread schedules, which can also lead to data races. Finally, this approach only checks the absence of data races, not the presence.

Symbolic execution. Symbolic execution is also used to verify GPU kernels and various tools are built to verify CUDA and OpenCL kernels based on the KLEE [107] symbolic execution engine [102, 101, 108, 103]. We discuss these techniques as follows.

GKLEE. Li et al. [102] present GKLEE to check data race for CUDA kernels. Since KLEE works on LLVM bytecode, GKLEE first compiles the CUDA kernel into bytecode with an additional CUDA syntax handler and creates memory objects based on the state model. To generate schedules and check for data races, GKLEE pursues one canonical schedule where each thread is fully executed within a barrier interval and

then the next thread is executed. During the execution of all the threads in a certain barrier interval, all read and write accesses to the shared memory is recorded and for each access pair with conflicts, the SMT solver is employed to determine if it is a data race. After the execution, detected bugs are reported and tests generated during the execution are produced to a concrete test file to replay the bug.
Limitations. GKLEE is restricted by the need to specify a certain number of threads, and the lack of support for custom synchronisation constructs. Although working with a certain number of threads allows precise defect checking, scalability is an issue with GKLEE. Li et al. [103] then scale GKLEE based on parameterised flows [103] to restrict race checking to only consider certain pairs of threads. This is similar to the 2-threaded model employed by PUG and GPUVerify but results in the loss of precise modelling of kernels and miss some inter-thread communications.

KLEE-CL. KLEE-CL [101] is built on top of KLEE and KLEE-FP [108], an extension to KLEE offering support on reasoning on the equivalence between floating-point values, for crosschecking the original C or C++ program against its GPU accelerated version using OpenCL and detecting data races in the OpenCL kernel. For the first capability, equivalence checking, KLEE-CL first builds symbolic expressions from the output of both the original program and the OpenCL version by applying a set of canonicalisation rules to make two expressions in a canonical form and compares the expressions syntactically. To detect data races, for each byte in the shared memory, KLEE-CL keeps a memory access record (MAR) consisting of the identifier of the thread that most recently makes the access to this byte, the identifier of the work-group that most recently access to this byte and four flags indicating whether this byte is written by one or more threads, read by one or more threads, read by many threads and read by many work-groups. When a new memory access is made, the MAR is queried and a data race is reported when write-read and write-write conflicts are identified according to the record.
Limitations. Similar to GKLEE, KLEE-CL also relies on the number of threads executing the kernel, which limits scalability.

ESBMC-GPU. Monteiro et al. present ESBMC-GPU [87] using Bounded Model Checking and Satisfiability Module Theories with explicit state-space exploration to check data races and function correctness. In addition, the novel combination of techniques such as 2-thread model and state hashing allows ESBMC-GPU to reduce the complexity of state-space exploration and eliminate redundant interleavings without ignoring program behaviours.

Limitations. The main limitation of ESBMC-GPU is false positives and false negatives [109]. In addition, although ESBMC-GPU can automatically check CUDA kernels, it still requires the user to place assertions for function correctness checks.

Hardware-accelerated. Detecting data races with the help of the hardware is also explored in the literature. Unlike techniques recording memory accesses per thread, Holey et al. [104] propose a hardware-accelerated mechanism, HAccRG, to track per-memory accesses on the hardware. For this purpose, they design the Race Detection Unit (RDU) which is able to record shadow entries of memory accesses. These entries are used to report data races when read-write and write-write conflicts exist within a given synchronisation point.
Limitations As RDU are not currently available on GPUs, HAccRG is now only implemented as an extension to a GPU simulator. In addition, the memory-recording mechanism is not applicable to detect barrier divergence.

3.3 Testing GPU Programs

Testing has some benefits than verification in that it can minimise false positives by executing the program, supports all program features by using real inputs and reduces overhead brought by the computation-intensive static analysis. Some existing work has explored test effectiveness measurement and test input generation.

3.3.1 Test Effectiveness Measurement

Measuring test quality in terms of code coverage and fault finding is common for CPU programs [110, 111]. GKLEE [102] is able to measure code coverage achieved by tests it generated by translating the GPU code to its sequential version using Perl scripts and applying the Gcov utility which is the code coverage measurement tool provided by the GNU Compiler Collection. However, this method disregards the GPU programming model. GKLEE can also report coverage achieved at 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.

Zhu et al. [112] apply mutation testing to evaluate the quality of tests for GPU kernels written in CUDA. Similar to our approach discussed in Section 4.2.2, they generate mutants from the original program and measure the quality of tests by checking if these tests are able to kill the mutants. Their work was published more than one years later than our CLTestCheck [2] and focuses more on CUDA-specific mutation operators on the CUDA host code (written on the CPU side) while our approach targets the kernel code. In addition, we port more traditional mutation operators to OpenCL and investigate the effect of code coverage on measuring mutation score.

3.3.2 Test Input Generation

Symbolic execution has been applied to generate test inputs for both CUDA and OpenCL kernels [102, 101] while other test input generation techniques remain unexplored. In this section, we discuss existing work on test input generation for GPU kernels.

GKLEE [102] is the only technique in the literature that provides test generation capability for CUDA kernels. As discussed in Section 3.2, GKLEE selects tests generated during symbolic execution to help reproduce the detected data race. However, test inputs generated by GKLEE are in the form of hexadecimal values that are meant to run on the KLEE virtual machine. They cannot be used directly to execute the original kernels and are not human readable. In addition, GKLEE has the disadvantage that it does not support floating-point data types which are widely used in scientific computation GPU kernels.

Finally, GKLEE suffers from high overhead in test generation as they rely on symbolic execution and SMT solvers. Scalability to large kernels is also an issue because of the high overhead and the path explosion problem associated with symbolic execution.

Fuzz testing. To mitigate the overhead and scalability problems associated with symbolic execution, we use fuzz testing in our test generation approach. Fuzz testing, or simply fuzzing, is based on randomly generating or mutating test inputs and has been shown to be fast and surprisingly effective [54, 55]. However, fuzz testing for GPU kernels has not been explored previously.

One noticeable limitation of fuzzing is that fuzzing based on random mutations, typically finds it hard to reach program parts protected by complex checks. As a result, other techniques including SMT 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 SMT solving and fuzzing has been effective in detecting security bugs in CPU, mobile and web applications [113, 114]. For instance, Sapienz [115] 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 [116, 53].

In Chapter 5, we discuss how we combine the fast and scalable nature of fuzz testing with the rigour of SMT solving to produce an effective test generator for GPU kernels.

3.4 Inter Work-group Synchronisation

Barrier functions in the OpenCL specification [25] help synchronise threads within the same work-group. There is no mechanism, however, to synchronise threads belonging to different work-groups, which highlights the importance of inter work-group data race detection.

One solution for this problem is to split a program into multiple kernels with the CPU executing the kernels in sequence providing implicit synchronisation. However, as discussed in Section 2.2.1, when running multiple kernels, the CPU needs to setup the device and transfer input data to the GPU before running the GPU kernel and read output data after the kernel finishes execution. Even some kernels are executed consecutively, they cannot pass data directly to each other. In summary, the drawback of this method is the considerable overhead incurred in launching multiple kernels.

Lock-baserd global synchronisation. Xiao et al. [117] propose two approaches for inter work-group synchronisation using barriers. The first approach is called lock-based synchronisation, uses a global mutex variable to count the number of work-groups that reach the synchronisation point and let work-groups to proceed when the variable is equivalent to the number of work-groups. In each work-group, the thread with ID 0 first adds 1 to the global variable and this operation is conducted by an atomic addition operation to avoid write-and-write conflicts. This thread then enters a loop which only exits when the value of the global mutex variable is equivalent to the number of work-groups. All other threads in the same work-group are blocked by an intra work-group barrier and can only proceed execution when the thread with ID 0 exits the loop.

Lock-free global synchronisation. The second approach, lock-free synchronisation, replaces the global mutex variables with two global arrays, *Arrayin* and *Arrayout* of sizes equivalent to the number of work-groups *N*. At the global synchronisation point, threads with ID 0 from each work-group set element *i* to a special value where *i* is the ID of the work-group that this thread belongs to. Threads with ID less than *N* in

Work-group 1 are used to check if all elements in *Arrayin* are equivalent to the special value, with thread *i* checking element *i* in *Arrayin*. After all are set to the special value, each checking thread then sets the corresponding element in *Arrayout* to the special value. Each block only continues execution once its leading thread sees that the corresponding element in *Arrayout* has the special value.
Limitations. These two approaches, however, rely on the information on the number of work-groups and are not portable as the number of launched work-groups depends on the device when the number of required work-groups is greater than the number of available compute units.

Portable global synchronisation. Sorensen et al. [118] extended [117] to be portable by developing an occupancy discovery protocol which dynamically computes a safe estimate of the work-group occupancy for a give GPU and kernel. To achieve this, all work-groups are required to execute the discovery protocol at the beginning and each work-group executes a routine that returns a value indicating whether this work-group is participating. A work-group is said to be participating if and only if it starts execution before any work-group finishes execution. The estimated number of work-groups is calculated after this protocol exits.

Their implementation of inter work-group barrier synchronisation is useful when the developer knows there are interactions between work-groups that needs to be synchronised. However, the detection of global data races remains a challenge for programs that do not apply these global synchronisation methods.

Chapter 4

Test Effectiveness Measurement for GPU Programs

4.1 Introduction

The research contributions presented in this chapter are the design and implementation of the test effectiveness measurement framework, with novel coverage metrics and fault seeding strategies designed specifically for OpenCL kernels. This chapter describes the underlying approach, the implementation and evaluation of the framework using 82 OpenCL kernels from industry-standard benchmarking suites.

We present a framework, CLTestCheck, that measures test effectiveness over GPU kernels written using the OpenCL programming model [25]. The workflow of the framework is illustrated in Figure 4.1. Taking an OpenCL kernels as input, CLCov instruments the kernel to add coverage measurement statements, CLMT generated mutations of the input kernel with seeded faults and CLSchedule generated different work-group schedules. The instrumented kernel with coverage measurement is then executed with test inputs to report code coverage achieved by these test inputs. Mutations of the kernel are also executed to check if the test inputs can reveal seeded faults which is the implication of their fault-finding capability. In addition, the kernel with its test inputs are executed using different work-group schedules to check if outputs are consistent. The inconsistency means inter work-group data races happens, resulting in data read and write conflicts.

CLTestCheck has three main capabilities.

The first capability is measuring code coverage for OpenCL kernels. The structures we chose to cover were motivated by OpenCL bugs found in public repositories such

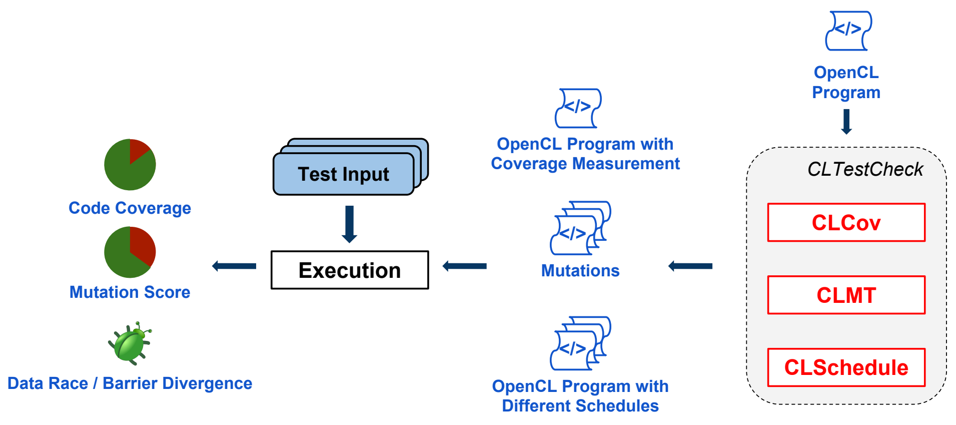

OpenCL Program
CLTestCheck
CLCov
CLMT
CLSchedule
OpenCL Program with Coverage Measurement
Mutations
OpenCL Program with Different Schedules
Test Input
Execution
Code Coverage
Mutation Score
Data Race / Barrier Divergence

Figure 4.1: CLTestCheck Workflow

as Github and research papers on GPU testing [23]. We define and measure coverage over synchronisation statements, loop boundaries and branches in OpenCL kernels.

The second capability of the framework is creating mutations by seeding different classes of faults relevant to GPU kernels. We assess the effectiveness of test suites in uncovering the seeded faults. Knowing the fault finding capability, we measure the correlation between coverage achieved and fault finding. As with empirical studies over sequential and concurrent CPU programs [119, 120, 121], we take a strong correlation between them to mean that coverage achieved is a good proxy for fault detection capability of that test suite.

The final capability of CLTestCheck is a technique called *schedule amplification* to check the execution of test inputs over several work-group schedules. Existing GPU architecture and simulators do not provide a means to control work-group schedules. In addition, the OpenCL specification provides no execution model for inter work-group interactions [27]. 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 by building 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.

