Automated Test Execution Using GPUs

Vanya Yaneva

Master of Science by Research

Laboratory for Foundations of Computer Science

CDT in Pervasive Parallelism

School of Informatics

University of Edinburgh

2016

Abstract

Functional software testing is a critical task in software engineering, involving the execution of a large number of test cases for any non-trivial system. This can be extremely time consuming, making functional testing incompatible with short development cycles. Accelerating test execution by running test cases in parallel on the GPU threads has been proposed before, but GPU limitations pose challenges to this approach's ease of use, scope and effectiveness.

This project propses automating test execution on the GPU as a way to increase its ease of use and scope. Two systems are developed to facilitate automation. The first is a code generation tool, called ParTeCL, which generates OpenCL kernel for the tested program, allowing its execution on the GPU. The second is a CPU runtime, which executes the test cases by building and launching the auto-generated kernel in parallel on the GPU threads.

Four benchmark applications from two repositories, containing different C features, are tested using this approach, demonstrating that ParTeCL generates valid OpenCL kernels and that testing results produced on the GPU are the same as those output by the CPU. In addition, a usability study with six programmers is performed, showing that automation greatly simplifies the testing process. Finally, preliminary performance results from two benchmarks are gathered and analysed, demonstrating speed-up of up to 90x for one of the applications and identifying optimisation areas to focus on in future research, in order to increase the approach's effectiveness.

Acknowledgements

I would like to thank my primary supervisor, Dr. Ajitha Rajan, for allowing me to work on this exciting project and providing me with plenty of support and direction in establishing its scope, goals and methodology. Her guidance has been invaluable.

I would also like to thank my secondary supervisor, Dr. Christophe Dubach, for helping me get through all the technical challenges I encountered, surrounding programming for the GPU, and for giving me guidance in collecting and interpreting performance data.

Special thanks to Fraser and Christopher Cormack for helping me name ParTeCL.

Finally, Thank You to all my coursemates from the 2015 cohort of the CDT in Pervasive Parallelism for making the office an extremely fun and supporting environment to work in. I will miss being in it with all of you.

This work was supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh.

Declaration

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

(*Vanya Yaneva*)

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In software engineering, careful testing and verification of the developed system is a crucial task, which provides confidence in its correctness and quality. However, rigorous testing of any non-trivial system involves the generation and execution of a huge number of test cases. This can be extremely time consuming, taking hours, days and even weeks, making testing impractical and even incompatible with development schedules. Therefore, accelerating the execution of large software test suites is an important problem and an active area of research.

This Master's project focuses on the acceleration of the execution of *functional* software tests - the tests which verify that the system behaves as intended. The current chapter introduces the challenges behind the acceleration of test execution and presents the solution proposed in this project, its goals and its contributions. Finally, it outlines the rest of the dissertation.

1.1 Challenges

Functional software testing involves the execution of the tested functionality with different inputs, which constitute the test cases, and checking that the results are as expected. For large systems, thorough verification requires a huge number of tests cases, ideally covering the whole input space, whose execution time on a single processing unit scales linearly with the size of the test suite, resulting in long testing times.

To accelerate the execution of the test suite, two characteristics of functional testing can be exploited:

- Tests are *data parallel*, as they involve the execution of the same functionality multiple times on different data inputs.
- Test executions are *independent*, as the results of one test do not affect the results of others.

These characteristics make functional tests an ideal candidate for *parallel* executions. **Tests can be simultaneously executed on separate processing units, reducing total execution time.**

Indeed, parallelisation of test executions exists as a practice [7]. Companies provision hardware infrastructure for testing and tools to facilitate parallel test runs are being developed. However, the acceleration achieved is limited by the level of parallelism available in the hardware. Executing all tests in a test suite simultaneously requires as many processing units as there are cores. Test suites consisting of thousands of tests require the equivalent of a super computer. This scaling can get prohibitively expensive for many companies. What is more, even when the infrastructure is available, some companies report that up to 90% of it can remain idle [17].

1.2 Proposed Solution

This dissertation examines the use of Graphics Processing Units (GPUs) for parallel execution of software tests. This approach runs all test cases in parallel, each on a separate GPU thread. GPUs are both readily available in many desktop systems, thus considerably cheaper than a parallel CPU testing infrastructure, and capable of executing thousands of threads simultaneously. In addition, the type of parallelism exhibited by testing, as seen earlier in Section 1.1, is a natural fit for the GPU hardware. GPUs have *Single Instruction Multiple Data (SIMD)* architecture, designed for the independent execution of the same instructions over multiple data inputs.

This approach is first proposed in [25]. The paper establishes the feasibility of the idea by analysing the results produced when testing four embedded systems programs on the GPU. It demonstrates speed-ups of up to 27x when compared to execution on the CPU.

However, [25] also recognises that GPUs have a number of limitations which pose challenges for the proposed approach:

1. Programming GPUs requires the use of a specialist low-level programming model, such as OpenCL [1] and CUDA [2], thus it is not available to every programmer.
2. Not all C/C++ features can be readily compiled for execution on the GPU, limiting the scope of applications which can be tested using GPUs. Unsupported features include *standard library calls*, *recursion* and *dynamic memory allocation*.
3. The achieved speed-up is limited by factors, such as the chosen GPU block/grid dimensions, control-flow divergence and data transfers.

Hypothesis

This Master's project focuses on challenges 1. and 2. It hypothesises that:

Launching tests on the GPU can be ***automated****, which will make it more accessible to general developers, as it will abstract away the low-level programming model.*

A tool is built to automatically generate an OpenCL wrapper for C programs, which allows their execution on the GPU threads. The tool also performs code transformations for C features which are not supported by the OpenCL compiler.

Thus, it achieves two goals:

- ease of use, as it alleviates the programmer from the need to write low-level GPU code
- increased scope, as it implements code transformations for C features which are not readily supported for execution on the GPU.

The tool's name is **ParTeCL**, standing for **Par**allel **Te**sting in Open**CL**. It is implemented using the Clang compiler's LibTooling library [3].

In addition, generic CPU runtime code is developed to build the OpenCL program output by ParTeCL and launch it together with the test cases in parallel on the GPU threads.

Finally, the project collects and analyses data for the speed-ups achieved when executing test cases on GPU, using the tools developed during the project. This analysis is used as a basis to plan future steps in addressing challenge 3. in the context of PhD work.

1.3 Project Goals

This project sets out to achieve the following particular goals:

1. Demonstrate that automation of test launching on the GPU is feasible and produces correct testing results.
2. Demonstrate that automation allows users to execute tests on the GPU without the need to understand GPU programming and write OpenCL code.
3. Investigate what performance can be expected and what optimisations may be beneficial.

1.4 Project Contributions

The project makes the following contributions:

- ParTeCL - a tool which automatically generates OpenCL code for the purposes of testing; it performs code transformations for features unsupported by the OpenCL compiler.
- A CPU runtime, which reads test cases, builds the OpenCL program, generated by ParTeCL, and launches its test cases in parallel on the GPU threads.
- Evaluation on four benchmark applications, demonstrating the following:
  - ParTeCL generates valid OpenCL kernels, which can be built and executed on the GPU without the need to manually edit OpenCL code.
  - Testing on the GPU, using the auto-generated OpenCL kernels, produces the same results as testing on the CPU.
- Performance analysis on the speed-ups achieved, using the auto-generated OpenCL kernels.

1.5 Dissertation Outline

The rest of the dissertation describes the work undertaken to achieve the above goals.

In particular, Chapter 2 provides relevant background information on functional testing, GPU architecture and programming models and the Clang LibTooling library.

Chapter 3 presents existing work in the areas of *test case acceleration*, *parallel execution of tests on the GPU* and *auto-generation of GPU code*, and relates it to the goals of this project.

Chapter 4 presents the process of executing tests on the GPU and uses this to introduce the design and implementation of ParTeCL and the CPU runtime.

In chapter 5, four benchmark applications are used to evaluate the feasibility and correctness of using the developed systems for test execution on the GPU. Performance data from two of them is gathered and analysed in order to investigate areas for optimisation. A usability study with six software developers is carried out to evaluate the ease of use of the proposed automatic approach.

Finally, Chapter 6 summarises the work carried out in this dissertation, suggests future work for the PhD phase of the project, and briefly reflects on the achievements of the project and lessons learnt during it.

Chapter 2

Background

To allow full understanding of the problem and solutions investigated in this Master's project, the current chapter provides relevant background information. It starts by describing functional testing in detail, the importance of an exhaustive test suite and the testing practices in which it is used. It briefly describes the GPU architecture and programming models, focusing on aspects most relevant to testing. Finally, it introduces the Clang compiler and LibTooling library [3] which are used in the implementation of ParTeCL.

2.1 Functional Testing

Functional testing is a method of testing, based on the functional specifications for the system's behaviour [37]. It aims to verify that the program behaves as intended by executing it using different inputs and checking that the outputs are as expected. Based entirely on the values of the inputs and outputs, functional testing does not involve the program's design or implementation structure. Thus, it is also referred to as *black-box* testing.

Figure 2.1 illustrates functional testing. A test suite consists of test cases, each of which is a different input to the tested program. The program is executed multiple times on the different inputs, and the produced outputs are examined. As discussed in Sections 1.1 and 1.2, the *data parallelism* and *independence* of test executions make functional testing suitable for parallelising on the GPU.

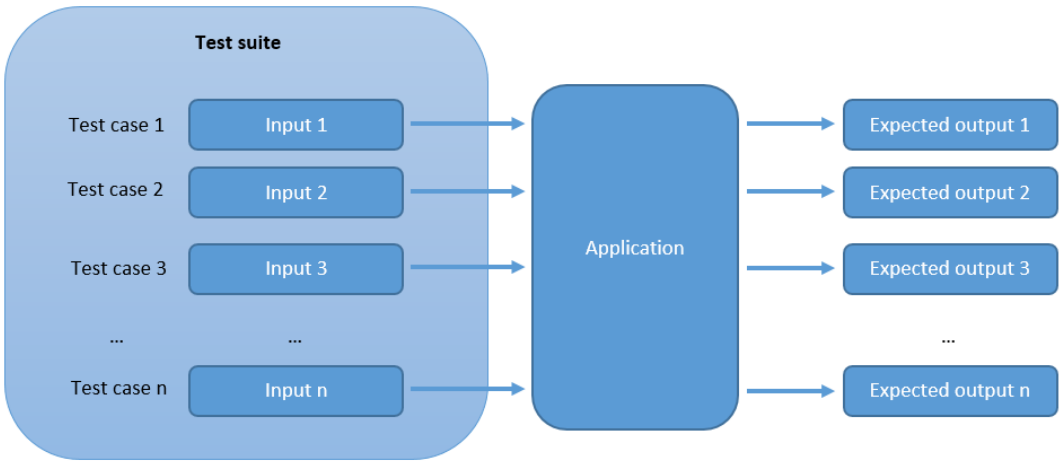

Figure 2.1: Functional testing.

There are many methods for the generation of functional tests, described in [37], whose aim is to discover faults in the implemented system. Theoretically, a full test suite that verifies the correctness of the system's behaviour will constitute of the entire input space of a program. In practice, this is infeasible to generate and execute, as even trivial programs would have many millions test cases. For example, a single 32-bit integer input has 2^{32} possible values. Thus, analytical processes for test case generation are employed. They involve partitioning of the input space based on all possible behaviours of the program and selecting representative test cases, which are likely to discover faults. As [37] states, this is a human-intensive and error-prone process, as there is always the chance of missing an important test case. Intuitively, the finer the partitioning and the larger the test suite, the higher its fault finding potential.

Since functional testing can be applied at any level of system granularity, from individual functions through modules to entire programs, it is often used in multiple testing practices, including *unit testing*, *system testing* and *regression testing*. Unit testing checks the behaviour of the smallest functional units of a program, usually individual functions and methods, while system testing checks the behaviour of the system as a whole. Regression testing is a specialised instance of system testing, which focuses on ensuring that changes of the existing code do not introduce new faults in the system.

