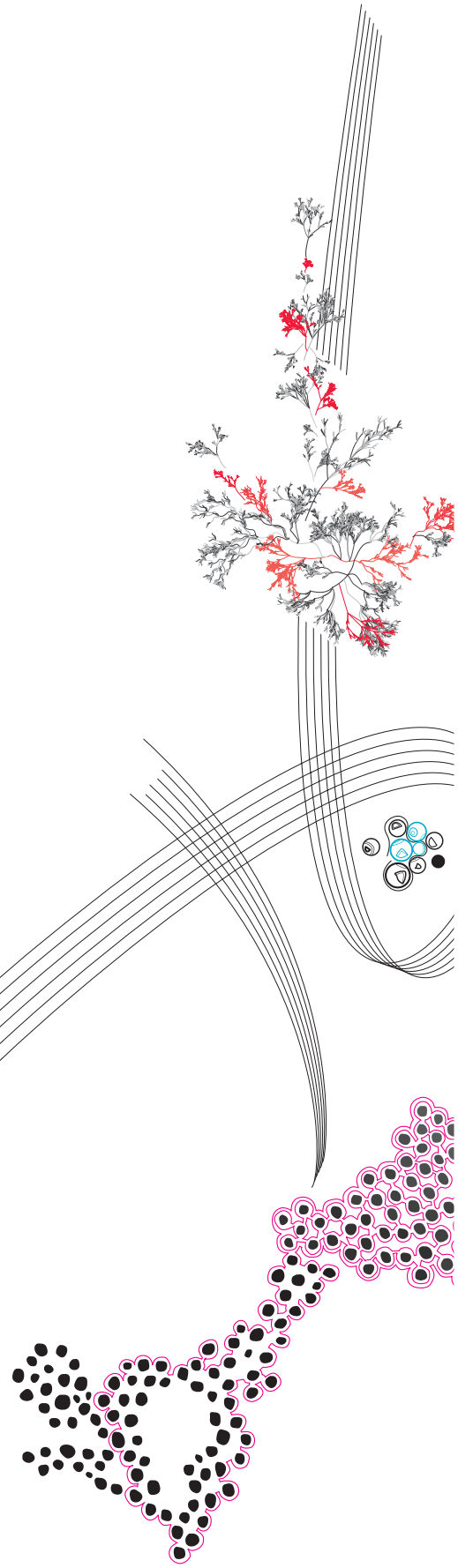

MSc Computer Science
Final Project

Testing and Mutation Testing for GPU Kernels

Yujie Liu

Supervisor: Marieke Huisman, Ben van Werkhoven,
Marcus Gerhold

November, 2023

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

The increasing GPU performance and maturing computational platform make it possible to handle general-purpose computing jobs traditionally computed by the CPU. Also, just like what we did in the CPU program, we use testing to verify the correctness of the GPU program. However, the quality of the tests may remain unknown, which inspires us to use mutation testing, a fault-based testing technique, to measure the effectiveness of a test. In this research, we conduct a feasibility study on applying testing and mutation testing to GPU programming, adapting existing research methodology, hypotheses, experiments, and optimization methods to the specific use case of the GPU kernel, and exploring the GPU-native mutation testing theories and techniques. A mutation testing tool is developed to validate and evaluate the theoretical analysis above, as a module of a kernel-tuning tool to offer an out-of-the-box mutation testing workflow for kernel developers. Our results reveal that testing can benefit the quality of kernels, and more test cases will lead to a higher possibility of detecting faults. We also discovered that mutation testing is able to quantify the testing quality for GPU kernels.

Keywords: Testing, Mutation testing, GPU programming, CUDA

Chapter 1

Introduction

In the last decade, GPUs have gained popularity in general-purpose computing (GPGPU) and high-performance computing (HPC) due to their significant performance increase, taking on tasks typically handled by CPUs as Moore's Law has stagnated. Compared to CPUs, GPUs have thousands of small, simple cores that can all work together to process data simultaneously. This makes GPUs particularly well-suited for tasks that involve a lot of data parallelism, such as machine learning, artificial intelligence, and computer vision. Some GPU computational platforms, such as CUDA [7] from NVIDIA and OpenCL [34] from Khronos Group, support the development of highly parallel GPU programs. These programs designed for parallel execution on GPUs are specifically called *kernels*.

However, new application scenarios also bring new challenges. As GPU-powered computing is widely used in all kinds of high-performance computing areas, kernel developers are aware that developing a correct and efficient kernel is still a challenging task.t This is because GPUs have a different computing architecture and programming model compared to CPUs.

The main differences between these two computing models are thread management and memory model. In CPU programming, threads are scheduled by the operating system, which assigns a time slice to each thread and switches between them as needed. However, in GPU programming, threads are managed by the GPU itself, and programmers need to seriously arrange the threads to operate data at the right memory address without causing deadlock and data race. Worse still, GPUs have significantly more threads compared to CPUs, which also makes this work much more challenging.

As for the memory model, GPU has separate device memory from the host and requires the kernel programmer to manage it manually. For example, moving data from the host memory to the device memory and retrieving results from the device back to the host memory. Much like CPUs, GPUs have multiple levels of cache/memory with different speeds and capacities. However, it requires the kernel developer to manage and optimize them carefully to maximize the kernel performance.

Therefore, ensuring the correctness and high performance of kernel code is a crucial aspect of kernel development. Concerning kernel performance optimization, tools such as Kernel Tuner [38] have been developed to measure kernel performance and tune the thread and cache arrangement.

However, testing, the most intuitive and popular way to assure the quality of a software, is not conducted extensively and formally on the GPU kernels. This is because in CPU programming, most of the languages have a sophisticated unit testing framework or a built-in unit testing module. However, this is not the case in GPUs. For example, NVIDIA provides only a very limited testing mechanism for its popular GPU computing platform,

CUDA, which requires kernel testers to refactor the code into a CPU-compatible one and test under CPUs [24]. However, this refactoring is not always possible because the code may use the exclusive features of GPUs, and the testers have to write their own code to manually execute the test cases on GPUs and compare the result for each test cases, handling all of the above threading and memory challenges.

This testing workflow is far from convenient for kernel testers. Therefore, we would like to introduce testing into the kernel development workflow in an effective way to ensure the correctness of kernel development. This goal can be achieved by solving the following two problems:

- **P1**: How to apply massive test cases to test the GPU kernels in a simple way?
- **P2**: On top of **P1**, how to ensure testing quality and effectiveness?

To address the first problem **P1**, we conduct our study on top of a previous research achievement called Kernel Tuner. Kernel Tuner offers a correctness verification mechanism that allows the kernel developers to focus on designing the kernel and verify the kernel output, regardless of thread and memory management of the kernel [38, 39]. However, this mechanism is designed for tuning the kernel with one input and expected output at a time. Therefore, we design a testing interface for massive test cases execution based on this. In this research, we design a data structure for test cases, implement a testing module to execute all the test cases with optimized kernel execution parameters in one click, and use two coverage criteria to estimate the quality of testing.

Then, to address the problem **P2**, we introduce mutation testing, a popular fault-based testing technique to measure the quality of the testing and the effectiveness of the proposed coverage criteria. Mutation testing can inject defined faults into the original code and see if the test suites can detect them. We conduct a series of feasibility studies on applying mutation testing to the GPU kernel. These studies including the exploration of mutation operators, verification of two fundamental hypotheses of mutation testing, the Coupling Effect and the Competent Programmer Hypothesis, and examination of the correlation between mutation testing and testing.

A mutation testing module is developed to support our research and for kernel developers and testers' usage. We use our this tool to conduct a set of experiments regarding design and execution of test cases, exploration and evaluation of our presented mutation operators, and analysis of the effectiveness of mutation testing.

Our experiment results reveal that testing is useful in the correctness assurance of GPU programming and our testing method is effective to apply massive test cases on GPU kernels. The mutation testing can be used to assure the quality of the testing on GPUs because our presented mutation operators can represent the real faults that a programmer can made.

In this research, we make contributions on applying testing and mutation testing to GPU programming and revealing the relationship between the mutation testing and testing on GPU kernels. We also proposed two more GPU-specific mutation operators for mutation testing, and develop an open-source testing and mutation testing tool for GPU kernel development.

This report is structured as follows: Chapter 2 firstly introduces the background of this research regarding testing, mutation testing, and GPU programming. Then, in Chapter 3, we list the related studies on applying testing and mutation testing on GPU programming and compare the differences from this research. We propose our goals and questions in Chapter 4, along with the design of our mutation testing tool and benchmark projects.

We introduce the details about applying testing to GPU kernels in Chapter 5, which is also a precondition of applying mutation testing. In Chapter 6, we explain a serious of experiments to apply mutation testing to GPU kernels in detail. We lists our results in Chapter 7, and discuss our proposed research questions in Chapter 8. We draw our conclusions and explain future work in Section 9.

Chapter 2

Background

As this study concentrates on incorporating testing and mutation testing into GPU programming, which includes three different research fields, this background chapter will include three sections as listed below. Each section describe a whole story of one field, with subsection titles added for indexing and speed reading.

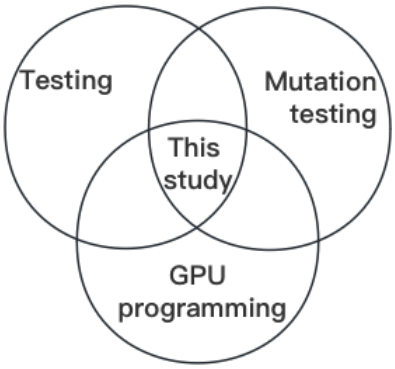

FIGURE 2.1: Research fields of this study and their relationships

- **Testing and coverage (Section 2.1)**: We briefly recall the concepts of testing, unit testing and performance testing in the context of GPU programming, and some code coverage criteria and corresponding measurements in the context of multi-threaded programming.
- **Mutation testing and related techniques (Section 2.2)**: We first introduce the basic of mutation testing and the relationship between mutation testing and testing. Then, we proceed with some core concepts of mutation testing and the processes of applying mutation testing.
- **GPU programming and kernels (Section 2.3)**: We introduce the basic concept of GPU programming with a running demo kernel, including the thread and memory management. Then, the Kernel Tuner is introduced to see how it addresses the thread and memory optimization problems. After that, we explain two typical fault models regarding thread and memory management: the data race and barrier divergence.

2.1 Testing

Software testing is an essential process in software development that involves validation and verification of the functionality and non-functionality requirements of software applications. The goal of testing is to identify and locate defects, vulnerabilities, and errors in the application, with the aim of preventing potential problems or risks that may arise from using it. The followings introduce some testing techniques involved, and their main differences when applying to GPU programming compared to CPU one.

2.1.1 Unit Testing

Unit testing is the most popular software testing technique where individual code components or units are tested in isolation to ensure that they have met the functional requirements. A "unit" typically refers to the smallest testable part of the software. In the context of GPU programming, it typically refer to a single kernel that can be launched by the GPU, which may contains several functions calls in a chain [22, 40] (details shown in 2.3). The primary goal of unit testing is to verify that each unit of code performs its intended function correctly to catch bugs and improve code quality early in the development cycle.

2.1.2 Performance Testing

Performance testing is a type of non-functional testing techniques that focuses on evaluating the speed and responsiveness of a software under different conditions and workloads. It is significant in the context of GPU programming because GPU programming is designed for speeding up the general-purpose computing on GPUs. For example, NVIDIA offers a set of best-practice metrics to measure the performance of a kernel [17], which allows the kernel developer to inspect the performance of a kernel. However, there are few studies concerning using these metrics to design a test suite to ensure a performance baseline of a kernel.

2.1.3 Coverage Criteria

As software systems grow in complexity, the efficacy of testing methodologies becomes increasingly significant. Therefore, coverage criteria come into play to measure the degree of software testing that has been carried out. These criteria are a range of metrics to provide a quantitative measure to qualify the comprehensiveness of the test objective [11]. Two main criteria involved in this research, statement coverage and branch coverage, are introduced below.

- Statement Coverage: This criterion measures the proportion of executable statements in the source code that have been executed during testing, providing a basic insight into the testing comprehensiveness of code [11].
- Branch Coverage: This criterion evaluates the coverage of branching constructs, such as condition statements, loop statements, and switch cases. It ensures that both true and false branches, or all cases in switch statement, are tested [11].

However, when introducing above criteria a multi-threaded situation, things are starting to get complicated. In the sequential computing model, the code will be executed from top to bottom one by one, so it is intuitive to understand which statements/branches of code

are covered and which are not. However, most of modern computing hardware support more than one parallel threads. In this situation, we need to consider the number of threads that visit the statements/branches of code. This is because some threads may cover some statements/branches while no threads visit the others, or a more complicated case, the threads get diverged when visiting some statements. For example, some threads may visit the *then* branch of a conditional statement while the other threads go to the *else* branch. This is called *thread divergence* and may cause critical concurrency problems when it goes wrong [3].

What's more, since coverage is a way of measuring the level of the test effort, we do hope that the coverage will be correlated with the possibility of revealing faults. However, this relationship between coverage and fault revelation is naturally falsifiable [5], which indicates that our intuition that higher test coverage will detect more errors is unreliable. Therefore, to quantify the quality and effectiveness of testing, and to evaluate the fault detection ability of test coverage, mutation testing comes into play.

2.2 Mutation testing

Mutation testing [19, 30] is a software testing technique that involves modifying the source code of a software application to create variations of the code, namely mutants. The mutants are created by changing the code in specific ways, such as changing arithmetic operators or deleting statements, to simulate the real faults that a programmer can make. These mutants are then tested using the target test suite to determine whether this test suite can detect the intentionally injected faults. If a test case in the target test suite fails when executed on a mutant, we consider the test case to have found the defect and 'killed' the mutant. However, if a mutant passes all the test cases, which means none of the test cases can figure out this specific problem, we consider the mutant as 'survived'. By measuring the ability to detect mutants, we can analyze the overall quality of the test suite.

2.2.1 Relationship between Mutation Testing and Testing

Although both mutation testing and software testing use the term 'testing', they appear to be distinct concepts. Software testing uses a designed test suite to evaluate the quality of software. However, mutation testing is a "bug simulation" process to measure the quality of software testing itself. Therefore, mutation testing is typically used in combination with other software testing techniques, such as unit testing and code coverage analysis mentioned above to improve the effectiveness of software testing.

Chekam et al. [5] conduct a study on the relationship between mutation testing and coverage criteria. They find that mutation testing has higher fault revelation than statement coverage and branch coverage. However, their finding also reveals that there exists a threshold level of coverage, and the testers need to achieve it before they can actually benefit from applying mutation testing or further increasing coverage.

This threshold is easy to understand because a fragment of uncovered code probably implies that the mutation in this fragment will also not be covered by the same test suite. In this case, all the mutants within this code fragment are not included in any execution path, let alone influence the final result.