We empirically evaluated CLTestCheck using 82 kernels with their associated test input workloads from industry-standard benchmarks. The schedule amplifier in CLTestCheck could detect deadlocks and inter work-group data races in benchmarks. We could detect barrier divergence and kernel code that requires further tests using the coverage measurement capabilities of CLTestCheck. Finally, the fault seeding capability could expose unnecessary barriers and unsafe accesses in loops.

The CLTestCheck framework aims to help developers assess how well the OpenCL kernels have been tested, kernel regions that require further testing, uncover bugs sensitive to work-group schedules. In summary, the main contributions in this chapter are:

1. Schedule amplification to evaluate test executions using different work-group schedules.
2. Definition and measurement of kernel code coverage considering synchronisation statements, loop boundaries and branch conditions.
3. Fault seeder for OpenCL kernels that seeds faults from different classes. The seeded faults are used to assess the effectiveness of test suites with respect to fault finding.
4. Empirical evaluation on a collection of 82 publicly available GPU kernels, examining coverage, fault finding and inter work-group interactions.

The rest of this chapter is organised as follows. CLTestCheck capabilities are presented in Section 4.2. The experiment setup of our empirical evaluation is then discussed in Sections 4.3. In Section 4.4, we discuss the coverage and fault finding results, along with their correlation for the subject OpenCL kernels in our experiment. Finally, Section 4.5 concludes this chapter.

4.2 CLTestCheck: Measuring Test Effectiveness for GPU Kernels

In this section, we present the CLTestCheck framework that provides capabilities for kernel code coverage measurement, mutant generation and schedule amplification.

To understand the kinds of programming bugs[1] encountered by OpenCL developers, we surveyed several publicly available OpenCL kernels and associated bug fix commits. A summary of our findings is shown in Table 4.1. We found bugs most commonly occur in the following OpenCL code constructs: barriers, loops, branches, global memory accesses and arithmetic computations. Atomic operators are not included in our approach as it does not have the implication to cause synchronisation bugs such as barrier divergence and data races based on the OpenCL specification

[1]These are kernel bugs that violate the specification of the program or are associated with executions that lead to undefined behaviour.

while barriers have undefined behaviours. In addition, atomic operators are well-studied in the literature focusing on performance issues [122, 123]. We seek to aid the developer in assessing the quality of test suites in revealing these bug types using CLTestCheck. A detailed discussion of CLTestCheck capabilities is presented in the following sections.

#	Code Structure	Bug Type	Repository
1	Barrier	Missing barriers	Winograd-OpenCL[124], histogram [100], reduction [100],OP2 [101]
2		Removing unnecessary barriers	Winograd-OpenCL[124]
3	Loop	Incorrect condition	mcxcl[125], particles[126]
4		Incorrect boundary value	clSPARSE[127]
5		Missing loop boundary	Pannotia [27]
6	Branch	Missing else branch	liboi[128]
7		Incorrect condition	mcxcl[125], ClGaussianPyramid[129]
8	Global memory access	Inter work-group data race	Parboil-spmv [130], lonestar-bfs [27], lonestar-sssp [27]
9	Arithmetic Computations	Incorrect arithmetic operators	mcxcl[125], ClGaussianPyramid[129]

Table 4.1: Summary of bug fixing commits we collected

4.2.1 Kernel Code Coverage

We define coverage over barriers, loops and branches in OpenCL code to check the rigour of test suites in exercising these code structures.

4.2.1.1 Branch Coverage

GPU programs are highly parallelised, executed by numerous processing elements, each of them executing groups of threads in lock-step, which is very different from the parallelism in CPU programs, where each thread executes different instructions with no implicit synchronisation, as seen in lock-step execution. Every GPU thread has its own thread ID, and typically processes a portion of the input data (that is assigned according to the scale of data and the number of threads). Kernel code for all the threads is the same. However, the threads may diverge, following different branches based on the input data they process. As seen in Table 4.1, uncovered branches and branch conditions are critical classes of OpenCL bugs. Lidbury et al. [130] report in their work that branch coverage measurement is crucial for GPU programs but is currently lacking. To address this need, we define branch coverage for GPU programs as follows,

$$branch\ coverage = \frac{\#covered\ branches}{total\ \#branches} \times\ 100\% \tag{4.1}$$

Branch coverage measures the adequacy of a test suite by checking if each branch of each control structure in GPU code has been executed by at least one thread.

4.2.1.2 Loop Boundary Coverage

In our survey of kernel bugs shown in Table 4.1, we found bugs related to loop boundary values and loop conditions were fairly common. For instance, bug #3 found in the `mcxcl` program allowed the loop index to access memory locations beyond the end of the array due to an erroneous loop condition. The bug was revealed on the NVIDIA GPU architecture.

We assess the adequacy of test executions with respect to loops by considering the following cases,

1. Loop body is not executed,
2. Loop body is executed exactly once,
3. Loop body is executed more than once
4. Loop boundary value is reached

$$Loop\ boundary\ coverage_{case\_i} = \frac{\#loops\ satisfying\ case\_i}{total\ \#loops} \times\ 100\% \tag{4.2}$$

where $case_i$ refers to one of the four loop execution cases listed above. Each of these cases is viewed as covered when any of the threads can execute the loop accordingly.
Loop boundary coverage measures the adequacy of a test suite by checking if each loop body in GPU code has not been executed, has been executed once, more than once, and if the boundary of the loop condition has been reached.

4.2.1.3 Barrier Coverage

Barrier divergence occurs when the number of threads within a work-group executing a barrier is not the same as the total number of threads in that work-group. Kernel behaviour with barrier divergence is undefined. Barrier related bugs, missing barriers and unnecessary barriers, are common classes of GPU bugs according to our survey. We define barrier coverage as follows.

$$barrier\ coverage = \frac{\#covered\ barriers}{total\ \#barriers} \times 100\% \tag{4.3}$$

Barrier coverage measures adequacy of a test suite by checking if each barrier in GPU code is executed correctly. Correct execution of a barrier without barrier divergence, *covered barrier*, is when it is executed by *all* threads in any given work-group.

4.2.2 Fault Seeding

Mutation testing is known to be an effective means of estimating the fault finding effectiveness of test suites for CPU programs [131]. We generate mutations using traditional mutant operators, namely, arithmetic, relational, bitwise, logical and assignment operator types. In Table 4.1, bug fixes #3, #7 and #8 show that traditional arithmetic and relational operator mutations remain applicable to GPU programs. In addition, we define three mutations specifically for OpenCL kernels: barrier mutation, image access mutation and loop boundary mutation inspired by bug fixes #1 to #5.

The barrier mutation operator we define is the deletion of an existing barrier function call, to reproduce bugs similar to #1 and #2 in Table 4.1.

OpenCL provides 2D and 3D image data structures to facilitate access to images. Multi-dimensional arrays are not supported in OpenCL. Image structures are accessed using read and write functions that take the pixel coordinates in the image as parameters. We perform image access mutations for 2D or 3D coordinates by increasing or decreasing one of the coordinates or exchanging coordinates.

Types of loop boundary mutations include:

1. Skipping the loop by injecting a false statement to the loop condition.

2. Allowing n-1 iterations of the loop by put a minus 1 to the boundary value, when applicable.

3. Allowing n+1 iteration of the loop by put an add 1 to the boundary value, when applicable.

The mutant operators we use in this paper are summarised in Table 4.2

Type of Operator		Mutants
Arithmetic	Binary	+, -, *, /, %
	Unary	-(negation), ++, --
Relational		<, >, ==, <=, >=, !=
Logical		&&, ||, !
Bitwise		&, |, ^, ~, <<, >>
Assignment		=, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
Barrier		Delete barrier function call
Image coordinates		Change coordinates when accessing images
Loop boundary		Change the boundary value in loop condition check

Table 4.2: Summary of mutation operators

4.2.3 Schedule Amplification

When a kernel execution is launched, the GPU schedules work-groups on compute units in a certain order. Presently, there is no provision for determining this schedule or setting it in advance. The scheduler makes the decision on the fly subject to the availability of compute units and readiness of work-groups for execution. The order in which work-groups are executed with the same test input can differ every time the kernel is executed. OpenCL specification has no execution model for inter work-group interactions and provides no guarantees on how work-groups are mapped to compute units. In our approach, we execute each test input over a set of schedules. In each schedule, we fix the work-group that should execute first. All other work-groups wait till it has finished execution. The work-group going first is picked so that we achieve a uniform distribution over the entire range of work-groups in the set of schedules. The order of execution for the remaining work-groups is left to the scheduler. For a test case, T over a kernel with G work-groups, we will generate N schedules, with $N < G$, such that a different work-group is executed first in each of the N schedules. The number of schedules, N, we generate is much lesser than the total number of schedules which is typically infeasible to check. The reason we only fix the first work-group in the schedule is that most data races or deadlocks involve interactions between two work-groups [23]. Fixing one of them and picking a different work-group each time, significantly reduces the search space of possible schedules. We cannot provide guarantees with this approach. However, with the little extra cost we are able to check significantly more number of schedules than is currently possible. This approach is shown to be effective in our experiments as discussed in Section 5.2.3.

To illustrate this, we consider the following kernel example co-running on four work-groups each with a single thread. This example is inspired by Betts et al. [23].

Listing 4.1: Inter work-group data race example with 4 threads

```
int tid = get_group_id(0);
int buf, x, y, i, j;
if (tid == 0) x = 4; else x = 1;
if (tid == 0) y = 1; else y = 4;
buf = i = 0;
while (i < x) {
    j = 0;
    while (j < y) {
        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
        A[(1 - buf) * 4 + tid] = A[buf * 4 + (tid + 1) % 4];
        buf = 1 - buf;
        j++;
    }
    i++;
}
```

The CLTestCheck schedule amplifier will insert code on the host and GPU side, shown in Listings 4.2 and 4.3, to generate different work-group schedules.

Listing 4.2: Schedule OpenCL kernel (CPU-side)

```
// Generate a value in the range of [0,4)
int target_group = randint(4);
// Pass the value as a macro to GPU code
sprintf(clOptions,"-DTARGET_GROUP=%d", target_group);
```

Listing 4.3: Schedule OpenCL kernel (GPU-side)

```
if (my_group_id == TARGET_GROUP){
    // Original code here executed by target group
    A[(1 - buf) * 4 + tid] = A[buf * 4 + (tid + 1) % 4];
    atom_increase(num_threads_finishes);
} else {
    while (num_threads_finishes != group_size) continue;
    // Original code executed by other groups
    A[(1 - buf) * 4 + tid] = A[buf * 4 + (tid + 1) % 4];
}
```

In this example, before the GPU kernel is launched, the host side generates a random value in the range of available work-group ids. This value is the id of the selected work-group to be executed first and is passed to the kernel code using a macro definition. On the kernel side, each thread determines if it belongs to the selected work-group. Threads in the selected work-group proceed with executing the kernel code while threads belonging to other work-groups wait. After the selected work-group completes execution, the remaining work-groups execute the original kernel in an order based on mapping to available compute units (occupancy bound execution model [118]). With different work-group schedules generated by the schedule amplifier, we could detect the presence of *inter* work-group data races using a *single* GPU platform. Betts et al. [23], on the other hand, focus on intra work-group data races on different GPU platforms.

4.2.4 Implementation

CLTestCheck is implemented using Clang LibTooling [32]. We instrument OpenCL kernel source code to measure coverage, generate mutations and multiple work-group schedules automatically. Our implementation is available at `https://github.com/chao-peng/CLTestCheck`.

Coverage Measurement. To record branches, loops and barriers executed within each kernel when running tests, we instrument the kernel code with data structures and statements recording the execution of these code structures. For each work-group, we introduce three local arrays, whose size is determined by the number of branches, loops and barriers accessible by threads in that work-group. The reason why we do not introduce arrays for every thread is that we do not want to occupy too much memory and according to the OpenCL specification, writing and reading data in the work-group memory using atomic operators is efficient enough.

To measure branch coverage, we add statements at the beginning of each then- and else-branch to record whether that branch is enabled. Similarly, statements to record the number of iterations of loops are added at the beginning of each loop body. An assignment expression with or operator in the format of $||(boundary\_not\_reached = false)$ is also added to every loop condition to record if the boundary of this loop is reached. When the boundary is reached, the loop condition turns to be false so that the added expression is enabled. On the other hand, if the loop condition is true, according to the rule of C99 standard which OpenCL complies to [25], the expression we added will not be executed because it does not affect the final value of the whole condition expression. At the end of the kernel, the information contained in the data structures is processed to compute coverage.

Fault Seeder and Mutant Execution. The CLTestCheck fault seeder generates mutants and executes them with each of the tests in the test suite to compute mutation score, as the fraction of mutants killed. In order to generate mutants effectively, our approach does not generate one mutant at a time as every time when running the tool, the overhead of calling the Clang library costs some time due to the expensive analysis of the abstract syntax tree. Instead, the CLTestCheck fault seeder translates the target kernel source code into an intermediate form where all the applicable operators are replaced by a template string containing the original operator, its ID and type. The tool then generates mutants from this intermediate form. An example of this process is illustrated as follows.

