On the Correctness of GPU Programs

Chao Peng
University of Edinburgh
Edinburgh, United Kingdom
chao.peng@ed.ac.uk

ABSTRACT

Testing is an important and challenging part of software development and its effectiveness depends on the quality of test cases. However, there exists no means of measuring quality of tests developed for GPU programs and as a result, no test case generation techniques for GPU programs aiming at high test effectiveness. Existing criteria for sequential and multithreaded CPU programs cannot be directly applied to GPU programs as GPU follows a completely different memory and execution model.

We surveyed existing work on GPU program verification and bug fixes of open source GPU programs. Based on our findings, we define barrier, branch and loop coverage criteria and propose a set of mutation operators to measure fault finding capabilities of test cases. CLTestCheck, a framework for measuring quality of tests developed for GPU programs by code coverage analysis, fault seeding and work-group schedule amplification has been developed and evaluated using industry standard benchmarks. Experiments show that the framework is able to automatically measure test effectiveness and reveal unusual behaviours. Our planned work includes data flow coverage adopted for GPU programs to probe the underlying cause of unusual kernel behaviours and a more comprehensive work-group scheduler. We also plan to design and develop an automatic test case generator aiming at generating high quality test suites for GPU programs.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

software testing, code coverage, fault finding, data race, GPU, OpenCL, test case generation

ACM Reference Format:
Chao Peng. 2019. On the Correctness of GPU Programs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3293882.3338989

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '19, July 15–19, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3338989

1 INTRODUCTION

1.1 Research Motivation

Graphics Processing Units (GPUs) have been widely used for general-purpose computations in a large variety of application domains [19] due to massive parallelism, energy efficiency and ease of programming using C-like programming models including CUDA [1] and OpenCL [9]. However, writing correct GPU programs (also interchangeably referred to as GPU kernels in the field of GPU programming) is challenging due to many reasons [11]– a program may spawn thousands to millions of threads which are clustered in work-groups, making the execution difficult to analyse; there is a greater risk of inter work-group data races due to the lack of a mechanism to synchronise threads from different work-groups; control flow divergence causes intra work-group synchronisation bugs as the behaviour of synchronisation functions called in different code blocks is undefined [3]; there is no explicit GPU thread and work-group scheduling scheme provided by either GPU vendors or programming standards for us to predict thread interleavings [21].

Existing techniques focus on static and dynamic code analysis to verify GPU programs. Some of them [7, 12, 13, 24, 25] suffer from the scalability issue due to the exponential growth of the number of thread interleavings. The 2-thread execution model proposed by GPUVerify is more feasible but may report false positives [3]. In addition, test suites usually reflect the specification of programs to ensure the intended program behaviour [23] but there is no existing work considering program execution with test suites.

Given these difficulties of writing correct GPU programs and the lack of test quality measurement, it becomes important to understand the extent to which a GPU program has been analysed and tested, and the code portions that may need further attention. Coverage metrics can help in monitoring the quality of testing, in creating tests for areas that have not been tested before, and with forming small yet comprehensive test suites [4]. The most commonly used coverage metrics are based on the structure of the program, such as statement coverage, branch coverage, or data-flow based coverage [23]. Empirical studies show that there is a strong correlation between test suites with high coverage and the defect-detection ability of these test suites [5, 15]. Some of these coverage criteria have been successfully implemented in tools that are used in industry. Many of the coverage metrics that exist over program code focus on sequential programs. Nevertheless, coverage criteria specifically targeting concurrent programs, like synchronisation coverage, also exist [10]. Such criteria help detect concurrency bugs that cannot be found with sequential program coverage metrics.

1.2 Goal

Our aim is to define coverage metrics in line with the characteristics of GPU architectures and provide a framework for measuring

test effectiveness and generating high quality tests. Our proposed contributions include:

(1) **Coverage metrics definitions.** Definitions of barrier, control flow and data flow coverage metrics specifically for GPU programs considering their memory and execution models.
(2) **Test effectiveness measurement.** A framework for measuring effectiveness of test suites in two aspects: code coverage analysis based on the proposed criteria, and fault finding capability measurement by seeding faults to GPU programs and reporting the number of faults uncovered by the test suite.
(3) **Kernel execution engine with schedule amplification.** A scheduler to amplify work-group schedules and check for unusual kernel behaviours including data races and deadlocks arisen during the execution of test inputs.
(4) **Automated test case generation.** Implementation of a test case generator aiming at high test quality and effectiveness.

