THE UNIVERSITY
of EDINBURGH

Accelerating Software Test Execution Using GPUs

Vanya Yaneva-Cormack

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2021

Abstract

Today, software is all around us, impacting our everyday lives in fundamental ways. Developing software whose behaviour is reliable, predictable and correct is therefore crucial. This has made software testing a critical part of the development process and has led to the emergence of rigorous testing practices and standards. Testing any non-trivial system, however, is time-consuming and takes up the bulk of development time. Modern software engineering practices involve the repeated execution of large test suites, as part of regular build, test and release cycles. A common approach to speeding up testing without sacrificing rigour is distributing test executions among computer clusters and cloud servers, but this can be complex and expensive due to the costs of testing infrastructure and energy consumption.

This thesis presents a novel approach to accelerating test execution by parallelising it using Graphics Processing Units (GPUs) - powerful and low-cost hardware accelerators that are readily available in the majority of modern desktops. It demonstrates that GPUs can be used to dramatically reduce test execution time at a lower cost compared to other parallel approaches. To achieve this, it addresses significant challenges related to usability, performance and scope, and makes three separate contributions:

First, a GPU testing framework, ParTeCL, is developed to automatically transform the system under test into GPU source code and launch test execution in parallel on the GPU threads. ParTeCL performs the entire testing process transparently without requiring any expert GPU programming and architecture knowledge.

Second, two types of systems are used to evaluate the applicability and effectiveness of the approach - sequential C programs form the embedded systems domain and Finite State Machine (FSM) models. To enable testing them on the GPU, compiler-based transformations and FSM implementations are developed and included in ParTeCL.

Finally, GPU performance is extensively analysed and optimised through a combination of standard and domain specific techniques. Evaluation on programs from the two domains demonstrates that the GPU outperforms a standard 16-core Central Processing Unit (CPU) by up to $4\times$ (avg. $1.4\times$) for embedded systems and up to $9\times$ (avg. $4.5\times$) for FSMs.

The techniques developed in this thesis demonstrate the exciting possibilities of using specialised hardware architectures, such as GPUs, for the acceleration of software test execution. Through integration into the testing process, they could provide rapid feedback, reducing the amount of costly bug-fixing in later stages of development.

Lay Summary

Software testing is the process of ensuring that a given computer program does what we intend it to. It involves the repeated execution of the program, giving it different parameters, and observing if it behaves as expected every time. The parameters used for testing are called test inputs. As a program is developed and grows in size and complexity, so does its set of test inputs. Normally, software engineers execute the set of tests after each change to the program to ensure that the change did not introduce new faults. This is a repetitive process, which is not performed manually by the engineers, but automatically by a computer. As most programs have many tests, executing all of them often can be a very time-consuming process, leading to delays in the software development process and loss of productivity. It is often said that testing accounts for at least 50% of a project's cost.

Graphics processing units (GPUs) are a type of very fast computer processor that is able to perform many of the same operations in parallel, using different inputs. This thesis proposes using them to speed up software testing by executing multiple tests in parallel on them. This is not easy to do, as GPUs are a niche type of processor that is challenging to use without knowledge of its hardware and programming models. The work in this thesis develops tools for software engineers to be able to use GPUs for software testing without the need to program them themselves. It then uses these tools to speed up test execution for two types of computer program and shows that, for these programs, GPUs are indeed faster than conventional computers when executing tests.

Acknowledgements

First, I would like to thank my supervisor, Dr. Ajitha Rajan, for her invaluable guidance in the past four years. Throughout this time she was a constant source of inspiration and encouragement, continuously helped me see the big picture in our research, and was always available when I needed motivation and support.

I would also like to thank Dr. Christophe Dubach for co-supervising my PhD and providing crucial advice that has helped shape the technical approaches, experimental designs and results analysis in this work.

I would like to thank my colleagues from the University of Edinburgh for the friendships, emotional support and good times. I am deeply grateful to have met you all. The PhD experience would not have been the same without you.

Finally, I would like to thank my lovely parents and sister for always being my champions, and my dear Fraser for being my rock.

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. Some of the material used in this thesis has been published in the following papers:

- Vanya Yaneva, Ajitha Rajan, and Christophe Dubach.
  "Compiler-assisted test acceleration on GPUs for embedded software".
  In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2017.

- Vanya Yaneva, Arnav Kapoor, Ajitha Rajan, and Christophe Dubach.
  "Accelerated Finite State Machine Test Execution Using GPUs"
  In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018.

(*Vanya Yaneva*)

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Software forms the basis of modern technology and is present in all areas of life. Today, software is embedded in household appliances, personal devices and everyday objects. It is widely used in diverse sectors including industry, healthcare, transport, finance and retail, among others.

With the growing use of software, developing reliable systems whose behaviour is predictable becomes critical. This has made rigorous software testing increasingly important, but thoroughly testing any non-trivial system is a complex and time-consuming process. Literature regularly states that testing effort accounts for the majority of the cost of development [1, 2]. This problem is heightened with the use of widespread software engineering practices, such as test-driven development and continuous integration [3–5], which rely on systematic testing. They provide improvements to overall system quality and reductions in development costs, but involve the generation of extensive test suites for each system component and regular test executions for every build and release, making testing an even larger portion of the development process.

An important part of these practices is *regression testing* - the process of re-running tests after every change in order to ensure that no errors have been introduced in the existing functionality. As software evolves, new tests are added to the test suites, making regular test execution increasingly expensive and time-consuming.

These observations are confirmed by industry accounts. Facebook [6] and Google [7] state that even with the enormous resources dedicated to testing, they are unable to perform regression testing using all tests for each code change due to the large number of tests and high rate of changes. Google report that on an average day they perform 150 million automatic test executions at enormous compute cost, and they experience increased lag in time between code check-ins and test result feedback to developers.

The research community has developed a body of work addressing this problem. The focus has been on developing techniques to reduce the number of executed tests without sacrificing testing rigour [8] and on distributing test executions across computer clusters and cloud servers [9]. However, the first set of approaches could lead to reduced testing effectiveness [10, 11], while the second could be complex to set up [12] and expensive in terms of the cost and maintenance of the testing infrastructure [7].

This thesis addresses the problem of accelerating test execution by leveraging the high degree of parallelism available on modern Graphics Processing Units (GPUs). GPUs are powerful, readily available and low-cost hardware accelerators aimed at graphics processing, which have been successfully applied to a wide range of applications in other domains [13–16]. This thesis demonstrates that through the parallel execution of tests on separate GPU threads, GPUs can be successfully incorporated into the testing process and used to speed up test execution.

1.1 Software Testing and GPUs

The focus of this work is the acceleration of software testing aimed at checking functional correctness. This is the type of testing that ensures that the system behaves as intended, which is distinct from testing for non-functional properties, such as speed, portability or usability. It involves the execution of the program with different test inputs and verifying that it produces the expected outputs. For complex systems, thorough testing requires a large number of tests for each component. As a system evolves, the number of tests in its test suite grows, along with the time necessary to execute them.

GPUs could prove well suited to parallelising test execution and reducing the time it takes. In the general case, tests are both data parallel and independent, as the tested functionality is executed multiple times over separate independent inputs. This is precisely the computation pattern for which GPUs are designed, making a compelling case for their use for test execution. GPUs are also a cost-effective option for acceleration, present in virtually any modern computer. With the right tooling, they could allow developers to execute tests and receive feedback quickly on their local machines, before submitting their changes to shared development environments.

Nevertheless, using GPUs to parallelise test execution has until now remained an unexplored area. This work aims to change this. The following sections present the key challenges and contributions of this thesis.

1.2 Challenges in Using GPUs for Software Testing

GPUs could offer reduced test execution time at lower cost compared to other parallel architectures, but there are significant challenges, related to usability, scope and performance, that must be overcome to achieve this goal.

1.2.1 Usability

As a niche architecture targeted at accelerating graphics computations, GPUs are notoriously challenging to program. Writing GPU programs requires expert knowledge and the use of specialist programming models, such as OpenCL [17], CUDA [18] and SYCL [19]. Of these, SYCL provides the highest level of abstraction, allowing programmers to write their GPU programs using standard C++, but even with it understanding of the underlying GPU hardware is necessary. This represents a high barrier to entry in adopting and experimenting with GPU approaches for test acceleration, as the majority of software engineers do not have the necessary GPU programming expertise. This makes automated frameworks that are flexible and intuitive to set up essential for the use of GPUs in software testing.

1.2.2 Scope

GPU programming models and compilers are based on the C/C++ programming languages. As a result, the scope of applications which can be tested using GPUs is limited to C/C++ programs, and to system models which can be implemented in C/C++ code. To evaluate the applicability of using GPUs for software testing, this thesis focuses on two types of systems: (1) sequential C programs from the embedded systems domain and (2) Finite State Machine (FSM) models, using examples from the network intrusion detection and digital signal processing domains.

In addition, due to hardware restrictions, there are standard C features that are not supported for compilation on the GPU. Such features include dynamic memory allocation, global scope variables and standard library functions. They pose further restrictions to the scope of the approach and their impact is evaluated in this thesis.

1.2.3 Performance and Scalability

GPUs achieve high speeds compared to Central Processing Units (CPUs) due to their highly parallel architecture, capable of launching thousands of threads at the same

time, but simply compiling and running an application on the GPU does not guarantee performance. GPU performance is subject to a number of factors related to the architecture, programming model, executed application and associated data. To establish the effectiveness of using GPUs to accelerate test execution, it is crucial to understand their importance in that context. Some of the most common factors that hinder GPU performance are:

Data transfer overhead. GPUs have their own memory and data needs to be explicitly copied between it and main memory before and after execution. Depending on the amount of data, these data transfers can be slow, adding considerable overhead to GPU time and severely impacting performance. With respect to testing, test inputs and outputs need to be transferred and the overhead could be significant for the target applications, as it is precisely programs with large test suites that are likely to benefit the most from parallel test execution on the GPU.

GPU Memory Bandwidth. In GPUs, memory bandwidth is shared among thousands of threads impacting the performance of memory intensive applications [20]. Applications written by expert programmers specifically for the GPU can alleviate this by taking advantage of *memory coalescing*, in which data used by neighbouring threads is stored in contiguous memory blocks and retrieved from memory at once. However, this is a GPU specific optimisation that cannot be expected in every tested application and shared memory bandwidth could have considerable performance impact.

Choice of GPU Parameter Values. Selecting appropriate values for GPU configuration parameters plays an important role in achieving optimal performance. They depend on the specific GPU hardware, the implementation of the program and the structure and size of program data. Finding the optimal configuration for a given GPU architecture is a challenging problem, which has been addressed through the use of dynamic techniques [21, 22].

Limited Memory. GPUs have a limited amount of memory which tends to be smaller than main memory. In addition, unlike CPUs, GPUs do not have access to a hard disk. This could pose a limitation to the scalability of the approach, as large test suites comprised of large amounts of data may not fit into GPU memory.

1.3 Problem Statement

Thorough software testing is a crucial part of the development process that ensures the correct behaviour, reliability and quality of the developed system. However, testing is often time-consuming and expensive, taking up the majority of development time. Regression testing involves the repeated test executions of large test suites as part of daily development and overnight builds and adds considerable strain to the software development schedule. GPUs are widely available and inexpensive parallel accelerators that could be well suited to speeding up test execution, but using them poses significant challenges in terms of usability, scope, performance and scalability.

The goal of this thesis is to address these challenges by improving the usability of GPUs for the purpose of test execution, evaluating the applicability and extending the scope of the approach and evaluating and improving its performance and scalability.

1.4 Contributions

This thesis makes three main contributions. Table 1.1 shows which challenges are addressed by each contribution.

1.4.1 Automated GPU Test Execution

A GPU testing framework, called ParTeCL[1], is developed to automatically perform test execution in parallel on the GPU, using the OpenCL programming model. ParTeCL performs two tasks: (1) it translates the tested application into an OpenCL program, which can be compiled and executed on the GPU and (2) it launches instances of the tested program on the GPU threads, each with a separate test input.

ParTeCL addresses the usability challenge (Section 1.2.1) by launching test execution on the GPU transparently, relieving programmers from the need to write any GPU-specific code. Furthermore, it facilitates the techniques used to address the scope and performance challenges (Sections 1.2.2 and 1.2.3) by providing an automated framework in which to implement them.