Listing 4.4: Sample original statement

```
if (id<size) i++;
```

Listing 4.5: Intermediate translation of sample statement

```
if (id${op1_<B}size) i${op2_++U};
```

Listing 4.6: Mutants generated from intermediate form

```
// 4 mutants generated
if (id<=size) i++;     /* Mutant 1 changes < to <=  */
if (id>size) i++;      /* Mutant 2 changes < to >   */
if (id>=size) i++;     /* Mutant 3 changes < to >=  */
if (id!=size) i++;     /* Mutant 4 changes < to !=  */
if (id<size) i--;      /* Mutant 4 changes ++ to -- */
```

Once mutants are generated, the tool executes each of the mutant files and checks if the test suite kills the mutant. We term the mutant as killed if one of the following occurs: program crashes, deadlocks or produces a result different from the original kernel code.

Schedule Amplification. As mentioned earlier, we generate several schedules for each test execution by requiring a target work-group to execute the kernel code first and then allowing other work-groups to proceed. The target work-group is selected uniformly across the input space of work-group ids. To achieve coverage of this input space, we partition work-group ids into sets of 10 work-groups. Thus if we have N work-groups, we partition them into $N/10$ sets. The first set has work-group ids 0 to 9, the second set has ids 10 to 19 and so on. We then randomly pick a target work-group, W_t, from each of these sets to go first and generate a corresponding schedule of work-groups, $\{W_t, S_{N-1}\}$, where S_{N-1} refers to the schedule of remaining $N-1$ work-groups generated by the GPU execution model which is non-deterministic. For $N/10$ sets of work-groups, we will have $N/10$ schedules of the form $\{W_t, S_{N-1}\}$ (a W_t first schedule). The test input is executed using each of these $N/10$ W_t first schedules. Due to the non-deterministic nature of S_{N-1}, we repeat the test execution with a chosen W_t first schedule 20 times. This will enable us to check if the execution model generates different S_{N-1} and evaluate executions with 20 such orderings.

4.3 Experiment

In our experiment, we evaluate the feasibility and effectiveness of the coverage metrics, fault seeder and work-group schedule amplifier proposed in Section 4.2 using OpenCL kernels from industry-standard benchmark families and their associated test suites.

4.3.1 Research Questions

We investigate the following questions:

Q1. Coverage Achieved: *What is the branch, barrier and loop coverage achieved by test suites over OpenCL kernels in our subject benchmarks?*

To answer this question, we use our implementation to instrument and analyse kernel source code to record visited branches, barrier functions, loop iterations along with information on executing work-group and threads.

Q2. Fault Finding: *What is the fault finding capability of test suites associated with the subject programs?*

For each benchmark, we generate all possible mutants by analysing the kernel source code and applying the mutation operators, discussed in Section 4.2, to eligible locations. We then assess the number of mutants killed by the tests associated with each benchmark. To check if a mutant is killed, we compared execution results between the original program and mutant. This comparison process is tailored for each benchmark based on the format of its results. Finally, we compare coverage achieved to the percentage of killed mutants for each of the benchmarks.

Q3. Deadlocks and Data Races: *Can the tests in the test suite give rise to unusual behaviour in the form of deadlocks or data races?* Deadlocks occur when two or more work-groups are waiting on each other for a resource. Inter work-group data races occur when test executions produce different outputs for different work-group schedules. For each test execution in each benchmark, we generate $20 * N/10$ different work-group schedules, where N is the total number of work-groups for the kernel, and check if the outputs from the execution change based on the work-group schedule.

4.3.2 Subject Programs.

We used the following benchmarks for our experiments.

1. *Parboil [132]* is an open-source benchmark suite with benchmarks collected from throughput computing application researchers in many different scientific and commercial fields including image processing, biomolecular simulation, fluid dynamics, and astronomy. Applications in the benchmarks perform a variety of computations including ray tracing, finite-difference time-domain simulation, magnetic resonance imaging, 3-D stencil operation. Each benchmark includes several implementations targeting specific CPU and GPU architectures. The benchmarks are also accompanied by test suites specific to the OpenCL programming model. We use 9 benchmarks with 23 OpenCL kernels from Parboil.

2. *Scan [133]* with 3 kernels, computes parallel prefix sum, a common task found in a significant portion of the kernels of real applications. Test inputs are large arrays of floating-point data designed to measure the performance of the parallel prefix sum algorithm.

3. *Rodinia [134]* is a benchmark suite for heterogeneous computing. Each of the applications in this benchmark suite has multiple implementations for different computing infrastructures including multi-core CPUs and GPUs. We use 5 applications containing 13 kernels in Rodinia benchmark suite. The applications we picked were based on the capability of verifying execution results, a feature needed for determining mutation scores.

4. *PolyBench [135]* is a collection of 30 benchmarks spanning linear algebra, data mining and stencil computations. The benchmarks are accompanied by test data sets that we evaluate coverage, mutation score and data races over. There are a total of 43 kernels in the 30 benchmarks.

4.3.3 Experiment Setup

The configuration of the machine we used for the experiments is presented in Table 4.3.

CPU	Intel Core i5-6500 CPU @ 3.20GHz
GPU	Intel HD Graphics 530 (Skylake GT2)
RAM	16 GB
Operating System	Ubuntu 16.04 LTS
Clang	6.0.0
OpenCL SDK	Intel OpenCL SDK 2.0

Table 4.3: Experiment platform configurations

4.3.3.1 Coverage Measurement

We first compile and run all the original benchmarking programs and record their execution time. For each program, we extract OpenCL kernel files from its source code directory and check the number of kernel functions in each kernel file. We then use our tool (discussed in Section 4.2) to instrument the kernel code to record coverage information. The tool also generates host code for allocating and initialising arrays, transferring these arrays from the host (CPU) to the kernel (GPU) and reading them back. In many cases, we modify the generated host code slightly because each program typically contains different variables to record OpenCL error code, different implementations to deal with OpenCL failures, and different wrappers to OpenCL SDK calls to set kernel function arguments.

We execute each of the benchmarks, after instrumentation, with associated test suites (or test data sets) to record coverage. Execution of the test suites produces data files that have information on triggered branch conditions, barrier functions, number of loop iterations executed by threads. We compute coverage by analysing the data files. Additionally, we record the execution time of the instrumented kernel, comparing it to the original, unmodified kernel.

4.3.3.2 Mutant Execution

The process of generating mutants for each benchmark is fully automated, as described in Section 4.2. The tool first reads the source code of kernel programs, replaces original operators with templates and computes all possible mutants. Each mutant is written to a single file.

To check if a mutant is killed, we compare execution results between the original program and mutant. This comparison process is tailored for each benchmark based on the format of its results. We do not use any extra tests in measuring the percentage of mutants killed than what was used for coverage measurement. We use Python scripts for each benchmark to automatically replace the original kernel file with one mutant at a time, execute the mutant 10 times with all tests and, finally, record the results in each of these 10 runs. The results in the 10 runs are not always consistent in all the benchmarks. This is further discussed in Section 4.4.

4.3.3.3 Schedule Amplification

Before scheduling the subject programs, we perform an initial run to see how many work-groups they launch. For programs with multiple kernels, we use the summation of the number of work-groups launched by each kernel as the total number for that program. The number of work-groups launched by different programs ranges from 4 to 277873 as each program has its own characteristics such as algorithm complexity and input data size. Catering to this big range of number of work-groups, we divide the work-group IDs into chunks with the size of 10. Then we randomly pick one work-group from each trunk, execute the scheduled version 20 times and compare the output against the original output.

To make kernels able to be scheduled, we manually insert pieces of code as described in Section 4.2 to original kernel programs. The host program is also modified to generate a random number within the range of work-group IDs of each chunk, and send the ID to the GPU side.

The process of execution and validation is automatically performed by a script which is similar to the one used mutation execution.

4.3.4 Experiment Summary

Tables 4.4, 4.5 and 4.6 summarise the different subject programs, their characteristics and number of generated mutants.

Benchmark	#Kernels	#Branches	#Barriers	#Loops	#Work-groups	#Mutants
bfs	1	12	3	2	4	40
cutcp	1	6	2	4	48410	322
histo	5	38	3	11	389	584
lbm	1	4	0	0	18000	666
mri-gridding	9	54	18	12	277873	1304
sad	3	6	0	5	144720	606
spmv	1	2	0	1	765	22
stencil	1	2	0	0	63240	57
tpacf	1	14	2	6	201	273

Table 4.4: Parboil experiment summary

Benchmark	#Kernels	#Branches	#Barriers	#Loops	#Work-groups	#Mutants
scan	3	12	8	4	192	163
lud	3	10	6	16	4033	565
nn	1	2	0	0	669	30
pathfinder	1	12	3	1	40000	91
srad	6	38	4	5	5334	522
gaussian	2	6	0	0	170	137

Table 4.5: Scan/Rodinia experiment summary

4.4 Results

For each of the subject programs presented in Section 4.3, we ran the associated test suites and report results in terms of coverage achieved, fault finding and overhead incurred with the CLTestCheck framework. We executed the test suites 20 times for each measurement. Our results in the context of the questions in Section 4.3 are presented below.

4.4.1 Coverage Achieved

Branch coverage pointed to code regions that were not adequately exercised by existing test suites. Barrier coverage identified a barrier divergence in the `scan` kernel.

Branch and Loop coverage (with 0, exactly 1 and > 1 iterations) for each of the

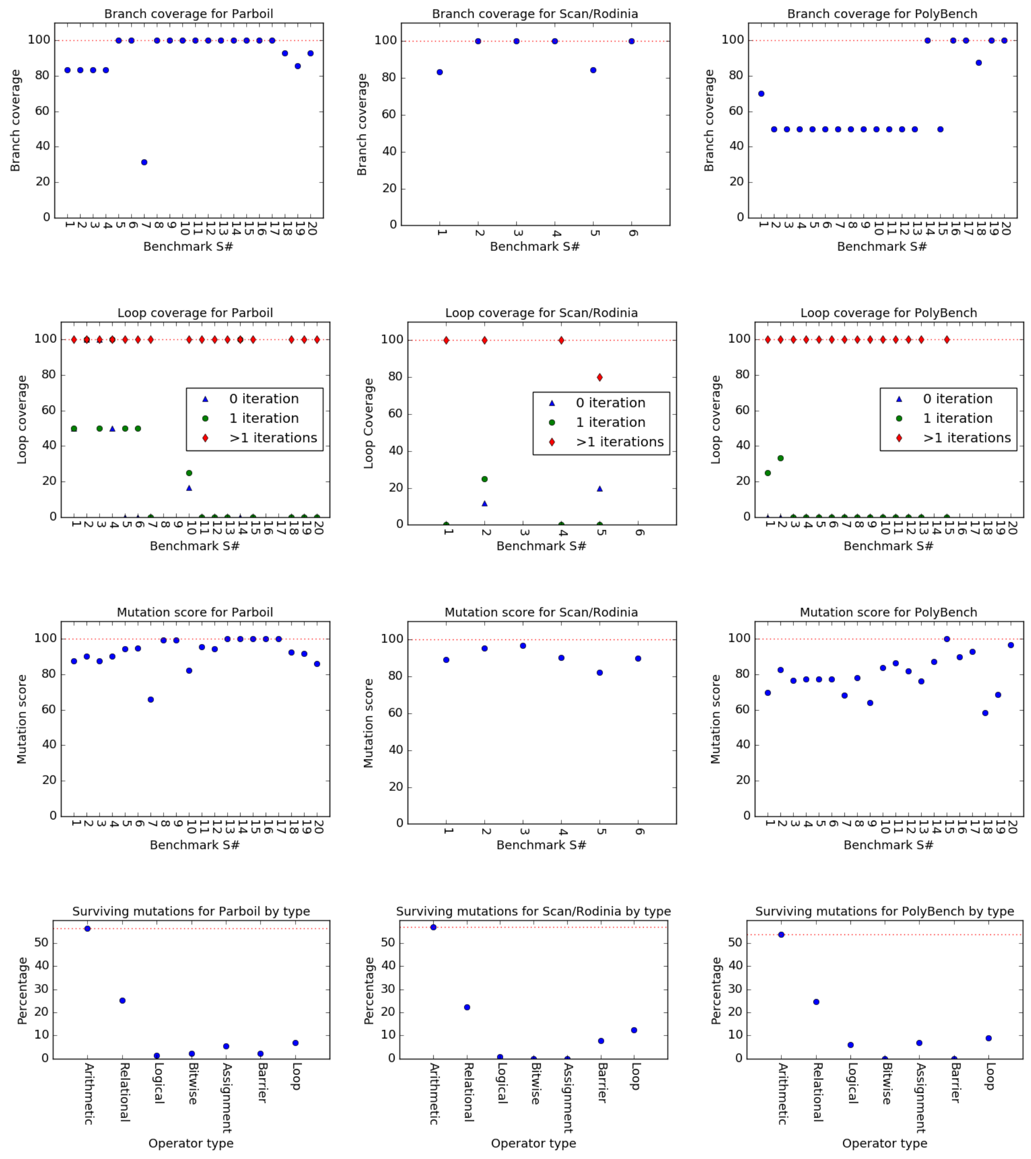


Figure 4.2: Coverage achieved - Branch and Loop, mutation score and percentage of surviving mutations by type for each subject program in the 3 benchmark suites.