In conclusions, mutation testing is a powerful complement to testing because it forms a bridge between faults and test effectiveness of a test suite: failing to detect certain types of mutants implies failing to reveal certain types of faults [2, 5]. This principle of mutation

is grounded in an assumption that the mutant is capable of representing a real fault that may happen in the production code. Two hypotheses [9], the *Coupling Effect* and the *Competent Programmer Hypothesis*, ensure this capability, which are explained below.

2.2.2 Coupling Effect and Competent Programmer Hypothesis

The Coupling Effect is one of the basic hypotheses in mutation testing. It assumes that if a test case can distinguish all programs, which differ from a correct one by only simple errors, it is so sensitive that it also implicitly distinguishes more complex errors [9]. This is an empirical hypothesis, and there is no hope of theoretical "proof" of it. Therefore, over the decades, a set of empirical studies were conducted on this hypothesis.

Offutt [27] extends the definition of the hypothesis to the Mutation Coupling Effect, which assumes that a simple error, which can be fixed by making a single change, can be represented by a *1-order mutant*. One step further, Offutt assumes that the complex error, which can only be fixed by multiple changes, can also be represented by a *higher-order mutant*. The higher-order mutant here refers to a type of mutant that has more than one mutation in the code, which can be interpreted as a composite of multiple 1-order mutants [27]. Based on this, Offutt made an assumption that the higher-order mutants are coupled to 1-order mutants in such a way that a test suite that detects the 1-order mutants in a program can still detect the corresponding composited higher-order mutants.

To validate the correctness of the extended hypothesis, Offutt conducts an experiment on the connected 2-order mutant, which is composed of a pair of mutants on the same execution path, using a first-order mutation-adequate test suite to verify the generated 2-order mutants, and analyzes the alive ones. All the alive mutants after mutation testing are classified into *strong* uncoupled and *weakly* uncoupled. A weakly coupled mutant means it just happened to be missed by this test suite, for example, a former mutation changes the execution flow, making the latter one uncovered. However, a strong uncoupled mutant means there is some characteristic that makes it unable to be killed, which indicates that the Mutation Coupling Effect is defective, and mutation testing fail to represent this type of fault.

Mutation Coupling Effect Hypothesis assumes that faults can be represented by the mutation operators. However, it does not explain whether these faults can be actually made by a real programmer. That's why the Competent Programmer Hypothesis is proposed.

The Competent Programmer Hypothesis (CPH) is proposed by DeMillo et al. [9], which states that most programmers are competent enough to create correct or almost correct source code. One step further, if we assume the Mutation Coupling Effect is true, we can infer that if a test suite cannot distinguish the original program and the mutants, then it is not able to distinguish correct and faulty code either [27].

Gopinath et al. [13] conduct a large-scale study on over 4000 open-source projects on GitHub, trace the issues proposed on the GitHub Issues of the repository and corresponding linked bug-fixes, and quantify the syntactic differences before and after the fix. They conclude that the difference is so significant that "the Competent Programmer Hypothesis, at least from a syntactical perspective, may not be applicable."

Stein et al. [33] use a reversed way to validate the CPH. They use 5 Java projects with real bugs and bug patches and compare the buggy code and the correct one. They conduct an experiment to find a chain of mutation operators that can transform the correct code to the buggy version, which means that a higher-order mutant does exist to reproduce the bug. The result reveals that the CPH seems to be true. However, the common mutation operators are not representative enough for real-world bugs.

2.2.3 Equivalent Mutant

There is a situation that even though the test suite is well designed, some mutants can still survive. These mutants which can never be killed because they always produce the same output as the original program, no matter what the input is, are called *equivalent mutants*.

Equivalent mutants refer to a type of mutants that can never fail any test cases because the program's behavior is always preserved. Accordingly, the *non-equivalent mutants* refer to the ones that can distinguish themselves from the original code as long as they receive a suitable input.

Because of the behavior-preserving nature, equivalent mutants are valuable to be analyzed as they imply that the original code is not sensible enough to these types of change represented by these mutants. By changing the behaviour of the original code, we may be able to remove some equivalent mutants.

Although it's not that easy to distinguish the equivalent mutant from the survived mutants and it takes a lot of human labour to find them out, spending time on it seems still worthy, especially when we are going to calculate the mutation score.

2.2.4 Mutation Score

After figuring out all the non-equivalent mutants, we can use them to quantify the result of mutation testing, which can then be used as one of the metrics to evaluate the quality of testing. Here the *mutation score* is introduced. This score is calculated on the number of non-equivalent mutants detected by the test suite, i.e., the ratio of killed mutants to the total number of non-equivalent mutants. The higher the mutation score, the more effective the test suite is.

However, equivalent mutants will still take plenty of time and resource when we really execute them. Therefore, figuring out all the non-equivalent mutants, or in other words, eliminating all the equivalent mutants before actually executing them, becomes a critical process in mutation testing. And one of the most important ways to reduce or prevent equivalent mutants is by using well-defined *mutation operators*.

2.2.5 Mutation Operators

To guide the generation of mutants, mutation operators, or mutators are used. The mutation operators are defined rules to insert specific faults into the original program. They are significant in mutation testing since the quality of operators will highly influence the quality of the mutants and the testing effectiveness, and a poor operator may derive a number of equivalent and/or trivial mutants.

2.3 GPU Programming and Kernels

In this section, we are going to introduce GPU programming with a running example written in CUDA, and explain the thread management and memory model of the GPU device. The example kernel adds two vectors in parallel and stores the results in a third vector.

```
1  __global__ void add(int n,float *x,float *y,float *z){
2     int i = blockIdx.x * block_size_x + threadIdx.x;
3     if (i < n) {
4       z[i] = x[i] + y[i];
```

```
5     }
6  }
7
8  int main() {
9    int N = 1<<20;
10   float *x, *y, *z;
11   float *dev_x, *dev_y, *dev_z;
     ... //Declear and allocate host and device memory
12   cudaMemcpy(dev_x, x, N, cudaMemcpyHostToDevice);
13   cudaMemcpy(dev_y, y, N, cudaMemcpyHostToDevice);
14   add<<<4096, 256>>>(N, dev_x, dev_y, dev_z);
15   cudaMemcpy(z, dev_z, N, cudaMemcpyDeviceToHost);
     ... //Release host and device memory
16   return 0;
17 }
```

There are two functions in this running example. The function starts with keyword $\_\_global\_\_$ is called global function, which can be executed on GPU and called by a CPU function. Therefore, the global function also serves as the entry point of a kernel. There are two other kinds of functions, device function starting with $\_\_device\_\_$ and host function starting with $\_\_host\_\_$, which can only be called and executed on the GPU or CPU respectively. The functions without any execution space specifier, like the second function in above example, are also recognized as the host functions [24].

2.3.1 Thread and Memory Management

As we mentioned before, threads are managed by the GPU itself in GPU programming. These hardware threads are typically organized into groups called *warps* (NVIDIA) or *wavefronts* (AMD). This is because GPUs use the Single Instruction, Multiple Thread (SIMT) architecture, where a single instruction is executed by multiple hardware threads simultaneously, with each thread in one thread group running the same code but operating on different data. For example, in order to add up two vectors, we often use a loop statement to iterate them in CPU programming. However, in GPU programming, the more effective way is to unroll the loop statement by assigning different software threads with different vector indices for addition. As shown in line 2 of the running example, we give each thread an unique calculated id, and this id also works as the index of the two vector to guarantee the integration.

However, a kernel may involve a large number of software threads, but the hardware threads is always limited. That is why NVIDIA introduces thread block and grid in CUDA to abstract the hardware threads for computing. At the abstract level, just like its name, one thread block contains up to 1024 software threads, which at the hardware level represents a composition of one or multiple warp(s). Multiple thread blocks are combined into a grid, and the included blocks in one grid must have the same number of threads. Therefore, under this computing model, the number of threads in a thread block is limited, but grids can be used for kernels that require a large number of thread blocks.

In our running example, we call the kernel function in line 14 after some initialization jobs. The statement $add<<<4096, 256>>>$ tells the CUDA runtime that the kernel *add* will be executed by 4096 thread blocks, and each thread block includes 256 threads, which means it will start $4096 * 256 = 2^{20} = 1,048,576$ parallel software threads in total. This is

the same as the size of the vectors, which indicates all the threads will be executed once and only once for one vector index.

Now we know that one software thread will only operate one vector index. The following job is to make sure this one-to-one relationship is unique. In line 2, *threadIdx.x* uniquely identifies each thread within its thread block, and *blockIdx.x* does so for the thread block within a grid. By combining them with $block\_size\_x$, we can uniquely identify the threads in block 0 as $thread\ (0)$ to $thread\ (block\_size\_x - 1)$, and in block 1 as $thread\ (block\_size\_x)$ to $thread\ (2 * block\_size\_x - 1)$ ..., thus extending the uniqueness across the entire grid.

Before and after launching the kernel, the kernel developer needs to explicitly manage different types of memory, including the shared/local memory and global memory of the GPU device, and the memory of the host. In the running example, the developer needs to manually initialize the host memory for the variables *x, y, z* with *malloc* statements and the device memory for dev_x, dev_y, dev_z with *cudaMalloc* statements, which are omitted in the example due to space limitation. After initialization, the *cudaMemcpy* statement is called to copy input from the host memory to the device memory in lines 12-13, and copy the result back to the host in line 15 after kernel execution. After that, both the host and device memory need to be explicitly released.

2.3.2 Kernel Tuner

To address the above resource management and performance optimization problems in GPU programming, Kernel Tuner is presented [38]. Kernel Tuner is a Python library that facilitates the optimization of compute kernels for parallel computing devices. It provides an interface to automatically tune kernel parameters to achieve the best performance for a specific hardware platform. It speeds up the kernel development processes in the following three critical aspects:

- Kernel execution: Kernel Tuner supports all CUDA, OpenCL, and C kernels with a set of Python-based backends to offer a comprehensive solution for kernel execution, including kernel code compilation, initialization and cleanup of runtime context, bi-directional host-device memory transformations, and launching kernels.

- Kernel tuning: The variety of thread block divisions and other code optimization parameters, like tiling or unrolling factors, results in different kernel execution performances. Instead of manually tweaking kernel parameters and testing performance, Kernel Tuner includes a series of built-in search optimization algorithms to automatically tune the GPU kernels to benchmark the best configurations for kernel execution.

- Result verification: Kernel Tuner allows the users to offer a reference answer for kernel tuning and it offers a verification mechanism to automatically compare the kernel output with the reference answer.

2.3.3 Data Race and Barrier Divergence

It is always up to the kernel developer to manage the thread and memory properly in GPU programming, and insufficient understanding of this computing model will lead to problems like data races and barrier divergence. A data race is a situation that two or more threads try to access a global or shared memory at the same time, and at least one thread among them tries to do the write operation. This can lead to nondeterministic behavior of the execution and compute invalid results.

There are two kinds of data races in the GPU kernel, the *inter-group data race* and the *intra-group data race* [3]. An inter-group data race represents that these threads are from at least two blocks (CUDA terminology)/groups (OpenCL terminology), and at least two threads from different blocks/threads try to read and write to the same location in the global memory at the same time. An intra-group data race occurs only among the threads in the same group/block, and the conflicted operation can happen in both global or shared memory.

To prevent data races in the GPU kernel, the kernel developer can use fence function and atomic operations. A fence function can make sure that all the write operation to global or shared memory made by the caller thread are visible to other threads before the fence. It can be used in combination with atomic operations, where within one atomic operation, the value in address can be read, modified, and written back to the same memory without the interference of other threads. Both the CUDA and OpenCL provide a set of out-of-the-box fence function and atomic operations, like *__threadfence()*, *atomicInc(_)* for CUDA and *mem_fence()*, *atomic_inc(_)* for OpenCL.

Unlike the fence function which only ensure the order of memory operation, the *barrier statement* (i.e., *__syncthreads() in CUDA and barrier(_) in OpenCL*) is a strict synchronization method between threads. The barrier is the place where all the threads in the same block/group have to wait until every thread has reached the barrier. This mechanism forces all the threads to synchronize the shared and/or the global memory before executing the barrier.

However, things can become buggy if the barrier statement is misused. *Barrier divergence* is the subtle situation that the threads in the same group/block get diverged and go to the different branches of a conditional statement, where each branch has a barrier statement, forcing all of the threads to do the impossible synchronization. This is ill-designed and will lead to nondeterministic behavior of the kernel, which may vary with the kernel programming languages and hardware architectures [3].

Chapter 3

Related Work

With some basic background information introduced above, here we would like to discuss some related work regarding the application of testing and mutation testing to GPU kernels. We also summarize the differences or show our improvements between their studies and ours.

3.1 Testing on GPU Kernels

3.1.1 Tools for Kernel Verification

Although the GPU programming platforms do not offer a mechanism for applying testing to GPUs, researchers in this area have proposed some tools to verify the correctness of the GPU kernels.

Li et al. [22] develop a testing tool called GKLEE to analyze the correctness and performance problems for CUDA kernels. This tool is able to identify various bugs such as data races, barrier divergence. It also allow the tester to do a concolic analysis to figure out some performance-related problems like low-efficient memory access and bank conflict. Based on this tool, Sun et al. [35] develops their tool called DSGEN, which improves the data race detection rate by supporting dynamic data structure like linked lists and trees.

However, both these tool above only support CUDA kernels. Betts et al. [3] propose a tool called GPUVerify, which supports both CUDA and OpenCL computing platforms. It can detect the potential data races and barrier divergence problems. Compared to CKLEE, it does not provide performance measurement functionality, but it is more intuitive to use, which is particularly useful for early or mid-stage kernel development [29].

3.1.2 Coverage Measurement

In the context of GPUs, measuring the code coverage is even harder because a kernel can spawn millions of threads. This leads to complex coverage patterns which are hard to reason about, but it is always up to the kernel developer to make sure the concurrently executing threads do not conflict [21]. This creates the need of coverage information to inspect the flows of threads for each test case. However, despite the increasing importance and need for it, research on coverage metrics or measurement tools in GPU programming is still vastly unexplored.

Firstly, there is no official support for coverage measurement from some GPU programming platforms. For example, the CUDA compiler developed by NVIDIA, the NVCC, does not offer a mechanism to measure the code coverage of the CUDA kernel [37, 20]. Tabani et al. [37] meet the same problem when measuring the code coverage for a CUDA project.

They bypass this problem by using a simple tool called *cuda4cpu* [4], running the device code on the CPU, although they admit that this might lead to some unforeseen problems. Several similar tools can be found, like *GPU Ocelot* [15]. However, whether these tools can be compatible with coverage measurement tools still remains unknown. Another option is warping the device function with the global function, and removing *static* and *inline* keywords if needed to make sure that these functions can also be accessed by the host [40]. But this option also has some drawbacks because some operations are incompatible with the CPU, like the synchronization call, and it is not fine-grain enough to measure the coverage.

There is also a commercial coverage measurement tool named *VectorCAST/QA* claiming that it supports collecting the statement coverage and branch coverage of CUDA kernels, but from the live demo of this tool, only the coverage on the host is displayed [12].