1.3 Completed Work

We maintain a repository of bug fixes of open source GPU programs and existing research on GPU program verification and it acts as a guide for our work. We define barrier and control flow (branch and loop) coverage metrics for GPU programs and propose a set of mutation operators inspired by the bug repository. A framework, CLTestCheck, is developed for measuring code coverage achieved by test suites based on our coverage metrics, seeding faults to GPU programs and amplifying work-group schedules. Empirical evaluation using 82 publicly available industry level GPU kernels with associated test input workloads shows that the framework is able to detect data races as well as barrier divergence and report kernel code that requires further tests using the coverage measurement capabilities. Deadlocks are also exposed by the scheduler.

2 CLTESTCHECK: TEST EFFECTIVENESS MEASUREMENT FOR GPU KERNELS

In this section, we present our design, implementation and empirical evaluation of CLTestCheck, a framework for measuring test effectiveness for GPU kernels. Some of the work has been completed and published [16].

2.1 CLTestCheck Overview

CLTestCheck measures test effectiveness over GPU kernels written using the OpenCL programming model [9]. Adapting the framework to CUDA is not complicated as they share a similar abstraction of GPU architectures and will only involve handling syntactic differences between these two GPU programming models [1]. The framework has three main components as illustrated in Figure 1.

2.1.1 CLCov: Coverage measurement. The first component, CLCov, is for measuring code coverage achieved by test suites. The constructs we chose to cover were motivated by OpenCL bugs found in GitHub repositories and research papers on GPU testing. We define coverage metrics over synchronisation statements (barriers), loop boundaries and branches in OpenCL kernels. Barrier coverage checks if each barrier is hit by all threads from the same work-group, otherwise, barrier divergence occurs and its behaviour is undefined, which may lead to potential data races and deadlocks [3]. For control flows, we consider a branch or a loop is covered if any of the threads execute it.

Implementation. Building on the Clang/LLVM compiler infrastructure [2], CLCov takes the original OpenCL program as input and adds extra statements and data structures to existing barriers and control flows to record the execution. After the execution of the kernel with a test input, the data structures containing execution information is processed to report code coverage.

2.1.2 CLMT: Fault seeding. The second component provides the capability of creating mutations by seeding different types of faults relevant to GPU kernels. CLMT assess the effectiveness of test suites in uncovering the seeded faults.

Implementation. Based on Clang/LLVM as well, the fault seeder generates mutants from the original program and executes them with the test input to compute mutation score, as the fraction of mutants killed. We consider a mutant is killed if the output produced is different from the original output, or the execution of the mutant crashes.

2.1.3 CLSchedule: Schedule Amplification. The purpose of CLSchedule is conducting *schedule amplification* to check the execution of test inputs over several work-group schedules. Existing GPU architectures and simulators do not provide a means to control work-group schedules. The OpenCL specification provides no execution model for inter work-group interactions [21]. As a result, the ordering of work-groups when a kernel is launched is non-deterministic and there is, presently, no means for checking the effect of schedules on test execution. We provide this monitoring capability. For a test case T_i in test suite TS, instead of simply executing it once with an arbitrary schedule of work-groups, we execute it many times with a different work-group schedule in each execution. We build a simulator that can force work-groups in a kernel execution to execute in a certain order. This is done in an attempt to reveal test executions that produce different outputs for different work-group schedules which inevitably point to problems in inter work-group interactions.

Implementation. To achieve thread scheduling, CLSchedule adds extra code blocks to the original OpenCL program to control the schedule.CLSchedule first randomly selects a work-group to be executed first. When the kernel starts, the inserted code blocks allow threads from that work-group to proceed with executing the original kernel code while other threads fall into a loop and remain in waiting. After the selected group finishes execution, threads in all other work-groups quit the loop and are enabled to execute the kernel then. For each test input, CLSchedule is able to check its execution over several different work-group orders when assessing quality.