In modern software development practices, regression testing is used regularly, as soon as a change to the program is made. Development teams often employ overnight builds and test runs, making the need of time efficient test executions even greater. Thus, the current project focuses on system testing, but the approaches used in it can also be extended to finer granularity.

2.2 GPU Architecture and Programming Models

GPUs are a parallel accelerators, used in High Performance Computing (HPC). Originally designed for graphics computations, they have been successfully employed in some areas of general purpose computing.

Generally, GPUs consist of one or more *compute units* (aka *streaming multiprocessors*), which in turn contain one or more *processing elements* (aka *streaming processors*). The processing elements execute the individual threads. The functions executed by the GPU threads are called *kernels*. In the GPU programming models, each thread executes the same kernel with different input data, which fits the execution of functional software tests, as seen in Section 2.1.

As compute units share the same instruction counter, execution on a single compute unit is done in *lock-step*, meaning that each thread executes the same instruction at all times. If there is control-flow divergence across threads within the compute unit, divergent instructions will be serialised, impacting performance negatively. This has implications for testing, when test cases take different control-flow paths in the tested program. While researching methods to mitigate this effect is outside the scope of the current project, it is planned to be included in future PhD work (Section 6.3).

GPUs have a memory hierarchy:

- **Global memory.** Large and slow, allocated by the CPU. Accessed by all threads in all compute units.
- **Constant memory.** Like global memory, but allows only read access.
- **Local memory.** Local to compute units, shared among the threads in a single unit.
- **Private memory.** Private to individual threads.

Memory is explicitly managed int the CPU and GPU code and this can have significant impact on performance.

The two most widely used programming models for GPU programming are CUDA [2] and OpenCL [1]. Based on the C/C++ programming languages, they expose low-level hardware details, which require the programmer to explicitly express the parallelism in their algorithms. The mapping of parallelism to compute units and threads is also explicitly managed. To do it in OpenCL, the programmer supplies *work-group* dimensions for the kernel execution, which can have a huge impact on the performance achieved on particular GPU architectures.

OpenCL is the programming model chosen for this project, since it provides cross-platform functional portability, which could potentially enable future research on testing using different accelerator architectures. The C-like language means that to translate a C program into an OpenCL kernel, generally, only a simple decoration of the code with OpenCL attributes is necessary.

However, GPUs are specialised and not all C/C++ features are supported by the OpenCL compiler, which has implications for the scope of C applications which can be tested on the GPU. ParTeCL's design takes this into consideration and includes methods for to handle it. Particular discussion on this is included in Section 4.4.

2.3 Clang LibTooling

LibTooling [3] is a C++ infrastructure, aimed at the development of standalone front-end compiler tools, built on top of the Clang compiler.

LibTooling uses Clang to build the program's Abstract Syntax Tree (AST) and provides two interfaces for using the AST - **RecursiveASTVisitor** and **AST Matchers**. As its name suggests, the RecursiveASTVisitor allows the recursive traversal of the AST. It provides hooks which visit the AST nodes, based on their types, and allow the developer to implement particular actions for specific node types. In contrast, The AST Matchers allow the programmer to find AST nodes, based on specific patterns, which are described using a Domain Specific Language (DSL). The AST Matchers are usually implemented together with corresponding AST Handlers, which contain the actions the tool takes for the matched AST nodes.

ParTeCL uses the AST Matchers and AST Handlers. This is presented in Section 4.5.

Finally, LibTooling provides a **Rewriter** class, which facilitates source code changes, by exposing deletions, insertions and replacements in the original source code, based on source locations, which are part of the Clang AST nodes.

2.4 Summary

This chapter presented brief background information on functional testing, the GPU architecture and programming models and the Clang LibTooling library [3].

The information on functional testing aims to aid better understanding of its importance and the reasons which make it time consuming. The description of functional tests in particular should be a useful background for understanding how they are input and executed by the automatic systems developed in this project.

The brief introduction to GPU architecture and programming models aims to help the reader understand the particular approaches to test execution on the GPU employed in this project, as well as the design decisions behind the developed systems and the performance challenges, which can be encountered.

Finally, as LibTooling is used for the implementation of ParTeCL, it was briefly described in order to show how it fits the purposes of the tool.

Chapter 3

Related Work

The problems addressed in this project and the proposed solutions fall within three separate areas of research. The broad problem belongs to the area of *acceleration of software testing*. The proposed solution, *execution of test cases in parallel on the GPU threads*, is a novel approach to this problem, recently presented in [25], but other approaches which make use of GPUs also exist. Finally, this project's main contribution, ParTeCL, relates to research in *auto-generation of GPU code*.

This chapter provides an overview of the existing work in all three areas and discusses and the ways in which it relates to the work carried out in this project.

3.1 Acceleration of Software Testing

Traditional approaches to acceleration of software testing include *test suite minimisation*, *test case selection* and *test case prioritisation*. A comprehensive survey is presented in [35]. The current section presents only a brief summary to enable comparison with the approach presented in this dissertation.

The particular goals of the existing approaches are:

- remove obsolete or redundant test cases (test suite minimisation)
- choose a subset of test cases to execute in a particular scenario (test case selection)

- reorder the test suite according to some desirable property (test case prioritisation)

Test suite minimisation relies on heuristics to determine the minimum number of test cases which satisfy some criteria, usually a measurement of code coverage. A large body of work exists in identifying and evaluating different heuristics and algorithms [14, 34, 31], as well as optimising their fault detection [19].

Test case selection is a technique similar to test suite minimisation, but instead of focusing on a single version of the tested program, it aims to select test cases covering the changes between the current and a previous version of the application. Particular methods have been developed based on multiple techniques and criteria, including data-flow analysis, program dependence graphs and code coverage [12, 29, 22].

Finally, **test case prioritisation** reorders test cases based on some desirable criterion, so that testing yields maximum useful results as early in the testing process as possible. The desirable criteria include structural coverage, rate of fault detection and specification requirements [30, 10, 21].

Existing work also combines some of the techniques in order to achieve test suites with better prioritisation, reaching code coverage faster, and faster testing time, due to minimisation [32].

As traditional approaches involve the use of heuristic algorithms, whose execution time grows with the size of the test suite, GPUs have been used to accelerate some of them. In particular, [36] presents a modified parallel evolutionary algorithm for test suite minimisation. It is implemented in CUDA and executed on an Nvidia GPU, achieving a speed-up of up to 25x for large applications. Another example is [38], which uses GPUs to accelerate the generation of new test cases.

A common aspect of all of the above approaches is the fact that they manipulate the test suite. While they achieve success in reducing test case numbers, and therefore overall testing time, they can have a detrimental effect on the effectiveness of testing. This has been observed in multiple empirical studies, which discover that the fault finding capabilities of a test suite correlate to its size [18] and removing test cases from the testing process leads to reduced fault detection [28, 13], making the existence of large test suites, as well as the execution of all test cases, highly desirable.

3.2 Parallel Test Case Execution on the GPU

In contrast to the approaches examined in Section 3.1, the current project suggests executing the entire test suite on the GPU by running all test cases in parallel on the GPU threads, thus reducing total execution time.

As discussed in Chapter 1, [25] presents the approach and performs preliminary studies on four benchmark applications from the embedded-systems domain. The paper makes four notable contributions:

- It shows speed-ups between 2 and 27 times in comparison to execution on the CPU.
- It shows that the outputs achieved by the executions on the GPU are the same as those achieved on the CPU.
- It shows that factors such as *the chosen grid and block dimensions*, *the time for data transfer between the host and device* and *the degree of control flow divergence* in the tested program play an important role in the achieved speed-up.
- It identifies the GPU limitations presented in Section 1.2, which pose challenges to the scope, effectiveness and ease of using GPUs for test execution.

This is a novel and promising idea and the current project continues this work by addressing some of the challenges and proposing solutions to others. In particular, it proposes automatic generation of the low-level code, which executes the tests in parallel on the GPU.

3.3 Code Generation for the GPU

Even outside of testing, GPU programming poses many challenges for the developer, both in terms of programmability and performance. The use of low-level programming models, such as CUDA and OpenCL, requires familiarity with the architecture in order to write correct parallel code, and fine-grained optimisations in order to reach the full performance potential of the GPU. Previous research addresses these challenges by proposing high-level programming frameworks, compilers and code generation tools. The current section briefly presents some of them.

For example, [33, 26] introduce and evaluate a framework, which automatically generates low-level OpenCL code from high-level parallel primitives. The work defines the primitives, which correspond to parallel functionalities, e.g. *map* and *reduce*, as well as a functional-style programming language which is used by the programmer to express parallel algorithms.

Another example is SYCL [4], which provides a high level-abstraction of OpenCL to allow programmers to write GPU code in standard C++. It includes C++ templates and libraries, which express parallel functions, and SYCL compilers which generate code executable on the GPU.

Purely compiler-based approaches include [8], targeting CUDA, and [11], aimed at lower-level code generation. Both of them use the polyhedral model for loop parallelisation in order to transform portions of the program into parallel code to be executed on the GPU.

The existing tools and frameworks provide high-level mechanisms to both express and discover parallelism and generate efficient parallel code for the GPU. In contrast, the parallelism inherent to functional testing is straightforward - the test cases are mapped to the GPU threads, which execute separate instances of the tested program. The code generation tool needs to wrap the tested functionality into an OpenCL kernel and transform C features for compilation by the OpenCL compiler. In other words, the tool developed in this project has a goal different to those of the existing solutions. While they aim to generate performant parallel code, this project's goal is to transform an application written for the CPU into a GPU kernel.

3.4 Summary

This chapter presented existing approaches to accelerating test execution, as well as the ways in which GPUs have been applied to them. These approaches focus on manipulating the test suite, either by minimising it, or by prioritising and selecting test cases for execution. However, they have been shown to reduce the effectiveness of testing, demonstrating that executing the whole test suite is highly desirable.

In contrast, the proposed approach executes the entire test suite and reduces total execution time by running test cases in parallel on the GPU threads. Its feasibility has been established in previous work, but limitations of the GPU architecture and programming models pose challenges to its effectiveness, scope and ease of use.

The current work addresses some of these challenges by proposing auto-generation of the OpenCL code which is executed on the GPU threads. Whereas previous research in code generation for the GPU proposes high-level frameworks for the expression of parallelism, the parallelism inherent in software testing is simple. What is required is translating the tested program in OpenCL to allow its execution on the GPU. Thus, a new tool, ParTeCL, to perform these modifications is developed in this project.

Chapter 4

Design & Implementation

The current chapter describes the design and implementation decisions behind ParTeCL and the CPU runtime, which launches the test cases in parallel on the GPU threads. It starts with a high level overview of the process of testing on the GPU and the two systems involved in it. It continues with descriptions of the individual inputs and outputs of ParTeCL and of the CPU runtime. The chapter then discusses the ways in which various C features are supported for execution on the GPU - either readily, out of the box, or through code transformations performed automatically by ParTeCL. It also presents C features which currently aren't supported and discusses ways in which they can be addressed in the future. Finally, the chapter describes the implementation of ParTeCL, based on the Clang compiler.

Example Program: add.c

To illustrate the design and implementation details presented in this chapter, a simple example is used throughout it.

```
int add(int a, int b){
  return a + b;
}

int main(int argc, char* argv[]){
  if(argc <= 2){
    printf("Please, provide two integers.\n");
    return 0;
  }

  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  printf("%d + %d = %d\n", a, b, add(a, b));
}
```

This program takes two integers command line arguments, adds them together, and prints the result to standard output.

A functional test case for this program will provide any two input integers to check that the program indeed computes their sum correctly.

4.1 High-Level Overview

As seen Section 2.1, executing functional tests on the CPU consists of writing of the test cases as inputs to the tested program, building and executing the program on the test case inputs and checking that results are as expected.

In contrast, executing tests on the GPU requires the following steps:

- Translate the tested program into an OpenCL kernel.
- Write a CPU runtime to
  - transfer the test cases to the GPU memory.
  - build the OpenCL kernel.
  - launch the tests in parallel on the GPU threads.
  - transfer the testing results back to the CPU memory and check that they are as expected.
- Build and execute the CPU runtime.