[1] ParTeCL - **Par**allel **Te**sting in Open**CL**

1.4.2 Accelerated Embedded System and FSM Testing

The proposed approach is applied to two types of applications: (1) sequential C programs from the embedded systems domain and (2) FSM models from the network intrusion detection and digital signal processing domains. Testing is of paramount importance to both of them and they can both be instrumented using the GPU programming models (Section 1.2.3). Nevertheless, there are still scoping challenges involved in using GPUs to test them. This thesis evaluates the applicability and feasibility of the approach and addresses these challenges by applying compiler-based transformations to C features that are not readily supported on the GPU. Furthermore, additional OpenCL implementations for FSM test execution are implemented in ParTeCL.

1.4.3 Performance Analysis and Optimisations

GPU performance for test execution is optimised using a combination of approaches. First, standard techniques to minimise the latency of data transfer are implemented in ParTeCL. Then, GPU performance for both application domains is analysed and optimisation approaches are developed, implemented and evaluated. Evaluation demonstrates that with optimisations, the GPU is faster than a standard 16-core CPU by up to $4\times$ (avg. $1.4\times$) for embedded systems and up to $9\times$ (avg. $4.5\times$) for FSMs. This addresses the performance challenge (Section 1.2.3) by demonstrating that GPUs can achieve better performance than their counterpart multi-core CPUs available on the same machine.

Contribution	**Challenge**
Automated GPU Test Execution (1.4.1)	Usability (1.2.1), Scope (1.2.2), Performance and Scalability (1.2.3)
Accelerated Embedded System and FSM Testing (1.4.2)	Scope (1.2.2)
Performance Analysis and Optimisations (1.4.3)	Performance and Scalability (1.2.3)

Table 1.1: Thesis contributions and the challenges they address.

1.5 Publications

The ideas and results presented in this thesis are based on three previous publications and one publication which is currently under revision. Table 1.2 summarises the thesis chapters, the contributions which they make and the publications to which they relate.

The framework for automatic test execution on the GPU (ParTeCL), presented in Chapter 4 was first presented in:

1. *Yaneva, V., Rajan, A. & Dubach, C. ParTeCL: Parallel Testing Using OpenCL. In ISSTA 2017.* [23]

Chapter 5 presents accelerated testing of embedded systems, which was previously published in:

2. *Yaneva, V., Rajan, A. & Dubach, C. Compiler-assisted Test Acceleration on GPUs for Embedded Software. In ISSTA 2017.* [24]

FSM testing using GPUs, presented in Chapter 6, was first published in:

3. *Yaneva, V., Kapoor, A., Rajan, A. & Dubach, C. Accelerated Finite State Machine Test Execution Using GPUs. In APSEC 2018.* [25]

Finally, the approach to improving the scale and performance of FSM testing presented in Chapter 7 is currently submitted and under revision in:

4. *Yaneva, V., Rajan, A. & Dubach, C. GPU Acceleration of FSM Input Execution: Improving Scale and Performance. In STVR 2020 (under revision).* [26]

This thesis reproduces the experimental results and analysis found in the above publications. It also offers background information (Chapter 2) and a survey of existing literature (Chapter 3), which includes references to recent work in the related fields.

1.6 Structure

This thesis is organised as follows:

Chapter 2 provides background information relevant to the motivation, methods, evaluation and results in this thesis. It describes the software testing process and provides details on GPU architecture and programming, embedded systems and FSMs.

Chapter	Contribution	Publication
2. Background		
3. Related Work		
4. Parallel Test Execution Using GPUs	1.4.1, 1.4.2, 1.4.3	[23]
5. Testing Embedded Software: Evaluating Applicability and Performance	1.4.2, 1.4.3	[24]
6. Testing Finite State Machine Models: Establishing Feasibility	1.4.2, 1.4.3	[25]
7. Testing Finite State Machine Models: Evaluating Performance and Scale	1.4.2, 1.4.3	[26]
8. Conclusion		

Table 1.2: Thesis chapters, contributions and related publications.

Chapter 3 provides an overview of existing work in five related areas: software test acceleration, the use of GPUs in software testing, testing of embedded systems and FSMs and automatic GPU code generation.

Chapter 4 describes the general method of executing tests in parallel on the GPU. It presents ParTeCL - the automated framework developed for this purpose and describes the general performance optimisations that are included in its implementation. The approach and tools presented in this chapter are used in the rest of the thesis.

Chapter 5 applies the approach to the testing of sequential C programs from the embedded system domain. It provides an evaluation of applicability and performance using applications from the EEMBC industry-standard benchmark suite [27].

Chapter 6 establishes the feasibility of using GPUs to accelerate FSM testing. Different designs for the FSM and test suites are considered, implemented and evaluated in order to choose the optimal implementation in terms of execution time. ParTeCL is extended to support FSM test execution.

Chapter 7 extends the work presented in Chapter 6 by improving the performance and scalability of the approach for FSM test execution. Evaluation using 15 large FSMs from the network intrusion detection domain is performed, demonstrating that, with optimisations, GPUs can execute FSM tests up to 9$\times$ faster than a 16-core CPU.

Chapter 8 summarises the main findings of this thesis, provides a critical review and discusses potential future work.

1.7 Summary

The growing dependence on software in all areas of human life has made software testing a crucial part of the development cycle in order to ensure the correctness, reliability and robustness of the developed systems. While essential, software testing is often a time-consuming and expensive part of the development process. GPUs could prove successful in accelerating test execution with greater performance and at lower cost compared to other parallel approaches. Achieving this requires overcoming three main challenges, concerning usability, scope and performance and scalability. The next two chapters provide technical background and discuss existing related work, while subsequent chapters present and evaluate techniques to address these challenges.

Chapter 2

Background

2.1 Introduction

This chapter provides relevant background information to aid the understanding of the problem and solutions presented in this thesis. First, it presents the software testing process and defines the terms used throughout this thesis in Section 2.2. This is followed by descriptions of the GPU architecture, memory hierarchy and programming model in Section 2.3, focusing on aspects relevant to parallel test execution and the optimisations presented in this work. The application domains to which the approach is applied, embedded systems and FSM models, are introduced in Sections 2.4 and 2.5, respectively. Finally, Section 2.6 concludes.

2.2 Software Testing

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.
>
> *Edsker W. Dijkstra*

Software testing is a process used to evaluate whether a system meets the *functional specifications* for its behaviour [28]. It aims to demonstrate that a system behaves as intended, by executing it using a range of inputs and checking that the outputs are as expected every time. Testing to ensure that a system meets other non-functional requirements, such as speed, portability or usability, is outside the scope of this thesis.

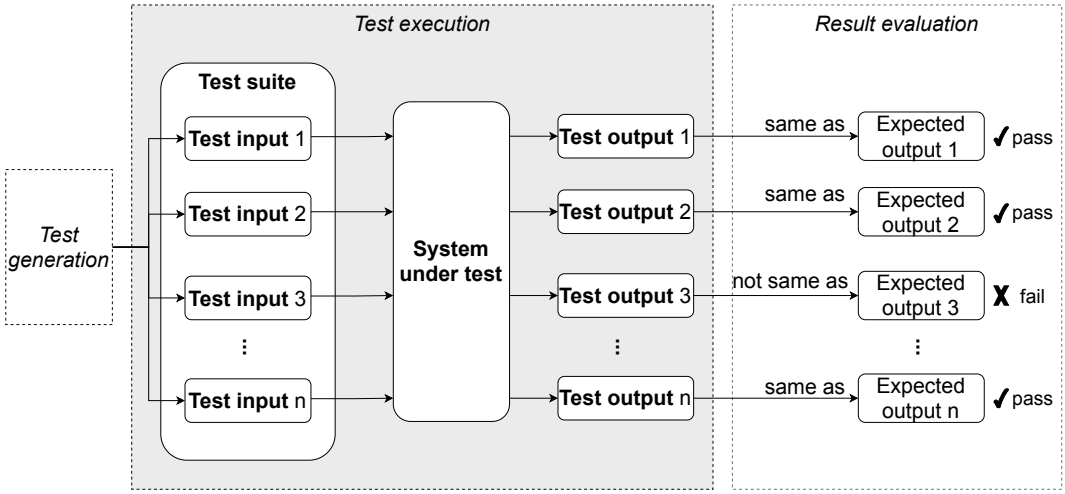


Figure 2.1: The software testing process. It consists of three stages: test generation, test execution and result evaluation. This thesis focuses on using GPUs to accelerate *test execution*.

Figure 2.1 illustrates the software testing process. First, tests are generated (*test generation*). This can be done either through the manual writing of tests or through the use of automated techniques. Tests consist of separate inputs to the system under test (SUT) and expected outputs. Next, the SUT is executed repeatedly, once with each test input, and its outputs are recorded (*test execution*). Finally, the outputs are compared to the expected outputs (*result evaluation*). A test passes if the two outputs are the same and fails if they are not.

2.2.1 Term Definitions

Informal words referring to aspects of software testing can be intuitive and testing terms are not always used consistently in literature. For this reason, it is useful to define the terms used throughout this thesis.

- **system/program/application under test (SUT/PUT/AUT)** - the software system or part of the system that is tested; for simplicity, SUT is used throughout this thesis

- **test case** or **test** - a set of inputs and a pass/fail criterion, also known as *test oracle*; typically, the oracle is given in the form of an expected output, but could also consist of other criteria

- **test suite** - a set of test cases; a test suite for a system may by made up of several test suites for individual modules, subsystems or units
- **test input** - input data for the SUT that is part of a test case
- **test output** - the output produced by the SUT during test execution
- **expected output** - the correct output for a given test input, based on the SUT's specification
- **test result** - indication whether a test execution passed (i.e. the system met its requirements) or failed.

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

int find(int array[], int n, int number) {
  for (int i = 0; i < n; i++) {
    if (number == array[i]) {
      return i;
    }
  }
  return -1;
}

int main(int argc, char **argv) {

  // input error checking is omitted for brevity

  int n = atoi(argv[1]);
  int array[n];

  printf("Enter %d array numbers.\n", n);
  for (int i = 0; i < n; i++) {
    scanf("%d", array + i);
  }
  printf("Enter a number to find.\n");
  int number;
  scanf("%d", &number);

  // perform a search
  int found_idx = find(array, n, number);

  // output answer
  printf("Number found at idx: %d\n", found_idx);
}
```

Listing 2.1: Linear search program, implemented in C.

2.2.2 Example: Linear Search

To illustrate software testing, Listing 2.1 shows a simple C program, which performs a linear search. The core function is `find()` on lines 4-11. It has three inputs: an array of integers *array*, its length *n* and a number *target*, for which to search in the array. It performs a linear search in *array* and returns the first index at which *target* is found, or -1 when *target* is not in the array. The `main()` function accepts values for *n* through the command line (line 17) and for *array* and *target* through standard input (lines 22 and 26) and calls `find()` to perform the linear search. Finally, it prints the output in standard output.

The functional specification for linear search is the following:

1. It accepts as inputs an array of integer values, the length of the array and an integer number for which to search in the array.

2. It returns the first index at which number is present in the array; it returns -1 if the number is not present in the array.

3. If the array is empty, it returns -1.

Table 2.1 shows an example test suite for the linear search program based on this specification, which contains four tests. Listing 2.2 shows an implementation for these tests in GoogleTest [29] - an automated testing framework for C/C++ programs. The SUT is the `find()` function and all tests are structured following the process illustrated in Figure 2.1: (1) the test inputs are declared and initialised, (2) the SUT is executed with the test inputs and the test output is recorded and (3) the test output is compared to the expected output (using the assertions available in GoogleTest, e.g. `EXPECT_EQ`).

Test id	**Test input**			**Expected output**
	n	**array**	**target**	
1	5	{1,2,3,4,5}	3	2
2	5	{1,2,1,2,1}	2	1
3	5	{1,2,3,4,5}	10	-1
4	0	{}	5	-1

Table 2.1: Example tests for the linear search program.