2.2 Evaluation

In our experiment, we evaluate the feasibility and effectiveness of the framework using 82 OpenCL kernels from 46 industry standard benchmarks and their associated test suites. We execute each kernel with the test suites 20 times for each measurement on the Intel CPU (i5-6500) and GPU (HD Graphics 530) using OpenCL SDK 2.0.

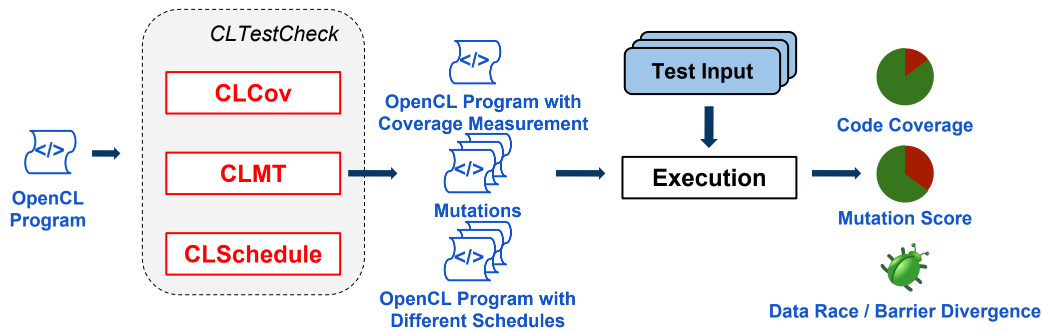

Figure 1: CLTestCheck

Subject Programs. We used the following benchmarks for our experiments, 1. Nine scientific benchmarks with 23 OpenCL kernels from Parboil benchmark suite [22], 2. scan benchmark [20], with 3 kernels, that computes parallel prefix sum, 3. Five applications containing 13 kernels from Rodinia benchmark [6] suite for heterogeneous computing, 4. 20 benchmarks from PolyBench [17] with 43 kernels spanning linear algebra, data mining and stencil computations.

Research Questions. The following research questions are instigated:

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

Q2. Fault Finding: *What is the mutation score of test suites associated with the subject programs?*

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?*

2.3 Experimental Results

Answering Q1: Coverage Achieved. CLTestCheck is able to automatically measure barrier, branch and loop boundary coverage for kernels in all benchmarks. Uncovered branches in the benchmarks typically have the condition of thread ID or data out of range and point to exception handling branches that were not exercised by the tests.

Barrier coverage for all but one of the kernels in the benchmarks is 100%. For synchronisations to work correctly, we should expect 100% barrier coverage. For the subject program that results in 87% barrier coverage, we uncovered a barrier divergence bug.

Answering Q2: Fault Finding. CLTestCheck is able to automatically generate different types of mutants for OpenCL kernels. Fault finding and total coverage are strongly correlated for these benchmarks by achieving 70.3% Pearson's correlation coefficient.

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 causes this uncertainty. 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.

Answering Q3: Deadlocks and Data Races. We observe deadlocks when applying the schedule amplifier to OpenCL kernels when the work-group ID selected to go first exceeds the number of available compute units of the GPU. It appears that the the GPU makes unstated assumptions that work-groups with lowest IDs are first allocated to compute units. However, they are asked by the scheduler to wait, while in the meantime the work-group selected by the scheduler is not assigned compute resources, which consequently causes the deadlock.

The schedule amplifier was able to reveal data races in two subject programs in successful schedules. These kernels produce inconsistent outputs with different schedules.

3 FUTURE WORK

In order to reveal the underlying reason of inconsistent kernel outputs, we plan to adapt data flow coverage to GPU models and improve the scheduler to have more control over the kernel execution. In addition, a high quality test case generator for GPU programs is also planed in our future work.

3.1 Efficient Kernel Execution Engine

3.1.1 Motivation. The evaluation of our framework uses test suites associated with subject programs and data races are reported if the test output is different from the expected output provided by the test suite. However, the presence of deadlocks as well as barrier divergence does not need to check the kernel output and data race can be proved if the output of one run is different from another. This means that detecting unusual kernel behaviours does not necessarily need test suites provided by humans.