Benchmark	#Kernels	#Branches	#Barriers	#Loops	#Work-groups	#Mutants
coorelation	4	10	0	4	16408	155
covariance	3	6	0	3	16400	97
2mm	2	4	0	2	8192	94
3mm	3	6	0	3	3072	132
atax	2	4	0	2	256	44
bicg	2	4	0	2	32	44
doitgen	2	4	0	1	128	123
gemm	1	2	0	1	1024	50
gemver	3	6	0	2	65568	78
gesummv	1	2	0	1	16	43
mvt	2	4	0	2	256	44
syr2k	1	2	0	1	4096	71
syrk	1	2	0	1	4096	50
lu	1	4	0	0	16392	79
adi	6	12	0	2	24	410
convolution-2d	1	2	0	0	65536	189
convolution-3d	1	2	0	0	256	512
fdtd-2d	3	8	0	0	49152	173
jacobi-1d-imper	2	4	0	0	32	35
jacobi-2d-imper	2	4	0	0	8192	117

Table 4.6: Polybench-ACC experiment summary

subject programs in the three benchmark suites[2] is shown in the plots in Figure 4.2. The first row shows branch coverage, the second loop coverage. Mutation score and surviving mutation types shown in the last two rows of Figure 4.2 are discussed in the next Section 4.4.2.

Barrier Coverage is not shown in the plots since for all, except one, applications with barriers, the associated test suites achieved 100% barrier coverage. The only subject program with less than 100% barrier coverage was `scan`, which had 87.5% barrier coverage. The uncovered barrier is in a loop whose condition does not allow some threads to enter the loop, resulting in barrier divergence between threads. We find that less than 100% barrier coverage is a useful indicator of barrier divergence in code.

[2]20 applications in Parboil counting different test suites separately, 6 in Scan/Rodinia, and 20 in PolyBench

Branch Coverage. For most subject programs in Parboil and Scan/Rodinia, test suites achieve high branch coverage ($> 83\%$). The `histo` benchmark is an outlier with a low branch coverage of 31.6%. Its kernel function, `histo_main`, contains 20 branches in a code block handling an exception condition (overflow). The test suite provided with `histo` does not raise the overflow exception, and as a result, these branches are never executed. We found uncovered branches in other applications, with $> 80\%$ coverage, in Parboil and Scan/Rodinia to also result from exception handing code that is not exercised by the associated test data.

Branch coverage achieved for 13 of the 20 applications in PolyBench is at 50%. This is very low compared with other benchmark suites. Upon investigating the kernel code, we found that all the uncovered branches reside within a condition check for out of range array index. Tests associated with a majority of the applications did not check out of range array index access, resulting in low branch coverage.

Loop Coverage. Test suites for nearly all applications (with loops) execute loops more than once. Thus, coverage for > 1 iterations is 100% for all but one of the applications, `srad` in Rodinia suite, that has 80%. The uncovered loop in `srad` is in an uncovered then-branch that checks exception conditions. We also checked if the boundary value in loop conditions is reached when > 1 iterations are covered by test executions. We found `pathfinder` in Rodinia to be the only application to have full coverage for > 1 iterations but not reach the boundary value. The unusual scenario in `pathfinder` is because one of the loops is exited using a break statement.

We find that test suites for most applications are unable to achieve any loop coverage for 0 and exactly 1 iteration. The boundary condition for most loops is based on the size of the work-groups which is typically much greater than 1. As a result, test suites have been unable to skip the loop or execute it exactly once. The only exceptions were applications in the Parboil suite - `bfs, cutcp, mri-gridding, spmv`, and two applications in Rodinia- `lud, srad`, that have boundary values dependent on variables that may be set to 0 or 1.

Overhead. For each benchmark and associated test suite, we assessed overhead introduced by our approach. We compared the time needed for executing the benchmark with instrumentation and additional data structures that we introduced for coverage measurement against the original unchanged benchmark. Overhead varied greatly across benchmarks and test suites. Overhead for Parboil and Rodinia benchmarks was in the range of 2% to 118%. Overhead was lower for benchmarks that took longer to execute as the additional execution time from instrumentation is a smaller fraction of the overall time. Overhead for most programs in PolyBench ranges from 2% to 70%, which is similar to Parboil and Rodinia benchmarks. The overhead for `lu`, `fdtd-2d` and `jacobi-2d-imper` programs are $>$ 100%. The code for kernel computations in these benchmarks is small with fast execution. Consequently, the relative increase in code size and execution time after instrumentation with CLTestCheck is high.

4.4.2 Fault Finding

Arithmetic operator and relational operator mutations that changed $<$ to $<=$, $>$ to $>=$ or vice versa were hard to be killed by existing test suites. More rigorous tests are needed to kill these mutations. Barrier mutations serve as an indicator to unnecessary barrier uses.

Fault finding for the subject programs is assessed using the mutants we generate with the fault seeder, described in Section 4.2. The mutation score, percentage of mutants killed, is used to estimate fault finding capability of test suites associated with the subject programs. Each test suite associated with a benchmark is run 20 times to determine the killed mutants. A mutant is considered killed if the test suite generates different outputs on the mutant than the original program in *all* 20 repeated runs of the test suite. In addition to killed mutants, we also report results on “Undecided Mutants”, that refers to mutants that are killed in at least one of the executions of the test suite, but *not all* 20 repeated executions. Changes in GPU thread scheduling between runs cause this uncertainty. We do not count the undecided mutants towards killed mutants in the mutation score. The mutation score for all subject programs in each benchmark suite is shown in the third row of plots in Figure 4.2.

Mutation Score. In general, we find that test suites for subject programs achieving high branch, barrier and loop coverage also have high mutation scores. For instance, for `spmv` and `stencil`, their test suites achieving 100% coverage, also achieved 100% mutation score. An instance of a program that does not follow this trend is `mri-gridding` that has 100% branch, barrier, and loop ($>$ 1 iterations) coverage but only 82% mutation score. On analysing the survived mutants, we found a significant fraction (160 out of 232) were arithmetic operator mutations within a function named *kernel_value* that contained variables defining a fourteenth-order polynomial and a cubic polynomial. The effect of mutations on the polynomials did not propagate to the output of the benchmark with the given test suite. The `histo` program with low branch coverage, 100% barrier and loop coverage has 65.9% mutation score. Nearly two-thirds of the branches in `histo` cannot be reached by the input data. As a result, all the mutations in the untouched branches are not killed, resulting in a low mutation score. A few of the programs in PolyBench have mutation scores that are between 60 – 70%. In these programs, most surviving mutations are arithmetic operator mutations.

As seen in the last row of Figure 4.2 showing surviving mutations by operator type, arithmetic operators are the dominant surviving mutations in all three benchmark suites. Control flow adequate tests can kill arithmetic operator mutations only if they propagate to a control condition or the output. Data flow coverage may be better suited for estimating these mutations. Around 20% of 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 suites provided with the benchmarks missed such boundary mutations.

Undecided mutants occur during executions of 9, out of the 46 subject programs and test suites across all three benchmark suites. The number of undecided mutants during the 9 executions is generally small ($<= 5$). The only exception is `tpacf` in the Parboil benchmark suite, that resulted in 18 undecided mutants when executing one of its test suites. Undecided mutants point to non-deterministic behaviour in the kernel, that is dependent on the GPU thread execution model. A large number of undecided mutants is alarming and developers should examine kernel code more closely to ensure that the behaviour observed is as intended.

Barriers were not used in all benchmarks. Only 5 out of the 9 benchmarks in Parboil, and 4 of the 6 in Scan/Rodinia had barriers. PolyBench programs did not use any barriers. Mutations removed barrier function calls in these benchmarks and we recorded the number of mutants killed by test suites. The percentage of killed barrier mutations is generally low across all benchmarks with barriers. For instance, removing 2 out of 3 barriers in the `histo` program in Parboil, and removing all barriers in the `cutcp` program had no effect on the outputs of the respective program executions. This may either mean that the test suites are inadequate with respect to the barrier mutations or it could be an indication that these barriers are superfluous with respect to program outputs, and the need for synchronisation should be further justified. For the programs in our experiment, we found barriers, whose mutations survived, to be unnecessary.

Coverage versus Mutation Score. The plots in Figure 4.2 illustrate the total mutation score over all types of mutations for each subject program and test suite. We also compute mutation scores specifically for branches, barriers, and loops using mutations relevant to them. We do this to compare against branch, barrier and loop coverage achieved for each of the subject programs. We found that mutation score for branches closely follows branch coverage for most subject programs. Outliers include `adi`, `nn`, `convolution-2d` and `convolution-3d`. Mutations that change $<$ to $<=$ are not killed in these kernels; these comprise one third of all branch mutations.

The mutation score for barriers is quite different from barrier coverage. This is because test suites are able to execute the barriers and achieve coverage. However, they are unable to produce different outputs when the barriers are removed. This may be a problem with the superfluous manner in which barriers are used in these programs.

Loop coverage with > 1 iterations is 100% for all but one subject program (`srad` in Rodinia). Mutation score for loops on the other hand is variable. In general, tests achieving loop coverage are unable to reveal loop boundary mutations. Histo and srad are worth noting with high loop coverage but low loop mutation scores. We find that mutations to the loop boundary value in these two benchmarks survive, which implies that access to loop indices outside the boundary go unchecked in these programs. These unsafe values of loop indices should be disallowed in these kernels and loop boundary mutations in our fault seeder help reveal them.

4.4.3 Schedule Amplification: Deadlocks and Data Races

The scheduler amplifier can uncover deadlocks when the work-group selected to execute first has an ID that exceeds the number of available compute units. It also revealed inter work-group data races in the `spmv` kernel from the Parboil benchmark.

Kernel Deadlocks: When we used the CLTestCheck schedule amplifier on our benchmarks, we found kernel executions deadlock when the work-group ID selected to go first exceeds the number of available compute units. As there are no guarantees on how work-groups are mapped to compute units, we allow work-group IDs exceeding the number of compute units to go first in some test executions using our schedule amplifier. However, it appears that the GPU makes unstated assumptions on what work-group IDs are allowed to go first. As noted by Sorenson et al. [118], "execution of a large number of work-groups is in any *occupancy bound* fashion, by delaying the scheduling of some work-groups until others have executed to completion." They observed deadlocks in kernel execution due to inter work-group barriers. However, in the benchmarks in our evaluation, there is no explicit inter work-group barrier. It may be the case that developers made implicit assumptions on inter work-group barriers using the occupancy bound model and our schedule amplification approach violates this assumption. Nevertheless, our finding exposes the need for an inter work-group execution model that explicitly states the details and assumptions related to the mapping of work-groups to compute units for a given kernel on a given GPU platform.

Inter Work-group Data Races: We could reveal a data race in the `spmv` application from the Parboil benchmark suite. We found that when work-groups 0 or 1 are chosen to go first in our schedules, the kernels execution always produces the same result. However, when we pick other work-group ids to go first, the test output is not consistent. Among twenty executions for each schedule, the frequency of producing correct output varies from 45% to 70%.

We observe similar behaviour in the `tpacf` application in Parboil when we delete the last barrier function call in the kernel. The kernel execution produces consistent outputs when we pick work-group 0 or 1 to go first. When we pick other work-groups to go first using our schedule amplifier, the kernel execution results are non-deterministic.

We observe no unusual behaviour in any of the PolyBench programs. These programs split the computation into multiple kernels and the CPU program launches GPU kernels one by one. The transfer of control from the GPU to the CPU between kernels acts like a barrier as the CPU will wait until a kernel finishes before launching the next kernel. In addition, care has been taken in the kernel code to ensure threads do not access the same memory location. As a result, we observe no data races in PolyBench with our schedule amplifier.

4.4.4 Threats to Validity

A potential threat to internal validity is bugs in CLTestCheck's implementation. To mitigate this threat, we conducted careful code reviews and extensive testing. We also conducted manual inspection to check if the instrumented kernels are correct. Further, the implementations are publicly available for other researchers and potential users to check the validity of our results.

A potential threat to the external validity is related to the fact that our dataset may not be an accurate representation of all aspects of OpenCL kernel development. We attempt to reduce the selection bias by using different benchmarking suites with kernels from a variety of application domains.

4.5 Summary

In this chapter, we have presented the CLTestCheck framework for measuring test effectiveness over OpenCL kernels with capabilities to measure code coverage, fault seeding and mutation score measurement, and finally amplify the execution of a test input with multiple work-group schedules to check inter work-group interactions.

Our empirical evaluation of CLTestCheck capabilities with 82 publicly available kernels revealed the following,

1. The schedule amplifier could detect deadlocks and inter work-group data races in Parboil benchmarks when higher work-group ids were forced to execute first. This finding emphasizes the need for transparency and clearly stated assumptions on how work-groups are mapped to compute units.
2. Barrier coverage served as a useful measure in identifying barrier divergence in benchmarks (`scan`).
3. Branch coverage pointed to inadequacies in existing test suites and found test inputs for exercising error handling code were missing.
4. Across all benchmark suites, we found arithmetic operator and relational operator mutations that changed $<$ to $<=$, $>$ to $>=$ or vice versa were hard to kill. More rigorous test suites to handle these mutations are needed.
5. The use of barrier mutations revealed several instances of unnecessary barrier use. Barrier usage and its implications are not well understood by developers. Barrier mutations can help reveal incorrect barrier uses.
6. Loop boundary mutations helped reveal unsafe accesses to loop indices outside the loop boundary.