The most promising research is from Li et al. [22]. They partially address the coverage problem by measuring the statement coverage in two different ways in their presented kernel verification tool GKLEE: the statements that are covered by all threads at least once, or by some threads at least once, which helped them to identify whether the states/paths that are covered by test cases or not.

However, all these research mainly focuses on using the coverage to make sure that the mutants are covered by test cases or not, without discussing whether these coverage criteria can be a way to quantify the testing quality and comprehensiveness. Therefore, in this research, we would like to go one step further to address this insufficient point.

Compared to CUDA, OpenCL has much more options regarding this problem. There are some device code coverage measurement tools for OpenCL, like the *clcov* [31] used by Peng et al. [32] and Li et al. [23]. The problem can also be bypassed since the CPU manufacturers like Intel offers the CPU-only runtime for the OpenCL [18], although a study reveals that the CPUs and GPUs may behave differently regarding the barrier divergence problem [3].

Compared to their studies, our research focuses on a different aspect. It requires applying testing to GPU kernels in an efficient way and further supports the following mutation testing process. Therefore, our research is based on an existing tool called Kernel Tuner [38]. It can tune the GPU kernels to find the optimized parameters for kernel executions, which is best suitable for optimizing the testing process.

3.2 Mutation Testing on GPU Kernels

Although both mutation testing and GPGPU are popular topics nowadays, only a few studies have been conducted in this cross-topic domain, i.e., the application of mutation testing to the GPU kernel.

The most relevant research is from Zhu et al. [40]. They conduct an empirical study on introducing mutation testing to the CUDA kernel. They propose nine GPU-specific mutation operators and develop a Python-script mutation testing tool to perform the mutation testing on six kernel projects from the CUDA SDK. They draw a conclusion that mutation testing can benefit GPU programming in simple test case writing and bug detection.

Peng et al. [32] also propose a tool that can inject faults into the OpenCL kernel and measure the fault detection capability of the existing test suite. All the mutation operators they used are from the CPU programming. They conclude that mutation testing is useful in this area, and find that some arithmetic operator and relational operator mutations are hard to kill, suggesting the tester design more specific test cases for them.

However, both of these studies mainly focus on developing a mutation testing tool for corresponding GPU programming language and use a demo input-output to apply the mutation testing to GPU kernels. This results in various shortcomings in their research:

- The test cases involved in these studies are not sufficient. Mutation testing is a way to measure the quality of a test suite. However, they did not clearly answer how to design the test suite and effectively apply testing and mutation testing to GPUs with this test suite.
- They did not adequately express why mutation testing can be applied to GPU kernels in the first place. The lack of analysis of the principles behind the application of mutation testing makes their experiments less convincing.

To address the insufficient points of these studies, we propose our research goals and questions.

Chapter 4

Approach

The purpose of this study is to rigorously investigate the application of testing and mutation testing on GPU kernels to either provide evidence of the usefulness of mutation testing on GPU programming, or show that mutation testing is less valuable on evaluating testing effectiveness on GPUs.

4.1 Research Goals

As the infrastructure and precondition of applying mutation testing, it is firstly required to demonstrate how to effectively apply testing on GPUs, including the test suite design, the result verification and the coverage criteria measurement.

After that, we would like to examine the feasibility and effectiveness of applying mutation testing to GPU kernels through the following four milestones:

1. Explore the mutation operators used for GPU computing platforms.
2. Develop a mutation testing tool for GPU kernels.
3. Do the mutation testing using the presented mutation operators and test cases.
4. Analyze the result and effectiveness of mutation testing on GPUs.

4.2 Research Questions

Based on the goals listed above, to steer our study, we propose the following three core questions. We limit the scope of each research question by proposing sub-questions addressing the core parts of each research question, showing our working direction to the core problem. These sub-questions need to be answered in order, and the answers to our research questions will be concluded on the answers to each sub-question

- Research question 1 (**RQ1**): *How to effectively apply testing to GPU kernels?* We focus on the design, execution, verification of the test suite, and the estimation of testing quality with the following sub-questions.
  - **RQ1-1**: How to effectively design a test suite for testing and mutation testing on GPUs?
  - **RQ1-2**: When having a large number of test cases, how effective are they when executing on GPUs?

  - **RQ1-3**: How effective are the statement/branch coverage criteria in estimating the testing comprehensiveness on GPUs?

- Research question 2 (**RQ2**): *How effective is applying mutation testing to GPUs* This question can be broke down into three sub-questions:
  - **RQ2-1**: How to explore the mutation operators that can be applied to GPUs?
  - **RQ2-2**: How effective are these mutation operators on CUDA computing platform assumed by Coupling Effect Hypothesis?
  - **RQ2-3**: How effective are the mutation operators representing the real faults assumed by Competent Programmer Hypothesis?
  - **RQ2-4**: How effective is the mutation testing evaluating the testing quality?

4.3 Research Objects

The benchmark objects for this research are drawn from several kernel benchmark projects, includes the CUDA samples from NVIDIA [25], existing kernel examples of Kernel Tuner project [38], tunable kernels from Sagecal [16], and some other benchmark projects are investigated in this research. Some kernels from them are imported into our research after modification to the tunable ones.

Project	Kernel	Tunable
Kernel-Tuner	VectorAddTemplate	Yes
	Matrix multiplication (share memory)	Yes
	Matrix multiplication (tiling)	Yes
	Diffusion (naive)	Yes
	Diffusion (tiling)	Yes
	Convolution	Yes
Sagecal	ArrayBean	Yes
	Coherency	Yes
	Sincos (cub)	Yes
	Sincos (manual)	Yes
Hetero-Mark	Histogram (Hist)	Yes
CUDA-Samples	Increment	Yes
Math	Saxpy	Yes

TABLE 4.1: Benchmark projects

4.4 Tool Development

We design and implement our mutation testing tool to apply testing and mutation testing on GPUs. To simplify the development and workflow of our tool, we built it as a part of Kernel Tuner and named it **Mutation Kernel Tuner**.

The following modules are developed to support our designed experiments shown in Chapter 5 and Chapter 6. The corresponding details of these two modules are introduced in Section 5.3 and 6.4 for a better reading context.

- Testing Module: This module is responsible for the compilation, kernel execution using the test cases, coverage measurement, and the host device data transportation.

- Mutation Testing Module: This module includes two main components, the Mutation Analyzer and the Mutation Executor. The mutation Analyzer is responsible for locating the mutants in kernel code using the mutation operators. A schema is also designed to allow the analyzer to serialize and export the mutants to a file. The Mutation Executor is designed for receiving the mutants from the Analyzer, mutating the kernel code, calling the Testing Module to execute the mutant for test cases, and verify the result of execution.

Chapter 5

Testing on GPU Kernels

Before we can actually do mutation testing on GPUs, we need to design the test cases for each benchmark kernel and do a dry run to make sure that the test cases are valid. This will answer the sub-question **RQ1-1** and **RQ1-2**, shown in Section 5.1. The coverage data are also collected at this stage shown in Section 5.2, the data will be analyzed later along with mutation testing results to answer **RQ1-3**. To apply the testing and coverage measurement to GPUs, we develop a testing module for our tool, which is introduced in Section 5.3

5.1 Apply Testing to GPU Kernels

We design the test suite for the testing and following mutation testing process and execute the designed test cases for evaluation and coverage collection. The goal of this experiment is answering the sub-question **RQ1-1** and **RQ1-2**.

5.1.1 Design, Execute and Verify the Test Suite

To apply effectively testing to GPU kernels, we first need to design a test suite for each kernel, find the best kernel execution parameters to launch the kernel, and verify the output.

Test suite design

As shown in Table 4.1 above, the benchmark projects we used are from several existing studies, which may use different result verification methods or even do not have a verification mechanism. Our test suite design may also be different varying among benchmark kernels. However, we conclude some general guidance for our test suite design.

For most of the benchmark projects that offer a reference function for generated inputs and expected outputs, we will reuse these mechanisms and generate 3 test cases for each benchmark projects. These reference functions include CPU functions written in Python or C/C++, and GPU device functions in CUDA. If the reference function requires to designate some functional parameters, these parameters will be selected by maximizing, minimizing, randomizing and assigning null (if available) based on the code logic. These parameters selecting methods will be combined to generate 3^n (or 4^n if null is available, n is the number of parameters) test cases except for the illegal input combination. In particular, if the parameter is closely related to the problem size of the kernel, which may influence the thread-block division, it will be selected at least once among $32, 256, 1024$ *and* 1048576 (2^{20}) depending on the specific kernels.

For some benchmark projects which already contain one or more test cases, we reuse the existing ones and make sure for each benchmark projects will have at least 3 test cases. For the projects which contain no verification mechanism, we design our own at least 3 test cases based on the code logic.

However, since the benchmark projects involved are diverse, there must still be some exceptions depending on the specific kernels.

Kernel execution parameters

As discussed in Section 2.3, the execution parameters for thread block division and memory management will dramatically influence the performance of the kernel, and sometimes even the correctness. If not otherwise specified, we will use the following parameters to tune the kernel when available to get the best execution parameters for test case executions, shown in Table 5.1.

Tunable parameters	Default values
block_size_x	[16, 32, 64, 128, 256]
block_size_y	[2, 4, 8, 16, 32]
tile_size_x	[1, 2, 4]
tile_size_y	[1, 2, 4]

TABLE 5.1: Tunable parameters and their default values

Execution result verification

Most of the return results of kernels are numeric values. For these the Kernel Tuner has a sophisticated function to verify them, as discussed in Section 2.3. It checks the consistency of data types and data lengths of the results and expected outputs before checking whether the numbers are equal or within a given absolute tolerance.

For other benchmark kernels that return non-numeric results, a customized verification function should also be provided along with the test cases. This function should receive an actual output and an expected output, and return a Boolean type indicating pass or fail of a test case.

5.1.2 Experiment Setup

1. Design and generate test cases for each benchmark project.
2. Conduct the test executions for each test case.
3. Verify the kernel outputs with expected outputs for each test case.
4. Generate the testing report.

This experiment also works as a validation of our Mutation Kernel Tuner tool. If anything goes wrong during these experiment steps, both the test suite and the tool are examined to check where it goes wrong.

If a test case fails the testing, a new test case will be designed using the method shown in the experiment design. Then this setup will be conducted again on this test cases to make sure that all test cases are passed and the coverage data are collected. However, if

a bug of our tool is detected during this experiment, all the test cases for all benchmark projects will be executed again.

After the experiment, the passed test suites will be used for the following mutation testing experiments in Chapter 6. The coverage data will be analyzed along with the result of mutation testing to quantify the comprehensiveness of testing on GPU kernels.

5.2 Code Coverage Measurement on GPUs

As introduced in Section 2.1 there are neither concurrency coverage criteria nor coverage measurement tools for GPU kernels provided by CUDA or OpenCL. We also have discussed in Section 3.1.2 that all the related studies mainly focus on using the coverage to make sure that the mutants are covered by test cases or not, without discussing whether these coverage criteria can be a way to quantify the testing quality and comprehensiveness. Therefore, we use the following criteria from Li et al. [22] 's related research to calculate the coverage, and propose a way to measure the concurrency statement coverage and branch coverage on the GPU side. The coverage data will be used to analyze the relationship between coverage and testing quality.

- *Least Branch Coverage*: It measures the number of branches covered at least by one thread across the whole kernel, averaged over the number of branches.
  We use this metric to measure whether our test cases have covered all the possible branches of the kernels. When a thread covers a branch, it also covers all the statements on that branch. The goal of this coverage for each kernel is 100% so that we can make sure every line of code is covered by at least one test case, assuring the future mutation testing without uncovered code.

- *Average Statement Coverage*: It measures the number of branches covered by threads across the whole kernel, averaged over the threads. We use this metric to measure the testing comprehensiveness of the kernels. It is an intuitive metric under the hypothesis that higher coverage means a higher possibility to detect a fault.

An example is provided below to explain how to calculate these coverage criteria.

```
// 128 parallel threads with an id from 0 to 127
if (id > 127) {
    return; // No thread goes here
} else {
    if (id < 32) {
        c[id] = a[id] + b[id]; // 32 thread go here
    } else {
        c[id] = a[id] - b[id]; // 96 thread go here
        c[id] = c[id] + 1;
    }
}
```

The code snippet above contains 4 branches and 6 lines of code (LOC). Assume we have 128 parallel threads with corresponding IDs. It is intuitive to find that there are one statement and one branch which are not covered by any threads. Therefore, the Least Branch Coverage are $3/4 * 100\% = 75\%$. For the Average Statement Coverage, there are 2 LOC covered by 128 threads (2 if statement), 1 LOC by 0 thread, 1 LOC by 32 threads, and 2 LOC by 96 threads, so the coverage is $(128*2/6+0*1/6+32*1/6+96*2/6)/128*100\% = 62.5\%$.

5.3 Testing Module Development

To conduct the experiments above, we implement a testing module in our tool Mutation Kernel Tuner. This module is responsible for the compilation, kernel execution using the test cases, coverage measurement, and the host device data transportation. Please note that this module can only measure the coverage for tunable CUDA kernels, which can be directly launched the global function by our Mutation Kernel Tuner. It does not support measuring coverage for the kernels which already have a host function and require that host function to start the kernel.

5.3.1 Testing Module Workflow

A designed workflow of the testing process is listed below. The details are shown in Figure 5.1.

1. The user provides a tunable kernel, a test suite and some optional testing parameters.
2. The Tuning Module tunes the kernel over the problem size of the test cases to get the optimized kernel execution parameters, and categorize the test cases based on problem size.
3. *[Optional]* If the user enables the coverage measurement, the coverage measurement code will be inserted into the kernel. The implementation details are explained below in Section 5.3.2.
4. For each problem size with optimized kernel execution parameters, the Testing Module executes the test cases with the same problem size using the same corresponding parameters.
5. After execution, the Testing Module fetches and verifies the kernel output with expected output.
6. The Testing Module returns the testing results with optional coverage data.