3.1.2 Expected Contribution. We plan to implement an efficient execution engine for testing GPU kernels by fuzzing both test inputs and work-group schedules. Given a GPU kernel, the engine first generates a number of random test inputs and different work-group schedules. It then executes the program with these inputs and schedules. Barrier divergence can be detected by the barrier coverage metrics and data races can be reported if the output is inconsistent.

As the engine eliminates the need for test suites, it can also help evaluate other work by enlarging the scope of subject programs that can be collected from open source projects by a web crawler.

3.2 Improved Scheduler

3.2.1 Motivation. Although some unusual behaviours can be detected by the current scheduler, it suffers from the limitation that it can only select one work-group to be executed first while it has no control over the remaining work-groups. In addition, the ID of the selected work-group cannot exceed the number of available compute units.

3.2.2 Expected Contribution. We plan to improve the scheduler in two ways. (1) Based on the current scheduler, we can modify the code block inserted by the scheduler to accept more work-groups to be executed first. (2) We also plan to shuffle work-group schedule by mapping work-group IDs. In practice, inputs of GPU programs are stored in contiguous arrays. Threads determine which set of elements of the array to process by querying their thread and work-group IDs. Based on this fact, we can rewrite the query interface to disrupt the order. For example, if 4 work-groups are launched in a GPU with two compute units, the scheduler may generate a random array {3, 0, 1, 2}. Work-groups 0 - 3 will be instead allocated with IDs 3, 0, 1, 2 respectively. Consequently, compute units will process data which should be processed by work-groups 3 and 0 and this can be considered as work-groups 3 and 0 are launched first. The combination of these two proposed approaches is more comprehensive than our existing scheduler and will allow more control over the schedules.

3.3 Data Flow Coverage

3.3.1 Motivation. As discussed in Section 2, some kernels cannot produce consistent output with different schedules. To reveal the underlying cause of this problem, we need to monitor the internal state of variables during the execution. A promising technique to solve this problem is tracking and analysing data flow.

3.3.2 Expected Contribution. We plan to define data flow coverage for GPU programs based on traditional definition-usage metrics for CPU programs. The challenge is to define definition-usage pairs for shared variables which do not exist in sequential programs. In addition, the order of accesses to shared variables is affected by thread scheduling. To solve this problem, the data flow analysis should be combined with the scheduler to fix work-group schedules.

3.4 High Quality Test Case Generator

3.4.1 Motivation. Existing techniques for generating test cases for GPU programs discussed in Section 4 rely on symbolic execution and target data race checks. When they measure code coverage achieved by the generated tests, they convert the GPU program to its sequential version and apply coverage tools for sequential programs, which disregards the GPU programming model and memory hierarchy. This motivates us to improve the test case fuzzer to achieve high coverage according to the criteria we defined reflecting characteristics of GPUs.

3.4.2 Expected Contribution. To make our random test case generator developed for the execution engine smarter, search-based test case generation [14] for GPU programs is planned. This technique requires the maintenance of a problem-specific fitness function to optimise the search for good solutions in a reasonable time frame. In our case, the contribution would be the definition of the fitness function taking control flow and data flow coverage into consideration.

4 RELATED WORK

GPU Kernel Verification. Static analysis methods including PUG [12] and GPUVerify [3] are thwarted by the complexity of the sharing patterns. PUG is infeasible for a larger number of threads due to the exponential growth of the number of thread interleavings. GPUVerify aims at data races and barrier divergence by 2-threaded execution but may report false positives and has limited support for atomic operations. GKLEE [13] and KLEE-CL [7] check for data races for GPU programs by exploiting dynamic symbolic execution. The verification requires the user to specify the number of threads for the dynamic execution and both tools do not support custom synchronisation constructs. GRace [24] and GMRace [25] combine both static and dynamic execution but both of them do not support inter work-group data race detection. Test amplification was also applied to the verification of GPU programs by amplifying test inputs [11]. The main drawback is that they can only check the absence of data races rather than the presence.