Thus, there are two major differences in the testing process:

1. The tester no longer builds and executes the tested program. They translate it into an OpenCL kernel instead and build and execute the CPU runtime.
2. The test cases are no longer taken directly as inputs by the tested program. They are read by the CPU runtime and transferred to the GPU memory, where the OpenCL kernel uses them as inputs. Thus, they need to to be written in a format,

which is parsed by the CPU runtime. For this project, the format chosen is CSV. It is presented in detail in Section 4.3.

This project provides two systems, which automate the testing process and abstract away the GPU programming details. They are:

- **ParTeCL - a code generation tool**, presented in Section 4.2.
  inputs: the source code of the tested program and a configuration file.
  functions: generates the OpenCL kernel, executed by the GPU threads, the data structures used to transfer test cases and results between the CPU and GPU memories and a small part of the CPU runtime.

- **CPU runtime - a test execution system**, presented in Section 4.3.
  inputs: test cases, in CVS format, and OpenCL kernel, generated by ParTeCL.
  functions: read the test cases and transfer them to the GPU memory; build the OpenCL kernel and launch it in parallel on the GPU threads; transfer testing results back to the CPU for inspection.

Figure 4.1 illustrates the two systems.

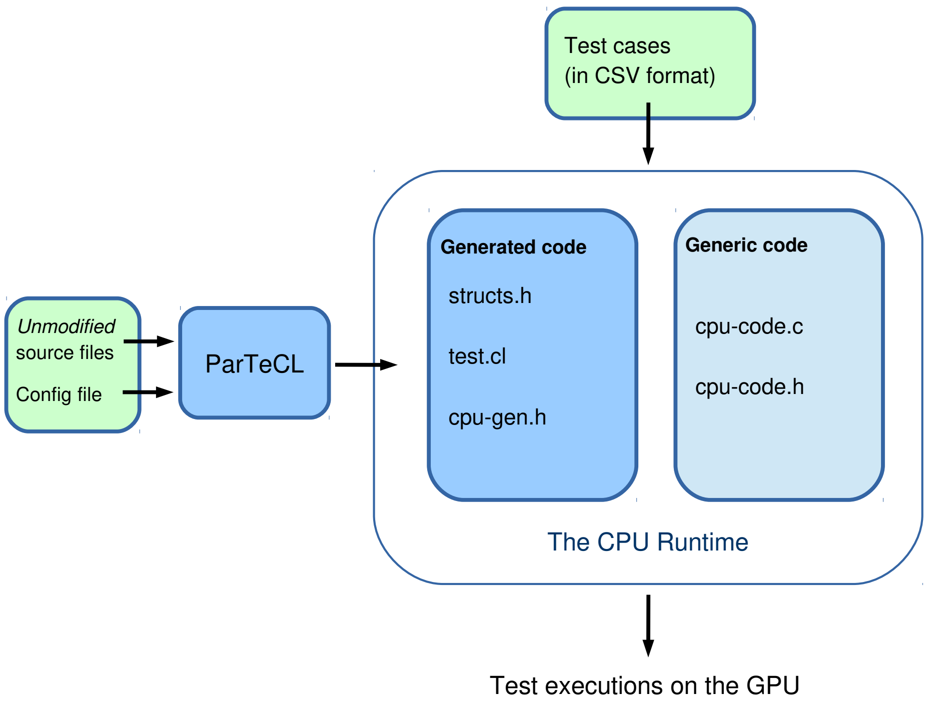

Figure 4.1: Parallel Testing on the GPU: automation

4.2 ParTeCL

The current section describes ParTeCL by presenting its inputs and outputs. Implementation details are presented in Section 4.5.

4.2.1 Inputs

ParTeCL takes two inputs, supplied by the user:

- the *unmodified* C source code of the program which is being tested
- a configuration file, which describes the test cases.

The unmodified C source code is used to generate the OpenCL kernel, which executes on the GPU. ParTeCL changes the signature of the `main` function to an OpenCL kernel function and performs automatic transformations on the code, which are described in Sections 4.2.2 and 4.4.

The configuration file provides information to describe the testing setup. It is necessary for the generation of the data structures, which are used to pass test case data between CPU and GPU memory. The data structures are presented in Section 4.2.2.

In particular, the configuration file contains the following:

1. the name of the function, which is being tested
2. description of the inputs to the tested program
3. description the expected result

ParTeCL parses the configuration file, line by line, with the help of the keywords shown in Table 4.1.

For example, the configuration file for the **add.c** program would look like this:

```
function-name: add RET
input: int a 1
input: int b 2
result: int c
```

It tells ParTeCL that the function to test is **add** and it *returns* the result, which is being tested (the **RET** keyword). If, for example, if **add** didn't return the result, but

Keyword	Usage	Examples
function-name	The name of the C function which is being tested.	
RET	The tested function *returns* the tested output.	`function-name: add RET`
ARG	The tested function writes the tested result in one of its arguments; followed by an integer index to specify which argument.	`function-name: add ARG 3`
input	A command line input: type, name and index on the command line.	`input: int a 1`
stdin	Standard input inputs (`stdin`). They need to be supplied by the user in a single string, separated by a new line character. More details in Section 4.2.2.	`stdin: char* stdin1`
result	The result that is being tested: type and name.	`result: int c`

Table 4.1: Configuration file keywords.

recorded it in the third argument passed to it instead, then line 1 would have been `function-name: add ARG 3`. The file also tells ParTeCL that the program takes two command line inputs, **a** and **b**, each an integer. They have command line indices 1 and 2 respectively, corresponding to **argv[1]** and **argv[2]** in the original C code. Finally, the configuration file tells ParTeCL that there is a single result to be stored in a variable called **c**, which is an integer. The **add.c** program does not take inputs through `stdin`, so the **stdin** keyword is not used.

4.2.2 Outputs

In addition to the OpenCL kernel, ParTeCL generates a part of the CPU runtime. ParTeCL's outputs are presented in this section, while the rest of the CPU runtime is presented Section 4.3.

In particular, ParTeCL generates three distinct outputs:

- Data structures used to transfer data between CPU and GPU memory
- CPU code to assign test case values to the input data structures
- Kernel code, which executes the tested program on the GPU

Data Structures - structs.h

In order for the GPU to execute test cases, the CPU code transfers them to the GPU memory, where the kernel code uses them as inputs. Each GPU thread reads and executes a different test case. Similarly, once the tests are executed, the results are transferred back to the CPU memory, where the CPU code can check if they are correct.

To store the values for the transfers, data structures generated by ParTeCL are used. The tool uses the configuration file, described in Section 4.2.1, to generate the structures and write them in file **structs.h**. There are always two data structures - **input** and **result**. Each of them contains the variables described in the configuration file, as well as a variable `test_case_num` to record the test case id. **input** also contains the variable `argc`, which corresponds to the number of command line input variables.

Thus, for the **add.c** program, the structs file looks like this:

```
typedef struct input{
  int test_case_num;
  int argc;
  int a;
  int b;
} input;

typedef struct result{
  int test_case_num;
  int c;
} result;
```

CPU Runtime Code - cpu-gen.c

ParTeCL generates a single function for the CPU runtime, which, given values for the test case inputs, assigns them to the variables in the **input** structure. As the **input** structure is auto-generated by ParTeCL, so is the function which assigns values to it.

For the **add.c** program, **cpu-gen.c** contains the following:

```
void populate_inputs(
            struct input *input,
            int argc, char** args,
            char* stdins)
{
  (*input).test_case_num = atoi(args[0]);
  (*input).argc = argc;
  if(argc >= 2)
    (*input).a = atoi(args[1]);
  if(argc >= 3)
    (*input).b = atoi(args[2]);
}
```

This function is called by the generic CPU code, described in Section 4.3. Before calling it, the CPU reads the test case values from the CSV file, supplied by the user, and then passes them to the function as the argument `char** args`. The function also takes argument `char* stdins` which is used when testing programs which take inputs through standard input. **add.c** does not use standard input, so in this example `stdins` is not used and it would be NULL at runtime.

Kernel Code - test.cl

The kernel code is generated from the unmodified source code of the tested C program. The tool turns the C source into OpenCL code, by turning the `main` function into an OpenCL kernel function. It also modifies the way inputs are being read, by replacing references to `argv` and standard input with references to the auto-generated **input** structure.

To illustrate this, consider the resulting OpenCL code for **add.c**:

```
int add(int a, int b){
  return a + b;
}

__kernel void main_kernel(
         __global struct input* inputs,
7          __global struct result* results)
8  {
9    int idx = get_global_id(0);
10   struct input input = inputs[idx];
11   int argc = input.argc;
12   results[idx].test_case_num = input.test_case_num;
13
14   if(argc <= 2){
15     printf("Please, provide two integers.\n");
16     return 0;
17   }
18
19   int a = input.a;
20   int b = input.b;
21   /*printf("%d + %d = %d\n", a, b, add(a, b));*/
22   results[idx].c = add(a, b);
23 }
```

The tool has made the following modifications to the original C code:

- Lines 5, 6 and 7: the `main` function is turned into an OpenCL kernel function, via the `__kernel` attribute. It is renamed to `main_kernel`. The arguments, `argv` and `argc` are also changed. They are replaced by two arrays
  - **__global struct input* inputs**: an array of **input** structures, generated by the tool; it contains the program's test cases and is populated and transferred to the GPU's global memory by the CPU code.
  - **__global struct input* inputs**: an array of **result** structures, generated by the tool; the GPU threads write the results of the tested function to it and the CPU then transfers them to its own memory.
- Lines 9, 10, 11 and 12: these lines are always inserted in the beginning of the kernel function. They determine, respectively:
  - **int idx**: the thread index of the current GPU thread; it determines which elements of **inputs** and **results** correspond to the current thread's test case.
  - **struct input input**: the element of **inputs** which has the current thread's test case inputs.
  - **int argc**: the number of inputs in the current test case.

  - the test case id of the current thread's test case; it is recorded in the corresponding element of **results**.

- Lines 19 and 20: the references to **argv** in the original source code are replaced with reads from the **input** structure.

- Line 21: the tool comments out any prints to standard output, as, generally, they are not required for testing.

- Line 22: the result from the tested function **add**, as specified in the configuration file, is assigned to a variable in the **result** structure corresponding to the current test case; ParTeCL scans the original code for function calls to the tested function and replaces them with the line which assigns their results to the **result** structure.

This simple example demonstrates the general code changes which ParTeCL performs to the original C source code in order to turn it into OpenCL kernel code. What is important to note is that the changes affect only the way in which the program handles inputs and outputs. *The original functionality remains unchanged in order to ensure that it is the one being tested.*

For less trivial applications, ParTeCL performs additional code transformations. They are discussed in Section 4.4.

4.3 CPU Runtime

The CPU runtime defines the behaviour of the CPU, which builds the tested program for execution on the GPU and launches its test cases on the GPU threads. It consists mostly of generic code which is always the same, irrespective of which program is being tested. The only exception is the CPU function which assigns test case values to the inputs structures, described earlier in Section 4.2.2.

The current section presents the tasks performed by the CPU runtime. It also describes the CSV format defined for the file, which contains the values for the test cases. This file is an input to the CPU runtime.

4.3.1 Generic CPU Code

The CPU runtime performs mostly general tasks which facilitate execution on the GPU, as well as some tasks with particular significance to testing. The latter are included in bald in the list below and discussed in more detail after it.

In particular, the code performs the following tasks:

1. Allocate memory for the **input** and **result** structures on the CPU and the GPU.
2. Declare the OpenCL command queue and create the OpenCL context.
3. Build the auto-generated OpenCL kernel code.
4. **Read the values for the test case inputs from the test case file.**
5. **Populate the input structures with test case values.**
6. Transfer the inputs to the GPU memory.
7. **Calculate dimensions for the GPU threads.**
8. Launch the kernel code on the GPU and wait for the execution to finish.
9. Transfer the results to the CPU memory.
10. **Compare the results to expected results.**

Read the values for the test case inputs from the test case file. The user supplies the test case values in a pre-defined CSV format, presented in Section 4.3.2. While this file will have a different number and type of tests for different programs, the CPU code that reads it remains the same, due to the general nature of the CSV format.