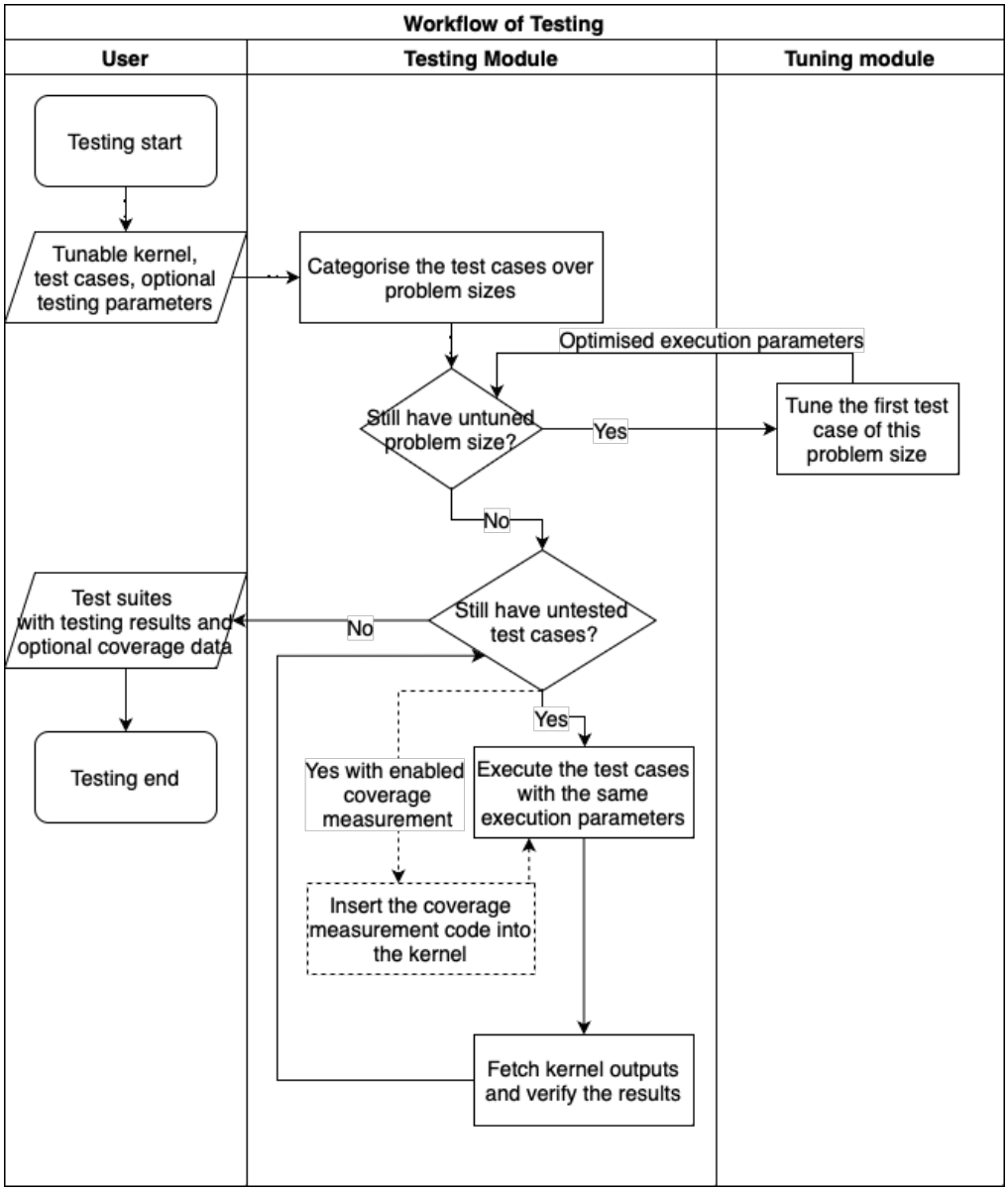

FIGURE 5.1: Workflow of testing module

5.3.2 Coverage Measurement

We propose and use two methods to measure the coverage criteria of GPU kernels, the **Branch Bucket** and **Thread Counter**. The Branch Bucket is used for measuring the Least coverage and the Thread Counter is for the Average Coverage. These two coverage metrics are explained in Section 2.1.

Branch Bucket for Least Coverage

The following code snippet reveals how we measure the Least Coverage using Branch bucket. A Boolean array is added to the kernel parameters, whose elements are initialized as False. For each branch of the code, a True value will be assigned to the corresponding position of the bucket when the assignment statement is covered by at least one thread.

```
__global__ void add(float *c, float *a, float *b, int n, bool *bucket) {
```

```
    bucket[0] = true;
    int i = blockIdx.x * block_size_x + threadIdx.x;
    if (i<n) {
        bucket[1] = true;
        c[i] = a[i] + b[i];
    }
    bucket[2] = true;
}
```

Thread counter for Average Coverage

We use the following code to measure the average coverage. A thread counter array is inserted to the parameter list, whose elements are all zeros. For each branch of the code, the counter will self-increase by 1 when it is covered by a thread. We use *atomicAdd* to make sure this self-increment is synchronized, and only one thread can access the counter memory at a time.

```
__global__ void add(float *c, float *a, float *b, int n, int *counter) {
    atomicAdd(&counter[0], 1);
    int i = blockIdx.x * block_size_x + threadIdx.x;
    if (i<n) {
        atomicAdd(&counter[1], 1);
        c[i] = a[i] + b[i];
    }
    atomicAdd(&counter[2], 1);
}
```

Chapter 6

Mutation Testing on GPU Kernels

To answer the sub-question **RQ2-1**, we conduct a study on collecting mutation operators, shown in Section 6.1.1. We adapt the existing operators for CPU programming to the GPUs, and categorize them as *traditional operators*. We also research on the operators specific to GPU programming, namely *GPU-native* operators.

After that, we validate the Coupling Effect Hypothesis to answer sub-question **RQ2-2** in Section 6.2, research on the git repositories of the benchmark project for Competent Programmer Hypothesis to answer sub-question **RQ2-3** in Section 6.3, and analyze the mutation testing result along with the collected coverage data to answering the **RQ1-3** and **RQ2-4**.

6.1 Apply Mutation Testing to GPU Kernels

In this section, we will introduce how we explore the mutation operators and apply mutation testing to the GPU kernel in this research, answering the sub-questions **RQ2-1**.

6.1.1 Mutation Operators, Coverage and Score

Before we can do the mutation testing, we need to explore the mutation operators guiding the mutant generation. To quantify the results of mutation testing, we need to calculate the mutation coverage and the mutation score.

Mutation operators

We conduct a study on adapting the mutation operators used for mutation testing on CPU programming from previous research and industrial practice [1, 6, 8, 36]. We collect and categorize them as traditional mutation operators.

Similarly, we categorize the operators that can only be executed on GPU computing platforms as GPU-specific mutation operator. We conduct a review and find some GPU-specific operators that are proved to be useful to the mutation of GPU kernel, presented by Zhu et al. [40] in their mutation testing tool for CUDA. These operators will also be included in this research for evaluation.

To explore and design new GPU-specific mutation operators, we go into two research directions, 1) focusing on the special grammar/pattern/syntax that is specific to that application, and 2) following the fault models, where each type of fault in that scenario can be used to design the mutation operator [10]. We will analyze the trees of git commits for some classic kernels and try to find out the fault model from their bugfix patches. Also, we will conduct a literature review on CUDA/OpenCL documentations [14, 24] and related

papers, and search for common mistakes that the kernel developer are likely to make on NVIDIA CUDA forum [26] and Stack Overflow [28].

Following the above exploration approaches, we find two kinds of new useful mutation operator specific to CUDA.

- *sync_ child_ removal*: This operator removes the synchronization between the child kernel from a parent block. An example can be found below.

```
__device__ void child(_){...}
__global__ void parent(_){
    ...
    child<<<Grid, Block>>>(_);
}
int main() {
    ...
    parent<<<Grid, Block>>>(_);
    cudaDeviceSynchronize();   //<-- original
    //cudaDeviceSynchronize(); //<-- mutant
    return 0;
}
```

- *fence_ removal*: This operator addresses the data race problem introduced in Section 2.3.3. It removes the fence function in the kernel code. There are three kind of fence function in CUDA, *__threadfence()*, *__threadfence_ block()*, and *__threadfence_ system()*. An example is shown below.

```
__device__ void readXY() {
    int B = Y;
    __threadfence();    //<-- original
    //__threadfence();  //<-- mutant
    int A = X;
}
```

Table 6.1 shows the traditional operators we use in this experiment. Since there are so many operators and we cannot apply all of them at once, we tentatively prioritize them based on their popularity in related research and estimated effectiveness, because some operators may have functional overlap with others.

Table 6.2 lists the operators specific to the CUDA GPU kernel. In this study, we only validate and benchmark these operators under the CUDA computing model. The grammar for these operators may differ between CUDA and OpenCL, but the offered functionalities are similar, so theoretically the alternatives of these operators can also be found in OpenCL.

In this experiment, we measure the number of mutants for each mutation operator to analyze the effectiveness of these operators.

Name and Ref	Operator(s)	Description	Priority
conditional_ boundary_ replacement (CBR) [6, 40, 32]	original: <, <=, >, >= mutant: <=, <, >=, >	replace the relational operators with corresponding boundary counterpart	High
arithmetic_ operator_ replacement_ short-cut (ARS) [6, 40, 32]	original: i++; i–; mutant: i–; i++;	replace increments short-cut with decrements and vice versa	High
conditional_ operator_ replacement (COR) [6, 40, 32]	original: &&, || mutant: ||, &&	replace conditional operator AND (&&) with OR (||) and vice versa.	High
math_ replacement (MR) [6, 40, 32]	original: +, -, *, /, %, &, |, ^, <<, >> mutant: -, +, /, *, *, |, &, &, >>, <<	replace binary arithmetic operations with corresponding math operation	High
negate_ conditional_ replacement (NCR) [6, 40, 32]	original: <, <=, >, >=, ==, != mutant: >=, >, <=, <, !=, ==	replace all conditionals found with corresponding counterpart	High
short-cut_ assignment_ operator_ replacement (ASR) [6, 32]	original: +=, -=, *=, /=, %= mutant: -=, +=, /=, *=, *=	replace assignment short-cut with corresponding math operation	Medium
arithmetic_ operator_ insertion (AIU) [6, 32]	original: a; mutant: -a;	inverts negation of integer and floating point variables.	Medium
conditional_ operator_ deletion (COD) [6]	original: !a; mutant: a;	delete the conditional operator	Medium
arithmetic_ operator_ deletion (AOD) [6]	mutant: ^a mutant: a	delete the arithmetic operator	Low
conditional_ statement_ deletion (CSD) [6]	original: a==b, a==b; mutant: true, false;	delete the conditionals statement and let it always be true or false (Partially overlap with NCR)	Low

TABLE 6.1: Traditional mutation operators

Name and Ref	Operator(s)	Description
alloc_swap(AS) [40]	original: add<<<4096,256>>> mutant: add<<<256,4096>>>	Replace the number of threads with the number of blocks in parallel processor allocations (and vice versa).
alloc_increment(AI) [40]	original: add<<<4096,256>>> mutant: add<<<4096+1,256>>>	Increase the number of parallel processors (in both threads and blocks) allocated by one
alloc_decrement(AD) [40]	original: add<<<4096,256>>> mutant: add<<<4096-1,256>>>	Decrease the number of parallel processors (in both threads and blocks) allocated by one
share_removal(SHR) [40]	original: __shared__ float cache[N]; mutant: float cache[N];	Remove the shared memory space specifier in variable declarations
gpu_index_replacement(GIR) [40]	original: int tid = blockIdx.x; mutant: int tid = threadIdx.x;	Replace the thread indexing variable with the block indexing variable (and vice versa)
gpu_index_increment(GII) [40]	original: int tid = blockIdx.x; mutant: int tid = blockIdx.x+1;	Increase the indexing variables (threadIdx and blockIdx) by one
gpu_index_decrement(GID) [40]	original: int tid = blockIdx.x; mutant: int tid = blockIdx.x-1;	Decrease the indexing variables (threadIdx and blockIdx) by one
sync_removal(SYR) [40]	original: __syncthreads(); mutant: //__syncthreads();	Remove the synchronization function call
atomic_replacement(AR) [40]	original: atomicAdd(), atomicSub(), atomicExch(), atomicMin(), atomicMax(), atomicInc(), atomicDec(), atomicCAS(), atomicAnd(), atomicOr(), atomicXor() (may have _system or _block suffix)	Replace the atomic operation with ordinary arithmetic operation or conditional operation
sync_child_removal(SCR)	original: cudaDeviceSynchronize(); mutant: //cudaDeviceSynchronize();	Remove the parent-child synchronization function call
fence_removal(FR)*	original: __threadfence() mutant: //__threadfence() (may have _system or _block suffix)	Remove the fence function call

TABLE 6.2: GPU-native mutation operators (CUDA)

Mutation Coverage

In this experiment we measure the mutation coverage over the test suite. Only the mutant that is covered by at least one thread will be considered as a covered mutant. In other words, a mutant must lie along at least one of the execution paths of the kernel; otherwise, it will not impact the kernel output and becomes an equivalent mutant. Mutation coverage is the ratio of covered mutants to the total number of mutants.

Mutation Score

As introduced in Section 2.2.4, the mutation score is measured as the ratio of the killed non-equivalent mutant to the total number of non-equivalent mutants. We plan to manually distinguish the equivalent mutants since there is no sophisticated tool or method to speed up this process. The mutants with compilation errors are not included in this calculation since the kind of errors are generated by defective mutation operators.

6.1.2 Experiment Setup

1. Explore and design the operator(s).
2. Use these operators to generate mutants for each benchmark kernels.
3. For each mutant, mutate and compile the code. If the compilation succeeds, go to the next step; if it fails, mark the mutant as COMPILE-ERROR status and continue to the next mutant.
   Note: the details about mutant status are explained in Section 6.4.2 below.
4. Execute the mutants for all test cases of a kernel. Check if the mutant is covered by the test case or not.
5. For each test case, if the execution goes well, go to the following step; If the kernel throws a runtime error or timeout, mark the mutant status as RUNTIME-ERROR or TIMEOUT respectively and continue the next mutant.
6. Verify the output of the mutated kernel with the expected output of the test case.
   - If the verification fails, mark the mutant as KILLED status and continue to the next mutant;
   - If the verification passes, execute the next test case for this kernel until running our of test cases, mark the mutant as SURVIVED status and continue to the next mutant.
7. Collect the results for mutants and test cases.

The described process of the mutation testing (step 3-6) is shown in Figure 6.1.