In sum, the CLTestCheck framework is an automated, effective and useful tool that will help developers assess how well OpenCL kernels have been tested, kernel regions that require further testing, uncover bugs with respect to work-group schedules.

In the next chapter, we explore automated test input generation for OpenCL kernels. We also address the limitation that CLTestCheck can only fix one work-group to be executed first by presenting an automated kernel execution engine which is able to test OpenCL kernels without the host code and provides the all work-group schedule capability. With this execution engine and test input generator, the developer can generate high-quality test inputs to uncover data races and barrier divergence, without bothering with writing the boiler-plate host code.

Chapter 5

Automated Test Input Generation for GPU Programs

5.1 Introduction

Chapter 4 provides a set of metrics with an automated framework, CLTestCheck, to measure the effectiveness of test suites developed for OpenCL kernels. CLTestCheck can report code coverage achieved and fault finding capabilities for kernels with their test suites. Our empirical evaluation demonstrates that the framework is useful in guiding kernel developers to uncover data races and barrier divergence.

However, there is an urgent need for a *fast*, *effective* and *scalable* technique to generate test inputs for GPU kernels. In this chapter, we present the CLFuzz framework, which takes only the kernel as input, generates test inputs and different work-group schedules, executes the kernel with these inputs and schedules without asking the developer to write boiler-plate host code and finally analyses the output to report code coverage and checks for data races as well as barrier divergence. The overall workflow is illustrated in Figure 5.1.

To make this happen, we propose a testing technique that combines fuzz testing with SMT 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 [136, 137] and found 76% more bugs when compared to a symbolic executor (angr) in a 24-hour period [138]. Currently, there are no available fuzz testers for GPU kernels. The first

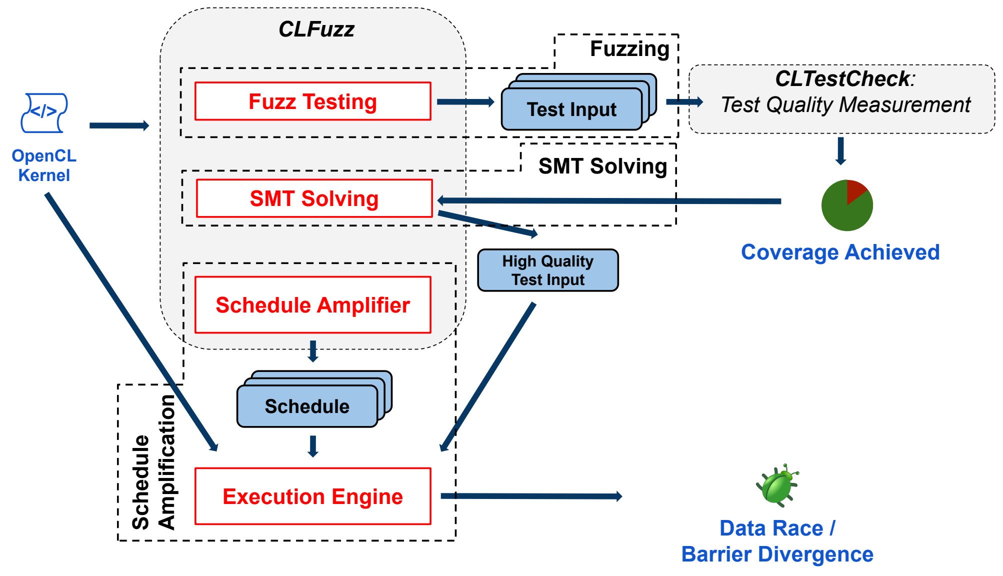


Figure 5.1: CLFuzz Workflow

contribution in this chapter 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 mutations) over the OpenCL kernels.

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 chapter addresses this weakness. When our fuzzer is unable to reach full (or high) coverage of the kernel, we complement it with an SMT solver[1] that is tasked with generating inputs for *uncovered* branches. The test inputs generated by the SMT 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*. As mentioned in Chapter 2, existing GPU architecture and simulators do not provide a means to control work-group schedules and the OpenCL specification provides no execution model for inter work-group interactions [27]. 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 present a partial scheduler by only fixing the first work-group in Chapter 4. Schedules generated by the partial scheduler are not realistic, and resulted in creating deadlocks in some kernel executions. In this chapter, we provide a simulator that can provide complete orderings over all the work-groups, addressing limitations in the par-

[1]We invoke an SMT solver to generate inputs satisfying constraints for uncovered paths.

tial scheduler. 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.

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 SMT solver to achieve full branch coverage. The average mutation score improved significantly (54% to 73%) with the combined approach for these kernels. We found most of the surviving 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 could uncover data races in 21 kernels. The 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 chapter are:

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

5.2 CLFuzz: Automated Test Generation and Execution for OpenCL Kernels

In this section, we present the CLFuzz framework that provides automated test input generation using 1. *Mutation-based fuzzing* and 2. *Selective SMT 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.

Test Input Definition. With the aim to check for the presence of data races and classify them, we consider a test input is comprised of a test input and a work-group schedule and we do not need an expected output. As discussed in the previous section, if the kernel produces inconsistent outputs by running it multiple times, the presence of the data race is confirmed. Work-group schedules can also affect the test output of OpenCL kernels when there are data races. When running the kernel with a fixed work-group schedule rather than the random schedule by default, inter work-group data race can be confirmed if outputs become consistent.

5.2.1 Mutation-based Fuzzing

Our technique for mutation-based fuzzing is illustrated in Algorithm 1.

Algorithm 1 Mutation-based Fuzzing Algorithm

```
Input: Kernel argument type list $K$
Output: List of test inputs $T$, branch coverage $B = [b_1, b_2, .., b_n]$ ($n$ = number of
    branches and $\forall b \in B$ is a boolean value indicating if the corresponding branch
    is covered
 1: $T \leftarrow []$ // Empty list
 2: Generate a random seed test $t$ based on type list $K$
 3: Add $t$ to $T$
 4: $B \leftarrow$ coverage achieved by running $t$ with coverage measurement
 5: while MAX_NUM_ATTEMPT not exceeded && $\exists b \in B$ is FALSE do
 6:     $t_{tmp} \leftarrow$ mutate one value of $t$
 7:     $B_{new} \leftarrow$ coverage achieved by running $t_{tmp}$ with coverage measurement
 8:     if $B_{new}$ has more TRUE than $B$ then
 9:         Add $t_{tmp}$ to $T$
10:         $B \leftarrow B_{new}$
11:         $t \leftarrow t_{tmp}$
12:     end if
13: end while
```

In the algorithm, test inputs is generated using the following steps,

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	scalar`n`	A vector of `n` scalar values, e.g., int2, float16
Struct	struct	A struct comprised of scalar and vector values
Image	image`nd`_t	An `n`-dimensional image, e.g., image2d_t

Table 5.1: Summary of kernel argument data types

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 result 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 5.1. We use CLTestCheck 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).

5.2.1.0.1 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 5.1: 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}).

5.2.2 Selective SMT Solving

We address the limitation of mutation-based fuzzing in determining specific inputs to pass complex checks using selective SMT solving with the algorithm shown in Algorithm 2.

When the fuzzer is unable to increase branch coverage after going through a predetermined amount of mutations (proportional to the number of kernel arguments), we consider the fuzzer to have reached its limit. We then invoke selective SMT solving to generate tests for uncovered branches in the kernel.

For each uncovered branch, selective SMT 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 [139] SMT solver to determine an input that will

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

Algorithm 2 Selective SMT Solving Algorithm

```
Input: Control flow graph $cfg$, not covered branches $B$
Output: List of test inputs $T$
1: $T \leftarrow []$ // Empty list
2: for $b \in B$ do
3:     $P \leftarrow$ path constraints to $b$ from $cfg$
4:     Solving result $s$, generated input $t \leftarrow$ Z3 SMT Solver solving $\leftarrow P$
5:     if $s ==$ TRUE then
6:         Add $t$ to $T$
7:     end if
8: end for
```

satisfy the constraints. If such an input can be found, it is added to the test suite as a test exercising the uncovered branch. We repeat this for all uncovered branches.

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

5.2.3 Schedule Amplification

As mentioned earlier in Chapter 2, no mechanism is provided by GPU vendors to manipulate and set work-group schedules. As a result, the 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 discrepancies 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 5.2 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 5.2 where work-groups $3,5,2,6$ are launched first, followed by $4,1,0,7$. Although the example shows a 1-dimensional work-group schedule execution model, our schedule amplifier is capable of supporting multi-dimensional work-group schedules.

Kernels usually launch hundreds of work-groups, which makes it impractical to generate all possible work-group schedules. In this thesis, 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 schedules than is currently possible, achieving better coverage of the work-group schedule space.

The partial schedule generator in CLTestCheck 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 cause 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

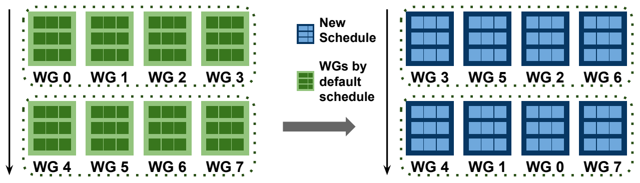


Figure 5.2: Schedule controlled by the enhanced scheduler

deadlocks for the subject kernels in our experiment. The kernel developer also has better control over work-group schedules with our schedule amplifier.

5.2.4 Host Code Generation and Kernel Execution

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. By unifying the format of the data file storing the generated test input and work-group schedule, the kernel execution engine is able to automatically read 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.

Our approach frees kernel developers from writing the redundant and tedious host code and allows them to focus on the kernel code itself.

5.2.5 Implementation of the Framework

The CLFuzz framework is implemented using Clang LibTooling [32]. 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 5.2. The developer can modify this data file to specify attributes, such as the desired size of arrays and if the argument is an input or output parameter. The framework is written in Python and uses PyOpenCL API [140] for kernel execution and PyZ3 [139] for SMT solving. We use

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	The user can specify an initial value or provide a file to initialise the parameter if needed.
initial_value	
result	True if it is used to store the output of the kernel.
pos	The position of the parameter in the interface. .

Table 5.2: Kernel interface information

the CLCov and CLMT tools from the CLTestCheck framework [2] 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`.

5.2.5.1 Kernel analysis and information extraction.

The framework analyses the kernel code using the AST (abstract syntax tree) traversing capability provided by Clang LibTooling and extracts the following information for test input generation:

1. Kernel interface for random input data generation.

   The framework iterates all kernel arguments stored in the AST, gathers their properties including data type and scope etc. and stores them in a data file. The content in the data file is shown in Table 5.2. The kernel developer can modify this data file to supply the desired size of the arrays and specify which array is used to store the output of the kernel.

2. Control flow for SMT solving.

As discussed in Section 5.2.2, basic blocks, conditions and the linkage of them representing the control flow of the kernel are gathered from the AST to build constraints.

5.2.5.2 Test Input Generation.

CLFuzz first randomly generates a test input for kernel arguments according to the kernel interface and measure branch coverage achieved by the CLTestCheck framework. The fuzzer keeps mutating the test input to increase branch coverage and when the coverage stops increasing, it turns to the SMT solver. With the control flow information, the SMT solver gathers constraints that can lead to the uncovered branch, solves them and produces new test input if the constraints are satisfiable.

5.2.5.3 Schedule Amplification

We provide wrappers for the standard OpenCL API functions to enable different work-group schedules as shown in the following code listing.

Listing 5.2: OpenCL API Wrapper for Schedule Amplification

```
int get_group_id(int dimindx, __global uint3* cl_schedule_map) {
  int id;
  int dimension = get_work_dim();
  if (dimension == 1) {
    id = get_group_id(0);
  } else if (dimension == 2) {
    id = get_group_id(0) + get_group_id(1) * get_num_groups(0);
  } else if (dimension == 3) {
    id = get_group_id(0) + get_num_groups(0) * (get_group_id(1) + ↩
        get_num_groups(1) * get_group_id(2));
  }
  if (dimindx == 0) {
    return cl_schedule_map[id].x;
  } else if (dimindx == 1) {
    return cl_schedule_map[id].y;
  } else {
    return cl_schedule_map[id].z;
  }

int get_global_id(int dimindx, __global uint3* cl_schedule_map) {
  int new_group_id = get_group_id_new(dimindx, cl_schedule_map);
  return new_group_id * get_local_size(dimindx) + get_local_id(↩
      dimindx);
}
```

The schedule generated on the host side is passed to the kernel by the parameter, `cl_schedule_map` array. For instance, if there are 4 work-groups in total and the new schedule is 3, 2, 1, 4, the new `get_group_id` function uses the real work-group ID as the index and returns the corresponding array element to the statement which queries the work-group ID. Therefore, the first launched work-group (Work-group 0) gets 3 and will execute the kernel for data originally allocated for Work-group 3. Similarly, for statements querying their global ID, the new `get_global_id` function also checks the map and returns the shuffled value based on the generated schedule.

As OpenCL supports launching kernels in 1-dimensional. 2-dimensional and 3-dimensional spaces, we also enabled the wrapper to check the launched number of dimensions and query the new schedule number accordingly.