Test Effectiveness Measurement. Measuring effectiveness of tests in terms of code coverage and fault finding is common for CPU programs [8, 18]. Support for GPU programs is scarce. To the best of our knowledge, there exists only one tool, GKLEE that measures code coverage for GPU kernels by converting them to sequential C programs. However, this method does not take features of GPU execution model into consideration.

5 CONCLUSION

The wide use of GPUs in the large variety of general computation domains emphasises the importance of the correctness of GPU programs. Existing work has made contributions to GPU kernel verification based on static and dynamic code analysis but they do not consider test quality measurement and program execution with test suites.

We have presented definitions of coverage metrics for GPU programs and the CLTestCheck framework for code coverage and fault finding capability measurement based on code instrumentation, and inter work-group data race detection by amplifying the execution across different work-group schedules. Our empirical evaluation using 82 OpenCL kernels shows that the framework can help developers review code blocks that require further testing and inter work-group synchronisation problems. In the future, we plan to add further data flow coverage metrics, develop an efficient kernel execution engine with the improved work-group scheduler to strengthen test adequacy measurement and an automated high quality test case generator specialised for GPU programs.

ACKNOWLEDGMENTS

This work is part of my PhD research supervised by Dr. Ajitha Rajan. Many thanks to my supervisor for her patient guidance and consistent encouragement and colleagues from our group for their kind help.

REFERENCES

[1] 2017. CUDA Zone. https://developer.nvidia.com/cuda-zone

[2] 2018. clang: a C language family frontend for LLVM. http://clang.llvm.org/

[3] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 113–132.

[4] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. 2005. Applications of synchronization coverage. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 206–212.

[5] Xia Cai and Michael R Lyu. 2005. The effect of code coverage on fault detection under different testing profiles. *ACM SIGSOFT Software Engineering Notes* 30, 4 (2005), 1–7.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 44–54.

[7] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. 2011. Symbolic testing of OpenCL code. In *Haifa Verification Conference*. Springer, 203–218.

[8] Gregory Gay, Ajitha Rajan, Matt Staats, Michael Whalen, and Mats PE Heimdahl. 2016. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 25.

[9] Khronos OpenCL Working Group. 2017. The OpenCL Specification Version 2.2.

[10] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. 2012. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 210–220.

[11] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU kernels by test amplification. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 383–394.

[12] Guodong Li and Ganesh Gopalakrishnan. 2010. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 187–196.

[13] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P Rajan. 2012. GKLEE: concolic verification and test generation for GPUs. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 215–224.

[14] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.

[15] Akbar Siami Namin and James H Andrews. 2009. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ACM, 57–68.

[16] Chao Peng and Ajitha Rajan. 2019. CLTestCheck: Measuring Test Effectiveness for GPU Kernels. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 315–331.

[17] LN Pouchet and T Yuki. 2015. PolyBench/C 4.1. *Retrieved May2015 from http://web.cse.ohio-state.edu/ pouchet/software/polybench* (2015).

[18] Ajitha Rajan and Mats PE Heimdahl. 2009. *Coverage metrics for requirements-based testing*. University of Minnesota.

[19] Ajitha Rajan, Subodh Sharma, Peter Schrammel, and Daniel Kroening. 2014. Accelerated test execution using gpus. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 97–102.

[20] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens. 2007. Scan primitives for GPU computing. In *Graphics hardware*, Vol. 2007. 97–106.

[21] Tyler Sorensen and Alastair F Donaldson. 2016. The Hitchhiker's Guide to Cross-Platform OpenCL Application Development. In *Proceedings of the 4th International Workshop on OpenCL*. ACM, 2.

[22] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

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

[24] Mai Zheng, Vignesh T Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: a low-overhead mechanism for detecting data races in GPU programs. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 135–146.

[25] Mai Zheng, Vignesh T Ravi, Feng Qin, and Gagan Agrawal. 2014. Gmrace: Detecting data races in gpu programs via a low-overhead scheme. *IEEE Transactions on Parallel and Distributed Systems* 25, 1 (2014), 104–115.