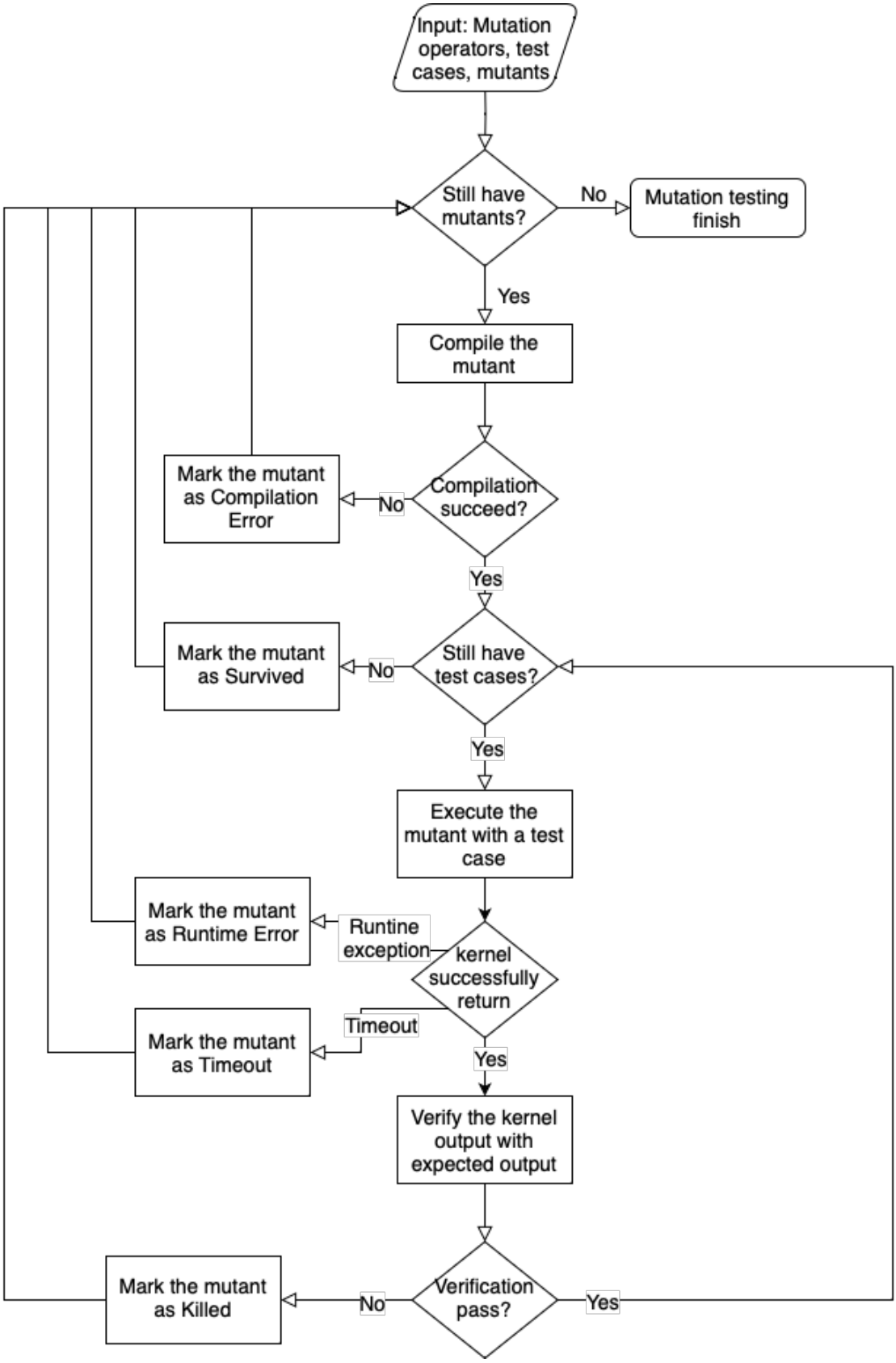

FIGURE 6.1: Process of Mutation Testing

6.2 Coupling Effect Hypothesis on GPUs

We conduct an experiment to verify whether the Coupling Effect also holds in the domain of the GPU kernel. This experiment is mostly following Offutt's approach [27] with some necessary modifications adapting to GPU programming.

6.2.1 2-order Mutant Generation

To generate the 2-order mutants, we pairwisely combine all first-order mutants from the previous experiment shown in Section 6.1 and exclude mutants that contain compilation error sub-mutants.

6.2.2 Experiment Setup

The whole experiment process is listed as follows.

1. Create a 1-order mutation-adequate test suite for each benchmark project.
2. Generate the 2-order mutants and execute these mutants with the same test suite from 1-order mutation testing.
3. Analyze the execution result. Try to figure out which and why the 2-order mutants are alive.

After the execution of 2-order mutation testing, we can get the mutation score of this high order mutation testing and analyze the alive 2-order mutant.

If the Coupling Effect also holds in GPU programming, the expected result will be that the 1-order mutation-adequate test suite can also kill most of the higher-order mutants. If it is not the case, the result will be carefully analyzed to seek potential reasons.

6.3 Competent Programmer Hypothesis on GPUs

As introduced in Section 2.2.2, the studies on this topic are controversial due to the opposite results from several related research. In this experiment, we focus on tracing the bug fixes on the git tree of some benchmark kernels and investigate if our explored mutation operators are capable of representing the bugs shown there.

6.3.1 Git Repository Analysis

The git repositories of all collected benchmark projects are included. This experiment requires that the benchmark projects have clear git commit trees with explicitly marked bug fixes. However, it is supposes to take a number of human works to find the suitable benchmark projects and analyze the Git tree. This is because the lacking maintenance of the projects we currently found makes them less meaningful due to the inadequate number of bug fixes.

We measure the bug-fix commits on the merged main branch over the benchmark project repositories, and the representative of bugs over the mutation operators. The format of the commit may vary from benchmark projects, but the description of a typical one should include keywords with 'fix' or 'bugfix', or with a hashtag to a concrete issue.

6.3.2 Representativeness of Bugs

The representativeness of the bugs over mutation operators is measured based on the real bug found and the fixing solutions. The operator should have a similar functionality to a real bug and the fixed code should preserve the behavior of the original code of the operator. We inspect whether our mutation operators can generate the mutants that can lead to the same faults and be fixed by these bug-fix patches. We also analyze the feasibility of using these models to design the new mutation operators.

6.3.3 Experiment Setup

The experiment process is listed as follows.

1. For each benchmark project, search the git commit tree for the commits with keywords 'fix' or 'bugfix'.

2. For the qualified commits, inspect all the code changes to check if they can be represented by one or multiple mutation operators, or if we can design new mutation operators to represent these faults.

6.4 Mutation Testing Module Development

To conduct the experiments above, we implement a mutation testing module in our tool Mutation Kernel Tuner. This module includes two main components, the Mutation Analyzer and the Mutation Executor. The mutation Analyzer is responsible for locating the mutants in kernel code using the mutation operators, explained in Section 6.4.2. A schema is also designed to allow the analyzer to serialize and export the mutants to a file. The Mutation Executor in Section 6.4.3 is designed for receiving the mutants from the Analyzer, mutating the kernel code, calling the Testing Module to execute the mutant for test cases, and verify the result of execution.

6.4.1 Workflow of Mutation Testing

A typical workflow of mutation testing of our tool is listed below. The details are shown in Figure 6.2.

1. The user provides a tunable kernel, a test suite, and some optional mutation testing parameters.

2. The Testing Module tests all the test suite to make sure all the test cases are valid and to get optional coverage data, optimized thread block parameters, and the best performance matrix over the problem sizes of the test cases.

3. The Mutation Analyzer generates the mutants using the designated operators.

4. The Mutation Executor executes the mutants, handle the exceptions and return the mutation result.

5. If the result is different, the mutant will be marked as killed. Or the mutant will be executed with other test cases. The mutant will be marked as survived if the test cases are exhausted.

6. The tester manually recognizes and eliminates the equivalent mutant from the survived mutants.

7. The tester analyzes the testing results and calculates the mutation testing score.

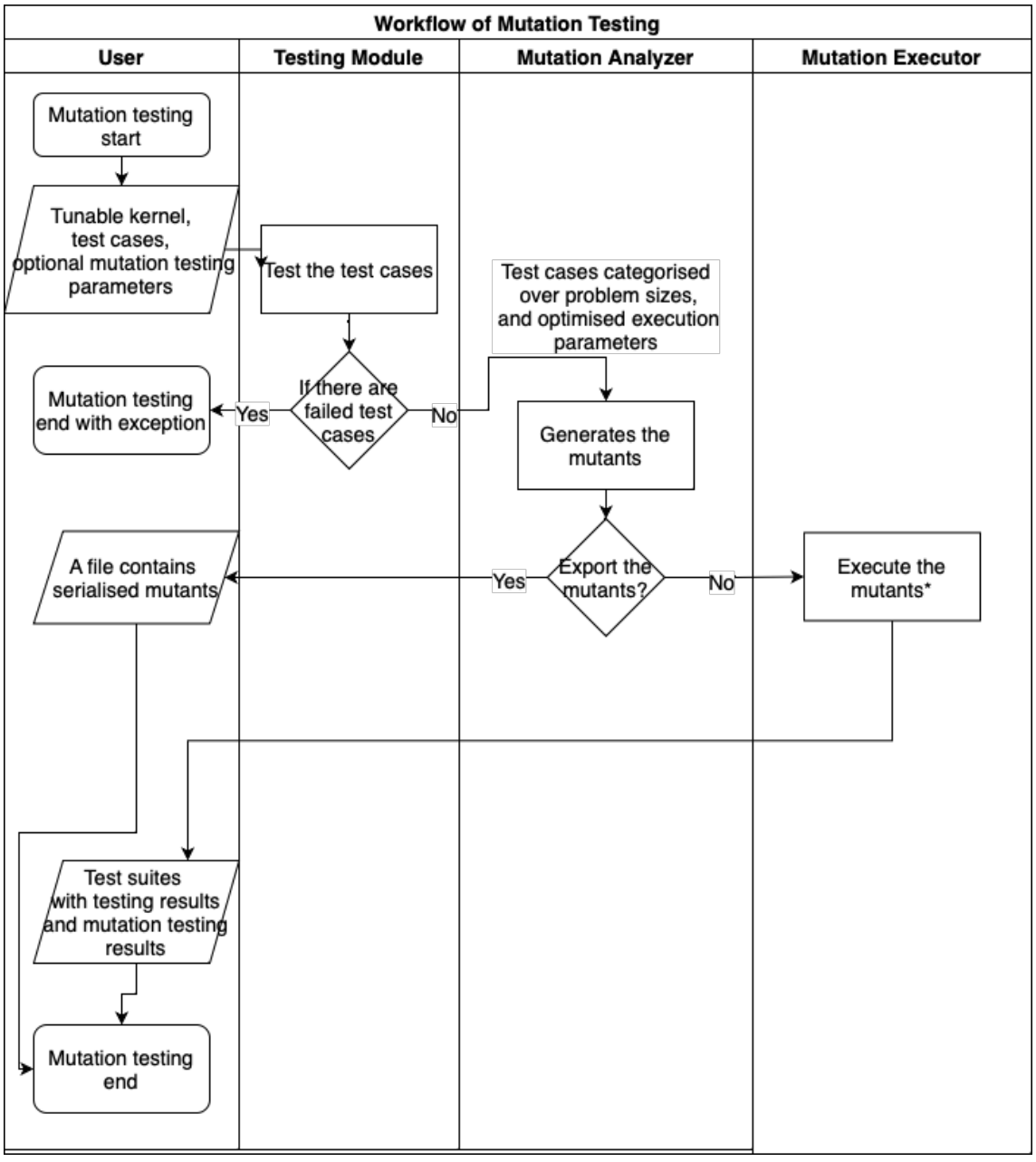

FIGURE 6.2: Workflow of mutation testing module
Note: The details of *Execute the mutants** is shown in Figure 6.1.

6.4.2 Mutation Analyzer

The mutation analyzer is designed to locate the mutants in kernel code, it supports three different ways to locate the syntax or statement(s) to mutate.

- Regular expression: The regular expression analyzer will scan the kernel code line by line and find the mutants using regular expression defined by mutation operators.

It is the easiest way to locate a mutant, however, it is not that precise and cannot distinguish some mutation operators with the same or similar syntax, for example, the pointer declaration and multiplication operation. Therefore, an optional skipping pattern is provided to ignore specific mismatched situations. The regular expression analyzer can be used for both user-customized and built-in mutation operators of the Mutation Kernel Tuner, but we should avoid using this if there is a risk of confusion.

- Helper function: A helper function receives the whole kernel code and returns zero or more mutants by analyzing the code using the programmed logic within the function. It requires significant human work to design the helper function for each mutation operator, so it is only recommended to be used for user-customized mutation operators.

- Parser: A parser analyzer will use the pre-defined language-specific matching patterns to split the kernel code into snippets. Each snippet represents a different part (e.g., a variable declaration in the parameter list) or statement (e.g., if statement) of the kernel, which allow an operator to easily locate a single snippet and replace it with a mutant. Since the code structure and grammar information are preserved, these snippets can be reconstructed to an executable kernel. It looks like it is the silver bullet of mutation analysis, however, it requires a lot of expert knowledge to design the matching patterns based on the syntax and grammar of the specific language. Therefore, it is only used for the built-in mutation operators of the Mutation Kernel Tuner.

After locating the mutant, the analyzer will pass the mutants to the mutation executor, or export the mutant following a schema defined below into a JSON file, allowing the mutation executor to import and execute the mutants later. In this way, we decouple the mutation analysis and execution, which enables the ability that running the mutants in off-peak hours when the GPUs are not that busy. A mutant example conforming to this schema is shown in subsection below.

Mutant status enumeration

There are nine kinds of mutant status implemented in this tool, cf. Figure 6.3. Each status is explained below:

- CREATED: The mutant is just created by the Mutation Analyzer
- PENDING: The mutant is received by the Mutation Executor, but haven't been applied to the kernel.
- KILLED: The mutant is killed by a test case.
- COMPILE-ERROR: The corresponding mutated kernel has compilation errors.
- RUNTIME-ERROR: The mutated kernel throws a runtime error when executing.
- TIMEOUT: The mutated kernel times out when executing
- NO-COVERAGE: The mutant is not covered by any test cases.
- SURVIVED: The mutant is not killed by any test cases.
- IGNORE: The mutant is not suitable for execution or contains unknown errors.

The mutant status transformation is shown in Figure 6.3.

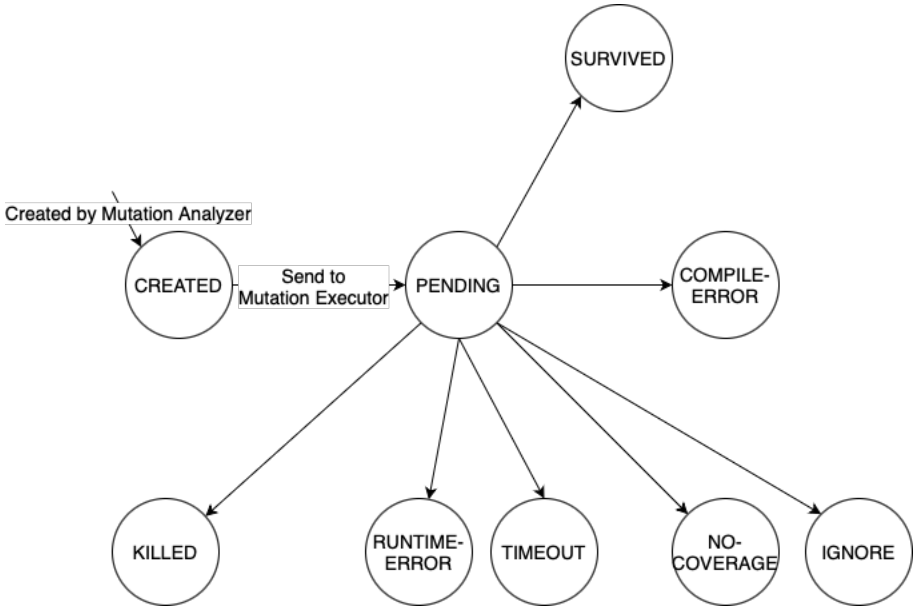

FIGURE 6.3: Mutant Status Transformation

Mutant schema example

The code snippet below shows a mutant example conforming to the JSON schema we designed to serialize the mutant.

```
{
    "id": 0,
    "operatorName": "math_replacement",
    "replacement": "-",
    "status": "Created",
    "killedById": [],
    "coveredById": [],
    "location": {
        "start": {
            "line": 3,
            "column": 41
        },
        "end": {
            "line": 3,
            "column": 42
        }
    }
}
```

6.4.3 Mutation Executor

A mutation executor is designed to execute the mutants generated by the Mutation Analyzer. For each mutant, all test cases will be executed at least once except the mutant has compilation/runtime error(s) or has already been killed.