```
TEST(linear_search_test, test_id_1) {
  int n = 5;
  int array[] = {1,2,3,4,5};
  int target = 3;
  int output = find(array, n, target);
  EXPECT_EQ(output, 2);
}

TEST(linear_search_test, test_id_2) {
  int n = 5;
  int array[] = {1,2,1,2,1};
  int target = 2;
  int output = find(array, n, target);
  EXPECT_EQ(output, 1);
}

TEST(linear_search_test, test_id_3) {
  int n = 5;
  int array[] = {1,2,3,4,5};
  int target = 10;
  int output = find(array, n, target);
  EXPECT_EQ(output, -1);
}

TEST(linear_search_test, test_id_4) {
  int n = 0;
  int array[] = {};
  int target = 5;
  int output = find(array, n, target);
  EXPECT_EQ(output, -1);
}
```

Listing 2.2: Example tests for the linear search program, implemented in GoogleTest.

2.2.3 Testing Practices

The purpose of software testing is to provide confidence that a system's behaviour conforms to specification. Ideally, test suites will exhaustively sample the entire input space of a system, but in practice this is infeasible, as even trivial programs would need many billions of tests, that would be impossible to execute in any practical amount of time. For example, to exhaustively test a program which takes two 32-bit integers as arguments, $2^{64} \approx 10^{21}$ tests are needed. At one nanosecond (10^{-9} seconds) per test, this would take approx. 10^{12} seconds, which is about 30,000 years [28]. For this reason, the goal of software testing is to uncover faults in the system which can be fixed. Sufficient testing should eventually provide enough confidence that no critical faults remain.

Testing can be applied at any level of system granularity - individual functions, modules and entire systems. *Unit testing* checks the behaviour of the smallest functional units of a program. *Integration testing* checks the correctness of the interactions between units and modules. *System testing* focuses on the behaviour of the system as a whole.

Regression Testing Regression testing aims to ensure that changes to the existing code do not introduce new faults in the system. It is used regularly in continuous integration and test-driven development, usually as soon as a change to the program is made. Developers often build and test multiple times a day, both locally on their own machines and remotely, on dedicated central servers. Automated overnight builds and test executions for the whole system are also a common practice. These testing practices are repetitive and, combined with large test suites necessary for thorough testing, they are time-consuming, creating a crucial need of time efficient test execution techniques.

2.3 GPU Architecture and Programming

GPUs are highly parallel hardware accelerators, designed for the efficient processing of large blocks of data. They do not work by themselves but together with a general-purpose CPU form a single heterogeneous system.

2.3.1 Architecture

Figure 2.2 illustrates the GPU architecture. GPUs consist of compute units, each of which contains a number of processing elements, which execute the individual GPU threads. The functions executed by the GPU threads are called **kernels**.

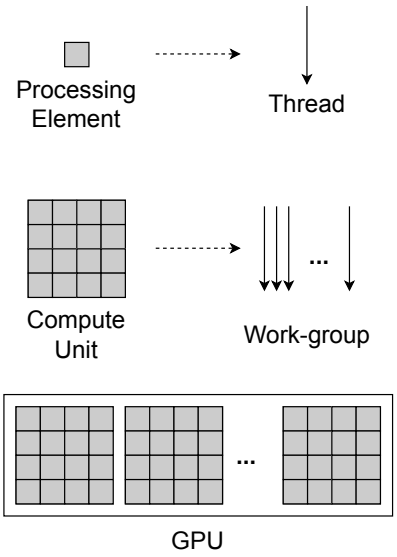


Figure 2.2: GPU architecture. GPUs consist of compute units, each of which contains multiple processing elements that execute the individual GPU threads. All threads belonging to a compute unit are organised into a work-group (in the OpenCL programming model). The functions executed by the GPU threads are called **kernels**.

All threads in a compute unit are organised into groups, which in OpenCL are called work-groups[1] [30]. Threads belonging to the same compute unit follow the Single Instruction Multiple Data (SIMD) execution model - each thread executes an instance of the same kernel over different input data. With respect to software testing, this corresponds to running instances of the same SUT with different test inputs.

Threads in the same compute unit share the same instruction counter, and instruction execution is performed in lock-step - all threads execute the same instruction at any one time. If there is control-flow divergence across threads, divergent instructions will be serialised, negatively impacting performance. Similarly, when the workload is not balanced across threads, some threads would stay idle, resulting in reduced performance.

2.3.2 Memory Hierarchy

GPUs have a memory hierarchy, which is illustrated in Figure 2.3. The placement of data during GPU execution can have significant impact on performance. The GPU memory hierarchy consists of the following regions:

[1] In OpenCL, GPU threads are referred to as work-items. For simplicity, this thesis uses the term *thread* throughout.

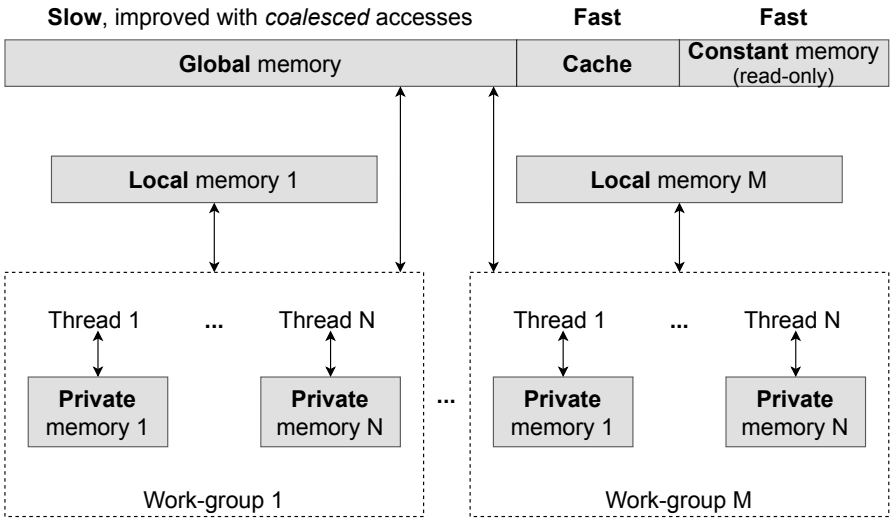


Figure 2.3: GPU memory hierarchy. Placement of data during GPU execution can have significant impact on performance.

- *Global memory* - large and slow, shared among all threads in all compute units. Performance is improved, when accesses are *coalesced* during execution, i.e. when all threads access consecutive addresses in global memory. Performance can also be improved through the use of the cache.
- *Constant memory* - a read-only portion of global memory, which contains a special cache allowing faster memory access.
- *Local memory* - local to compute units, shared among threads in a work-group.
- *Private memory* - private to individual threads.

GPU memory is usually smaller than main memory and is the only storage available to the GPU, as it has no access to the hard disk. For this reason, input data needs to be moved from main memory to GPU memory before kernel execution. Similarly, output data needs to be moved from GPU memory back into main memory after kernel execution. These data transfers can add considerable overhead to GPU execution time, resulting in negative impact on performance.

2.3.3 Programming Model - OpenCL

GPUs require the use of specialist programming models. They include CUDA [18], OpenCL [17] and SYCL [19]. Based on the C/C++ programming languages, they

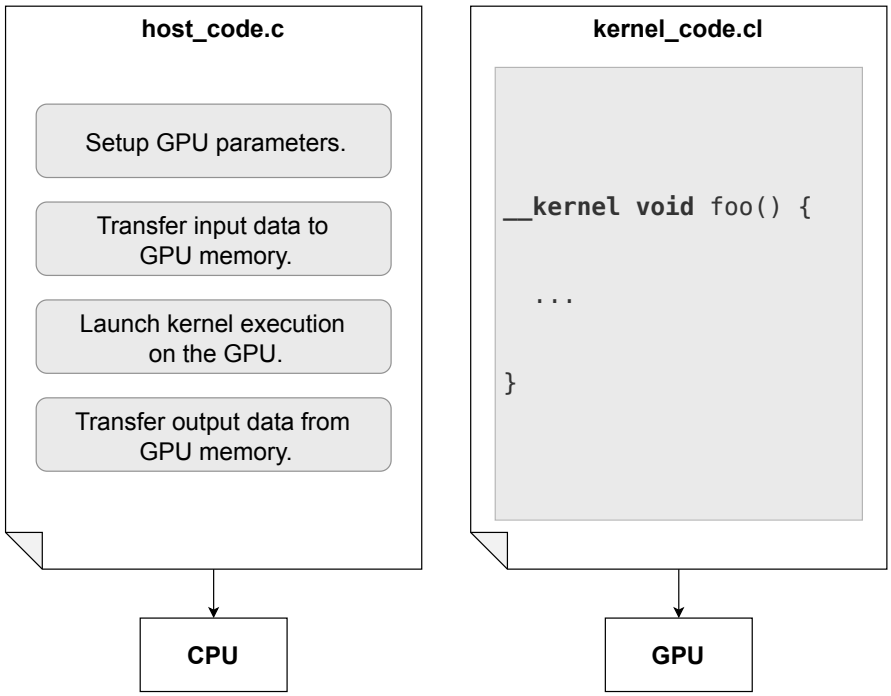


Figure 2.4: The OpenCL programming model. An OpenCL program consists of a host part, executed by the CPU, and a kernel part, executed by the GPU.

expose low-level hardware details and require the programmer to explicitly express the parallelism in terms of the architecture. Each model differs in the degree to which they abstract GPU architecture, but all of them require specialist knowledge from the programmer in order to achieve fast GPU performance.

This work uses OpenCL for multiple reasons. It is an open standard, which is maintained by an industry consortium of over 140 companies that include hardware designers such as the Intel Corporation, NVIDIA, Qualcomm, AMD and ARM. Its C-like syntax allows for the automatic translation of the SUT into an OpenCL kernel. In addition, it provides cross-platform portability, making possible future research on testing with other parallel heterogeneous architectures. Finally, it provides a relatively low level of abstraction for the GPU hardware, allowing for fine-grained performance optimisations and analysis. At the time of writing, OpenCL 1.2 [30] is the version most commonly supported by hardware vendors and is the one used in this thesis.

Figure 2.4 illustrates the basic structure of an OpenCL program. It consists of a host part, executed by the CPU, and a kernel part, executed by the GPU. The host part is written in C or C++ and uses the OpenCL host API. Its role is to setup GPU execution parameters, transfer data to/from GPU memory and compile and run the GPU

kernel[2]. The kernel part is written in OpenCL. During GPU execution, each thread executes an instance of the kernel application, using the input data transferred by the host application.

2.4 Embedded Software

Embedded systems are a type of computer that is integrated into a larger system. They consist of hardware and software components that form a computation unit designed to perform a particular function within another system. Crucially, unlike conventional software, embedded systems often operate in interaction with the physical world by reading and reacting to inputs from different environmental sensors.

Embedded systems are ubiquitous. They are found in the consumer, industrial, transportation, telecommunication, medical and military sectors, among others. Developing dependable embedded systems is a critically important, but difficult process, which relies on the use of rigorous methods and standards [31].

Characteristics Common characteristics of embedded systems are:

1. Interactions with the environment; embedded systems are designed to react to inputs from the physical world and other components in the larger system, leading to many possible interactions and a high degree of complexity in the software.

2. Strict requirements for safety, reliability and availability; as embedded systems are connected to the physical environment and have a direct impact on it, it is critical that their behaviour is dependable.

3. Restrictions on resource consumption; these could be memory, power, run-time and code size.

4. Majority of embedded software is written in the C programming language.

These characteristics have an impact on the design, development and testing of embedded software. They require a high degree of engineering expertise and knowledge of the application domain and hardware. Particular emphasis is placed on the requirements and design phases of development. Formal methods and model-driven approaches are often used as a way to ensure that all quality requirements and resource consumption constraints are incorporated into the system design [31].

[2]The GPU kernel can be compiled by the host application at runtime. Alternatively, it can be compiled beforehand and read as a binary by the host application.

Testing Thorough and rigorous testing is another critical part of embedded software development. Ebert and Jones [32] estimate that testing takes up 15% to 50% of total project duration and is a major cost driver for embedded software development.

One of the key challenges associated with embedded software testing is generating tests which accurately represent environmental inputs [33]. Most interfaces in embedded systems are non-human interfaces (e.g. temperature and pressure sensors) which produce a large range of potential inputs. It is crucial for embedded software to behave predictably in the face of different, often unpredictable, inputs. Several methods are used for the generation of embedded software tests. These include tests derived from the system requirements and design models, sampling of input parameter combinations, the use of rigorous code coverage criteria, and the use of static and dynamic analysis [31,34]. In practice, these methods result in large test suites which are time-consuming to execute repeatedly as part of regular regression testing [32, 33].

2.5 Finite State Machines

This thesis considers FSMs in the context of model-based development - a widespread software development approach, in which software is implemented and verified based on a model of the required system. FSMs are a useful abstraction which is used to model a large variety of systems, including embedded systems, control circuits, signal processing tools and communications protocols. Industry tools which use FSMs to model computer systems include Simulink [35], IBM Rational Rhapsody [36] and Sparx Systems Enterprise Architect [37].

FSMs are commonly classified as two types that are very similar, Moore and Mealy machines, with the Mealy definition being more general. Mealy machines have a finite number of states and given a current state and an input, they transition to a new state and produce an output. A formal definition of a Mealy FSM is found in [38]:

Definition 1. A finite state machine M is a quintuple

$$M = (I, O, S, \delta, \lambda)$$

where I, O and S are finite non-empty sets of input symbols, output symbols and states, respectively.

$\delta : S \times I \to S$ is the state transition function and

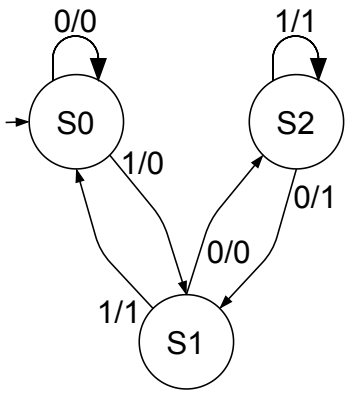
$\lambda : S \times I \to O$ is the output function.

When the machine is in a state s in S and receives an input a from I it moves to the next state specified by $\delta(s,a)$ and produces an output $\lambda(s,a)$.

An example FSM is shown in Figure 2.5a as a state-transition diagram. The input to this FSM is a binary number and the output is the input divided by 3. Thus, the sets of input and output symbols I and O are both $\{0,1\}$ and the set of states S is $\{S0,S1,S2\}$.

Starting State Generally, FSMs have a single initial state which they enter when they are reset. Throughout this thesis, this is referred to as the *starting state*. The starting state for the divide-by-3 FSM in Figure 2.5 is $S0$.

Accepting State Some FSMs may have one or more accepting states, indicating whether the received input sequence is accepted or not. Such FSMs may not produce outputs at each transition, but a single output once all input has been processed, indicating if the reached state is accepting or not.