Populate the input structures with test case values. As the CPU code reads test cases from the CSV file, it assigns their values to the **input** structure, declared in task 1. To do this, it calls the auto-generated CPU runtime function, described in Section 4.2.2.

Calculate dimensions for the GPU threads. As mentioned in Chapter 2, the work-group dimensions chosen for any OpenCL program can have significant effect on the achieved performance. For this project, only a naive approach to calculating the dimensions is implemented. The test cases are split equally across a single dimension of the work-group. Future research will be carried out to identify more optimal strategies.

Compare the results to expected results. Currently, the CPU runtime compares the testing results returned by the GPU threads to those from the CPU. This is used to confirm that the GPU provides the same testing results as the CPU.

The CPU runtime is implemented in standard C, using the OpenCL API.

4.3.2 Test Cases

The values for the test cases are supplied by the user and read by the CPU runtime. They need to be in CSV format, in which:

- Each row corresponds to a test case.
- The first column contains the id of the test case. The subsequent columns contain the input variables, in the order in which they are read from the command line by the tested program.
- If the program uses standard input (`stdin`), these inputs are saved in separate text files for each test case, separated by a new line character. In the CSV file, the name of the file containing the `stdin` inputs is recorded, preceded by the '<' character. This informs the CPU code, that a file is being piped, similarly to the way it is done in bash scripting.

For the **add.c** example, a test case file looks like this:

```
1 1 2
2 2 2
3 3 2
4 4 2
5 5 2
```

This file contains 5 test cases. The first column shows the test case id, from 1 to 5. The second column contains values for test input `a` and the third column contains values for the test input `b`.

Suppose **add.c** didn't take `a` and `b` through the command line, but through standard input instead. Then, for each test case, the values would be kept in a separate text file, say `test_{test_num}.inp`, on a new line each. The CSV file would look like this:

```
1 < test_1.inp
2 < test_2.inp
3 < test_3.inp
4 < test_4.inp
5 < test_5.inp
```

The CSV format assumed for the test cases was chosen for a few reasons.

- It is general and likely to fit many testing scenarios.
- It is straightforward to parse.
- It is widely used and likely to be familiar to many potential users.

4.4 C Features

As discussed in Chapter 2, not all C features are readily compiled for execution on the GPU. For some of these which are not, code transformations are possible. Table 4.2 groups C features in three groups: features which run on the GPU readily out of the box, features which are currently being automatically transformed by ParTeCL and features which currently aren't supported. Applications which have the latter would fail to build for execution on the GPU. Furthermore, the section describes the code transformations performed by ParTeCL and discusses the features which are currently unsupported.

4.4.1 Code Transformations

The current section describes the code transformations for some of the features in Table 4.2, namely *global scope variables*, *standard library calls* and *standard input and input*. The transformations for *command line arguments* are shown in Section 4.2.2.

Out of the box	• simple data types, structs, vectors • pure functions, function calls, double precision (for OpenCL 1.2)
With transformations	• command line arguments • standard in/out • global scope variables • standard library calls (partial support)
Unsupported	• recursion • dynamic memory allocation • file I/O

Table 4.2: C features handled by ParTeCL.

Global Scope Variables

OpenCL does not support assignment to global scope variables. Thus, if a C program uses them, they need to be moved to local scope in order for it to be tested on the GPU. ParTeCL does this is by moving the declaration of these variables to the `kernel_main` function and then passing a pointer to them as argument to the functions which use them. Thus, the variables are no longer in the program's global scope. By using pointers to them, the tool ensures that any changes made to their values would be visible within the scopes of all functions which use them.

Consider the **add.c** example. Suppose that instead of declaring the variable `b` inside the `main` function, the programmer had decided to make it a global scope variable, accessible by all methods, like in the code below.

```c
int b;

int add(int a){
  return a + b;
}

int main(int argc, char* argv[]){
  if(argc <= 2){
    printf("Please, provide two integers.\n");
    return 0;
  }

  int a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("%d + %d = %d\n", a, b, add(a));
}
```

In order to port the above program to compilable OpenCL code, the tool performs the following transformations to the global scope variable b.

```
 1 /*int b;*/
 2 
 3 int add(int a, int* b){
 4   return a + *b;
 5 }
 6 
 7 __kernel void main_kernel(
 8          __global struct input* inputs,
 9          __global struct result* results)
10 {
11   int idx = get_global_id(0);
12   struct input input = inputs[idx];
13   int argc = input.argc;
14   results[idx].test_case_num = input.test_case_num;
15 
16   int b;
17 
18   if(argc <= 2){
19     printf("Please, provide two integers.\n");
20     return 0;
21   }
22 
23   int a = input.a;
24   b = input.b;
25   /*printf("%d + %d = %d\n", a, b, add(a));*/
26   results[idx].c = add(a, &b);
27 }
```

- The declaration of b is moved inside main_kernel (line 16).
- An argument int* b is added to function add (line 3).
- The reference to b inside add is replaced with dereferencing of the pointer argument int* b (line 4).
- The address of b is passed as an argument to the function call of add (line 26).

These automatic transformations make it possible for the program to be built in OpenCL without changing its behaviour.

Standard Library Calls

Generally, the OpenCL compiler is not be able to use system files for the C Standard Library. Thus, custom OpenCL implementation of the Standard Library is required for applications which make calls to its functions.

While there are no OpenCL implementations for all of the Standard Library, developing them from scratch is possible. An example of a specific Standard Library implementation, targeting embedded systems, is **uClibc** [5]. Similarly, in the current project, a small subset of Standard Library functions were implemented in OpenCL. These were functions found in the evaluation benchmarks, described in Chapter 5. In particular, the functions implemented were:

- int **atoi**(const char *str)
- char* **fgets**(char *s, int maxsize, char* input, int* idxptr)
  Here the argument `FILE *stream` is replaced by two arguments, namely `char* input` and `int* idxptr`. In other words, a file stream is replaced by a C string and a pointer to its elements. This is necessary as OpenCL cannot work with file strams.
- functions in **ctype.h**

As a supporting part of this project, OpenCL implementations of other standard library functions will be added, as the need for them arises. This sub-project is named **clClibc** and it will be open sourced, in order to allow external contributions and feedback.

Standard Input and Output

The transformations necessary for handling standard input and output on the GPU are briefly described in Section 4.2. Here, they are described in more detail - standard input first and standard output second.

In short, a value for every input to the tested program needs to be supplied as part of the test case. When the program takes them through standard input (`stdin`), the reference to it is replaced with a reference to the **input** structure.

Once more, consider the **add.c** example. Suppose that the inputs `a` and `b` were not command line arguments, but read through `stdin` instead, as in the code below:

```c
1  int add(int a, int b){
2    return a + b;
3  }
4
5  int main(int argc, char* argv[]){
6    char a_str[10];
7    fgets(a_str, 10, stdin);
8    int a = atoi(a_str);
9
10   char b_str[10];
11   fgets(b_str, 10, stdin);
12   int b = atoi(b_str);
13
14   printf("%d + %d = %d\n", a, b, add(a, b));
15 }
```

The program uses `fgets` and `stdin` in lines 7 and 11. ParTeCL replaces them with lines 19 and 23 in the code below. In particular, `fgets` now references the **clClibc** implementation, described in the previous paragraph and `stdin` is replaced with a reference to the **input** structure.

```c
1  int add(int a, int b){
2    return a + b;
3  }
4
5  __kernel void main_kernel(
6          __global struct input * inputs,
7          __global struct result * results){
8
9    int idx = get_global_id(0);
10   struct input input = inputs[idx];
11   __global struct result *result_gen = &results[idx];
12   int argc = input.argc;
13   (*result).test_case_num = input_gen.test_case_num;
14
15   int stdin_count_gen;
16   stdin_count_gen = 0;
17
18   char a_str[10];
19   fgets(a_str, 10, input.stdin1, &stdin_count_gen);
20   int a = atoi(a_str);
21 
22   char b_str[10];
23   fgets(b_str, 10, input.stdin1, &stdin_count_gen);
24   int b = atoi(b_str);
25 
26   /*printf("%d + %d = %d\n", a, b, add(a, b));*/
27   (*result).result = add(a, b);
28 }
```

The variable `stdin_count_gen`, declared on line 15, is added by ParTeCL and is used by `fgets` to record to which character of `input.stdin1` it has read. `fgets` stops at the new line character. The next call to it will start reading from there. In this way, subsequent calls to `fgets` will read from subsequent lines in `input.stdin1`, replicating the behaviour of `stdin`. This is why, as discussed in Section 4.3, the user is required to enter the values for the `stdin` inputs on a new line in the test case input files. In this case, those will be test values for `a` and `b`.

For standard output, ParTeCL comments out calls to functions that output to it (for example `printf`), as they are usually a supporting functionality.

Exception is made when the user wants to test what values are output through standard output. If this is the case, the user needs to enter the name of the standard output function in the configuration file, just as if they were interested in testing the output of any other function. ParTeCL then handles it in a standard way.

4.4.2 Unsupported Features

Table 4.2 displays C features which are not supported for compilation by the OpenCL compiler and which are currently not handled by the tool. This section discusses them in greater detail together with possible solutions.

Recursion

A major limitation of GPU hardware is the lack of support of recursion. However, any recursive function can be rewritten as a non-recursive function [23]. Thus, if the tested program contains recursive calls, they could be removed without changing functionality.

To demonstrate this in practice, one of the benchmark programs, used in Chapter 5, is considered. This is the **replace** program, which is part of the SIR repository [9]. Replace contains a single recursive function, called `amatch`, which is manually turned into a non-recursive function in order to execute it on the GPU. For brevity, the detailed description of the function and its transformation are omitted from this chapter and included in Appendix A instead.

`amatch` it is a complex recursive function, with a recursive call contained within two nested `while` loops and an `if` statement. Via analysis of the control flow of this function, it was manually transformed into a non-recursive one with the use of a stack.

To confirm that this modification does not alter **replace's** functionality, its test cases were executed on the CPU and their results were compared to those obtained with the original recursive implementation.

ParTeCL does not offer automatic code transformation to remove recursive functions, since, as demonstrated by the **replace** example, this is not a trivial task for general functions. However, as seen in this example, when necessary, it is possible to perform the transformation manually, even for non-trivial functions.

Dynamic Memory Allocation

Currently, dynamic memory allocation is not possible on the GPU and the OpenCL compiler throws an error if it encounters it. The GPU can allocate memory only statically. Dynamic memory allocation of the GPU memory can be performed only by the CPU runtime.

Therefore, transformations to eliminate dynamic memory allocation would be only possible for data structures which are either test inputs or test results. Such transformations would not be trial.

For this reason, ParTeCL currently targets applications which do **not** use dynamic memory allocation. This may change in the future, with the advance of GPU architectures and tools, as well as in future work aimed at extending the scope of this project's approach.

File I/O

File I/O, similar to other system calls, is not possible on the GPU. Special OS abstractions are needed to allow the GPU to access the file system, which can have significant performance penalties [27].

For the purposes of testing, the tested program's kernel can be partitioned, so that file I/O is performed by the CPU and overlapped with kernel execution where possible. This will be examined and implemented in future work.

4.5 Implementation

ParTeCL was implemented in C++14, using the Clang LibTooling library [3], presented briefly in Section 2.3. The current section describes the implementation, starting from the system structure, continuing with the components performing the source code transformations and finishing with a discussion of the reasons behind the implementation decisions and the process involved in ensuring ParTeCL's correctness.

The implementation is based on ParTeCL's outputs and on the code transformations it performs, seen earlier in Sections 4.2.2 and 4.4 respectively.