5.3 Experiment

In our experiment, we evaluate the feasibility and effectiveness of mutation-based fuzz testing, selective SMT solving and schedule amplification proposed in Section 5.2 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. 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 the 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 SMT Solving: *Can selective SMT 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 SMT 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.

5.3.1 Subject Kernels.

We use 217 kernels collected from the following benchmark suites for our experiments:

- 24 kernels collected from open-source projects bilateral, clpractical, 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 the 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.

5.4 Results

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 5.3 is presented below.

5.4.1 Effectiveness of Fuzz Testing

Mutation-based fuzzing on its own produced 100% branch coverage for 186 of the 217 kernels and $>$ 90% mutation score for 110 kernels. Across all kernels, the fuzzer achieved 91.5% branch coverage 74.9% mutation score. Kernels with branches requiring a specific value, boundary checks of arrays or nested control flows with strict conditions were hard to get full coverage by the fuzzer.

Figure 5.3 presents a frequency plot with the 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 the median is 29. The kernel with the 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 5.3, 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.

5.4.1.1 Branch coverage

For 31 out of 217 kernels, fuzz testing does not achieve full branch coverage with the generated test inputs as shown in Table 5.3. Upon investigation, we found the following principal 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. In this example, 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

Benchmark	Kernel	Coverage by Fuzzing
OpenDwarfs	nqueens1	14.71%
	calculate_potential	18.75%
	bwa_hmm_opencl_11	50.00%
	bwa_hmm_opencl_16	50.00%
	bwa_hmm_opencl_17	50.00%
	kmean_2	66.67%
	bwa_hmm_opencl_13	75.00%
	nqueens2	83.33%
	cfd_kernel_1	83.33%
Parboil	lbm	50.00%
	sad2	50.00%
	convs	75.00%
	cutcp	75.00%
	tpacf	83.33%
	bfs	91.67%
Rodinia	heartwall	17.07%
	srad_3	40.00%
	dwt2d_3	50.00%
	srad_6	50.00%
	srad_4	62.50%
	leukocyte_track_ellipse_kernel	73.08%
	cfd_4	75.00%
	leukocyte_find_ellipse_kernel_1	75.00%
	leukocyte_find_ellipse_kernel_2	75.00%
	srad_5	75.00%
	hotspot	80.00%
	B+tree_2	85.71%
	hybridsort_mergesort_2	85.71%
	B+tree_1	87.50%

Table 5.3: Kernels with less than 100% coverage by fuzzing

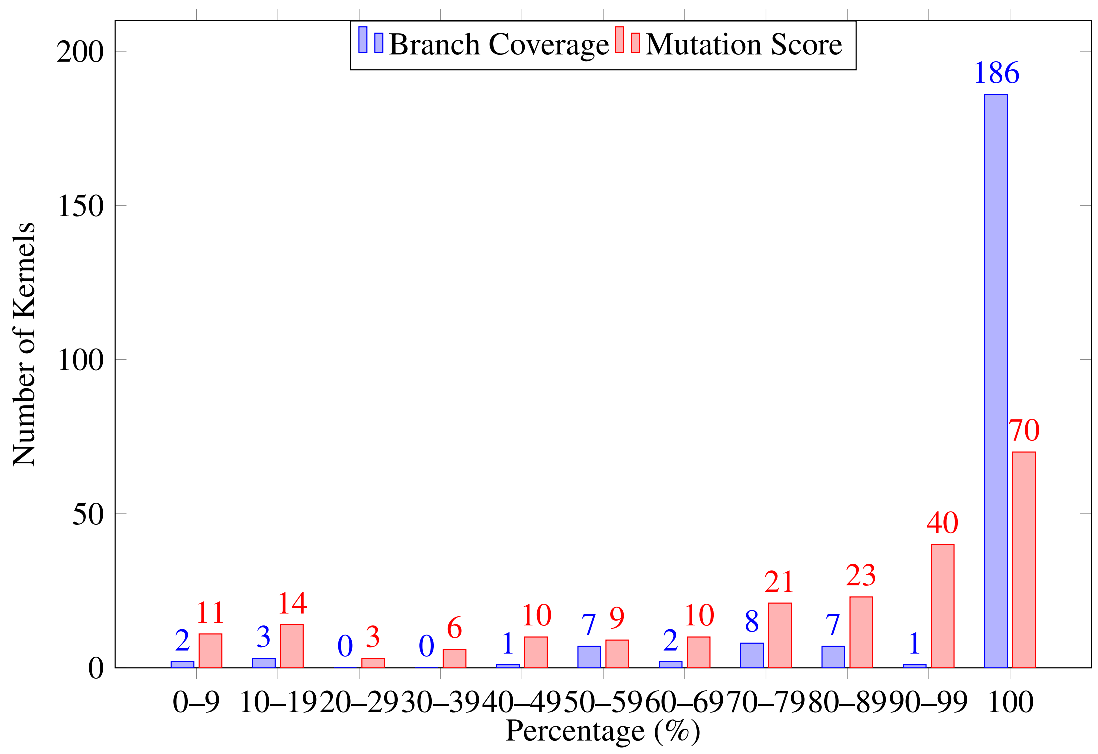


Figure 5.3: Frequency of branch coverage and mutation score across subject kernels

that it will match the exact value of *idy* is very low.

Listing 5.3: 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
    }
}
```

The following example illustrates a similar issue in the condition check. Instead of asking a specific value of a variable, it requires a data element in the input array to be 0 and the possibility is further less, making the fuzzer hard to enter the true branch.

Listing 5.4: The scale_b_dev kernel from bwa_hmm_opencl_13

```
__kernel void scale_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 (b_d[(idy * nstates) + idx] == 0) {
        //Computations using idx and idy
    }
}
```

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. 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. For this kind of circumstances, it is obvious that the SMT solver can easily give us a value of *no_of_nodes* that is less than the number of available nodes and the false branch can be exercised. However, there is no out-of-boundary error handling in these kernels and developers usually provide the kernel with the original number used for allocating the array in the memory. We do not envision that tests not covering these false branches is a big problem.

Listing 5.5: 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
    }
}
```

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. We discuss the effectiveness of the SMT solver addressing this issue in the next Section.

5.4.1.2 Fault Finding

Fault finding for the subject kernels is assessed with the help of the mutant generation component in the CLTestCheck framework. 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 a 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 scores achieved are 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, 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) has 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 SMT solving helps increase the mutation score for these kernels.

5.4.2 Effectiveness of SMT Solving

The SMT Solver can generate test inputs for uncovered branches by the mutation-based fuzzer. For these 31 kernels, the average mutation score was increased from 61% to 77%.

For the 31 kernels that do not have full branch coverage with fuzz tests, we run the SMT solver to generate tests for uncovered branches. We find that 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 SMT 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 5.4.1 are easily covered by the SMT 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 SMT 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 the 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 and nested control flow of 5 levels but only takes 1.3 seconds for the SMT 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 SMT 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. The mutation score of this kernel is increased from 53% to 95% with the SMT solver tests.

Figure 5.4 illustrates a comparison of mutation scores achieved by fuzz tests versus fuzz + SMT solver tests for the 31 kernels that used the SMT solver. The average mutation score for the 31 kernels increases from 54.3% to 73.3% with the combined approach. We find that including tests from the SMT solver improves the mutation score significantly for 14 out of the 31 kernels. Among the remaining kernels with unchanged mutation scores, for 8 of them, the SMT 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

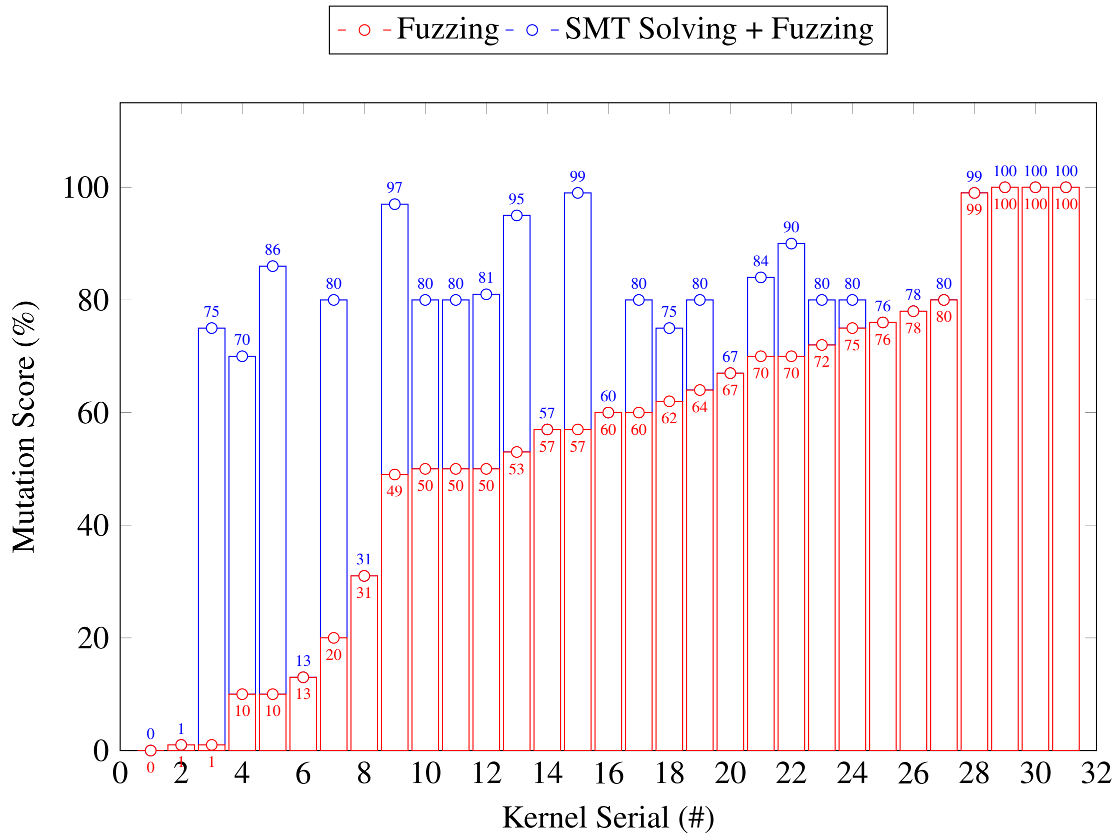


Figure 5.4: Mutation score achieved over the 31 kernels using “Only Fuzzing” versus “Fuzzing + SMT solving”.

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 5.4: Kernels with data races

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.

5.4.3 Effectiveness of Schedule Amplification

The scheduler amplifier can shuffle work-group schedules for subject kernels and uncovered data races in 21 kernels (2 with intra work-group data races and 19 with inter work-group data races). These inter work-group data race was hard to be uncovered with a fixed work-group schedule but can be reproduced using the schedule amplifier with shuffled schedules.

Each test generated by the fuzzer and selective SMT 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]

Our schedule amplification technique could detect data races in 21 kernels, as shown in Table 5.4. Tests executed on these kernels produced different outputs on at least 2 out of 10 different work-group schedules. Each work-group schedule is ad-

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

ditionally 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 observation 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.

When further investigating the root cause of data races, we found that some of them can be avoided by providing valid inputs instead of randomly generated inputs. All kernels from Polybench fall into this case and we use the following code listing from the 2mm_1 kernel as an example.

Listing 5.6: 2mm_1 kernel from Polybench

```
__kernel void mm2_kernel1(
      __global DATA_TYPE *tmp,
      __global DATA_TYPE *A,
      __global DATA_TYPE *B,
      int ni,
      int nj,
      int nk,
      int nl,
      DATA_TYPE al) {
  int j = get_global_id(0);
  int i = get_global_id(1);

  if ((i < ni) && (j < nj)) {
    tmp[i * nj + j] = 0;
    int k;
    for (k = 0; k < nk; k++) {
      tmp[i * nj + j] += al * A[i * nk + k] * B[k * nj + j];
    }
  }
}
```

The output array tmp is access by using the expression $i*nj+j$, which is a calculation based on the number of threads and thread IDs. The number of threads (variable nj) is passed as a kernel argument rather than queried at runtime. Our random fuzzer cannot understand the usage of this argument and generate a random number for it, which leads to duplicated write accesses to the same location by the wrong result of the expression.

5.4.4 Scalability and Overhead

Our approach can scale to the largest kernel with 2235 lines of code. Random fuzzing took 0.01 to 2 seconds and SMT solving took 0.2 seconds to 1 minutes for subject kernels. The average overhead of our approach is 0.8 second which is negligible.

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 could verify that CLFuzz could 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.

SMT solving takes between 0.2 seconds to 1 minute across the 31 kernels to generate tests for uncovered branches. However, since we only use the SMT 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 used in the previous chapter does not make efficient use of the compute units. When the first work-group is executing, the initial approach requires other work-groups to wait till execution of the first one is finished, making kernel execution slower.

5.4.5 Threats to Validity

A potential threat to internal validity is bugs in CLFuzz's implementation. We used Clang LibTooling for kernel interface parsing the control flow graph generation, and Python to implement the test generation and execution engine. We had a careful code review and made our work publicly available.