(a) State-transition diagram. Each transition is labelled as $input/output$.

	0	1
S0	(S0, 0)	(S1, 0)
S1	(S2, 0)	(S0, 1)
S2	(S1, 1)	(S2, 1)

(b) State-transition table.

Figure 2.5: An example of a divide-by-3 FSM. The machine takes a binary number, as an input sequence of bit values, and produces the number divided by 3 as an output.

2.5.1 Complete and Partial FSMs

A *complete* FSM is one, in which every state has a transition for each input. These FSMs are also known as *deterministic* FSMs. In contrast, in a *partial* FSM, an input could have no transition for a given state, i.e. a partial FSM has missing transitions.

Density Chapters 6 and 7 refer to the *density* of a given FSM. It is calculated as the percentage of transitions present in the FSM out of the number of all possible transitions for that FSM, which is $|S| * |I|$. Partial FSMs have densities that are less than 100%.

FSMs can be represented as state-transition tables, in which rows represent states and columns represent inputs from the input set. Each entry in the table is a tuple (next state, output), which encodes the transition of the FSM, corresponding to the respective state and input. In other words, the element for row s and column a of the matrix is the tuple $(\delta(s,a), \lambda(s,a))$. A *sparse* transition table indicates that transitions are missing for a large number of state/input pairs. On the other hand, a *dense* table indicates that there is a transition for most state/input pairs.

Figure 2.5b shows the transition table for the divide-by-3 FSM. This is a complete FSM, which has a full transition table and a density of 100%.

2.5.2 Testing FSMs

FSM testing can refer to two distinct activities.

The first aims to confirm that a system implementation generated from an FSM model, manually or automatically, is behaviourally equivalent to the model. This is *not* the FSM testing activity that is the focus in this thesis, but it is related. In this activity, the FSM is used to generate suitable tests which, when executed on the implemented system, will confirm that the implementation conforms to requirements. It relies on the assumption that the FSM is an accurate model of the system requirements. This is the activity which is commonly referred to as *Finite State Machine Testing* in literature, covering problems like state identification, state verification and machine verification. There is extensive literature on these problems dating back to the 1950's [38, 39].

The second aims to check that an FSM model accurately captures the high level requirements. Its goal is to confirm that a system that is developed based on a model will deliver what is required of it. A practical approach to confirming the behaviour of FSM models, employed in industry, is the generation and execution of large test suites, which can be costly and time-consuming. This problem is the focus of Chapters 6 and 7 of this thesis. A test checking the behaviour of an FSM consists of an input sequence to the machine and an expected output sequence. Executing such a test involves applying the inputs in the sequence one by one, commencing at the starting state, transitioning through the states of the FSM, and recording the outputs associated with each transition. The test passes if the output sequence is the expected one and fails otherwise.

Example: Digital Oscilloscope The problem with time consuming FSM test runs was first brought to the author by an industrial partner, Keysight Technologies [40]. Keysight provide electronic measurement solutions to the wireless communications, aerospace and semiconductor industries. Their systems are modelled using FSMs that get tested extensively. They report that testing to validate their FSM models is part of the test cycles for test-driven development and there is tension between the need for test suites that achieve full coverage and the need for short test execution.

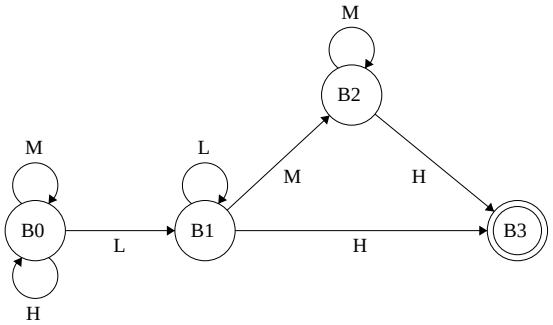


Figure 2.6: An FSM model of a digital oscilloscope from Keysight Technologies.

To illustrate this problem, consider an example designed by Keysight. Figure 2.6 shows an FSM, which is used to identify and trigger particular measurements of interest performed by a digital oscilloscope [41]. The inputs L, M and H correspond to Low, Medium and High frequencies and this particular FSM is designed to identify a rising edge in the digital signal. In order to ensure that the FSM identifies exactly the required type of digital pattern, the system engineers at Keysight perform testing on it. They execute input sequences and observe the output behaviour, checking both for false positives and false negatives.

While this is a simple example, the aforementioned test execution problem is encountered with large FSMs with thousands of states, depending on the input pattern they are designed to identify. Such FSMs require millions of tests to ensure that all parts of the FSM are tested adequately and provide confidence in its correctness. Executing all the tests can take arbitrarily long times, adding cost to the project and strain to the project schedule.

2.6 Summary

This chapter provides background information related to the problem and methodology presented in this thesis. Information on the software testing process demonstrates the problem of time-consuming test executions and the importance of solving it. Introduction to the GPU architecture, memory hierarchy and programming model is essential for the understanding of the main approach and performance optimisations in this thesis. Finally, the sections on embedded systems and FSMs introduce the application domains, to which the techniques proposed in this thesis are applied.

Chapter 3

Related Work

3.1 Introduction

The problems, solutions and applications examined in this thesis relate to five separate areas of research. The main problem belongs to the field of *accelerating software testing*. Section 3.2 reviews the existing work in this field, including traditional approaches and the use of parallel architectures. The proposed solution, execution of test cases in parallel on the GPU, falls within the broader area of *using GPUs for software testing*, which is reviewed in Section 3.3. This thesis applies the proposed solution to two types of systems, *embedded software* and *finite state machines*. Sections 3.4 and 3.5, respectively, provide summaries for the existing work on testing for both of them. Finally, the automated approach presented in Chapter 4 compliments existing research on *automated GPU code generation*, which is summarised in Section 3.6.

3.2 Accelerating Software Testing

Accelerating software testing is an important problem, which has received a lot of interest, examining both test generation and test execution. The related work in the context of this thesis is with respect to test execution.

3.2.1 Minimisation, Selection and Prioritisation

Optimising test execution is a long-standing problem and research in this area spans four decades. The objectives are reduced test execution costs in terms of both time and resources, without sacrificing effectiveness in terms of fault finding. Traditional

approaches are primarily focused on manipulating the test suite and popular techniques are test suite minimisation, test case selection and test case prioritisation. Yoo and Harman [8] provide a comprehensive survey of these approaches.

- **Test suite minimisation** [42], also called **test suite reduction**, is the process of systematic removal of tests from a test suite, aiming to reach the minimum number of tests which satisfy some criteria, usually a measurement of code coverage. The goal is to optimise the test suite by removing tests that over time have become redundant with respect to the testing requirements for which they were generated. Finding the minimum test suite is an NP-hard problem and existing work has been focused on developing heuristics [43–45] and algorithms [46, 47] to guide this process.

- **Test case selection** [48] is a technique similar to test suite minimisation, but instead of focusing on a single version of the SUT, it aims to select a subset of tests for execution, covering the changes between two versions of the applications. In the context of regression testing, the goal is to avoid executing tests whose outcome will not have changed between the current and the previous versions of the program. Therefore, the selected set of tests will be different for each regression. Particular techniques for test case selection all aim to identify the test cases in the given test suite that cover the modifications in the program. Existing approaches are based on data-flow analysis [49–51] and graph-walking for control flow and control dependence graphs [52–55]. Test case selection techniques are surveyed by Biswas et al. [56], while Kazmi et al. [57] review recent empirical studies on their effectiveness in terms cost, coverage and fault finding, all of which are important objectives. In [58], Yoo and Harman formulate test case selection as a multi-objective problem and evaluate three algorithms. They allow software engineers to optimise test suites based on multiple objectives and observe trade-offs between them.

- **Test case prioritisation** [59] is a technique which reorders test cases based on some desirable criterion, with the aim of detecting faults as early during execution as possible. This is particularly useful in situations in which there are fixed time and resource budgets allocated for testing. Popular criteria used for test case prioritisation include code coverage [60–62], fault detection rates [63–66] and system requirements [67–69]. A systematic literature review on test case prioritisation criteria and techniques is provided by Khatibsyarbini et al. [70].

Recent work attempts to apply these techniques in real-world regression testing at scale. Elbaum et al. [71] report that traditional test selection and prioritisation approaches that rely on code coverage and instrumentation are too expensive for Google's large codebases. They propose two new cost-effective selection and prioritisation techniques, based on historical data of tests that were recently executed and revealed faults. Similarly, Herzig et al. [72] from Microsoft dismiss code coverage for test selection due to its runtime overhead, and present a cost-based test selection strategy based on historic test performance data. Machalica et al. [6] from Facebook also use historical testing data for test case selection, but in a novel way. They train a statistical model to select a subset of tests for a particular code change using basic machine learning techniques on data on previous code changes and test outcomes on those changes. All three papers report significant reductions in regression testing time and costs.

A major risk associated with test suite optimisation approaches is omitting the execution of tests which reveal faults in the SUT. Yoo and Harman [8] summarise the findings of four studies on the effect of test suite minimisation on fault finding. The studies provide contradictory results, with [73, 74] finding negligible reduction in fault finding, while [75, 76] show a considerable negative effect. Yu et al. [10] apply ten different test suite minimisation techniques to a set of eight programs and report that higher reduction in test suite size tends to negatively impact fault finding. A similar conclusion is reached by Heimdahl and George [77] for test suites form model-based test generation. Elbaum et al. [11] compare five different techniques for test case prioritisation over eight programs and conclude that the rate of fault detection after the application of test suite prioritisation varies considerably across different attributes of the SUT, test suites, and program modifications. In addition, Inozemtseva et al. [78] show that structural coverage of the code, one of the common criteria used for these approaches, does not have a strong correlation with test effectiveness when test suite size is controlled for, while a study by Namin and Andrews [79] concludes that both coverage and test suite size are important for fault finding. The results in these studies demonstrate the difficulty in guaranteeing that these approaches will not negatively impact fault finding for a given program and testing scenario. Therefore, when software correctness is of critical importance, executing the entire test suite would still be preferred. In these situations, parallel hardware can be leveraged to accelerate test execution without the need to modify the test suite. Furthermore, even in cases when minimisation, selection and/or prioritisation are successfully used, parallel test execution can still be utilised as a complimentary approach to further speed up testing.

3.2.2 Parallel Test Execution