4.5.1 System Structure

ParTeCL's implementation has a very straightforward system structure. It consists of two components:

- **The Main Class.**
  Entry point to the system, it performs two functions:
  - Read the configuration file and generate the Data Structures (**struct.h**) and a part of CPU Runtime Code (**cpu-gen.c**).
  - Launch the Kernel Generator and pass to it parameters from the configuration file.
- **The Kernel Generator.**
  Generates the OpenCL kernel code from the original C code.

The Main Class performs a straightforward translation of the configuration file into the two outputs - **structs.h** and **cpu-gen.c**.

The Kernel Generator performs the code transformations which translate the original C program into valid OpenCL kernel code. The Kernel Generator uses LibTooling's AST Matchers to find portions of the original program which it needs to transform. It then uses the Rewriter class inside the corresponding AST Handlers to perform the transformations at source code level. The particular AST Matchers and AST Handlers currently implemented in ParTeCL are described in the next section.

4.5.2 AST Matchers & Handlers

In LibTooling, each AST Matcher is usually implemented together with a corresponding AST Handler, which implements the desired actions to be taken for the matched AST nodes. ParTeCL uses AST Handlers to save necessary intermediate information between matchers and to perform the code transformations.

Each of the matchers implemented in ParTeCL is described below, together with the actions implemented in its handler. The matchers are grouped based on the code transformations they perform, as described in Section 4.4, or on their supporting functions.

Command line arguments

- `argvInAtoiMatcher`
  Used for programs, which take integers as command line arguments, it finds references of the type `atoi(argv[i])`, and replaces them with references to the corresponding variable inside the **input** structure.
  For example, `int a = atoi(argv[1])` becomes `int a = input.a`.

- `argvMatcher`
  Same as `argvInAtoiMatcher`, but used in programs, in which the reference to `argv` is not inside a call to `atoi`.

Standard output

- `commentOutMatcher`
  Finds calls to functions which perform standard output and comments them out.

Can be extended to comment out calls to other functions too.

Build 'function call to caller map'

- `calleeToCallerMatcher`

  Builds a map between function calls and the function inside which they are called. It is used in handlers, which add either new parameters to function declarations or, correspondingly, new arguments to function calls. For example, when a global variable is being turned into a local variable, its declaration is moved to the `main` method and then added as a parameter to the functions which use it, e.g. `foo`. `foo` may not be called directly by `main` though. There may be a function `bar` which calls `foo`, which in turn is called by `main`. Thus, the call to caller map will be:

  `foo ->bar`

  `bar ->main`

  Thus, the global variable will need to be added as a parameter to `bar` as well, and then passed as an argument to all function calls of `bar`. The 'call to caller map' allows the matchers, which deal with new function parameters and arguments, to recursively handle such scenarios of arbitrary depth.

Standard input

- `stdinMatcher`

  Finds references to `stdin` and replaces them with references to the **input** structure. Creates a list of functions, which make a reference to `stdin`.

- `stdinAsParamsMatcher`

  Adds the counter, which keeps track of how much of `stdin` has been read, as a parameter to the function declarations, which use it. Uses the 'call to caller map'. An exmaple of this counter is found in Section 4.4 under **Standard Input and Output**.

- `stdinAsArgsMatcher`

  Adds the counter from `stdinAsParamsMatcher` as an argument to the function calls of the functions, which now have it as a parameter.

Global scope variables

- `globalVarMatcher`
  Finds the program's global variables and comments out their declarations.

- `globalVarUseMatcher`
  Finds the functions which use global variables and changes their references to the global variable to the dereferencing of a pointer to it (if the global variable is not already a pointer). An example of this can be found in Section 4.2.2 under **Kernel Code - test.cl**.

- `globalVarsAsParamsMatcher`
  Adds the global variables as parameters to the function declarations, which need them. Uses the 'call to caller map'.

- `globalVarsAsArgsMatcher`
  Adds the global variables as arguments to the function calls of the functions, which now have them as parameters.

Input and Result structures inside functions

- `inputsMatcher`
  Finds functions, which require a reference to the **input** structure. Those are functions, which reference `stdin`.

- `resultsMatcher`
  Finds functions, which require a reference to the **result** structure. Those are functions which make a call to the tested function, as specified in the configuration file.

- `inputsAndResultsAsParamsMatcher`
  Adds the **input** and **result** structures as parameters to the function declarations, which need them. Uses the 'call to caller map'.

- `inputsAndResultsAsArgsMatcher`
  Adds the **input** and **result** structures as arguments to the function calls of the functions, which now have them as parameters.

Write the result into the result structure

- `functionToTestMatcher`

Finds function calls to the tested function, as specified in the configuration file. Replaces those function calls with calls, which write the result in the **result** structure.

Transformations in `main`

- `mainMatcher`
  Performs the following transformations in `main`:
  - Changes the signature and the name to turn it into an OpenCL kernel function.
  - Adds the generic code in the beginning of `main`, as described in Section 4.2.2 under **Kernel Code - test.cl**.
  - Adds declarations for the global scope variables.
  - Adds declarations for the necessary string counters.
- `returnInMainMatcher`
  If the `main` function returns a value, comments the return statement out, since OpenCL kernels do not return values.

Strandard library calls

- `includesMatcher`
  Finds calls to standard library functions. If the function is implemented in **clClibc**, it adds an `#include` for the respective header file.

4.5.3 Discussion

Using Clang LibTooling

The use of Clang's LibTooling was chosen for ParTeCL's implementation, as the AST Matchers it provides fit with the objective to perform transformations on specific parts of the source code. This meant that most of the work went into deciding what the transformations needed to be and not into implementation effort.

In addition, the AST Matchers and Handlers are implemented in their own classes, which keeps the code decoupled and easy to maintain and extend for additional code transformations.

Testing

To ensure that ParTeCL performed the intended code transformations correctly, small example programs were used, similar to the **add.c** example used in this chapter. For each code transformation, a small program containing only the relevant feature would be developed and run through ParTeCL to check that the transformation was as intended. These programs are kept as a set of tests.

4.6 Summary

The current chapter presented the process of testing on the GPU threads, as well as the design and implementation of the two systems which automate it, namely ParTeCL and the CPU runtime. Testing on the GPU involves:

- generation of an OpenCL kernel, which executes the tested program on the GPU threads. This step is automated by ParTeCL.
- execution of the CPU runtime, which launches the tests in parallel on the GPU threads. In particular, it reads the values for the test cases, transfers them to GPU memory, builds and launches the OpenCL kernel on the GPU threads and transfers the results back to CPU memory for verification. The CPU runtime requires the values for the test cases to be supplied in a particular CSV format, also presented in this chapter.

ParTeCL

This chapter presented ParTeCL by describing its inputs and outputs. It has two inputs: the unmodified source code of the tested program and a configuration file, written by the user. It generates three outputs: the OpenCL kernel, data structures to transfer data between the CPU and GPU memories and a CPU runtime function which assigns values to the auto-generated data structures.

In addition, the chapter described the code changes performed by ParTeCL in detail and discussed particular C features, which pose a challenge to the OpenCL compiler and are handled via specific transformations. Transformations alter the input/output interface of the program, but not its functionality.

Finally, the chapter described ParTeCL's implementation in the C++14 programming language, which uses the AST Matchers available in the Clang LibTooling library [3].

The CPU runtime

The CPU runtime consists of mostly generic CPU code, which remains the same, irrespective of which C program is being tested. The current chapter presented the functionalities performed by the CPU runtime, as well as the CSV format for the test case values, used by the CPU runtime. The CSV format was chosen for being general, easy to parse and already widely adopted.

Chapter 5

Evaluation

The Master's project has three goals, identified in Chapter 1. To clarify them and to evaluate the extend to which they are met, four evaluation questions are identified. They are presented in the beginning of the current chapter and used to evaluate the results achieved in this project. In addition, the evaluation uses benchmark applications from two software repositories. They are presented in Section 5.2. Finally, the evaluation results are presented in Section 5.3.

5.1 Evaluation Questions & Setup

To evaluate the extend to which the project meets its goals, the current chapter answers the following evaluation questions:

Q1. Valid OpenCL code.
Does ParTeCL produce valid OpenCL code, which can be built and ran on the GPU without manual modifications?

Q2. Correct Testing Results.
Do ParTeCL's output kernels produce the same testing results as the tests ran on the CPU?

Q3. Usability.
Does ParTeCL eliminate the need of OpenCL knowledge? Can it be used by general software developers?

Q4. Speed-up and Optimisations.
What speed-up is achieved by ParTeCL's output kernels? What future optimisations should be researched?

To answer question Q1, four benchmark applications, containing different C features, were compiled with ParTeCL. The resulting OpenCL kernels were built by the CPU runtime and executed on an Nvidia Tesla K40m GPU. The benchmarks are from the SIR [9] and EEMBC [6] repositories and are described in detail in Section 5.2.

To answer question Q2, the testing results obtained for each benchmark when ran on the GPU were compared to those obtained when executed on the CPU. The OpenCL API contains profiling functions, which are used to measure execution times on the GPU. Execution time on the CPU is measured, suing the standard C function `gettimeofday`. Executions were performed 1000 times and the median value is reported.

To answer question Q4, execution time for two different benchmarks was measured and compared to execution time on the CPU. This performance data was analysed and used to identify further steps for the optimisation of the achieved performance.

To answer question Q3, usability testing was performed with six programmers. For this purpose, a simple benchmark application was written, which takes inputs both through command line and standard input. The programmers were asked to write tests for it, generate an OpenCL kernel using ParTeCL and use the CPU runtime to execute the tests on the GPU. Their actions were observed and any problems and questions they had were noted down. Finally, they were asked to rate the ease of use of the different steps involved in the testing process. The benchmark program, as well as a full list of the actions and questions, can be found in Appendix B.

Hardware

The benchmark applications were executed on an Nvidia Tesla K40m GPU. The GPU has 15 compute units with a maximum work-group size of 1024 work-group items.

The final results for all evaluation questions are presented in Section 5.3.

5.2 Benchmarks

Benchmark programs with existing test suites were chosen for evaluation. They are all written in the C programming language and found in two separate benchmark repositories, namely SIR [9] and EEMBC [6]. SIR contains benchmarks programs aimed at software testing research, together with test suites for them. EEMBC [6] contains embedded systems benchmarks from a variety of industry domains, including mobile devices, automotive and telecom.

SIR benchmarks were chosen for the large test suites with which they are supplied, making them suitable for performance evaluation, while EEMBC benchmarks are representative of the types of applications which testing on the GPU could target.

Details of the particular programs are presented in this section and summarised in Table 5.1.

5.2.1 SIR: tcas

- **Size:** 173 LOC, 9 functions
- **Different test cases:** 1608

tcas is an aircraft collision avoidance system, developed at Siemens Corporate Research for the study presented in [16].

It takes 13 integer inputs through the *command line* and outputs a single integer. The program implements 8 functions in addition to `main` and uses *global scope variables*.

5.2.2 SIR: replace

- **Size:** 564 LOC, 21 functions
- **Different test cases:** 5542

replace performs pattern matching and substitution in strings. It is developed for the same study as **tcas** [16].

replace takes two string inputs through *command line* and multiple strings through *standard input*. It matches occurrences of the first string in the *stdin* inputs and replaces them with the second string. It outputs the result, character by character, in standard output.

replace's implementation has a number of interesting aspects, different to **tcas**:

- It performs string manipulation.
- It contains loops and a high degree of control-flow divergence, determined by the size of the test case inputs, i.e. the length of the input strings. This has implications for the achieved performance.
- It contains a recursive function.
- It takes inputs through standard input.
- The testing result is not performed by a function call, but output in standard output.

5.2.3 EEMBC: autcor00

- **Size:** ∼35 LOC, 1 function
- **Different test cases:** 3

autcor00 is a small system from the TeleBench benchmark suite in EEMBC, containing telecom applications. It is a mathematical algorithm for signal processing, which computes a cross-correlation of a signal with itself.

autcor00's implementation has the following interesting aspects:

- Custom data types; **autcor00** defines its own data types in a separate header file, by renaming `signed short` to `e_f32` and `signed int` to `e_s32`. This is important as it demonstrates that testing on the GPU handles custom data types, as long as the definitions are available to the OpenCL compiler, similar to the CPU.
- Results are written into an argument of the tested function.