For threats to external validity, we used 217 kernels from industry-standard benchmarks and open-source OpenCL projects in the experiment to evaluate our approach. Although they are from different application categories and their code covers a variety of features of OpenCL programming, a thorough study on more OpenCL kernels can further improve the validity of our approach.

A threat to construct validity is caused by restricting the number of attempts for the mutation-based fuzzer to 50. Restriction to 50 attempts was based on our observation that for most of the kernels, if the coverage was not improved after 50 random tests, the branch coverage remained unchanged.

5.5 Summary

In this chapter, we present a test generation technique for OpenCL kernels that combines mutation-based fuzzing and selective SMT 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 SMT 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 SMT 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 SMT 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 could uncover data races in 21 kernels with our schedule 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 SMT solving, and schedule amplification is fast, effective and scalable.

Chapter 6

Conclusion

This thesis presents novel techniques to assess the correctness of GPU kernels by test effectiveness measurement, automated test input generation and execution with schedule amplification.Our proposed techniques address challenges related to the SIMT execution model, GPU programming and memory management restrictions and the lack of customised thread scheduling and global synchronisation, which is not encountered in CPU systems and is not addressed by existing techniques.

6.1 Contributions

This thesis makes three main contributions:

Test Effectiveness Measurement

Test effectiveness is an important indicator of the quality of test inputs used for GPU kernels. The thesis proposed CLTestCheck, a test effectiveness measurement framework based on our coverage metrics including branch, loop and barrier (synchronisation) coverage specifically designed for GPU kernels considering their memory and execution model. The framework is also able to measure fault finding capabilities of test inputs by seeding faults using traditional operator and GPU-specific mutants. The empirical evaluation on 82 publicly available kernels and associated test inputs from industry-standard benchmarks illustrated that the framework was useful in accessing how well OpenCL kernels were tested by their test inputs. Code coverage measurement could reveal kernel regions that required future testing and the fault seeding capability could expose unnecessary barriers and unsafe loop accesses.

Test Case Generation

Based on fuzz testing and SMT solving, we proposed CLFuzz, an automated test input generation framework for OpenCL kernels. The random fuzzer is first used to analyse the interface of the subject kernel and generate seed inputs and mutate these inputs to achieve higher code coverage. If full coverage is not achieved after a threshold number of iterations, the SMT solver is then deployed to analyse uncovered branches and generate inputs by solving constraints of conditions of these branches. Experimental results showed that mutation-based fuzzing on its own could achieve full branch coverage and reveal 75% seeded faults for the majority of kernels in our data set with negligible overhead. The SMT solver, although time-consuming compared to the fuzzer, could achieve full coverage for the remaining kernels and increased the overall fault finding capability from 61% to 77%.

Test Execution with Schedule Amplification

To study how work-group schedules affect the execution of OpenCL kernels, we implemented the first simulator, CLSchedule, that is capable of shuffling work-group schedules rather than following the arbitrary default schedules. CLSchedule is also able to automatically execute the kernel with test inputs generated by CLFuzz without the host code, which eliminates testers' effort in writing boiler-plate code and help them focus on testing. With the scheduler, CLFuzz uncovered inter work-group data races in 21 OpenCL kernels in our experiment.

6.2 Critical Analysis

The novel approach presented in this thesis remains a prototype and several issues need to be addressed before it can be applied to test real world GPU applications.

Limited Scope of GPU Programming Frameworks

The thesis discusses test effectiveness and test input generation techniques for OpenCL kernels as the target GPU programming framework as it supports most GPU vendors as discussed in Section 2.2. However, there exist many other GPU programming frameworks such as CUDA for NVIDIA GPUs. It is unclear and hard to estimate how many of existing GPU programs are written in CUDA but it is reported that NVIDIA has the

77% market share for desktop GPUs as of 2020 [141]. On top of CUDA itself, NVIDIA has devoted much effort to CUDA libraries such as cuFFT [142] and cuDNN [143] for computing Fourier transforms and training deep learning models respectively. cuDNN, for example, has been used in many popular deep learning frameworks such as TensorFlow [144] and PyTorch [145].

Given that NVIDIA GPUs and CUDA libraries are widely-established, measuring test effectiveness for CUDA kernels in these libraries can serve as a promising ground for their developers to understand the extent to which these kernels are tested. In addition, CLFuzz can also help find potential synchronisation bugs for these kernels.

Scalability to Large GPU Kernels

Subject kernels used in our experiments presented in Sections 4.3 and 5.3 are collected from benchmarking suites and open-source projects. These kernels, although represents real-world algorithms for different application domains, are domain-specific functions, each for one fine-grained computation procedure. However, in industrial development, a kernel function can be complicated and designed in the context of business logic, with a set of algorithms with more nested control flows. Achieving full branch coverage for these kernels are not as straightforward as benchmarking kernels used in the thesis and those deep and nested control flows may challenge our selective SMT solver.

Limited Scope of Accelerators

With the fast development of computing devices, there are more types of accelerators in addition to GPUs. Tensor Processing Units (TPUs), for instance, is an accelerator developed by Google as a application-specific device for TensorFlow applications. Compared with GPUs, TPUs support 8-bit low-precision floating point computation while the the OpenCL abstraction only supports 16-, 32- and 64-bit floating point numbers as presented in Table 5.1 in Section 5.2. In addition, with a more energy-efficient input and output mechanism, TPUs are better suited for CNNs (Convolutional Neural Networks) while GPUs have more advantage for fully-connected neural networks [146]. These application-specific accelerators have more benefit in their domains where GPU kernels are not competitive and prominent.

6.3 Future Work

All the issues discussed in the previous section serve as a direction of future work. In this section, we discuss potential solutions for the future research.

Adoption for Kernels Written in CUDA and OpenACC

As discussed in Section 2.2, there exist other programming frameworks for heterogeneous systems including CUDA [26], OpenMP 4 [147], OpenACC [148], SYCL [149], etc. Among them, CUDA was developed by NVIDIA specifically for NVIDIA GPUs but shares the similar 3-level memory hierarchy and execution model with OpenCL. Therefore, mitigating our test effectiveness measurement, test input generation and schedule amplification techniques to CUDA kernels should involve only engineering efforts.

OpenMP was originally developed to program multi-threaded C programs. Beginning with version 4.0, OpenMP supports offloading to GPUs. OpenACC (Open ACCelerators) is designed to simplify parallel programming of heterogeneous CPU/GPU systems. Both OpenMP and OpenACC use directives (pragmas) in their programs to mark code blocks that can be executed in parallel and provide a set of support functions to initialise the parallel model, control synchronisation and query runtime information.

SYCL is another high-level programming model for hardware accelerators developed by the Khronos Group in addition to OpenCL. However, instead of putting the GPU code in kernels, SYCL is a single-source embedded language based on pure C++ 17. SYCL shares a similar memory and execution model with OpenCL.

These programming frameworks are supported by mainstream GPU vendors and compiler developers including NVIDIA, Intel, GNU Compiler Collection (GCC), etc. We plan to mitigate our testing techniques designed for OpenCL kernels to these frameworks taking into consideration their own characteristics.

Support for Other Types of Accelerators

In addition to GPUs, there exist other hardware accelerators such as FPGAs (field-programmable gate arrays), DSPs (digital signal processors) and TPUs (tensor processing units). These accelerators are either general-purpose (FPGAs) or domain-specific (DSPs and TPUs). OpenCL kernels and SYCL programs are also compiled to these platforms or hybrid CPU/accelerator systems. As the abstracted memory and execution model is unchanged, we plan to apply our techniques proposed in the thesis to more accelerators and hybrid systems.

6.4 Concluding Remarks

Although costly and time-consuming, software testing plays a critical role in the process of software development. Owing to distinct characteristics of the programming framework, three-level memory hierarchy and SIMT execution model, OpenCL kernels require special testing techniques when testing them. This thesis proposes a novel approach to generate test inputs for OpenCL kernels automatically, measure the effectiveness of test inputs and execute subject kernels with different work-group schedules to check for synchronisation bugs.

Our experimental results on industry-standard benchmarks are promising but there is also work to be done to enable the adoption of this approach to other heterogeneous devices such as FPGAs and GPU kernels written in programming frameworks other than OpenCL. In addition, the evolving GPU architectures and programming models can introduce new challenges in the future.

However, current computing demands rely heavily on accelerators like GPUs and this reliance is meant to go up as artificial intelligence and machine learning is applied more widely in our daily lives. The author of this thesis hopes that the presented research and techniques can inspire developers and testers, and serve as a basis for future research on testing heterogeneous computing systems.

Bibliography

[1] C. Peng, “On the correctness of gpu programs,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 443–447, 2019.

[2] C. Peng and A. Rajan, “Cltestcheck: Measuring test effectiveness for gpu kernels,” in *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pp. 315–331, Springer, 2019.

[3] C. Peng and A. Rajan, “Automated test generation for opencl kernels using fuzzing and constraint solving,” in *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU)*, pp. 61–70, 2020.

[4] C. Peng, S. Akca, and A. Rajan, “Sif: A framework for solidity contract instrumentation and analysis,” in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 466–473, IEEE, 2019.

[5] S. Akca, A. Rajan, and C. Peng, “Solanalyser: A framework for analysing and testing smart contracts,” in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 482–489, IEEE, 2019.

[6] C. Peng, A. Rajan, and T. Cai, “Cat: Change-focused android gui testing,” in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021.

[7] S. Akca, C. Peng, and A. Rajan, “Testing smart contracts: Which technique performs best?,” in *2021 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ACM, 2021.

[8] M. AbdelBaky, M. Parashar, H. Kim, K. E. Jordan, V. Sachdeva, J. Sexton, H. Jamjoom, Z.-Y. Shae, G. Pencheva, R. Tavakoli, *et al.*, “Enabling high-

performance computing as a service," *Computer*, vol. 45, no. 10, pp. 72–80, 2012.

[9] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," in *International Conference on Learning Representations*, 2019.

[10] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar, "Gpgpu processing in cuda architecture," *Advanced Computing*, vol. 3, no. 1, p. 105, 2012.

[11] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *IEEE Transactions on Parallel and Distributed systems*, vol. 22, no. 1, pp. 78–90, 2010.

[12] E. Wu and Y. Liu, "Emerging technology about gpgpu," in *APCCAS 2008-2008 IEEE Asia Pacific Conference on Circuits and Systems*, pp. 618–622, IEEE, 2008.

[13] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W.-m. Hwu, "Performance analysis and tuning for general purpose graphics processing units (gpgpu)," *Synthesis Lectures on Computer Architecture*, vol. 7, no. 2, pp. 1–96, 2012.

[14] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-m. Hwu, "Gpu clusters for high-performance computing," in *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–8, IEEE, 2009.

[15] M. J. Harris, "Fast fluid dynamics simulation on the gpu.," *SIGGRAPH Courses*, vol. 220, no. 10.1145, pp. 1198555–1198790, 2005.

[16] R. Kelly, "Gpu computing for atmospheric modeling," *Computing in Science & Engineering*, vol. 12, no. 4, pp. 26–33, 2010.

[17] M. Cárcamo, P. E. Román, S. Casassus, V. Moral, and F. R. Rannou, "Multi-gpu maximum entropy image synthesis for radio astronomy," *Astronomy and computing*, vol. 22, pp. 16–27, 2018.

[18] V. Campmany, S. Silva, A. Espinosa, J. C. Moure, D. Vázquez, and A. M. López, "Gpu-based pedestrian detection for autonomous driving," *Procedia Computer Science*, vol. 80, pp. 2377–2381, 2016.

[19] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1110–1121, 2020.

[20] G. Li and G. Gopalakrishnan, "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.

[21] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "Grace: a low-overhead mechanism for detecting data races in gpu programs," in *ACM SIGPLAN Notices*, vol. 46, pp. 135–146, ACM, 2011.

[22] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "Gmrace: Detecting data races in gpu programs via a low-overhead scheme," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 104–115, 2014.

[23] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: a verifier for gpu kernels," in *ACM SIGPLAN Notices*, vol. 47, pp. 113–132, ACM, 2012.

[24] G. Srinisvasan Desikan, *Software Testing*. Pearson India, 2007.

[25] K. O. W. Group, "The opencl specification version 2.2," May 2017.

[26] NVIDIA, "Cuda zone," Sep 2017.

[27] T. Sorensen and A. F. Donaldson, "The hitchhiker's guide to cross-platform opencl application development," in *Proceedings of the 4th International Workshop on OpenCL*, p. 2, ACM, 2016.

[28] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," in *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 153–162, 1998.

[29] S. U. Farooq, S. Quadri, and N. Ahmad, "Software measurements and metrics: Role in effective software testing," *International Journal of Engineering Science and Technology*, vol. 3, no. 1, pp. 671–680, 2011.

[30] F. Del Frate, P. Garg, A. P. Mathur, and A. Pasquini, "On the correlation between code coverage and software reliability," in *Proceedings of Sixth International*

Symposium on Software Reliability Engineering. ISSRE'95, pp. 124–132, IEEE, 1995.

[31] A. S. Namin, J. Andrews, and D. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, pp. 351–360, IEEE, 2008.

[32] A. Inc., "clang: a c language family frontend for llvm," March 2018.