Distributing test executions on parallel hardware infrastructure has become a widespread practice that has received attention both in industry and academia, using clusters of multiple machines to execute tests in parallel. Kushneryk and Barnett [80] use an auxiliary test environment, comprised of additional PCs, laptops and/or servers, to run test cases in parallel with the primary test environment. Misailovic et. al. [81] present a constraint-based algorithm that combines test generation and execution of structurally complex test inputs in parallel. They apply their approach to an application developed at Google, using 1024 machines from Google's infrastructure. Garg and Datta [9] combine test prioritisation with parallel test execution. They use a functional dependency graph to partition the test suites for web applications into prioritised test sets that can be distributed for execution on multiple machines. Gupta et al. [82] present a method for automatic machine configuration for parallel test execution, which targets not only large test suites, but also multiple software configurations.

The main drawback of these approaches is the costs associated with building, maintaining and operating parallel CPU clusters at a large scale [7]. This has led to the exploration of using cloud-based services as a way to reduce the costs associated with building dedicated testing infrastructure. Parveen and Tilley [83] explore this idea by taking into account the characteristics of the SUT and the types of testing performed on the application, stating that in certain situations cloud-computing "can aid in reducing the execution time of large test suites in a cost-effective manner". In [84], they propose a parallel test execution environment for cloud services using the MapReduce programming model [85] and Hadoop [86]. Yu et al. [87] present an elastic Test-as-a-Service platform to automatically cluster, schedule, and manage unit testing, including test generation, test execution and result reporting. More recently, Gambi et al. [88] present a framework for cloud unit testing, targeting Java unit tests, which automatically allocates computational resources and efficiently schedules test execution on them, including in the presence of test dependencies. Despite its advantages, cloud-based test execution might not be suitable for all projects, due to networking and bandwidth challenges, loss of autonomy and security, and potential lack of support for particular features or technologies, on which the SUT is dependent [83]. It could also still be prohibitively expensive for projects with limited budgets and large testing workloads. For these reasons, there is still a need for low-cost efficient test acceleration achievable on local infrastructure.

Another drawback of these approaches are the usability challenges associated with creating and maintaining a parallel test environment, which requires knowledge of testing frameworks and parallel technologies. A survey on test execution parallelisation for open-source projects by Candido et al. [12] found that only 15% of 110 project surveyed used parallel test execution. Among the teams who did not use it, the most common reasons cited are the extra work to organise testing, as well as lack of continuous integration services and unfamiliarity with the underlying technology.

Unlike CPU clusters, GPUs provide large scale parallelism at low cost, but they too pose significant challenges associated with ease of use (Section 1.2). The work in this thesis addresses this issue by presenting an automated approach to leverage GPU parallelism to accelerate test execution.

3.3 Using GPUs for Software Testing

There is a growing interest in the software testing community in utilising the massive performance advantages offered by GPUs. Yoo et al. [89, 90] successfully use GPUs to parallelise three search-based [91] algorithms for multi-objective test suite minimisation. Their results demonstrate that due to the data parallel nature of search-based algorithms, they are a perfect fit for the GPU and the approach scales well with the sizes of the SUT and test suite. Li et al. [92] build on this work by proposing a parallel GPU search-based algorithm for multi-objective test case prioritisation. More recently, in [93], Celik et al. use GPUs to accelerate test input generation for bounded-exhaustive testing [94], which generates test inputs, up to a given bound, based on a formal specification for the properties of desired test inputs. They define an abstract representation for candidate inputs and use it to derive a new technique for parallel test generation on the GPU.

The work carried out in these papers is complimentary to this thesis. They use GPUs for test generation and test suite optimisation, while this research focuses on execution. This idea is first presented by Rajan et al. in [95]. They define three key points of the approach: (1) the program and its logic remain unchanged, (2) the changes required to run tests on the GPU are only to the program interface and (3) the program is launched as a GPU kernel with each thread using a different test input. The approach in [95], however, uses manual code transformation and does not address GPU limitations with respect to ease of programming and performance optimisations. The paper presents promising preliminary performance results, but lacks a detailed evaluation and analysis of the applicability and scope of the approach. This thesis addresses these issues.

3.4 Testing Embedded Software

Embedded software is characterised by high requirements for quality, safety, reliability and availability in the face of often unpredictable inputs, which are supported by rigorous testing practices. To allow for the cost-effective testing of embedded software, industry and academia have proposed different approaches over the years. Garousi et al. [96] present a recent comprehensive literature survey of the area.

The majority of work has been focused on the design, generation and evaluation of effective test suites. There are many approaches to test generation for embedded systems, including model-based and requirements-based testing [97–99], partition testing [100] and coverage-based testing [101]. As the focus of this thesis is on test execution, thorough survey of test generation methods is outside of its scope.

Due to the complexity of embedded systems, test generation methods tend to lead to large test suites, resulting in long test execution, adding to the overhead of testing within development [32]. Traditional approaches for test suite optimisation (Section 3.2), are generally not suited to embedded software, as they do not take into account features specific to it, e.g. time-dependent tasks, and could lead to loss of fault finding which is a crucial concern for embedded software testing. Biswas et al. [102–104] address this by proposing test case selection for embedded software that takes into account additional features present in it, and compare their approach to existing test case selection techniques. Their results show their approach selects an additional 28% test cases with a 36% increase in the fault revealing effectiveness compared to other techniques [104]. However, these conclusions are based on a small set of eight embedded C programs of limited size. Netkow and Brylow [105] present a framework for automatic parallel regression testing of embedded systems on a pool of dedicated target hardware, but their focus is not on performance, but on automation of the testing process, and they do not report performance results. The work in this thesis accelerates embedded test execution without making any modifications to the test suite.

Static Analysis Due to the high quality requirements for embedded software, researchers have also looked into static analysis methods [31] to verify their behaviour. Static analysis is based on abstract interpretation theory [106] and aims to verify the run-time behaviour of a system by performing automatic code inspection and verifying program properties, based on system specification. For example, a static analyser can verify that a program never executes an instruction with undefined behaviour. Multiple

static analysers have been proposed [107–109] and static analysis tools are developed and used in industry. Examples include the Clang Static Analyser [110], Facebook's Infer [111] and tools developed by Coverity [112] and GrammaTech [113]. However, fully automated analysers take a long time to run and produce a large volume of false positives which need to be manually inspected by developers [108]. Therefore, thorough testing using rigorous test suites remains the most common approach for verifying embedded software behaviour.

3.5 Testing Finite State Machines

In model-based software development, the traditional testing process is split into two distinct activities: one activity that tests the model to *validate* that it accurately captures the high-level requirements, and another testing activity that *verifies* whether the code generated (manually or automatically) from the model is behaviourally equivalent to the model [114]. Chapters 6 and 7 use GPUs to accelerate the first activity, but as the two are closely related, this section provides a survey of related work in both.

Testing for FSM Model Validation

The formal verification of models against specification is referred to in literature as *model checking* [115, 116]. Though powerful, in practice, model checking methods can be costly in terms of memory, execution time and effort involved in learning and using them. Notoriously, they suffer from the *state-explosion* problem, where model checking tools fail to process large and complex systems [116].

Thus, as a practical addition to model checking, industry often uses testing to validate that their models behave as expected. In academia, work by Whalen, Rajan et al. [117] improves the effectiveness of this approach by introducing the notion of requirements-based coverage. They define coverage metrics for the adequacy of model and system validation tests, based on formalised high-level system requirements. Follow up work then uses the metrics for the generation of test suites than can be used to validate executable models against the requirements [118].

Testing Based on FSM Models

Testing a program that is based on an FSM model involves the use of the model as a tool for test generation and as a test oracle. In literature, this is what is most commonly

referred to as Finite State Machine Testing. This process has generated extensive literature that can be traced back to the 1950's [119, 120], which is surveyed by Lee and Yannakakis [38] and Broy by et al. [39] in the context of reactive systems. Test inputs for the SUT are generated from the FSM model by using techniques based on the construction of distinguishing sequences [121–124], unique I/O sequences [121, 125–127] and characterisation sets [128–131]. These techniques have high computational complexity and can lead to exponentially long testing sequences, incurring high execution costs. For this reason, there is a body of research focused on generating *minimised* testing sequences [132–135].

An alternative approach is structural testing, in which test suites are generated based on some type of coverage criteria for the FSM. Popular criteria choices are *all-transition*, *all-transition pair* and *full predicate* coverage, formalised in [136], as well as *transition tree* [137], based on the W-method introduced by Chow in [128]. Briand et al. [138] present an empirical investigation into the cost and fault detection effectiveness of the four criteria.They conclude that while the other criteria are inadequate for fault detection or too expensive, all-transition pair coverage offers strong fault detection guarantees, as it ensures that events in the system are tested not only individually, but also in relation to one another. While much more expensive than all-transition coverage, it is also much more rigorous.

In recent years, Hierons and Türker have been using GPUs to accelerate the *generation* of testing sequences for FSMs based on unique I/O sequences [139, 140], state harmonised state identifiers and characterising sets [141] and distinguishing sequences [142].

To the best of the author's knowledge, the work presented in this thesis is the first to explore using GPUs for the execution of FSM model validation tests.

3.6 GPU Code Generation

General-purpose computing on GPUs is successfully used in a wide range of domains [13–16], but GPUs are notoriously difficult to program and extract performance from. As a result, there is a rich body of research dedicated to automatic techniques for GPU programming. This section presents existing work in two relevant areas, automatic parallelisation and high-level programming frameworks.

Automatic Parallelisation