Since EEMBC is not a testing repository, **autcor00** is supplied with only 3 test cases. This is sufficient to evaluate the correctness of the OpenCL kernel which ParTeCL outputs (Q1 and Q2), but not enough to be used in performance evaluation (Q4).

5.2.4 EEMBC: viterb00

- **Size:** ∼193 LOC, 6 functions
- **Different test cases:** 4

viterb00 is another program from the TeleBench suite in EEMBC. It is a decoder for encoded streams in embedded IS-136 channel coding applications.

Its implementation is very similar to that of **autcor00**, making use of the same custom data types, but **viterb00** is a larger program, which also uses global scope variables and custom structures.

Similar to **autcor00**, **viterb00** has only a small number of test cases and is not used in the performance evaluation (Q4). It is used in the evaluation of questions Q1 and Q2.

Benchmark	**Test cases**	**Interesting features**	**Data types**
SIR: tcas	1608	• global scope variables	`int`
SIR: replace	5542	• string manipulation • standard input • recursion • results in standard output	`string`

EEMBC: autcor00	3	• custom data types
		• result as an argument to the tested function
			`typedef signed short`
			`typedef signed int`

EEMBC: viterb00	4	• custom data structures • global scope variables	`typedef struct`

Table 5.1: Benchmark summary.

5.3 Results

5.3.1 Q1. Valid OpenCL Code

ParTeCL successfully produces valid OpenCL kernels for the benchmark programs, shown in Section 5.2. In other words, without the need of manual editing, the kernels produced by ParTeCL can be built by the CPU runtime and launched on the GPU threads.

The benchmark programs use different C features, presented in Table 5.1, demonstrating that ParTeCL can produce valid OpenCL code for a range of C applications.

5.3.2 Q2. Correct Testing Results

For all test cases of the benchmarks, the testing results produced when executing ParTeCL's OpenCL kernels on the GPU were compared to those produced by the CPU. In all cases, GPU testing produced the exact same results, confirming empirically the correctness of testing on the GPU.

What is more, the benchmarks use different data types, as seen in Table 5.1, which demonstrates that the GPU outputs correct testing results irrespective of the data types used by the tested application.

5.3.3 Q3. Usability

One of ParTeCL's goals is to make testing on the GPU available to general programmers, by eliminating the need to manually write OpenCL code. To evaluate the extend to which this goal is achieved, usability testing was performed with six users, unfamiliar with the project. As seen in Section 5.1, they were asked to perform all steps involved in testing on the GPU, using ParTeCL, and then rate the different aspects of it in terms of ease of use. The average ratings are presented in Table 5.2.

The ratings show that overall, once users were familiar with the formats defined for the test cases and configuration file, they found the testing process clear and straightforward to follow, demonstrating that **ParTeCL successfully abstracts away the GPU programming details.**

Step in the testing process	**Rating**	
Writing test cases in the CSV format	4.00	★★★★☆
Writing the configuration file	3.41	★★★⯨☆
Running ParTeCL	4.83	★★★★★
Building the CPU runtime	4.85	★★★★★
Running the test cases	4.33	★★★★⯨
Overall process	**3.91**	★★★★☆

Table 5.2: Ease of use ratings. The scale is from 1 to 5, where 1 is *not at all easy* and 5 is *very easy*.

In addition, the usability testing highlighted some areas of improvement, which will be considered and included in future work. This was expected, as the systems are not a fully developed yet.

- Test cases in CSV format:
  - Extend the format to accommodate complex data structures.
- Configuration file:
  - Auto-generate names for the inputs and result, instead of expecting them from the user.
- Overall process:
  - Merge the test cases and the configuration file in one.
  - Provide a single front-end for both systems: code generation and test execution.

5.3.4 Q4. Speed-up and Optimisations

The two SIR benchmarks, **tcas** and **replace**, are used to asses the performance achieved by executing test cases in parallel on the GPU, using ParTeCL and the CPU runtime. The following performance data is gathered:

- Time to transfer test cases to the GPU memory (transfer inputs).
- Time to execute the kernels on the GPU (kernel execution).

- Time to transfer results back to the CPU memory (transfer results).
- Time to execute all tests on the CPU (CPU time).

In order to simulate larger test suites, test cases are read in a loop by the CPU runtime. For example, if 1000 test cases are available, but 1500 are needed, all 1000 test cases are read once and then the first 500 are read again.

SIR: tcas

Figure 5.1 displays the CPU and GPU execution times for **tcas**, when running different numbers of test cases. The following is observed:

- Overall execution time on the GPU is consistently much lower than execution time on the CPU, even for small numbers of test cases.
- Kernel execution time and the time to transfer inputs both contribute significantly to the execution time on the GPU.
- As the number of test cases grows, the time to transfer inputs increases faster the kernel execution time. In fact, the kernel execution time remains almost constant for up to 8192 test cases, as there are still parallel resources available on the GPU. The time to transfer inputs grows linearly.

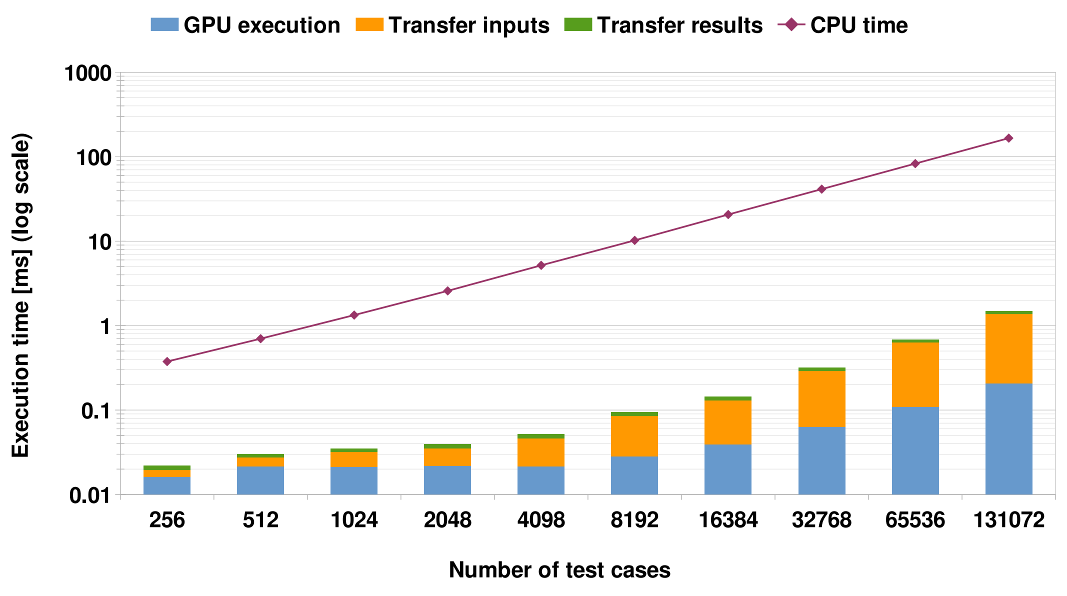

Figure 5.1: Execution times of the **tcas** benchmark on the GPU and the CPU for test suites of different sizes.

The last observation has implications for the speed-up achieved in comparison to execution on the CPU. Figure 5.2 displays the speed-ups when kernel execution time is considered both alone and together with the time for data transfers. While the speed-up of the overall GPU execution (kernel + data transfers) is considerable at up to 90x, this is almost 10 times smaller than the speed-up achieved when data transfers are not considered. This demonstrates that optimising for data transfers could be extremely beneficial in terms of performance.

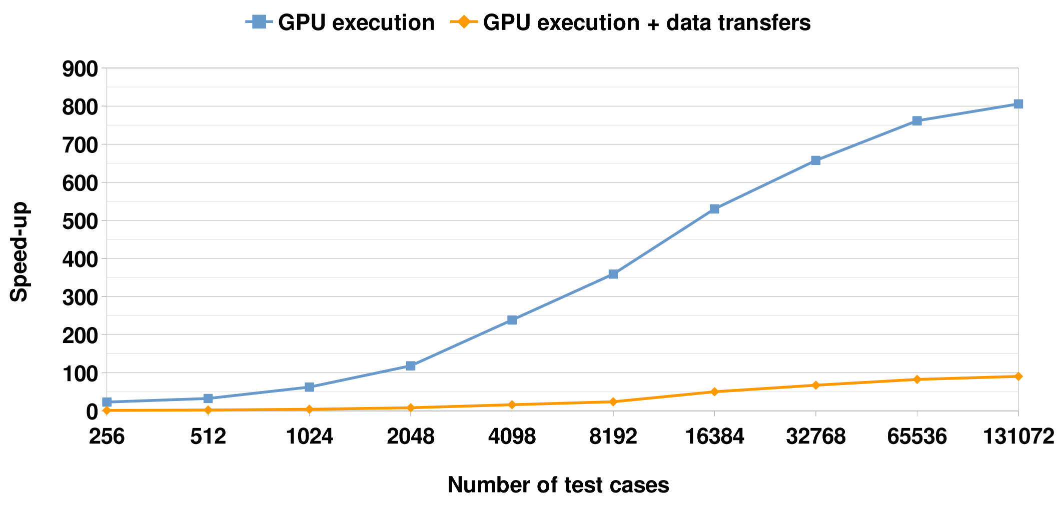

Figure 5.2: Speed-up for the **tcas** benchmark, when compared to execution on a single CPU, on test suites of different sizes.

These performance results are promising and as expected, as **tcas** is a simple application, with only a small degree of control-flow divergence, which uses little memory. Thus, it is an ideal candidate for testing on the GPU.

SIR: replace

In contrast, **replace** contains many nested loops, whose number of iterations depend on the lengths of the input strings. This constitutes a large degree of control-flow divergence, which would have a detrimental impact on the performance. In addition, the test cases of **replace** are much larger. In cases when not all of them fit in the thread's local memory, the GPU would need to store them in its global memory, forcing the threads to perform expensive reads from it.

Both of these factors are possible explanations for the long kernel execution times, seen in Figure 5.3. For test suites of up to 2048 test cases, the kernel execution time

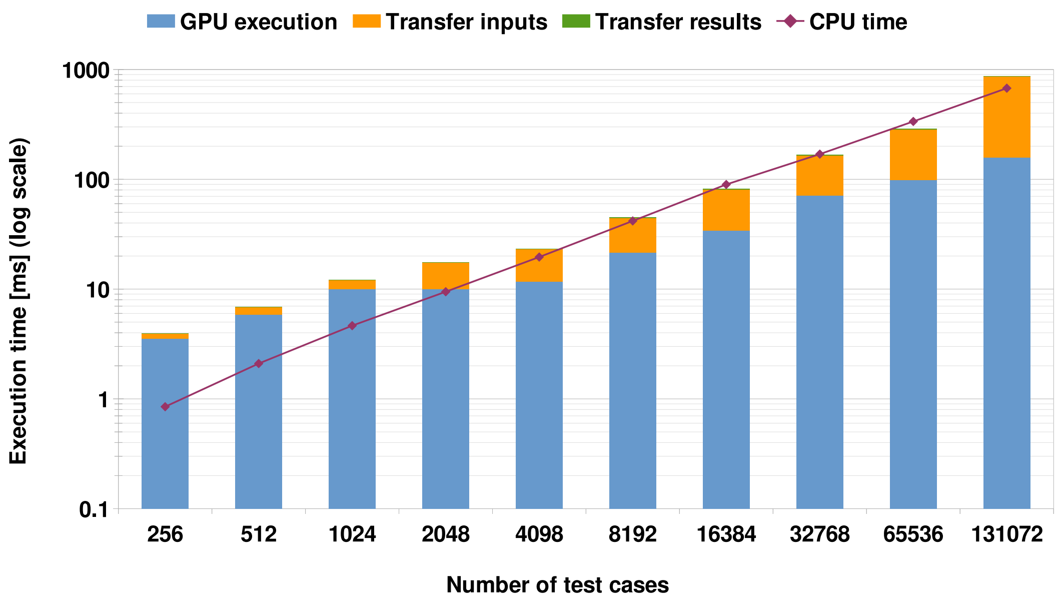

Figure 5.3: Execution times of the **replace** benchmark on the GPU and the CPU for test suites of different sizes.