For each mutant, the executor will firstly mutate the code and then compile it to check if it contains compilation error(s). After that, the execution backends of Kernel Tuner will

take over the initialization jobs and launch the kernel. There are three situations that may happen at this stage:

- Everything is fine and the kernel returns the result: The actual output will be verified with the expected output to decide if the mutant is killed or survived.
- The mutant execution cannot be normally completed and reaches a timeout threshold: A timeout mechanism is designed for this situation, which is shown below in subsection below. The mutant status will be marked as timeout.
- The mutant execution throws a runtime exception: This is the most complicated situation. The whole kernel execution context needs to be cleaned and re-initialized before being ready for the next mutant. The design of this recovery is explained in subsection below. The status of erroneous mutant will be marked as runtime error.

Handling runtime exception

Since mutation testing aims at inserting designed faults into the origin code, there is always a possibility that the mutated kernel will throw an exception during runtime. However, handling the runtime error is not that intuitive. The CUDA context will be dead after raising an error, but the corresponding GPU process will not be terminated automatically. Therefore, we need to kill this GPU process before another context can be re-initialized to continue the next mutant execution, and the only way to do so is to kill the CPU caller process, since the GPU process shares the same PID number and has the same life-cycle as the CPU one. That is why we have a separate runner from the executor, which is running on a sub-process, shown in Figure 6.4. If the GPU context throws a runtime exception, the Mutation kernel Tuner will kill the runner and initialize a new one to recover the GPU context.

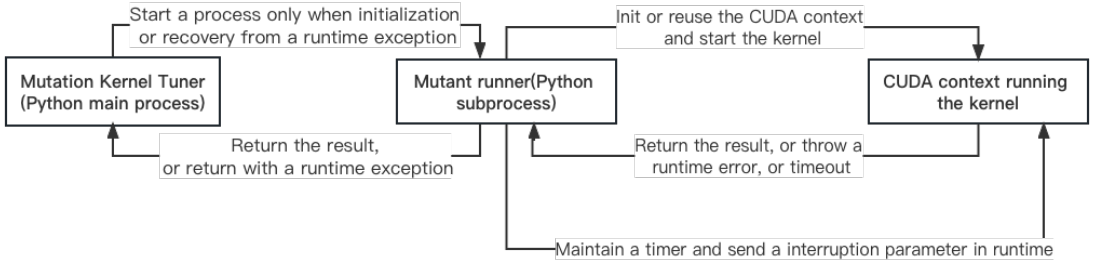

FIGURE 6.4: Sub-process runner for mutant

Timeout mechanism

For the mutated kernels that may be involved in an infinite loop and cause a timeout problem, two methods are used to make sure the whole mutation testing process can be recovered. The first method happens on the GPU side. By adding a Boolean flag to the kernel arguments and inserting code to all loop statements constantly monitoring this flag, we can pass an interruption signal to the kernel and force it to return during runtime when reaching timeout threshold. This flag needs to be passed to the GPU memory by another GPU stream as all the operations on the same stream happen serially.

A CUDA code snippet below addresses how this method works. The *interrupt* parameter is declared with keyword *volatile* to suppress the compiler optimization and make sure the threads will always check the flag memory when they pass by the if statement.

```
__global__ void itrpt(volatile bool *interrupt){
    while (true) {
        if (*interrupt){
            printf("Kernel interrupted");
            return;
        }
    }
}
```

The second method is an add-on of the sub-process runner. We can simply kill the sub-process on CPU side to force the GPU driver kill its corresponding process, also shown in Figure 6.4. However, this method cost a lot more computing resources than the first one, and is used as a global timeout solution when the kernel timeout mechanism does not work.

Chapter 7

Results

7.1 Testing Results and Coverage

To answer the research question **RQ1** about the effectiveness of applying testing on GPUs, we followed the experiment setup in Section 5.1 to design out test suites and execute them with measuring the coverage data for each benchmark kernels. The testing results is shown in Table 7.1.

Kernel \Test cases	Built-in test cases	Generated test cases	Human-designed test cases	Testing Results
Saxpy	0	5	0	All Passed
VectorAddTemplate	0	5	0	All Passed
Matris multiplication (tiling)	0	5	0	All Passed
Matrix multiplication (share memory)	0	5	0	All Passed
Diffusion (naive)	0	3	2	All Passed
Diffusion (tiling)	0	3	2	All Passed
ArrayBean	0	2	0	All Passed
Coherency	0	1	0	All Passed
Sincos (cub)	0	2	0	All Passed
Sincos (manual)	0	2	0	All Passed
Convolution	0	1	0	All Passed
Histogram	0	3	0	All Passed
Increment	0	5	0	All Passed

TABLE 7.1: Testing results

We also collect the coverage data to answer the question **RQ1-3** regarding the effectiveness of coverage criteria on GPUs, which is shown in Table 7.2, following the experiment setup in Section 5.2. These data will be further used to investigate the relationship between coverage and testing quality.

Kernel \Coverage	Loc	Least Branch Coverage (Highest test case)	Average Statement Coverage (Highest test case)
Saxpy	10	100%	100%
VectorAddTemplate	8	100%	100%
Matris multiplication (tiling)	52	100%	100%
Matrix multiplication (share memory)	30	100%	100%
Diffusion (naive)	14	100%	99.97%
Diffusion (tiling)	41	100%	93.75%
Convolution	127	75% (limited by macro, depending on the execution parameters)	87.5% (limited by macro, depending on the execution parameters)
Histogram	32	100%	88.90%
Increment	5	100%	100%

TABLE 7.2: Testing coverage results

7.2 Mutation Testing Results

Towards the questions about the effectiveness of coverage criteria and mutation testing (**RQ1-3**, **RQ2-1** and **RQ2-2**), we used the designed test suites and did a mutation testing using our presented mutation operators under the experiment setup shown in Section 6.1. The mutation results for each benchmark kernels are shown in Table 7.3.

Kernel \Mutants	Total mutants	Covered mutants	Killed mutants	Equivalent mutants	Survived mutants	Mutation Score
Saxpy	14	14	12	2	0	100%
VectorAddTemplate	13	13	11	2	0	100%
Matris multiplication (tiling)	93	93	87	4	2	96.66%
Matrix multiplication (share memory)	36	36	36	0	0	100%
Diffusion (naive)	54	54	51	1	2	98.07%
Diffusion (tiling)	93	93	87	3	3	96.666%
ArrayBean	38	34	32	1	1	94.12%
Sincos(cub)	18	18	16	0	2	88.89%
Sincos(manual)	28	28	26	2	0	100%
Convolution	142	142	142	0	0	100%
Histogram	29	29	20	6	3	86.96%
Increment	11	11	10	1	0	100%
Total	**569**	**565**	**530**	**22**	**13**	**97.61%**

TABLE 7.3: Mutation testing results

For detailed analysis of the mutants, the numbers of generated mutants and corresponding mutation operators for each kernels are shown in Table 7.4 for the traditional operators and in Table 7.5 for the GPU-specific operators.

For further investigation of the equivalent mutants and survived mutants, the survived mutants per kernels is shown in Table 7.6.

Kernel	Operator(s) of equivalent mutants	Operator(s) of survived mutants
Saxpy	1*CBR, 1*GID	-
VectorAddTemplate	1*CBR, 1*GID	-
Matris multiplication (tiling)	4*CBR	2*SR
Matrix multiplication (share memory)	-	-
Diffusion (naive)	1*GII	1*MR, 1*GID
Diffusion (tiling)	3*CBR	3*MR
ArrayBean	1*CBR	1*CBR
Sincos(cub)	-	2*SR
Sincos(manual)	1*CBR, 1*NCR	-
Convolution	-	-
Histogram	5*CBR, 1*GID	1*CBR, 1*ARS, 1*SR
Increment	1*GID	-

TABLE 7.6: Operator(s) of equivalant and survived mutants per benchmark kernels

For convenience, the same data per mutation operators is also provided in Table 7.7 for tradition operators and in Table 7.8 for GPU-specific operators.

kernel \Operators	CBR	ARS	COR	MR	NCR	ASR	AIU	COD	AOD	CSD	Total traditional mutants
Saxpy	1			4	1	0	0	0	0	0	6
VectorAddTemplate	1			3	1	0	0	0	0	0	5
Matris multiplication (tiling)	10	9	0	40	10	2	0	0	0	0	71
Matrix multiplication (share memory)	2	1	0	13	2	2	0	0	0	0	20
Diffusion (naive)	4	0	3	31	4	0	0	0	0	0	42
Diffusion (tiling)	12	2	6	45	12	2	0	0	0	0	79
ArrayBean	6	1	0	31	0	0	0	0	0	0	38
Sincos(cub)	1	0	0	5	2	3	0	0	0	0	11
Sincos(manual)	3	0	0	9	3	5	0	0	0	0	20
Convolution	14	8	5	76	22	3	0	0	0	0	128
Histogram	6	3	0	4	5	1	0	0	0	0	19
Increment	0	0	0	3	0	0	0	0	0	0	3

TABLE 7.4: Number of mutants generated by corresponding traditional operators

kernel \Operators	AS	AI	AD	SHR	AR	GIR	GII	GID	SYR	SCR	BR	Total GPU-specific mutants
Saxpy	0	0	0	0	0	4	2	2	0	0	0	8
VectorAddTemplate	0	0	0	0	0	4	2	2	0	0	0	8
Matrix multiplication (tiling)	0	0	0	2	0	12	3	3	2	0	0	22
Matrix multiplication (share memory)	0	0	0	2	0	8	2	2	2	0	0	16
Diffusion (naive)	0	0	0	0	0	8	2	2	0	0	0	12
Diffusion (tiling)	0	0	0	1	0	8	2	2	1	0	0	14
ArrayBean	0	0	0	0	0	0	0	0	0	0	0	0
Sincos(cub)	0	0	0	1	0	2	1	1	2	0	0	7
Sincos(manual)	0	0	0	2	0	2	1	1	2	0	0	8
Convolution	0	0	0	1	0	8	2	2	1	0	0	14
Histogram	0	0	0	0	0	5	2	2	1	0	0	10
Increment	0	0	0	0	0	4	2	2	0	0	0	8

TABLE 7.5: Number of mutants generated by corresponding GPU-specific operators

Operators	Covered mutants	Killed mutants	Equivalent Mutants	Survived Mutants	Mutation Score
conditional_boundary_replacement (CBR)	60	43	15	2	95.56%
arithmetic_operator_replacement_short-cut (ARS)	24	23	0	1	95.83%
conditional_operator_replacement (COR)	14	14	0	0	100%
math_replacement (MR)	264	260	0	4	98.48%
negate_conditional_replacement (NCR)	62	61	0	0	98.39%
arithmetic_short-cut_operator_replacement (ASR)	18	17	0	1	94.44%
arithmetic_operator_insertion (AIU)	0	-	-	-	-
conditional_operator_deletion (COD)	0	-	-	-	-
arithmetic_operator_deletion (AOD)	0	-	-	-	-
conditional_statement_deletion (CSD)	0	-	-	-	-
Total	**442**	**418**	**15**	**8**	**97.89%**

TABLE 7.7: Mutation testing results per traditional operators

Operators	Covered mutants	Killed mutants	Equivalent Mutants	Survived Mutants	Mutation Score
alloc_swap (AS)	0	-	-	-	-
alloc_increment (AI)	0	-	-	-	-
alloc_decrement (AD)	0	-	-	-	-
share_removal (SHR)	9	6	0	4	66.67%
atom_removal (AR)	0	-	-	-	-
gpu_index_replacement (GIR)	65	65	0	0	100%
gpu_index_increment (GII)	21	20	1	0	100%
gpu_index_decrement (GID)	21	16	4	1	94.12%
sync_removal (SYR)	11	6	0	5	54.55%
atomic_replacement (AR)	0	-	-	-	-
sync_child_removal (SCR)	0	-	-	-	-
fence_removal (FR)	0	-	-	-	-

TABLE 7.8: Mutation testing results per GPU-specific operators

To answer the question **RQ2-4** about the correlation between testing and mutation testing, we show the killed mutants that are either killed by test cases, by runtime error, or by timeout in Table 7.9. Among the mutants killed by test cases, we analyzed how many test cases are used to killed them, shown in Table 7.10.

kernel \Mutant Status	KILLED	RUNTIME_ERROR	TIMEOUT	Total killed mutants
Saxpy	12	0	0	12
VectorAddTemplate	11	0	0	11
Matris multiplication (tiling)	65	20	2	87
Matrix multiplication (share memory)	28	8	0	36
Diffusion (naive)	34	17	0	51
Diffusion (tiling)	65	22	0	87
ArrayBean	26	6	0	32
Sincos(cub)	15	1	0	16
Sincos(manual)	19	6	1	26
Convolution	110	30	2	142
Histogram	20	0	0	20
Increment	10	0	0	10
Total	**415**	**110**	**5**	**530**

TABLE 7.9: Specific status of killed mutants

Kernel \Test cases	Kill by all test cases	Missed 1 case	Missed 2 case	Missed 3 case	Missed 4 case	Missed 5 case	Total KILLED Mutants
Saxpy	1	5	5	0	0	-	11
VectorAddTemplate	5	6	0	0	0	-	11
Matris multiplication (tiling)	6	24	31	4	0	-	65
Matrix multiplication (share memory)	1	7	17	3	0	-	28
Diffusion (naive)	6	21	7	0	0	-	34
Diffusion (tiling)	20	34	14	0	0	-	68
ArrayBean	2	24	-	-	-	-	26
Sincos(cub)	15	-	-	-	-	-	15
Sincos(manual)	18	1	-	-	-	-	19
Convolution	110	-	-	-	-	-	110
Histogram	10	7	0	-	-	-	17
Increment	0	2	7	1	0	-	10

TABLE 7.10: The number of mutants and the number of test cases used to kill these mutants

7.3 2-Order Mutation Testing Results for Coupling Effect Hypothesis

As introduced in Section 6.2, the Coupling Effect Hypothesis assume that a test suite that detects the 1-order mutants in a program can still detect the corresponding composited higher-order mutants. Therefore, to validate the hypothesis, we conducted the 2-order mutation testing on the mutation-adequate test suites to validate the Coupling Effect Hypothesis to answer the question **RQ2-2**. The results are shown in Table 7.11.