Automatic parallelisation targeting GPUs (and other heterogeneous hardware) relies on the use of compiler techniques. A large portion of this research uses the polyhedral model [143] - a powerful mathematical framework for automatic optimisation and parallelisation of statically predictable loops. Polyhedral optimisations have been integrated into mainstream compilers, most notably through the Polly extension for the LLVM infrastructure [144]. Polly works at the level of intermediate representation (IR), recognises parts the program that fit the polyhedral model and transforms them into a suitable representation, allowing the application of further optimising transformations. Grosser and Hoefler [145] extend Polly to provide code generation for GPU hardware targeting CUDA. Other tools which use the polyhedral model to generate GPU code are C-to-CUDA [146] and the Polyhedral Parallel Code Generator (PPGC) [147]. Instead of targeting compiler IR, both of these tools perform source-to-source transformations, that take sequential C programs and generate optimised CUDA kernels. More recently, Baghdadi et al. [148] presented Tiramisu - a polyhedral framework that targets the generation of high-performance code for heterogeneous hardware, including multi-core CPUs, GPUs and distributed architectures. It performs optimisations on multiple layers of IR, resulting in better performance compared to previous approaches. However, unlike other tools, it does not detect polyhedral code automatically, but requires the programmer to express the parallel algorithms explicitly using a dedicated C++ API. In this regard, Tiramisu is similar to the high-level programming frameworks for heterogeneous programming.

High-Level Programming Frameworks

High-level programming frameworks and domain-specific languages have been proposed to aid programmers in writing efficient code for GPUs and other heterogeneous systems. They allow the programmer to explicitly guide the compiler towards those parts of the application which are suitable for offloading to the GPU.

Recent work in this area includes Lift [149–153] - a functional-style programming language, which provides high-level primitives for the explicit implementation of data-parallel algorithms that is suitable for dense linear algebra applications, stencil computations and some irregularly shaped data structures. Lift relies on a set of rewrite rules to transform the high-level primitives into low-level representations that are then compiled into efficient OpenCL kernels. Other functional programming languages

for GPU code generation include NOVA [154], which also uses high-level primitives and generates CUDA code, and Halide [155], which targets the optimisation of image processing pipelines, but can also be used for other computations [156].

Some GPU programming frameworks have been inspired by the notion of *algorithmic skeletons* [157]. Motivated by the need of higher-level parallel programming models, algorithmic skeletons represent an abstraction of parallel algorithms based on their use of generic patterns of computation and interaction, such as pipelines, task queues and fixed degree divide and conquer. The concept has been widely used in multi-core CPU programming, but has also inspired some GPU programming approaches. SkelCL [158] is framework which provides implementations for algorithmic skeletons in the form of a C++ API, that can be executed on the GPU using CUDA. Bones [159, 160] is a C-to-CUDA compiler, which relies on programmer annotations to determine which skeletons to use and what additional optimisations to perform.

Relevance to Test Execution

The body of work presented in this section provides diverse mechanisms for the automatic generation of GPU code, but ultimately their goal is different to that of this thesis. The related work greatly simplifies the expression and automatic detection of parallelism within software applications, but the challenge for GPU test execution is not in the identification of parallelism - it is inherent in the mapping of test cases to separate GPU threads. Instead, automating test execution on the GPU requires a code generation tool that takes a sequential CPU program and transforms it into a GPU kernel, without affecting the core program functionality. To meet this goal, Chapter 4 presents ParTeCL - the automated testing framework for test execution on the GPU, which is used throughout this thesis.

3.7 Summary

This chapter surveys the relevant literature in the fields of software test acceleration, the use of GPUs in software testing, testing of embedded systems and FSMs and automatic GPU code generation, outlining the ways in which this thesis relates to existing work in these areas.

Chapter 4

Parallel Test Execution Using GPUs

4.1 Introduction

The research contributions of this thesis are built on a novel method for parallel test execution using GPU architectures. This chapter describes the underlying approach and its implementation which is then applied and evaluated using two separate application domains in Chapters 5 to 7.

The proposed approach executes software tests in parallel with each test running on a separate GPU thread. The key idea behind it is that test executions are inherently data parallel and independent, which makes them well suited to parallelisation on the GPU. This idea is first explored by Rajan et al. in [95]. They demonstrate its feasibility on a set of four benchmark applications by manually transforming the program and tests to run on the GPU. Their approach, however, is incomplete in tackling GPU limitations with respect to ease of use, unsupported program features, and performance optimisations.

The approach presented in this thesis addresses these challenges in three stages. First, the method is generalised and automated through the development of a testing framework for the GPU, called ParTeCL. It consists of two parts, (1) ParTeCL CodeGen - a code generation tool that targets C programs and generates OpenCL kernels for them and (2) ParTeCL Runtime - a host application which executes on the CPU, builds the OpenCL kernel and automatically launches tests for execution on the GPU. Through ParTeCL, test execution is fully automated for the GPU, removing any need of specialist GPU programming knowledge.

Second, performance optimisations are implemented, targeting the performance overhead of transferring tests between main memory and GPU memory. These optimisations are: (1) using hardware optimisations for faster data transfer and (2) hiding

data transfer latency, by splitting test suites into groups and transferring them to GPU memory in batches, overlapping data transfer with test execution.

Finally, extensive empirical evaluation is performed, targeting two separate application domains, embedded systems and FSM models. Their unique challenges and results are presented and analysed in Chapters 5 to 7.

The rest of this chapter is organised as follows. Section 4.2 presents the generalised approach for parallel test execution on the GPU. Section 4.3 presents the automated GPU testing framework and its implementation. Section 4.4 describes how automation is used to address the scope challenge presented in Section 1.2.2. Section 4.5 describes the performance optimisations targeting data transfer. Finally, Section 4.6 concludes the chapter.

4.2 General Approach

Figure 4.1 illustrates the general approach for parallel test execution on the GPU. As any OpenCL GPU program, it involves a host application, executed by the CPU, and a kernel application, executed by the GPU (Section 2.3.3). The host application reads the test inputs, transfers them to GPU memory, builds and launches the GPU kernel and transfers the test outputs back from GPU memory to main memory. The kernel application is the system under test (SUT). The GPU executes multiple instances of it in parallel, each on a separate test input.

Figure 4.1 outlines five distinct steps involved in this process:

Step 1. CPU: Read test inputs.

Step 2. CPU: Transfer test inputs to GPU memory.

Step 3. CPU: Build the SUT kernel and launch test execution on the GPU.

Step 4. GPU: Execute the tests in parallel.

Step 5. CPU: Transfer test outputs from GPU memory.

This approach differs from conventional test execution in two ways. First, the SUT is not built by the tester or an automated build system, but by the OpenCL host application. To achieve this, the SUT needs to be wrapped into an OpenCL kernel, which can be compiled by the OpenCL compiler. This requires the use of source-to-source transformations, which are not trivial to perform manually. For this reason, automating

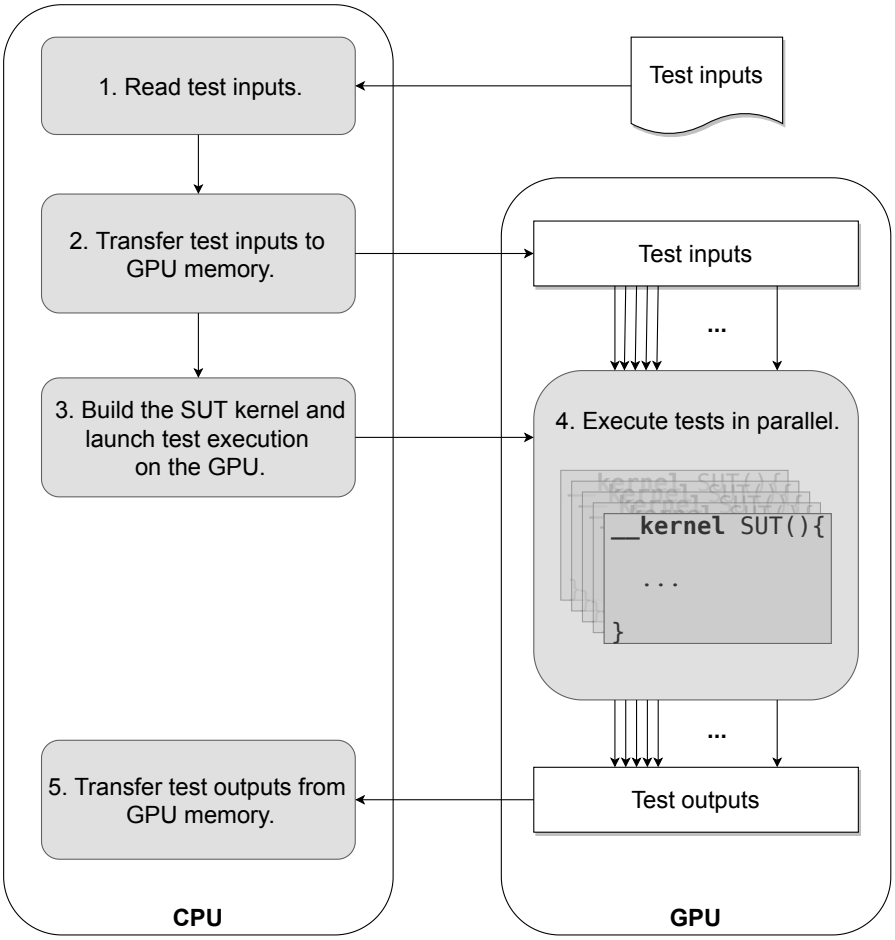


Figure 4.1: Executing tests in parallel on the GPU. The CPU (host) application is responsible for transferring test inputs and outputs to/from GPU memory and launching test execution on the GPU. The GPU executes multiple instances of the system under test (SUT) in parallel, each on a separate test input.

code transformations of the SUT into an OpenCL kernel is crucial. In addition, code transformations can be used to extend the scope of applications which can be tested on the GPU. An automatic tool for code generation for the SUT is described in Section 4.3, while code transformations targeting application scope are presented in Section 4.4.

Second, the tests input values are not read directly by the SUT, but by the host application and transferred to GPU memory. Similarly, test outputs are transferred back from the GPU to main memory after GPU test execution. The time to perform these data transfers adds an overhead to GPU performance. Depending on the size of the data, this overhead could be considerable, outstripping the performance gains of parallel GPU execution. Therefore, it is crucial to optimise data transfers. Such optimisation techniques are presented in Section 4.5.

Restrictions This approach relies on the use of the OpenCL programming model to compile and execute the SUT on the GPU. The extent to which this is possible is dependent on both OpenCL and the GPU architecture and is subject to restrictions. Presently, they include dynamic memory allocation, recursion, file I/O, function pointers and concurrency. Applications which contain these features cannot be currently tested using the GPU. Of these, dynamic memory allocation is the most common feature, causing the biggest restriction to the wide adoption of the approach. Section 4.4 contains more details on currently unsupported features, together with possible solutions to overcoming them in the future.

4.3 ParTeCL - Automating Test Execution on the GPU

This section describes ParTeCL - a framework, which automates test execution on the GPU. Through automation, ParTeCL abstracts away low level GPU details, making the approach accessible to all programmers, and allows for automatic code transformations of program features typically unsupported on the GPU. As discussed in Section 1.2.2, ParTeCL targets the testing of applications written in the C programming language, since GPU programming models are limited to C/C++. All optimisations, experiments and results presented in this thesis are performed using ParTeCL.

ParTeCL consists of two systems, illustrated in Figure 4.2:

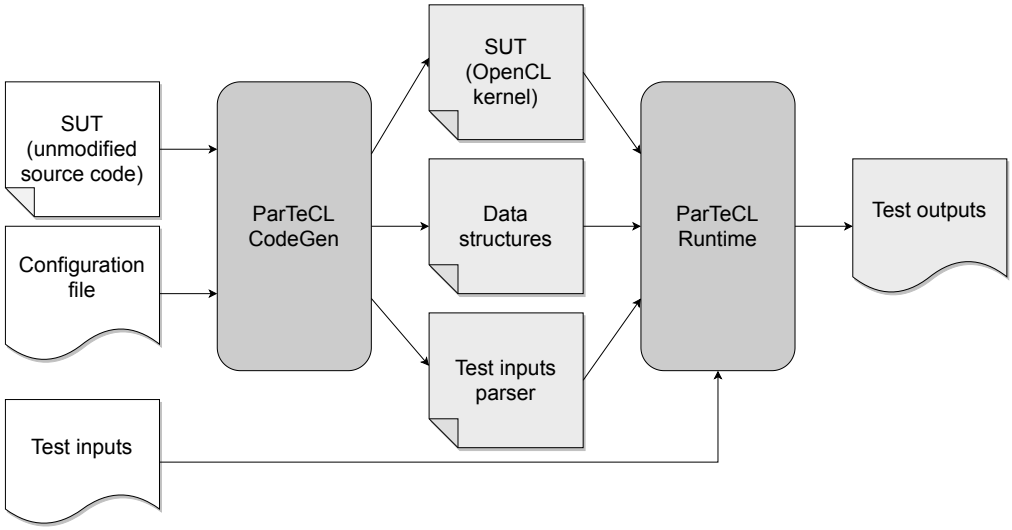


Figure 4.2: ParTeCL system overview. ParTeCL CodeGen uses the unmodified source files for the SUT and a configuration file to generate an OpenCL kernel for the SUT, along with data structures and auxiliary functionality for ParTeCL Runtime. ParTeCL Runtime uses the generated code to transfer test data to/from the GPU, build the SUT and launch test execution in parallel on the GPU, producing the test outputs.

(1) **ParTeCL CodeGen** is a code generation tool, which translates the SUT into an OpenCL kernel. It also generates data structures and functions, which are used by the host application to transfer test inputs and outputs between main memory and GPU memory.

(2) **ParTeCL Runtime** is the host application. It uses the code generated by ParTeCL CodeGen to transfer test data to/from the GPU, build the SUT and launch test execution in parallel on the GPU.

The framework requires three user inputs - the unmodified source files for the SUT, a configuration file and the test inputs. It uses them to transparently execute the tests on the GPU and produce the test outputs. Since ParTeCL's focus is on accelerating test execution, the framework does not perform result evaluation, but produces the test outputs which can be compared to the expected outputs outside of the system.

Example The following sections describe ParTeCL CodeGen and ParTeCL Runtime in detail. To aid understanding, they use the linear search program from Section 2.2.2 as an example, demonstrating how ParTeCL transforms it into an OpenCL kernel and executes its tests on the GPU.

4.3.1 ParTeCL CodeGen

ParTeCL CodeGen performs two tasks: (1) it converts the SUT into an OpenCL kernel and (2) generates data structures and auxiliary functions for ParTeCL Runtime. Its inputs are the unmodified C source code of the SUT and a configuration file, as illustrated in Figure 4.2. This section describes the configuration file and data structure generation first and the OpenCL kernel generation second.