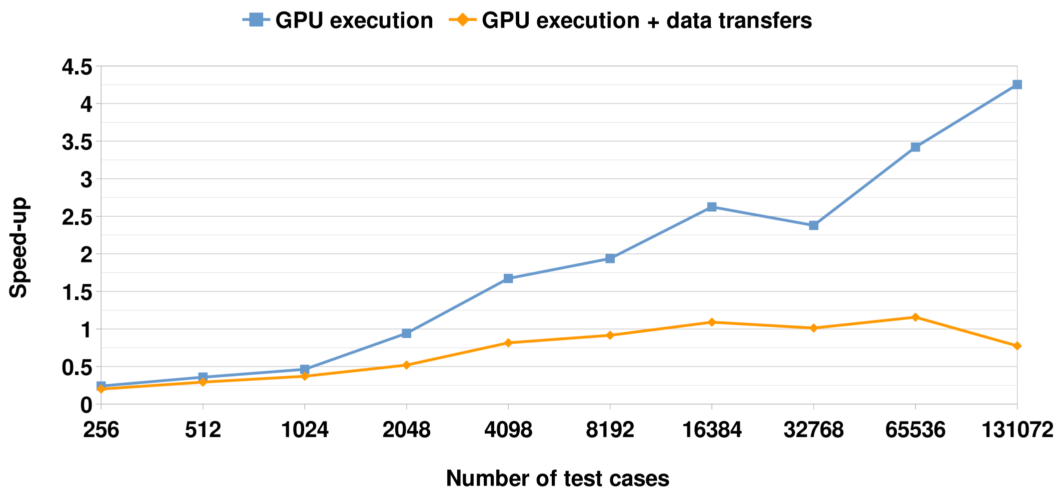

Figure 5.4: Speed-ups for the **replace** benchmark, when compared to execution on a single CPU, on test suites of different sizes.

alone is longer than the CPU execution time, meaning that the GPU is in fact slower. As both figures 5.3 and 5.4 show, this effect diminishes as more test cases are executed, leading to a speed-up of over 4x for test suites of 131072 test cases when only kernel execution time is considered. Similarly to what was observed for **tcas**, the speed-up is significantly smaller when both kernel execution and data transfer times are taken into account, effectively removing any benefit of using the GPU. Thus, **replace** shows that in addition to optimisations for data transfers, optimisations minimising the effect of control-flow divergence and large data inputs, not fitting into local memory, should be researched.

Performance optimisations will be the focus of future PhD work. This is discussed in 6.3.

5.4 Summary

To assess the extend to which the current project meets its goals, this chapter used four evaluation benchmarks from the SIR [9] and EEMBC [6] benchmark suites. The test cases of the four benchmarks were executed on the GPU, using ParTeCL and the CPU runtime, demonstrating the following results:

- ParTeCL generates valid OpenCL code, which can be successfully built and executed on the GPU without manual modifications.
- The OpenCL kernels generated by ParTeCL, when executed on the GPU, output the same testing results as the CPU.

Performance results were collected for the benchmarks from SIR, revealing the following:

- High speed-ups (up to 90x in this example) can be easily achieved for applications with a low degree of control-flow divergence.
- Data transfers between the CPU and the GPU take a significant portion of the execution time and have a huge detrimental impact to the achieved speed-ups.
- Performance for applications with a high degree of control-flow divergence and large data inputs is limited and optimisation methods should be researched in future work.

Finally, a usability study was performed with six programmers, which confirmed that ParTeCL and the CPU runtime successfully hide the OpenCL layer of implementation and allow programmers to execute test cases on the GPU easily and transparently. Additionally, user feedback was gathered during the study to be used for future improvements to the process.

Chapter 6

Conclusions

This Master's project focussed on the challenges encountered in using GPUs to accelerate functional software testing. Its goal was to demonstrate that *automation* of test launching can successfully abstract away low-level programming model details, making it more accessible to general programmers, who do not have GPU knowledge. The project also aimed to empirically demonstrate that automatically generated code can be successfully built and executed on the GPU, and that tests ran on the GPU threads produce the same testing results as when executed on the CPU.

For this purpose, two systems were developed - ParTeCL, which automatically generates OpenCL kernels for the tested program, and a CPU runtime, which reads the test cases and launches them on the GPU threads, using the auto-generated OpenCL kernels. ParTeCL was implemented in C++14, using the Clang LibTooling library and the CPU runtime was implemented in standard C. Four C benchmarks from two repositories were used for the evaluation of the systems. A usability study was carried out to determine their ability to enable general programmers to execute tests on the GPU. Finally, preliminary performance data was gathered and analysed in order to investigate potential performance and identify future areas for optimisation.

This chapter summarises the design and implementation of the developed systems and the evaluation results presented in this project. It also discusses what future work should be carried out in the PhD phase of this research and briefly analyses the current project and lessons learnt in it.

6.1 Design & Implementation

Running functional tests on the GPU requires two steps:

- **Code generation**
  Turn the original C application into an OpenCL kernel, which can be executed on the GPU.

- **Test execution**
  Transfer test case values to the GPU memory, build the OpenCL kernel and launch it in parallel on the GPU threads. Then transfer the results back to the CPU memory, where they can be checked.

Two systems were developed to automate the steps above - ParTeCL to automate code generation and a CPU runtime to automate test launching. They are developed separately, but the CPU runtime requires the output produced by ParTeCL.

Code generation

ParTeCL translates the original application which is being tested into an OpenCL kernel via a series of code transformations, which alter the way data is being passed to the program, but leave the original functionality unchanged. In short, the `main` function is turned into an OpenCL kernel function and inputs and results are being read and written to the GPU memory.

ParTeCL also performs code transformations for C features, which are not readily supported for compilation on the GPU. Features supported via ParTeCL's code transformations are *global scope variables*, *standard input/output* and *standard library calls*. Features which are currently unsupported are *recursion*, *dynamic memory allocation* and *file I/O*. Benchmarks which use recursion are tested by manually turning the recursive functions into equivalent non-recursive ones.

ParTeCL is implemented in C++14 and uses the Clang LibTooling library. The library supports code-to-code transformations by building an Abstract Syntax Tree (AST) for the input program and providing AST Matchers, which are used to identify portions of the code to be transformed. ParTeCL defines its own AST matchers and uses LibTooling's Rewriter class to apply particular changes to the original source code.

In addition the OpenCL kernel, ParTeCL also generates data structures for the test case inputs and results. They are used by the OpenCL runtime to transfer data between the CPU and GPU memories. Finally, ParTeCL generates a CPU function, which is used by the CPU runtime to populate the input structures with the test case values during test execution.

Test execution

The CPU runtime automates test execution. It performs mostly generic CPU actions, independent of the program which is being tested. It transfers inputs to the GPU memory, builds and launches the OpenCL kernel in parallel on the GPU threads and transfers the results back to the CPU. In addition, it performs some actions specific to testing. It reads values for the test cases from a text file in an assumed CSV format and assigns them to the input structures generated by ParTeCL. The CSV format was designed as part of this project. The CPU runtime is implemented in standard C and uses the standard OpenCL API.

6.2 Evaluation

The evaluation process used four benchmark applications from the SIR [9] and EEMBC [6] repositories. It consisted of three different stages.

Firstly, OpenCL kernels for all four benchmarks were generated by ParTeCL and their test cases were executed on an Nvidia Tesla k40m GPU, demonstrating the validity of the auto-generated kernels. In addition, the testing results produced by the GPU were compared to those from the CPU in order to empirically establish correctness of testing on the GPU.

Secondly, performance data from the two SIR benchmarks was collected and compared to test executions on a single CPU. This demonstrated that high speed-ups are possible, but also negatively influenced by factors like time for data transfers between the CPU and GPU memories, large input data size and high degree of control-flow divergence.

Finally, a usability study for the two systems developed in this project was carried out with six programmers, using a simple example application. It identified new desirable features for the systems from a usability point of view and demonstrated that automation successfully abstract away the low-level OpenCL programming model from the testing process and makes testing on the GPU accessible to general programmers.

6.3 Limitations & Future Work

This Master's project carried out preliminary work in automating test launching on the GPU threads, paving the way for future extensions to the scope and effectiveness of this approach. This section briefly presents current limitations and future work, which can be carried out during the PhD phase of the project. The work can be roughly split into three categories: empirical evaluation, performance optimisation and extending the systems' scope and usability.

6.3.1 Empirical Evaluation

Currently, the automated systems are evaluated on only four benchmark applications, the largest of which has only 564 LOC and 5542 test cases. To conclusively establish the benefits of the proposed approach and automation, extensive evaluation with more and larger benchmark applications need to be performed. This evaluation would also be helpful in guiding and evaluating future performance optimisations.

In addition, the approach can be evaluated on different heterogeneous architectures in order to determine to what degree its scope and performance depend on hardware architecture. The effects of different OpenCL thread/block dimensions can also be examined on different architectures and techniques for choosing the optimal ones can be researched.

6.3.2 Performance Optimisation

Presently, no performance optimisations are performed when tests are executed on the GPU, making the effectiveness of the approach susceptible to GPU and OpenCL limitations. This is observed in the performance results achieved for **replace**, presented in Section 5.3.

Thus, the following performance optimisations are considered for future work:

- **Data transfer.** To mitigate the effects of long data transfers, the CPU runtime can split test cases in batches and overlap their transfer with kernel execution. Additionally, *pinned* memory can be used, as described in [24].

- **Large test inputs.** Techniques for efficient memory management on the GPU should be researched and implemented. Dynamic memory allocation for the input and results structures on the CPU runtime should also be implemented in order to ensure that test cases of different sizes take up exactly as much memory as they need.
- **Control-flow divergence.** Using control-flow analysis, test cases which take similar control-flow paths can be identified and grouped together for execution on the same GPU compute units, resulting in threads executing in lock-step having the same instructions. This is promising, as applications with large test suites are likely to have many test cases with similar control-flow paths.

6.3.3 Extending Scope & Usability

Currently, ParTeCL and the CPU runtime make assumptions about the source code and type of testing which is being carried out. In particular, ParTeCL assumes that all functions of the tested program are in a single source file. Additionally, the tool can presently support functional testing for the full program (system testing), but not for individual functions (unit testing). It would be easy to remove both of these assumptions.

In addition, transformations for the unsupported C features, discussed in Section 4.4, should be added, as well as for C features found in any new evaluation benchmarks. Finally, system features improving usability, as suggested by the feedback of the usability study presented in 5.3, can also be added when found useful.

6.4 Critical Analysis

Overall, the Master's project is successful in achieving its goals. It provides two systems, which automatically execute test cases in parallel on the GPU threads and produce correct testing results. While there are still many features which can be added to make the systems more robust and easy to use, their modular implementation allows that. Even more importantly, the systems can be used to aid future research in addressing the performance limitations of the approach.

The rest of this section discusses some lessons learnt during this project.

Firstly, there is a fine balance between perfecting the tools developed during research, which can take a lot of time for, arguably, not much gain, and spending just enough effort on them to make them usable. The two extremes were encountered during this project and, in hindsight, high-level system design should be considered early in implementation, irrespective of how trivial the system seems. However, it should be kept simple in order to save time consuming changes later in the project.

Secondly, identifying a number of benchmarks to target early on would have been extremely helpful in implementing robust code transformations and estimating effort. Ideally, benchmarks should be chosen and evaluated first, their features analysed, and a common strategy for handling them in the tool should be designed before implementation.