7.4 Repository analysis results for Competent Programmer Hypothesis

To answer the question **RQ2-3**, we followed the experiment setup introduced in Section 6.3 to figure our whether the mutation operators in this research can represent the real faults. We analyzed the git repositories of our benchmark projects to collect the bug-fix-related commits. We collected our findings regarding bug-fix commits in Table 7.12, and the real faults and corresponding mutation operators in Table 7.13.

Kernel \Mutants	Total 1-order mutants	Total 2-roder mutants	Killed 2-order mutants	Survived 2-order mutants
Saxpy	14	76	74	2
VectorAddTemplate	13	63	61	2
Matris multiplication (tiling)	93	4238	4222	16
Matrix multiplication (share memory)	36	618	616	2
Diffusion (naive)	54	1376	1371	5
Diffusion (tiling)	93	4532	4521	11
Convolution	142	9981	9981	0
Histogram	29	388	351	37
Increment	11	41	40	1
Total	**485**	**21313**	**21237**	**76**

TABLE 7.11: 2-order mutation testing results

Repository	Total commits	Total bug-fix commits	Traditional operators related	GPU-specific operators related
Hetero-Mark	836	109	1	2
CUDA-Samples	71	12	0	0
Kernel-tuning -for-Sagecal	32	4	0	0
kernel_tuner	175	26	0	0
gpu-parboil	9	4	0	0
polybenchGpu	57	7	0	0
gpu-rodinia	108	18	0	0
Total	1288	180	1	2

TABLE 7.12: Git tree analysis results of benchmark projects

Repository	Commit hash	Related mutation operator
Hetero-Mark	ebc6174	sync_child_removal
	d41a443	sync_child_removal
	80e6412	conditional_boundary_replacement (CBR)

TABLE 7.13: Real faults can be represented by our presented mutation operators

Chapter 8

Discussion

8.1 Testing on GPU Kernels

In **RQ1** we would like to know how to effectively apply testing to GPU kernels. We divided this question into three sub-questions targeting different aspects, the test suite design **(RQ1-1)**, the execution **(RQ1-2)**, and coverage measurement **(RQ1-3)**. We discuss these three sub-questions in separate sub-sections.

8.1.1 Discussion of RQ1-1: Test Suite Design

In **RQ1-1** we would like to know how to effectively design a test suite for applying testing and mutation testing to GPU kernels. Therefore, in Section 5.1.1 we presented a method to design and generate the test suites for GPU programming by using reference functions. By using this method, one can easily generated a set of test cases in one click, just like what we did shown in the replicate package. We then used these test suites for applying testing and mutation testing to GPUs. The testing results shown in Table 7.1 reveals that this method can properly guide the kernel developer to design a set of legal test cases for kernels. The coverage data in Table 7.2 show that the designed test suites can reach a 100% Least Branch Coverage for most of the kernels, which indicate that these test cases can successfully test all the lines of code with at least one thread.

This coverage results fulfil the requirement of a threshold that the testing coverage may have to reach before mutation testing can benefit the testing quality measurement, as discussed in Section 2.2.1. The mutation testing results shown in Table 7.3 further confirm our analysis, which can detect an average of 97.61% tiny faults that can be represented by our presented mutation operators, indicating that the testing quality is relatively high.

Based on this, we can conclude that our presented test suite designing method is effective for applying testing and mutation testing to GPU kernels, and it is also efficient enough for a tester to easily receive a sufficient testing coverage and testing quality.

8.1.2 Discussion of RQ1-2: Test Case Executions

In **RQ1-2** we focused on the execution of these test cases to figure our how efficient are them when executing on GPUs. To answer this sub-question, we analyze the kernel execution process of Kernel Tuner and do a simple experiment to explore the most time-consuming jobs in this process. We select three kernels with different complexity from our benchmark kernels. The results shown in Table 8.1 reveal that compiling the kernel takes more than five to hundreds of times as long as the other steps combined. Therefore, our Mutation

Kernel Tuner compiles the kernel only once for each mutant and execute them for all the test cases.

Kernel \Time (in ms)	Compilation time	Kernel execution and framework consumption
Increment	274	1.35
Histogram	274	10.6
Matrix multiplication	341	66

TABLE 8.1: Time Consumption for the kernel execution process

Kernel	Minimum	Maximum
Increment (Simple)	0.059	0.092
Histogram (Medium)	9.62	10.88
Matrix multiplication (Complex)	65.12	282.62

TABLE 8.2: Kernel execution time with different execution parameters

Also, this experiment shows that how much the execution parameters can affect the kernel runtime. The maximum and minimum time are shown in Table 8.2. As the size and complexity of the problem rises, the performance gap becomes more and more significant. That is why in our architecture design we categorize the test cases by problem sizes, and tune one of the test cases for each problem size to get the best execution parameters for all test cases with the same size.

In these two ways, we speed up the kernel execution process when having a large set of test cases for testing and mutation testing, answering the question **RQ1-2**.

8.1.3 Discussion of RQ1-3: Effectiveness of Coverage Criteria

Sub-question **RQ1-3** focusing on the effectiveness of coverages when using them to measure the testing quality. To answer this sub-question, we need to combine the testing coverages and the mutation testing results for each test cases. For better interpretation of the results for this sub-question, we draw a figure for the Average Statement Coverage per test cases, shown in Table 8.3.

Kernel \ Average Statement Coverage	Test Case 1	Test Case 2	Test Case 3	Test Case 4	Test Case 5
Saxpy	67.71%	100%	100%	100%	100%
VectorAddTemplate	67.71%	100%	100%	100%	100%
Matrix multiplication (tiling)	100%	100%	100%	100%	100%
Matrix multiplication (share memory)	100%	100%	100%	100%	100%
Diffusion (naive)	99.97%	99.97%	99.97%	99.97%	99.97%
Diffusion (tiling)	93.75%	93.75%	93.75%	93.75%	93.75%
Convolution	87.5%	-	-	-	-
Histogram	17.59%	88.90%	51.86%	-	-
Increment	100%	-	-	-	-

TABLE 8.3: Average statement coverage per test cases

Notice that the average statement coverage for the four kernels of *Matrix multiplication* and *Diffusion* are always the same for different test cases, which means, the these test cases should share the same ability of revealing faults. However, the test cases used to kill the mutants of these kernels are varies, cf. Table 7.10, which indicate that these test cases do have different ability on detecting faults. Therefore, we can conclude that this coverage metric is at least not suitable for estimating the testing comprehensiveness of these kernels.

We go one step deeper to use the kernel *Histogram* as an example to analyze the relationship between test case coverages and killed mutants. We find that among the 19 killed mutants, Test Case 1 kills 11 mutants, Test Case 2 kills 18 mutants, and Test Case 3 kills 19 mutants, which does not conform the order shown by the coverage data.

Based on the analysis above, we can conclude that the Average Statement Coverage is not effective in estimating the testing quality and comprehensiveness.

8.2 Mutation Testing on GPU Kernels

In **RQ2** we focus on the effectiveness of applying mutation testing to GPUs. We proposed four sub-questions targeting the mutation operators exploration (**RQ2-1**), validations of the two fundamental hypotheses of mutation testing, Coupling Effect Hypothesis (**RQ2-2**) and Competent Programmer Hypothesis (**RQ2-3**), and the effectiveness of measuring testing quality when using these operators.

8.2.1 Discussion of RQ2-1: Mutation Operator Exploration

The sub-question **RQ2-1** focuses on the methodology of exploring the mutation operators for GPU programming. To answer this question, we designed a method to explore the mutation operators shown in Section 6.1.1 and found 21 mutation operators that can be applied to GPU programming, including 10 traditional operator shown in Table 6.2 and 11 GPU-specific operators shown in Table 6.2. Our mutation testing results, shown in Table 7.3, indicate that our methodology is useful for identifying existing operators and discovering new ones for GPU programming.

8.2.2 Discussion of RQ2-2: Effectiveness of Mutation Operator and Coupling Effect

Sub-question **RQ2-2** requires us to evaluate the effectiveness of the mutation operator from both empirical and theoretical perspectives. Therefore, in the following sections, we will first discuss the application of these operators to our benchmark kernels and analyze the surviving mutants. Then, we evaluate the validity of one of the fundamental hypotheses of mutation testing, the Coupling Effect Hypothesis under the context of GPUs.

Empirical Prospective

From the answers to **RQ1-1** and **RQ1-2**, we have designed a set of test suites and developed a high-efficient way to execute these test suites. On top of these answers, we can apply mutation testing to GPU kernels to evaluate the effectivity of these mutants.

From Table 7.3 we notice that the mutation coverage is 100% for all the kernels, which means that the mutated line(s) of code are covered by at least one thread for at least one test case, indicating that there is no excuse for a mutant that it is not killed because it is uncovered.

With this point in mind, we analyzed the mutation results shown in Table 7.3 and the equivalent mutants and survived mutants shown for each operators shown in Table 7.6. The analysis for some related mutation operator is listed below, along with our recommendation solutions to address the problems.

- *gpu_index_decrement (GID)*: This operator generates a number of equivalent mutants. As we mentioned in Section 2.3, the most typical use case of this index is iterating an array. When applying this operator to mutate the kernel, the iteration range will be -1 to $(TOTAL\_THREAD\_NUM - 1)$. However, the actual number of threads is always equal to or larger than necessary (i.e., the size of the array) because we always round up upward when calculating the thread block division on problem size, and we can get the correct answer most of the time when there are extra threads. Another reason for this is that there is no array-out-of-boundary check during CUDA runtime, so the -1 index of a array is legal when it does not cause other problems.
  This can also answer why mutants generated by the operator *gpu_index_increment (GII)* are easier to be killed because the range is 1 to $(TOTAL\_THREAD\_NUM + 1)$, and the first position is uncovered in this case, which will lead a wrong answer.
  To address this problem, we recommend the kernel developer to manually limit the range of the index after index calculation.

- *conditional_boundary_replacement (CBR)*: This operator always adds or deletes an equal condition for a conditional statement. The most typical mutants for this operators among our benchmark kernels are changing $<$ to $<=$ in loop statements to limit the iteration of an array. It met the same story above that accessing an out-of-boundary array will not directly cause an error.
  To address this problem, we recommend the kernel developer to precisely limit the valid range of the index during array iteration.

- *math_replacement (MR)* The cases for the survived mutant from this operator is because of the GPU index calculation of the *Diffusion Kernel*. This time, there are manual range limitations for the index, and the threads with out-of-range indexes are discarded without influencing the final result. The mutant survived because these

two *Diffusion* kernels are launched with sufficient extra threads to cover the influence. So the mutant results highly depend on the execution parameters for thread block division and tiling and the test cases. **We do not have any specific improvement recommendation for this case, and we suppose these mutants will be killed with other execution parameters or well-designed test cases.**

- *sync_ removal (SR)* As introduced in Section 2.3.3, removing barrier statement may cause nondeterministic behaviour of the code. There is a chance that the result is still correct depending on the thread scheduling and memory access. **We do not have any specific improvement recommendation for this case, the barrier statement is used to assure the correctness instead of gambling whether the kernel without it will still give a correct answer.**

To sum up, both the presented traditional operators and GPU-specific operators can represent the common used statements or syntax in GPU programming. And the survived mutants can also indicate some defects in the kernel code.

Theoretical Perspective (Coupling Effect Hypothesis)

As introduced in Section 6.2, we conducted the 2-order mutation testing on the mutation-adequate test suites to validate the Coupling Effect Hypothesis. The results are shown in Table 7.11. Based on these results, we further analyze every survived higher-order mutants, shown in Table 8.4.

Kernel \Mutants	Survived 2-order mutants	By 2 equivalent mutants	By 2 survived mutants	By both equivalent and survived mutants	By 1 equivalent or survived mutant	Others
Saxpy	2	2	-	-	-	-
VectorAddTemplate	2	1	-	1	-	-
Matris multiplication (tiling)	16	6	1	9	1	-
Matrix multiplication (share memory)	2	-	-	-	-	2
Diffusion (naive)	5	-	1	1	2	1
Diffusion (tiling)	11	2	-	8	-	1
Convolution	0	-	-	-	-	-
Histogram	37	15	3	18	1	-
Increment	1	-	-	-	1	-
Total	**76**	**26**	**4**	**37**	**5**	**4**

TABLE 8.4: 2-order mutants composition

We found that most of these mutants are composed by at least one survived or equivalent 1-order mutant. It is reasonable that the higher order mutants composed by two equivalent or survived 1-order mutants, or by both equivalent and survived 1-order mutant will survive because these 1-order mutants will not influence the final result. So we focus

on the higher-order mutants composed by at least one killed 1-order mutants. By analyzing them one by one manually we found the following cases.

- The two 1-order mutants overlap with each other, and the latter mutation will overwrite the change made by the former mutant.
  This happened among the GPU-specific operators *gpu_index_replacement (GIR)*, *gpu_index_increment (GII)* and *gpu_index_decrement (GID)*. The mutant generated from *GID* will overwrite the change made by the previous mutant, and we have shown that the *GID* will always generate a equivalent mutant in the answer of **RQ2-2**. So this kind of higher-order mutants with only one changes are invalid. This happens in kernel *Histogram* and *VectorAddTemplate*
  This can also happen between the two counterpart forms of GPU-specific operator *gpu_index_replacement (GIR)*, where the latter mutant precisely recovers the code mutated by the former mutant, back to the original version. This explained the 2 survived higher-order mutants for kernel *Matrix multiplication (share memory)*.

- The former mutant coincidentally solves the error that will be caused by the latter mutant on that execution flow. For example, the only higher-order mutant composed with a killed 1-order mutant in kernel *Matrix multiplication (tiling)*, which the previous mutant on the flow coincidentally initializes a out-of-boundary position of an array where the latter mutant should have an random buggy access.

- The two surviving higher-order mutants composed of two killed 1-order mutants happen to survive because of code logic. All the killed 1-order mutants are related to the GPU thread index calculation, and they survived because the kernel limits the number of the index within a legal range, and the kernel is launched with extra threads to compensate for the kernel execution, leading to correct final outputs, and surviving the higher-order mutants.

Based on the above case analysis, all the survived higher-order mutants are weakly uncoupled, which means we did not find any clues that will destabilize the validity of the Coupling Effect Hypothesis, influencing the effectiveness of mutation testing on GPUs.

8.2.3 Discussion of RQ2-3: Real Faults Representative and Competent Programmer Hypothesis

The answer to **RQ2-3** needs to ensure that the mutation operators can represent the real faults happened in real codes, which is assumed by the Competent Programmer Hypothesis. Therefore, we made a repository analysis specific to the fault representation ability of these operators to validate the hypothesis under the context of GPU programming. We followed the experiment setup introduced in Section 6.3, analyzing the git repositories of our benchmark projects to collect the bug-fix-related commits and to find the real faults which can be represented by our presented mutation operators. We collected our findings regarding bug-fix commits in Table 7.12, and the real faults and corresponding mutation operators in Table 7.13.