```
input: int n 1
stdin: char* array
stdin: char* number
output: int value variable: found_idx
```

Listing 4.1: ParTeCL configuration file for the linear search program shown in Listing 2.1. It shows that the program has three inputs and one output. One of the inputs is a command line argument and two are standard inputs. The output is an integer and corresponds to a variable called `found_idx` in the program.

```
#ifndef STRUCTS_H
#define STRUCTS_H

#define POINTER_ARRAY_SIZE 1024

typedef struct partecl_input {
  int test_id;
  int argc;
  int n;
  char array[POINTER_ARRAY_SIZE];
  char number[POINTER_ARRAY_SIZE];
} partecl_input;

typedef struct partecl_output {
  int test_id;
  int value;
} partecl_output;

#endif
```

Listing 4.2: Data structures for the linear search program, generated by ParTeCL. They correspond to the program's configuration file, shown in Listing 4.1.

Configuration File and Data Structure Generation The configuration file required by ParTeCL CodeGen describes the test inputs and outputs for the SUT and is used by the tool to generate data structures for them. ParTeCL CodeGen parses the configuration file, line by line, using the keywords and syntax outlined in Table 4.1.

To illustrate this, consider the linear search program from Section 2.2.2 and its tests shown in Table 2.1. Listing 4.1 shows its configuration file. It describes the input/output interface of the program as follows:

- Line 1: The program has one command line input argument, an integer `n`, that is passed at index 1, corresponding to `argv[1]` in the program's implementation.
- Lines 2-3: It has two more inputs, which are passed through standard input, which correspond to variables `array` and `number` in the program.
- Line 4: It has one output, an integer `value`, which corresponds to variable `found_idx` in the program.

ParTeCL CodeGen translates the configuration file into data structures for the test inputs, `partecl_input`, and test outputs, `partecl_output`. The data structures generated for the linear search program are illustrated in Listing 4.2. They correspond directly to the configuration file and contain additional fields for a test id and the value of `argc` for the test.

The current prototype of ParTeCL CodeGen sets a limit on the size of arrays in the data structures by defining a compile-time constant `POINTER_ARRAY_SIZE` (line 4). This size can be changed in the generated code, depending on the SUT and tests. For the purposes of memory efficiency, this size should be set to the smallest possible value which ensures that all test inputs and outputs will be accommodated. If any test input or output exceeds the size set by `POINTER_ARRAY_SIZE`, ParTeCL will not warn the user and tests will fail silently.

In addition, ParTeCL CodeGen generates a parser for ParTeCL Runtime, which is used to read the test input values and assign them to the `partecl_input` data structure. This code is then compiled and used by ParTeCL Runtime for test execution.

OpenCL Kernel Generation ParTeCL CodeGen translates the source of the SUT into an OpenCL kernel which can be compiled and executed on the GPU threads. To achieve this, it uses compiler-based transformations on the source code's Abstract Syntax Tree (AST), using the C front-end of the Clang compiler. ParTeCL CodeGen performs two types of transformation. The first instruments the functions in the SUT to be recognised as OpenCL kernels and changes their input/output interface to read and write test inputs/outputs from/to GPU memory. It does not change the core algorithm of the SUT, ensuring that the tested functionality remains the same. The second type of

Keyword	Usage & Examples

input	Test input supplied as a command line argument:
	type, name and index on the command line.
	Could also be an array (see end of table).
	`input: int a 1`
	`input: int array[10]`

stdin	Test input supplied through standard input; its type is always `char*`. `stdin: char* stdin1`
output	Test output: type, name and what it corresponds to in the program's implementation.
• **function *name* RET**	• when the output is the return value of a function *name*
• **function *name* ARG *idx***	• when the output is the calculated by a function *name* and is its *idx*th argument after execution

• **variable *name***	• when the output is a variable *name*
	`output: int out function: add RET`
	`output: int out function: add ARG 1`
	`output: int out variable: sums`

include	For header files from the SUT, which declare custom types for the inputs and outputs; these headers will be included in the file with data structures generated by ParTeCL CodeGen. `include: algo.h`

	arrays:
	The configuration file supports arrays of constant or variable
	lengths. When variable lengths are used, that variable should
	also be specified in the configuration file.
	`input: int array[n]`
	`input: int n 1`

Table 4.1: ParTeCL configuration file syntax.

transformation handles features which are not supported for compilation on the GPU out of the box, in order to extend the scope of the approach. It is described in more detail in Section 4.4.

To illustrate the transformations, Listing 4.3 shows the kernel generated for the linear search program shown in Listing 2.1. The tool makes the following key changes to the code:

- Lines 1-2: Includes for header files for an OpenCL implementation of the C standard library. It is described in more detail in Section 4.4.
- Line 3: An include for a header file that contains the data structures, generated by ParTeCL, shown in Listing 4.2.
- Lines 16-17: The signature of the `main()` function is changed to turn it into an OpenCL kernel. Its arguments are transformed into two arguments - (1) the test inputs, values for which are initialised by the CPU, and (2) the test outputs, which are to be calculated by the kernel.
- Lines 19-26: Included by ParTeCL CodeGen, in these lines the GPU kernel identifies the GPU thread on which it is running and selects the corresponding test. In particular, the input values for the corresponding test are contained in the variable `partecl_testin`, which is of type `struct partecl_input`, shown in Listing 4.2. The kernel will calculate the test outputs and store them in the variable `partecl_testout`, which is of type `struct partecl_output`. Each test input is identified by an id, which is copied to the output (line 24). This allows identifying the produced outputs as the result of their respective test inputs, once parallel execution is complete and outputs are transferred to CPU memory to be checked.
- Lines 30, 35, 39: References to the command line and standard input inputs are replaced with references to the test inputs inside `partecl_testin`.
- Line 46: Finally, the value of the variable `found_idx` is assigned to the data structure, containing the test output.

ParTeCL CodeGen also removes any printing to standard output, as shown on lines 33, 37, 41 and 45, in order to avoid producing a large volume of messages when large test suites are executed in parallel. This is safe to do under the assumption that printing to standard output has no side effects. To correctly handle printing statements that

```
#include "cl-stdio.h"
#include "cl-stdlib.h"
#include "structs.h"
//#include <stdio.h>
//#include <stdlib.h>

int find(int array[], int n, int number) {
  for (int i = 0; i < n; i++) {
    if (number == array[i]) {
      return i;
    }
  }
  return -1;
}

__kernel void main_kernel(__global struct partecl_input *partecl_testins,
                          __global struct partecl_output *partecl_testouts) {

  int partecl_threadidx = get_global_id(0);
  struct partecl_input partecl_testin = partecl_testins[partecl_threadidx];
  __global struct partecl_output *partecl_testout =
      &partecl_testouts[partecl_threadidx];
  int argc = partecl_testin.argc;
  partecl_testout->test_id = partecl_testin.test_id;
  char *partecl_arrayptr = partecl_testin.array;
  char *partecl_numberptr = partecl_testin.number;

  // input error checking is omitted for brevity

  int n = partecl_testin.n;
  int array[POINTER_ARRAY_SIZE];

  /*printf("Enter %d array numbers.\n", n);*/
  for (int i = 0; i < n; i++) {
    scanf("%d", array + i, &partecl_arrayptr);
  }
  /*printf("Enter a number to find.\n");*/
  int number;
  scanf("%d", &number, &partecl_numberptr);

  // perform a search
  int found_idx = find(array, n, number);

  // output answer
  /*printf("Number found at idx: %d\n", found_idx);*/
  partecl_testout->value = found_idx;
}
```

Listing 4.3: OpenCL kernel for the linear search program, shown in Listing 2.1, generated by ParTeCL CodeGen.

contain expressions with side effects, ParTeCL CodeGen could be extended to replace the printing statement with the expression. In this way the printing itself would be removed, but the side effects will still take place.

These transformations illustrate the changes which ParTeCL CodeGen makes to the input/output interface of the program. Crucially, the tool performs no transformations to the core functionality of this program in lines 7-14. Additional transformations which relate to extending the scope of the applications testable on the GPU, are discussed in Section 4.4.

Implementation ParTeCL CodeGen is implemented in C++14, using the Clang LibTooling library [161]. It consists of two main components, a data structure generator and a kernel generator, corresponding to the two tasks performed by the tool. The data structure generator performs a straightforward translation of the configuration file into data structures and a parser for the test inputs. The kernel generator uses LibTooling's AST Matchers, AST Handlers and Rewriter to apply the code transformations which translate the SUT into an OpenCL kernel. It executes sequential compiler passes, which find the relevant portions of the SUT that need to be transformed. It then uses the Rewriter class to perform the transformations at source code level, producing a readable OpenCL kernel for the SUT, which can be compiled and executed on the GPU. The source code for ParTeCL CodeGen is hosted at [162].

4.3.2 ParTeCL Runtime

ParTeCL Runtime implements the host application's functionality which is executed on the CPU. It uses the code generated by ParTeCL CodeGen to launch tests for parallel execution on the GPU. It performs the four steps presented in Figure 4.1: it reads the test inputs, transfers them to GPU memory, builds the OpenCL kernel and launches test execution on the GPU, and transfers the test outputs into main memory. It is implemented in standard C, using the OpenCL API to perform the GPU related operations. The source code for ParTeCL Runtime is hosted at [163].