[33] G. Candea and P. Godefroid, "Automated software test generation: some challenges, solutions, and recent advances," in *Computing and Software Science*, pp. 505–531, Springer, 2019.

[34] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 1–49, 2015.

[35] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 416–419, 2011.

[36] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 201–211, IEEE, 2015.

[37] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[38] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[39] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[40] L. van den Haak, A. J. Wijs, M. G. van den Brand, and M. Huisman, "Formal methods for gpgpu programming: is the demand met?," in *Integrated Formal Methods*, Springer, 2020.

[41] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[42] A. Habermaier and A. Knapp, "On the correctness of the simt execution model of gpus," in *European Symposium on Programming*, pp. 316–335, Springer, 2012.

[43] R. Rost, "Overview of the khronos group," in *ACM SIGGRAPH 2006 Courses*, pp. 2–es, 2006.

[44] M. J. Harvey and G. De Fabritiis, "Swan: A tool for porting cuda programs to opencl," *Computer Physics Communications*, vol. 182, no. 4, pp. 1093–1099, 2011.

[45] G. Martinez, M. Gardner, and W.-c. Feng, "Cu2cl: A cuda-to-opencl translator for multi-and many-core architectures," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pp. 300–307, IEEE, 2011.

[46] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing opencl applications on fpgas," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 114–125, IEEE, 2016.

[47] M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[48] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 72–82, 2014.

[49] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[50] B. Korel, "Automated software test data generation," *IEEE Transactions on software engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[51] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[52] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[53] C. Miller, Z. N. Peterson, *et al.*, “Analysis of mutation and generation-based fuzzing,” *Independent Security Evaluators, Tech. Rep*, vol. 56, pp. 127–135, 2007.

[54] V. Ganesh, T. Leek, and M. Rinard, “Taint-based directed whitebox fuzzing,” in *Proceedings of the 31st International Conference on Software Engineering*, pp. 474–484, IEEE Computer Society, 2009.

[55] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[56] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, “Compiler fuzzing: How much does it matter?,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[57] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, “Compiler fuzzing through deep learning,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 95–105, 2018.

[58] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[59] A. Fromherz, K. S. Luckow, and C. S. Păsăreanu, “Symbolic arrays in symbolic pathfinder,” *ACM SIGSOFT Software Engineering Notes*, vol. 41, no. 6, pp. 1–5, 2017.

[60] D. Mahrenholz, O. Spinczyk, and W. Schroder-Preikschat, “Program instrumentation for debugging and monitoring with aspectc++,” in *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*, pp. 249–256, IEEE, 2002.

[61] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O’Connor, and S. W. Keckler, “Flexible software profiling of gpu architectures,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 185–197, IEEE, 2015.

[62] Z. Wang, A. Sanchez, and A. Herkersdorf, “Scisim: a software performance estimation framework using source code instrumentation,” in *Proceedings of the 7th International Workshop on Software and Performance*, pp. 33–42, 2008.

[63] J. K. Hollingsworth, B. P. Miller, and J. Cargille, “Dynamic program instrumentation for scalable performance tools,” in *Proceedings of IEEE Scalable High Performance Computing Conference*, pp. 841–850, IEEE, 1994.

[64] K. S. Templer and C. L. Jeffery, “A configurable automatic instrumentation tool for ansi c,” in *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No. 98EX239)*, pp. 249–258, IEEE, 1998.

[65] M. Chabot, K. Mazet, and L. Pierre, “Automatic and configurable instrumentation of c programs with temporal assertion checkers,” in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 208–217, IEEE, 2015.

[66] F. Horváth, T. Gergely, Á. Beszédes, D. Tengeri, G. Balogh, and T. Gyimóthy, “Code coverage differences of java bytecode and source code instrumentation tools,” *Software Quality Journal*, vol. 27, no. 1, pp. 79–123, 2019.

[67] H. Fu, Z. Wang, X. Chen, and X. Fan, “A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques,” *Software Quality Journal*, vol. 26, no. 3, pp. 855–889, 2018.

[68] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics,” in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pp. 329–339, 2008.

[69] C. Flanagan and S. N. Freund, “Type-based race detection for java,” in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 219–232, 2000.

[70] M. Abadi, C. Flanagan, and S. N. Freund, “Types for safe locking: Static race detection for java,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 2, pp. 207–255, 2006.

[71] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur, “Framework for testing multi-threaded java programs,” *Concurrency and Computation: Practice and Experience*, vol. 15, no. 3-5, pp. 485–499, 2003.

[72] M. Naik, A. Aiken, and J. Whaley, “Effective static race detection for java,” in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 308–319, 2006.

[73] M. Naik and A. Aiken, “Conditional must not aliasing for static race detection,” *ACM SIGPLAN Notices*, vol. 42, no. 1, pp. 327–338, 2007.

[74] M. Vaziri, F. Tip, and J. Dolby, “Associating synchronization constraints with data in an object-oriented language,” *ACM Sigplan Notices*, vol. 41, no. 1, pp. 334–345, 2006.

[75] J. W. Voung, R. Jhala, and S. Lerner, “Relay: static race detection on millions of lines of code,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 205–214, 2007.

[76] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multithreaded programs,” *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[77] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu, “Efficient data race detection for distributed memory parallel programs,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2011.

[78] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Concurrency: the Works of Leslie Lamport*, pp. 179–196, 2019.

[79] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, “Sound predictive race detection in polynomial time,” *ACM Sigplan Notices*, vol. 47, no. 1, pp. 387–400, 2012.

[80] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan, “Efficient and precise datarace detection for multithreaded object-oriented programs,” in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 258–269, 2002.

[81] Y. Yu, T. Rodeheffer, and W. Chen, “Racetrack: efficient detection of data race conditions via adaptive tracking,” in *Proceedings of the twentieth ACM symposium on Operating systems principles*, pp. 221–234, 2005.

[82] S. Qadeer and J. Rehof, “Context-bounded model checking of concurrent software,” in *International conference on tools and algorithms for the construction and analysis of systems*, pp. 93–107, Springer, 2005.

[83] A. Lal and T. Reps, “Reducing concurrent analysis under a context bound to sequential analysis,” *Formal Methods in System Design*, vol. 35, no. 1, pp. 73–97, 2009.

[84] S. K. Lahiri, S. Qadeer, and Z. Rakamarić, “Static and precise detection of concurrency errors in systems code using smt solvers,” in *International Conference on Computer Aided Verification*, pp. 509–524, Springer, 2009.

[85] I. Rabinovitz and O. Grumberg, “Bounded model checking of concurrent programs,” in *International Conference on Computer Aided Verification*, pp. 82–97, Springer, 2005.

[86] P. Pereira, H. Albuquerque, H. Marques, I. Silva, C. Carvalho, L. Cordeiro, V. Santos, and R. Ferreira, “Verifying cuda programs using smt-based context-bounded model checking,” in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pp. 1648–1653, 2016.

[87] F. R. Monteiro, E. H. d. S. Alves, I. S. Silva, H. I. Ismail, L. C. Cordeiro, and E. B. de Lima Filho, “Esbmc-gpu a context-bounded model checking tool to verify cuda programs,” *Science of Computer Programming*, vol. 152, pp. 63–69, 2018.

[88] L. H. Sena, I. V. Bessa, M. R. Gadelha, L. C. Cordeiro, and E. Mota, “Incremental bounded model checking of artificial neural networks in cuda,” in *2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC)*, pp. 1–8, IEEE, 2019.

[89] Y. Xing, B.-Y. Huang, A. Gupta, and S. Malik, “A formal instruction-level gpu model for scalable verification,” in *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–8, 2018.

[90] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper, *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, vol. 1032. Citeseer, 1996.

[91] D. Peled, "All from one, one for all: on model checking using representatives," in *International Conference on Computer Aided Verification*, pp. 409–423, Springer, 1993.

[92] A. Valmari, "Stubborn sets for reduced state space generation," in *International Conference on Application and Theory of Petri Nets*, pp. 491–515, Springer, 1989.

[93] C. Wang, Z. Yang, V. Kahlon, and A. Gupta, "Peephole partial order reduction," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 382–396, Springer, 2008.

[94] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," *ACM Sigplan Notices*, vol. 40, no. 1, pp. 110–121, 2005.

[95] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv, "Cartesian partial-order reduction," in *International SPIN Workshop on Model Checking of Software*, pp. 95–112, Springer, 2007.

[96] V. Kahlon, A. Gupta, and N. Sinha, "Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions," in *International conference on computer aided verification*, pp. 286–299, Springer, 2006.

[97] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 328–342, Springer, 2010.

[98] C. Wang, S. Kundu, M. Ganai, and A. Gupta, "Symbolic predictive analysis for concurrent programs," in *International Symposium on Formal Methods*, pp. 256–272, Springer, 2009.

[99] P. Li, X. Hu, D. Chen, J. Brock, H. Luo, E. Z. Zhang, and C. Ding, "Ld: Low-overhead gpu race detection without access monitoring," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 1, pp. 1–25, 2017.

[100] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner, "Verifying gpu kernels by test amplification," in *ACM SIGPLAN Notices*, vol. 47, pp. 383–394, ACM, 2012.

[101] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic testing of opencl code," in *Haifa Verification Conference*, pp. 203–218, Springer, 2011.

[102] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "Gklee: concolic verification and test generation for gpus," in *ACM SIGPLAN Notices*, vol. 47, pp. 215–224, ACM, 2012.

[103] P. Li, G. Li, and G. Gopalakrishnan, "Parametric flows: automated behavior equivalencing for symbolic analysis of races in cuda programs," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–10, IEEE, 2012.

[104] A. Holey, V. Mekkat, and A. Zhai, "Haccrg: Hardware-accelerated data race detection in gpus," in *2013 42nd International Conference on Parallel Processing*, pp. 60–69, IEEE, 2013.

[105] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011, p. 1, Citeseer, 2011.

[106] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*, pp. 364–387, Springer, 2005.

[107] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.

[108] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic crosschecking of floating-point and simd code," in *Proceedings of the sixth conference on Computer systems*, pp. 315–328, 2011.

[109] T. Cogumbreiro, J. Lange, D. L. Z. Rong, and H. Zicarelli, "Checking data-race freedom of gpu kernels, compositionally," in *International Conference on Computer Aided Verification*, pp. 403–426, Springer, 2021.

[110] A. Rajan and M. P. Heimdahl, *Coverage metrics for requirements-based testing*. University of Minnesota, 2009.

[111] G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. Heimdahl, "The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 25, 2016.

[112] Q. Zhu and A. Zaidman, "Massively parallel, highly efficient, but what about the test suite quality? applying mutation testing to gpu programs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 209–219, IEEE, 2020.

[113] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.," in *NDSS*, vol. 8, pp. 151–166, 2008.

[114] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM Sigplan Notices*, vol. 43, pp. 206–215, ACM, 2008.

[115] K. Mao, M. Harman, and Y. Jia, "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.

[116] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[117] S. Xiao and W.-c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, IEEE, 2010.

[118] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamarić, "Portable inter-workgroup barrier synchronisation for gpus," in *ACM SIGPLAN Notices*, vol. 51, pp. 39–58, ACM, 2016.

[119] X. Cai and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[120] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 57–68, ACM, 2009.

[121] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, “Testing concurrent programs to achieve high synchronization coverage,” in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 210–220, ACM, 2012.

[122] Z. Wang, D. Grewe, and M. F. O’boyle, “Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, pp. 1–26, 2014.

[123] Z. Jin and H. Finkel, “Optimizing an atomics-based reduction kernel on opencl fpga platform,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 532–539, IEEE, 2018.

[124] “Winograd-based convolution implementation in opencl,” 2017.

[125] “Monte carlo extreme for opencl (mcxcl).”

[126] “Cinematic particle effects with opencl.”

[127] “clsparse: a software library containing sparse functions written in opencl,” 2016.

[128] “Opencl interferometry library (liboi),” 2018.

[129] “Experiments on gaussian pyramid implemented using opencl.”

[130] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, “Many-core compiler fuzzing,” *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 65–76, 2015.

[131] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[132] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[133] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, “Scan primitives for gpu computing,” in *Graphics hardware*, vol. 2007, pp. 97–106, 2007.

[134] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Ieee, 2009.

[135] L. Pouchet and T. Yuki, "Polybench/c 4.1," *Retrieved May2015 from http://web.cse.ohio-state.edu/ pouchet/software/polybench*, 2015.

[136] M. Zalewski, "American fuzzy lop (afl) fuzzer," 2017.

[137] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138, ACM, 2018.

[138] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," in *NDSS*, vol. 16, pp. 1–16, 2016.

[139] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[140] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[141] N. Otterness and J. H. Anderson, "Amd gpus as an alternative to nvidia for supporting real-time workloads," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[142] NVIDIA, "cufft," Sep 2021.

[143] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[144] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[145] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[146] Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking tpu, gpu, and cpu platforms for deep learning," *arXiv preprint arXiv:1907.10701*, 2019.

[147] T. O. A. R. Boards, "Openmp," Jul 2019.

[148] O. Standard, "Openacc," Sep 2020.

[149] T. K. Group, "Sycl," Jul 2021.