As shown in Table 7.12, we analyzed the git trees of 7 benchmark projects which includes 1288 commits in total. We manually inspect 180 commits that are related to making a bug fix. However, only one project is detected with three real faults. Two of them are about fixing a missing *cudaDeviceSyncronize()* function, and another one is about change conditional operator $>$ to $>=$, as shown in Table 7.13.

Based on the above results, although we did find some evidence that the mutants can represent real faults a kernel developer can make, we have to admit that this experiment is not persuasive enough to justify whether the Competent Programmer Hypothesis holds or not in the context of GPU programming. The reasons for the failure are discussed in the following.

1. **The git trees of some benchmark projects are incomplete** The repositories *gpu-parboil*, *polybenchGpu* and *gpu-redinia* are initialized only after their corresponding studies are finished, and the git trees did not cover their actual development and verification processes during their research. The *CUDA-Samples* was shipped to this repository from elsewhere recently, and all previous code is aggregated into one single initialization commit.

2. **The bug-fix commit sample size is too small** Although there seems to be a lot of commits are involved in this research, only 13.9% of them are separate bug-fix commits. Among these bug-fix commits, more than half of them are naive patches for typos or code formatting instead of a real fault in the kernel code.

We also discovered some typical and simple fault models during this experiment, which happened multiple times across projects. For example, some patches will limit the number of threads to prevent a array from being read out of bounds by these extra threads. However, these faults were hard to be represented by a mutation operator, and they were not considered in this experiment.

8.2.4 Discussion of RQ2-4: Effectiveness of Mutation Testing

The mutation score analysis shown in the answer of **RQ2-2** has proved that our presented operators are effective to GPU programming with a high mutation score. In this section, we are going to analyze the correlation between testing and mutation testing, answering the question **RQ2-4** about why a higher mutation score indicates a higher quality test suite.

To answer this question, we show the killed mutants that are either killed by test cases, by runtime error, or by timeout in Table 7.9. Among the mutants killed by test cases, we analyzed how many test cases are used to killed them, shown in Table 7.10.

Table 7.9 shows that in most of the cases, the mutated kernel will return a wrong answer, causing the mutant to be killed during the result verification stage. This means that for most of the mutants, the mutated kernel can finish the test case executions instead of throwing runtime errors. For these completed executions, it indicates that the wrong output is just because the mutated code makes a buggy computation instead of other untracked reasons, ensures the effectiveness of mutation testing. In other words, the mutants which will easily cause runtime errors or timeouts for a kernel are not that ideal, because we cannot directly infer that this mutated kernel is interrupted only by this input from the test case, or by any possible kernel inputs, or by other reasons in the kernel code.

Furthermore, for the mutants killed by test cases, we did a further analysis on how many test cases are used to killed them, shown in Table 7.10. We discard the *Convolution* kernel during this analysis because it is too fragile and all the mutants are killed by the only one random generated test case. For other kernels, we notice that most of the mutants will miss one or two test cases during mutation testing. This finding reveals that a test with more test cases has a higher possibility to detect more faults, even if these test cases are mostly random generated by reference functions.

Based on above two analysis, we can conclude that mutation testing is an effective way to measure the quality of testing.

8.3 Threats to Validity

8.3.1 External

- **Benchmark kernels**: We collect a set of kernels from different benchmark projects for applying testing and mutation testing. Most of the selected projects are widely used in all kinds of research in this area. The other kernels are selected because they are involved in the research of Kernel Tuner. In this way, we minimise the threats related to the effectiveness of benchmark projects.

- **Computing platforms**: The mutation operators and benchmark kernels are collected based on CUDA computing platform. To minimize this threat, we also discussed corresponding alternatives or similar functionalities in OpenCL. Theoretically, the methodology of research is also feasible to other computing platforms.

- **Nondeterministic nature of mutated kernels**: We believe that the kernels we collected are well-designed to get a repeatable output for benchmarking. However, we cannot make sure that the results from the mutated kernels can still be repeatable for each execution when using different execution parameters, different graphic cards, and different (random) inputs. To minimize this threat, we did our experiments multiple times to make sure that they always gave us the same results, or that the differences were reasonable to explain.

8.3.2 Internal

- **Correctness of Mutation Kernel Tuner**: The most concerning threat regarding internal validity is the correctness of our Mutation Kernel Tuner tool. It is possible that there are still bugs within our code which may influence the results of mutation testing. To minimize the influence of the final experiment results, we designed an automated script for each benchmark kernel and followed the Test Driven Development process to do regression experiments every time we made a change in our tool.

- **Inconsistent results of kernel tuning**: We tune the kernels before we test them. However, the best parameters may be different among tunings. We preferred to use larger problem sizes to make the performance among different combinations of parameters more distinguishable.

8.3.3 Construct

- **Our method of manually detecting equivalent mutants**: For example, it is possible that we may miss the equivalent mutants or misclassify the survived mutants into equivalent mutants. However, the benchmark kernels are relatively fragile to resist a single change in the code in practice. The survived mutants after mutant execution are not that much, and it's not that time-consuming to distinguish them.

Chapter 9

Conclusion

This section aims to provide a summary of the study and address the research questions, discuss our contributions, and explore the potential areas of future research.

In summary, the goal of this project is to investigate the effectiveness of testing and mutation testing on GPUs, in the context of the complexity of memory and thread management of GPU programming compared to CPU one.

Regarding the first question **RQ1** about how to effectively apply testing to GPU kernels, we present a method to effectively design and generate the test suites for GPU kernels (**RQ1-1**) with a relatively good testing quality, develop a framework to effectively execute massive test cases (**RQ1-2**). In this way, we conclude a methodology and a tool to effectively apply testing to GPU kernels.

During this process, we also discovered that the two coverage criteria involved were NOT effective in estimating the quality and comprehensiveness of testing (**RQ1-3**). This remains the question about how to measure the testing quality of GPU kernels. However, in other words, this also emphasizes the importance of applying mutation testing as an alternative method to measure testing quality.

To address the question **RQ2** about the effectiveness of applying mutation testing to GPUs, we firstly explore 21 mutation operators that can be applied to GPU programming (**RQ2-1**). Then, we demonstrate the effectiveness of these mutation operators from both empirical and theoretical perspectives, by analyzing the surviving mutants and validating the one of the fundamental hypothesis of mutation testing, the Coupling Effect, on our benchmark GPU kernels (**RQ2-2**).

After proving these mutation operators are effective, we still want to assure that these operators can represent the real faults in the real world. That is why we try to prove another fundamental hypothesis of mutation testing, the Competent Programmer Hypothesis (**RQ2-3**). However, our experiment results are not persuasive enough to justify whether the Competent Programmer Hypothesis holds or not because of the lack of available benchmark project size.

We also analyze the the correlation between testing and mutation testing to justify the effectiveness of mutation testing (**RQ2-4**). Our finding reveals that a test with more test cases has higher chances to kill more mutants, which indicates a higher possibility to detect more faults.

With these in mind, we can concluded that testing and mutation testing can benefit the GPU programming in the following forms.

1. Testing is useful in the correctness assurance of GPU programming, and more test cases indicates a higher chance of detecting a fault. Coverage measurement is a way

to assure the quality of mutation testing in GPU programming. However, using the coverage alone to measure the testing comprehensiveness seems a wrong approach.

2. Mutation testing can be used to measure the quality of the test on GPUs, both the traditional and GPU-specific mutation operators involved are effective in simulating the faults on GPUs because the Coupling Effect hypothesis holds. However, we cannot justify that the "simulated faults" are the "real faults" that a GPU programmer can make, which is assumed by Competent Programmer Hypothesis.

9.1 Contributions

Our research makes the following contributions:

1. A comprehensive research on applying testing and mutation testing to GPU programming and the relationship between the mutation testing and testing on GPU kernels.

2. High order mutation testing experiment for Coupling Effect and repository analysis for Competent Programmer Hypothesis on GPU programming.

3. Two more GPU-specific mutation operators.

4. An open-source testing and mutation testing tool for GPU kernel development. The tool is integrated with Kernel Tuner for kernel developer's usage.

9.2 Future Work

1. More computing platforms can be integrated into our tools. In the architecture, our mutant analyzer and executor are able to accept operators and kernels from different computing platforms with corresponding language backend.

2. More mutation operators and benchmark kernels. We would like to explore more mutation operators and kernels to test them using our tool, making our tool more comprehensive and powerful.

Bibliography

[1] Hiralal Agrawal, Richard A DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, EW Krauser, RJ Martin, and Aditya P Mathur. Design of mutant operators for the c programming language. 1989.

[2] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006. doi:10.1109/TSE.2006.83.

[3] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: A verifier for gpu kernels. *SIGPLAN Not.*, 47(10):113–132, oct 2012. doi:10.1145/2398857.2384625.

[4] Javier Cabezas. cuda4cpu, 2023. URL: https://github.com/javier-cabezas/cuda4cpu.

[5] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 597–608, 2017. doi:10.1109/ICSE.2017.61.

[6] Henry Coles. Pit, 2023. URL: https://pitest.org/quickstart/mutators/.

[7] NVIDIA Corporation. Cuda tookit, 2023. URL: https://developer.nvidia.com/cuda-toolkit.

[8] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. Assessment of class mutation operators for c++ with the mucpp mutation system. *Information and Software Technology*, 81:169–184, 2017. doi:10.1016/j.infsof.2016.07.002.

[9] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. doi:10.1109/C-M.1978.218136.

[10] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. Mutation operators for testing android apps. *Information and Software Technology*, 81:154–168, 2017. doi:10.1016/j.infsof.2016.04.012.

[11] Xavier Devroey, Gilles Perrouin, Axel Legay, Maxime Cordy, Pierre-Yves Schobbens, and Patrick Heymans. Coverage criteria for behavioural testing of software product lines. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014,*

Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I 6, pages 336–350. Springer, 2014.

[12] Vector Informatik GmbH. Code coverage for cuda code using vectorcast/qa, 2023. URL: https://www.vector.com/int/en/events/global-de-en/webinar-recordings/2021/coffee-with-vector-code-coverage-for-cuda-code-using-vectorcastqa/.

[13] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 189–200, 2014. doi:10.1109/ISSRE.2014.40.

[14] Khronos Group. Opencl-guide, 2023. URL: https://github.com/KhronosGroup/OpenCL-Guide.

[15] gtcasl. Gpu ocelot project, 2023. URL: https://github.com/gtcasl/gpuocelot.

[16] Ben van Werkhoven Hanno Spreeuw. Kernel-tuning-for-sagecal, 2023. URL: https://github.com/HannoSpreeuw/Kernel-tuning-for-Sagecal.

[17] Mark Harris. How to implement performance metrics in cuda c/c++, 2012. URL: https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/.

[18] Intel. Opencl runtimes for intel processors, 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/opencl-drivers.html.

[19] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi:10.1109/TSE.2010.62.

[20] Lavkush. Code coverage using nvcc compiler, 2023. URL: https://forums.developer.nvidia.com/t/code-coverage-using-nvcc-compiler/46623.

[21] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying gpu kernels by test amplification. *SIGPLAN Not.*, 47(6):383–394, jun 2012. doi:10.1145/2345156.2254110.

[22] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. Gklee: Concolic verification and test generation for gpus. *SIGPLAN Not.*, 47(8):215–224, feb 2012. doi:10.1145/2370036.2145844.

[23] Wentao Li, Zhiwen Chen, Xin He, Guoyun Duan, Jianhua Sun, and Hao Chen. Cvfuzz: Detecting complexity vulnerabilities in opencl kernels via automated pathological input generation. *Future Generation Computer Systems*, 127:384–395, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0167739X21003526, doi:10.1016/j.future.2021.09.006.

[24] NVIDIA. Cuda c programming guide, 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html.

[25] NVIDIA. Cuda samples, 2023. URL: https://github.com/nvidia/cuda-samples.

[26] NVIDIA. Nvidia developer forums, 2023. URL: https://forums.developer.nvidia.com.

[27] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, jan 1992. doi:10.1145/125489.125473.

[28] Stack Overflow. Stack overflow, 2023. URL: https://stackoverflow.com.

[29] Anmol Panda, Philipp Rummer, and Neena Goveas. A comparative study of gpu verify and gklee. page 112 – 117, 2016. Cited by: 0. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85019179747&doi=10.1109%2fPDGC.2016.7913126&partnerID=40&md5=56edde052ed9533c7f62baa64be03637, doi:10.1109/PDGC.2016.7913126.

[30] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275–378. Elsevier, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0065245818300305, doi:10.1016/bs.adcom.2018.03.015.

[31] Chao Peng. Check opencl kernel code coverages, 2023. URL: https://github.com/chao-peng/clcov.

[32] Chao Peng and Ajitha Rajan. *CLTestCheck: Measuring Test Effectiveness for GPU Kernels*, pages 315–331. 04 2019. doi:10.1007/978-3-030-16722-6_19.

[33] Eike Stein, Steffen Herbold, Fabian Trautsch, and Jens Grabowski. A new perspective on the competent programmer hypothesis through the reproduction of bugs with repeated mutations. *CoRR*, abs/2104.02517, 2021. URL: https://arxiv.org/abs/2104.02517, arXiv:2104.02517.

[34] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66 – 72, 2010. doi:10.1109/MCSE.2010.69.

[35] Xiaofan Sun and Rajiv Gupta. Dsgen: Concolic testing gpu implementations of concurrent dynamic data structures. page 75 – 87, 2021. Cited by: 0; All Open Access, Bronze Open Access. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85107484321&doi=10.1145%2f3447818.3460962&partnerID=40&md5=29e960f7d5cacdf9bd4286adaa5b9b40, doi:10.1145/3447818.3460962.

[36] Info Support. Stryker mutator, 2023. URL: https://stryker-mutator.io/docs/stryker-net/mutations/.

[37] Hamid Tabani, Leonidas Kosmidis, Jaume Abella, Francisco Cazorla, and Guillem Bernat. Assessing the adherence of an industrial autonomous driving framework to iso 26262 software guidelines. pages 1–6, 06 2019. doi:10.1145/3316781.3317779.

[38] Ben van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019. doi:10.1016/j.future.2018.08.004.

[39] Ben van Werkhoven. Writing testable gpu code, 2023. URL: https://blog.esciencecenter.nl/writing-testable-gpu-code-23bbda3a5d62.

[40] Qianqian Zhu and Andy Zaidman. Massively parallel, highly efficient, but what about the test suite quality? applying mutation testing to gpu programs. page

209 – 219, 2020. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85091553948&doi=10.1109%2fICST46399.2020.00030&partnerID=40&md5=1ef7d17f6ecb93617ffcea0923aae63b, doi:10.1109/ICST46399.2020.00030.