Finally, specifying the scope and goals of the tools early on would have allowed for a greater range of testing practices to be supported. In particular, the early goal of accelerating software tests was clear when implementation first started, but the level of granularity of testing was not decided, resulting in only system testing being currently supported by the automated systems. They can be easily extended to support unit testing, which is included in future work, but this could have been done from the beginning, enabling more evaluation and results.

Bibliography

[1] `https://www.khronos.org/opencl/`.

[2] `http://www.nvidia.com/object/cuda_home_new.html`.

[3] `http://clang.llvm.org/docs/LibTooling.html`.

[4] `https://www.khronos.org/sycl/`.

[5] `https://www.uclibc.org/`.

[6] `http://www.eembc.org/`.

[7] *Benefits of parallel testing*, tech. rep., National Instruments Corporation, Austin, TX, USA, June 2016.

[8] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, *Automatic C-to-CUDA code generation for affine programs*, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 6011 LNCS, Springer Berlin Heidelberg, 2010, pp. 244–263.

[9] H. Do, S. G. Elbaum, and G. Rothermel, *Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.*, Empirical Software Engineering: An International Journal, 10 (2005), pp. 405–435.

[10] S. Elbaum, A. Malishevsky, and G. Rothermel, *Incorporating Varying Test Costs and Fault Severities into Test Case*, Proceedings of the 23rd International Conference on Software Engineering, (2001), pp. 329–338.

[11] P. Group, T. Grosser, A. Groesslinger, C. Lengauer, P. Group, T. Grosser, A. Groesslinger, and C. Lengauer, *POLLY - PERFORM-*

ING POLYHEDRAL OPTIMIZATIONS ON A LOW-LEVEL INTERMEDIATE REPRESENTATION, Parallel Processing Letters, 22 (2012), p. 1250010.

[12] M. J. HARROLD AND M. L. SOFFA, *Interprocedural Data Flow Testing*, TAV3 Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification, (1989), pp. 158–167.

[13] M. P. E. HEIMDAHL AND D. GEORGE, *Test-suite reduction for model based tests: Effects on test quality and implications for testing*, in Proceedings of the 19th IEEE International Conference on Automated Software Engineering, ASE '04, Washington, DC, USA, 2004, IEEE Computer Society, pp. 176–185.

[14] J. R. HORGAN AND S. LONDON, *Data Flow Coverage and the C Language*, in Proceedings of the Symposium on Testing, Analysis, and Verification, 1991, pp. 87–97.

[15] J. R. HORGAN AND A. P. MATHUR, *Handbook of software reliability engineering*, McGraw-Hill, Inc., Hightstown, NJ, USA, 1996, ch. Software Testing and Reliability, pp. 531–566.

[16] M. HUTCHINS, H. FOSTER, T. GORADIA, AND T. OSTRAND, *Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria*, in Proceedings of the 16th International Conference on Software Engineering, ICSE '94, Los Alamitos, CA, USA, 1994, IEEE Computer Society Press, pp. 191–200.

[17] IBM, *Press release: Ibm extends development and test to the ibm cloud.* `http://www-03.ibm.com/press/us/en/pressrelease/29685.wss`, March 2010.

[18] L. INOZEMTSEVA AND R. HOLMES, *Coverage is not strongly correlated with test suite effectiveness*, in Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, New York, NY, USA, 2014, ACM, pp. 435–445.

[19] D. JEFFREY AND N. GUPTA, *Improving fault detection capability by selectively retaining test cases during test suite reduction*, IEEE Trans. Softw. Eng., 33 (2007), pp. 108–123.

[20] D. B. KIRK AND W.-M. W. HWU, *Programming Massively Parallel Processors : A Hands-on Approach*, Morgan Kaufman, 2013.

[21] R. KRISHNAMOORTHI AND S. A. SAHAAYA ARUL MARY, *Factor oriented requirement coverage based system test case prioritization of new and regression test cases*, Information and Software Technology, 51 (2009), pp. 799–808.

[22] D. LEON, W. MASARI, AND A. PODGURSKI, *An empirical evaluation of test case filtering techniques based on exercising complex information flows*, Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., (2005), pp. 412–421.

[23] Y. A. LIU AND S. D. STOLLER, *From recursion t o iteration : what are the optimizations ?*, Proceedings of PEPM'00: the ACM SIGPLAN 2000 Workshop on Partial Evaluation and Semantics-Based Program Manipulation, (2000), pp. 73–82.

[24] NVIDIA, *NVIDIA OpenCL Best Practices Guide*, Optimization, 181 (2009), pp. 2175–2184.

[25] A. RAJAN, S. SHARMA, P. SCHRAMMEL, AND D. KROENING, *Accelerated test execution using gpus*, in Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, New York, NY, USA, 2014, ACM, pp. 97–102.

[26] T. REMMELG, T. LUTZ, M. STEUWER, AND C. DUBACH, *Performance Portable GPU Code Generation for Matrix Multiplication*, in GPGPU: Workshop on General Purpose Processor Using Graphics Processing Units, New York, New York, USA, 2016, ACM Press, pp. 22–31.

[27] C. J. ROSSBACH, J. CURREY, AND M. SILBERSTEIN, *PTask: Operating System Abstractions To Manage GPUs as Compute Devices*, ACM Symposium on Operating Systems Principles, (2011), pp. 1–16.

[28] G. ROTHERMEL, M. HARROLD, J. OSTRIN, AND C. HONG, *An empirical study of the effects of minimization on the fault detection capabilities of test suites*, in Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), IEEE Comput. Soc, 1998, pp. 34–43.

[29] G. ROTHERMEL AND M. J. HARROLD, *Selecting tests and identifying test coverage requirements for modified software*, in Proceedings of the 1994 international symposium on Software testing and analysis - ISSTA '94, 1994, pp. 169–184.

[30] G. ROTHERMEL, R. H. UNTCH, C. CHU, AND M. J. HARROLD, *Test Case Prioritization: an Empirical Study*, Proceedings of the IEEE International Conference on Software Maintenance, (1999), p. 179.

[31] P. J. SCHROEDER AND B. KOREL, *Black-box test reduction using input-output analysis*, in Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '00, New York, NY, USA, 2000, ACM, pp. 173–177.

[32] A. M. SMITH, J. GEIGER, G. M. KAPFHAMMER, AND M. L. SOFFA, *Test suite reduction and prioritization with call trees*, in Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, New York, NY, USA, 2007, ACM, pp. 539–540.

[33] M. STEUWER, C. FENSCH, S. LINDLEY, AND C. DUBACH, *Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code*, in Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, New York, NY, USA, 2015, ACM, pp. 205–217.

[34] S. TALLAM AND N. GUPTA, *A concept analysis inspired greedy algorithm for test suite minimization*, SIGSOFT Softw. Eng. Notes, 31 (2005), pp. 35–42.

[35] S. YOO AND M. HARMAN, *Regression testing minimization, selection and prioritization: a survey*, Software Testing, Verification and Reliability, 22 (2012), pp. 67–120.

[36] S. YOO, M. HARMAN, AND S. UR, *Highly scalable multi objective test suite minimisation using graphics cards*, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 6956 LNCS, Springer Berlin Heidelberg, 2011, pp. 219–236.

[37] M. YOUNG AND M. PEZZE, *Software Testing and Analysis: Process, Principles and Techniques*, John Wiley & Sons, 2005.

[38] Z. YU, J.-H. CHO, B.-W. OH, AND L.-S. LEE, *Parallel Algorithm for Generation of Test Recommended Path using CUDA*, International Journal of Engineering and Technology, 5 (2013), pp. 438–443.

Appendix A

Recursion

This appendix shows the manual rewrite of the recursive `amatch` function into a non-recursive function. The function is found in the **replace** benchmark, which is part of the SIR repository [9]. The source code of the function is given below.

It is not necessary to completely understand what the function does, in order to study its structure and see that the recursion is not trivial. The recursive call is made on line 18, inside two nested `while` loops (lines 7 and 17) and an `if` statement. The value returned by it determines whether the inner `while` loop is exited. Even then, the outer `while` loop may continue executing, causing further recursive calls. The function terminates when the outer `while` loop is exited.

```
1  int amatch(char *lin, int offset, char *pat, int j)
2  {
3    int   i, k;
4    bool  result, done;
5
6    done = false;
7    while ((!done) && (pat[j] != ENDSTR))
8      if (pat[j] == CLOSURE) {
9        j = j + patsize(pat, j);
10       i = offset;
11       while ((!done) && (lin[i] != ENDSTR)) {
12         result = omatch(lin, &i, pat, j);
13         if (!result)
14           done = true;
15       }
16       done = false;
```

```
      while ((!done) && (i >= offset)) {
        k = amatch(lin, i, pat, j + patsize(pat, j));
        if (k >= 0)
          done = true;
        else
          i = i - 1;
      }
      offset = k;
      done = true;
    }
    else {
      result = omatch(lin, &offset, pat, j);
      if (!result) {
        offset = -1;
        done = true;
      }
      else
        j = j + patsize(pat, j);
    }

  return offset;
}
```

While this is not a trivial recursion to modify, studying its the control flow helps identifying how a stack can be used to turn it into a non-recursive function. The resulting code is the following:

```
int amatch(char *lin, int offset, char *pat, int j, int recur)
{
  int   i;
  bool  result, done;
  done = false;

  int stack[10];
  int iter = 0;
  int jrec, offrec;

  stack[iter++] = offset;
  stack[iter++] = j;
  stack[iter++] = recur;

  while(iter > 0)
```

```
{
  recur = stack[--iter];
  j = stack[--iter];
  offset = stack[--iter];

  while ((!done) && (pat[j] != ENDSTR))
  {
    if (pat[j] == CLOSURE)
    {
      j = j + patsize(pat, j);
      i = offset;
      while ((!done) && (lin[i] != ENDSTR)) {
        result = omatch(lin, &i, pat, j);
        if (!result)
          done = true;
      }
      done = false;
      jrec = j;
      offrec = offset;

      if(!done && i >= offrec)
      {
        stack[iter++] = i;
        stack[iter++] = jrec + patsize(pat, jrec);
        stack[iter++] = 1;
        recur = 0;
        break;
      }

      done = true;
    }
    else
    {
      result = omatch(lin, &offset, pat, j);
      if (!result)
      {
        offset = -1;
        done = true;
      }
      else
        j = j + patsize(pat, j);.inp
    }
    }

    if(recur > 0)
    {
      done = false;
      if(offset >= 0)
        done = true;
      else
        i = i - 1;

      if(!done && i >= offrec)
      {
        stack[iter++] = i;
        stack[iter++] = jrec + patsize(pat, jrec);
        stack[iter++] = 1;
      }
    }
  }
  return offset;
}
```

Appendix B

Usability Testing

B.1 Test Program - add-multi.c

```c
#include <stdio.h>
#include <stdlib.h>

int add(int a, int b){
  return a + b;
}

int main(int argc, char* argv[]){
  if(argc < 2){
    printf("Please, enter how many integers you'd like to add.\n");
    return 0;
  }
  int num_ints = atoi(argv[1]);

  int sum = 0;
  for(int i = 0; i < num_ints; i++){
    char str[10];
    fgets(str, 10, stdin);
    sum = add(sum, atoi(str));
  }

  printf("sum = %d\n", sum);
}
```

B.2 Actions

1. Become familiar with the **add-multi.c** program:
   - Identify what the inputs and outputs to the program are.
   - Identify what a test case should look like.
2. Write 5 test cases in the CSV format required by ParTeCL.
3. Write the configuration file, required by ParTeCL.
4. Run ParTeCL on the source code and configuration file.
5. Build the CPU runtime.
6. Run the tests.
7. Inspect results. Are they what you expected?

B.3 Questions

On a scale from 1 to 5, where 1 is *not at all easy* and 5 is *very easy*:

1. How easy to use do you find the CSV format for test cases?
2. How easy to use do you find the format for the configuration file?
3. How easy did you find to run ParTeCL?
4. How easy did you find to build the CPU runtime?
5. How easy did you find to run the tests?
6. Overall, how easy did you find the whole process?

Do you have any comments and suggestions?