```
1 1 5 "1 2 3 4 5" "3"
2 2 5 "1 2 1 2 1" "2"
3 3 5 "1 2 3 4 5" "10"
4 4 0 "" "5"
```

Listing 4.4: ParTeCL test input file, containing example tests for linear search.

Test Inputs ParTeCL Runtime accepts test inputs in a standard Space-Separated Value file. In it, each row corresponds to a test, the first column contains the test id while the rest of the columns contain the input values, in the order in which they are given in the configuration file. Listing 4.4 shows the test file for the linear search example, containing the tests shown in Table 2.1. Based on the configuration file, ParTeCL CodeGen generates code that is used by ParTeCL Runtime to automatically assign test input values to the custom data structures, shown in Listing 4.2.

Work-group Size The choice of work-group size impacts performance, depending on the GPU architecture, kernel and dataset involved in the execution. Choosing an appropriate value is a difficult problem [21, 22]. To enable dynamic experimentation, the work-group size can be supplied as an input parameter to ParTeCL Runtime.

For the performance experiments in Chapter 5, the work-group sizes for all benchmarks were chosen experimentally. The full test suites were executed with work-group sizes 32, 64, 128, 512 and 1024 and the ones that achieved the fastest GPU execution for each benchmark were used for all performance experiments in Chapter 5. The specific values for each benchmark are shown in Table 5.3.

For the performance experiments in Chapters 6 and 7, a fixed work-group size of 256 was used in order to reduce experimentation effort, but choosing different sizes could lead to different performance results.

In future work, existing approaches for work-group tuning could be integrated into the testing process, allowing the evaluation of their impact on the performance of the approach, as outlined in Section 8.3.

4.4 Extending Application Scope

As outlined in Section 1.2.2, there are standard C/C++ features which are not supported for compilation on the GPU and thus limit the scope of programs that can be tested on it. This is due to inherent GPU hardware and programming model restrictions. ParTeCL CodeGen addresses this by performing code transformations for such C features. This section describes the features that are handled by the tool and lists the features which are currently still unsupported.

Global Scope Variables OpenCL does not support the use of global scope variables which are not constant, since writing to them could lead to concurrency issues on

the GPU. Nevertheless, global scope variables are a common feature of C programs. To support these programs, ParTeCL CodeGen moves global scope variables to the local scope of the main kernel function (`main_kernel`), preventing any sharing of the variables among tests. If other functions in the application use a global scope variable, ParTeCL adds a pointer argument for it to the function's argument list. Using a pointer ensures that the variable is passed by reference and if its value is changed by the function, that would be available to the all other functions that may use it, just as it would be with the original global scope variable.

Command Line Arguments, Standard Input and Output These features form the input/output interface of the SUT and are transformed by ParTeCL CodeGen, replacing them with references to the tests' inputs/outputs, as discussed in Section 4.3.1 and illustrated in Listing 4.3. Standard output is commented out by ParTeCL CodeGen, unless it is specified as the test output in the configuration file. In this case, the tool replaces it with a write to the test output data structure.

Standard Library Calls There is no OpenCL implementation for the C standard library, making it impossible to compile code that uses it for execution on the GPU. To address this, an implementation of the C standard library in OpenCL, called clClibc, is started as an auxiliary project to ParTeCL. It is inspired by uClibc [164], a small C standard library typically used for embedded systems.

The code generated for the linear search example in Listing 4.3 illustrates this. Lines 1 and 2 contain includes for cl-stdio.h and cl-stdlib.h, which replace the includes for stdio.h and stdlib.h on Lines 4 and 5. cl-stdio.h contains an OpenCL implementation of the standard `scanf()` function, which is used by the program on Lines 33 and 37.

clClibc currently implements functions from stdlib.h, ctype.h, string.h and stdio.h, and will be extended further. The code for clClibc is hosted at [165].

Unsupported Features C features which are still unsupported for compilation and execution on the GPU include dynamic memory allocation, recursion and file I/O. Testing of applications which contain these features cannot be currently accelerated using the methods presented in this thesis.

Dynamic memory allocation. OpenCL kernels cannot use dynamic memory allocation. Dynamic allocation of GPU memory can be performed only by the host application prior to kernel execution. For applications in which the amount of allocated memory is

known at compile-time, a partial solution for the purposes of testing could be replacing dynamic memory allocation with static allocation in ParTeCL CodeGen. Another, more general solution, can be the implementation of a memory allocator on the GPU, similar to the CUDA programming model [18]. This can be achieved through the pre-allocation of a large enough memory buffer for each OpenCL thread and the use of auxiliary variables to record which portions of the array are allocated and which are free.

Recursion. GPU hardware lacks support for recursion. Removing recursion as part of compiler optimisations has been studied in literature [166]. In the context of testing, this means that the recursive calls can be automatically removed from the SUT without changing functionality. Nevertheless, removing recursion in general is a complex task depending on the particular implementation of the recursive function. It is simplest in the case of tail-recursion, where the recursive call is the last operation in the function and can be replaced with a loop. In the general case, removing recursion involves the use of a stack to track the state and parameters for the function calls, which would require the use of dynamic memory allocation.

File I/O. GPUs do not have access to the file system, unless special OS abstractions are used, which can have significant performance penalties [167]. One way to address this issue could be partitioning the SUT into several kernels, leaving file I/O to be performed by the CPU with data explicitly transferred to GPU memory between separate kernel executions.

Function pointers. OpenCL does not support the use of function pointers on the GPU. This can be addressed by implementing automatic elimination of function pointers in ParTeCL CodeGen. Static analysis can be used to identify the functions to which a function pointer may refer and assign unique integer ids to them. The function pointer itself can be replaced by an integer variable that is assigned the id of the concrete function to which it points. The function call that uses the function pointer can then be replaced with a switch statement which uses the id to identify and call the respective function. This approach is used by Cooper et. al. [168] to handle function pointers and virtual functions for automatic cross-compilation on heterogeneous hardware.

Concurrency. The approach does not support automatic testing of concurrent programs. Testing concurrent applications is a difficult task which requires specific approaches, because concurrency failures often show only under particular circumstances. In addition, significant complexity would be involved in handling both program concurrency and parallel test execution on the GPU. For these reasons, testing of concurrent applications is outside the scope of this approach.

Further research is needed to lessen the impact of the first four features on the applicability of the approach. As GPU architectures and programming models evolve, new ways to tackle these limitations are likely to emerge.

4.5 Optimising Test Transfers to Improve Performance

GPU programs have to copy data back and forth between main memory and GPU memory. In testing, the data is the test inputs and outputs, illustrated in Figure 4.1. This is a slow process that adds considerable overhead to total GPU execution and negatively impacts performance.

This section presents two optimisation strategies, implemented in ParTeCL Runtime. Section 4.5.1 presents the use of Direct Memory Access (DMA) - a hardware optimisation which accelerates data transfer between the CPU and GPU memories. Section 4.5.2 presents Data Transfer Overlap - a strategy in which tests are transferred to GPU memory in groups, overlapping data transfer with test execution and hiding data transfer latency. This is a well-known optimisation strategy for GPU applications, with recent implementations in several CUDA libraries [169, 170].

4.5.1 Direct Memory Access

GPU programming models require for the CPU to explicitly move test inputs from main memory to GPU memory before kernel execution. Similarly, the CPU needs to move the test outputs from GPU memory back into main memory after kernel execution. To do this, the CPU issues explicit reads and writes to transfer data via the bus and is occupied for the duration of the data transfer, adding an overhead to GPU performance.

To address this issue, modern GPUs are equipped with a DMA controller - a special-purpose processor which allows data to be transferred between main memory and a device (the GPU) without occupying the CPU. The DMA controller temporarily takes over control of the bus from the CPU and generates memory addresses and control signals to transfer data between main memory and GPU memory. Figure 4.3 illustrates this process. This results in faster data transfers compared to explicit reads and writes. It is also required for OpenCL to be able to overlap data transfer with GPU execution.

To make use of DMA, specific OpenCL directives are used to setup memory and transfer data in ParTeCL Runtime:

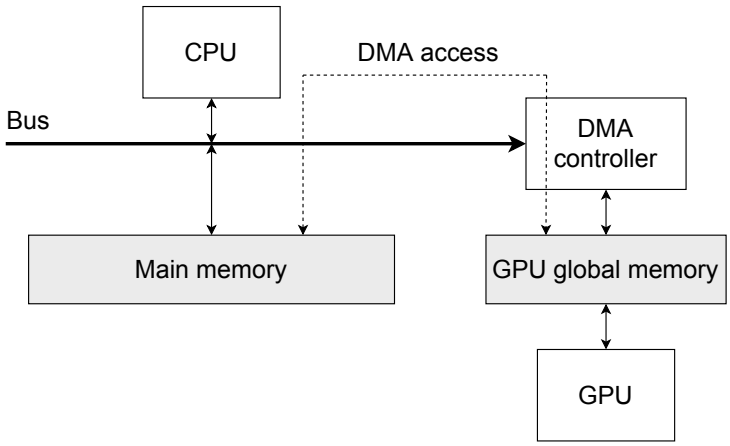


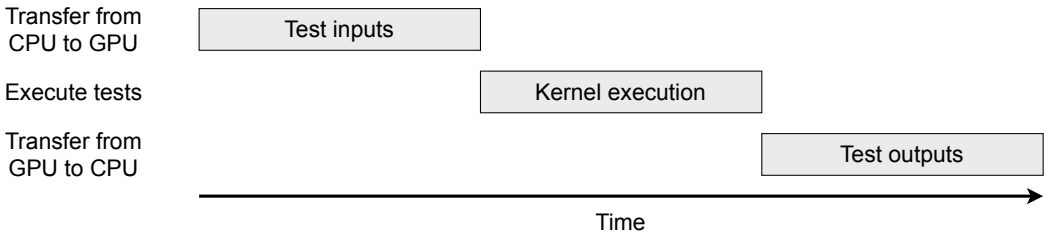
Figure 4.3: DMA between CPU and GPU memory.

- *clCreateBuffer* with flag *CL_ALLOCATE_HOST_PTR* - create OpenCL buffers into main memory, in which test inputs and outputs are stored; these are called *pinned* buffers.
- *clEnqueueMapBuffer* - map the main memory buffers to pointers, which are used to access the data in them directly.
- *clCreateBuffer* - create OpenCL buffers into GPU memory, for test inputs and outputs; these buffers are used directly by the GPU kernel during test execution.
- *clEnqueueWriteBuffer* and *clEnqueueReadBuffer* - transfer the test inputs and outputs between main memory and GPU memory.

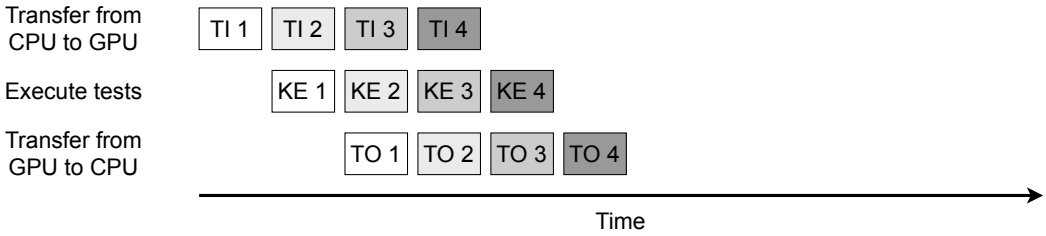
This memory setup allows the OpenCL implementation to perform the data transfers using the DMA controller, which not only accelerates the data transfers, but also enables the overlapping of data transfer with test execution to improve performance. It is important to note that the particular OpenCL directives for DMA access are dependent on the GPU architecture and OpenCL implementation. The setup implemented in ParTeCL Runtime will use DMA on the NVidia Tesla GPU, which is used in the experiments in Chapters 5 to 7. A different implementation might be necessary for other GPU architectures.

4.5.2 Data Transfer Overlap

Figure 4.4a shows the workflow of testing on the GPU without any data transfer overlapping, illustrating the large performance overhead of transferring test inputs and



(a) Without data transfer overlap.



(b) With data transfer overlap.

Figure 4.4: Test execution on the GPU without and with data transfer overlap. Without data transfer overlap, test inputs are transferred to GPU memory as one whole group. Test execution begins after all tests are transferred. Then, after all tests are executed, test outputs are transferred from GPU to CPU memory. In contrast, with data transfer overlap, tests are split into groups and transferred to GPU memory one after the other. Test execution begins as soon as the first group is transferred, overlapped with the data transfer for the next group. Test outputs for each group are transferred back to CPU memory as soon as test execution for the group is finished.

outputs between CPU and GPU memories. For large test suites that take considerable time to transfer, the overhead could be large enough to remove the advantage parallel kernel execution. In addition, due to the limited amount of GPU memory, storing the entire test suite in it at once may not be possible.

To mitigate these problems, ParTeCL Runtime implements data transfer overlap, as illustrated in Figure 4.4b. The test suite is split into groups of tests, which are transferred to the GPU memory one after the other. As soon as a group of tests is transferred, the GPU can start executing them, overlapping execution with the data transfer for the next group. Similarly, as soon as kernel execution is finished, output transfer back to the CPU can start, overlapping data transfer to and from the GPU. This is possible on GPUs equipped with dual copy engines and requires the use of additional OpenCL buffers for implementation.

This approach achieves two objectives:

1) *Performance.* Overlapping data transfer with kernel execution reduces total time spent on the GPU, resulting in faster testing and mitigating data transfer overhead.

2) *Scalability to larger test suites.* Splitting the test suite into groups reduces the amount of memory necessary on the GPU. There is no longer need to allocate GPU memory for all tests, but only for the group that is being executed at each cycle. This allows the execution of larger test suites than is possible otherwise, improving the scalability of the approach.

Implementation Data transfer overlap is implemented within ParTeCL Runtime. Once the tests are split into groups of equal sizes, ParTeCL Runtime performs a loop which transfers each group to the GPU memory, launches the test execution in the GPU kernel, and transfers the outputs back to main memory. Due to the use of DMA for data transfer, the OpenCL implementation takes care for overlapping transfers of test inputs and outputs for each group with their execution.

4.6 Summary

This chapter presents the general approach of parallel test execution on the GPU, along with the design and implementation of an automated testing framework. The developed framework, called ParTeCL, uses compiler-based source-to-source transformations to generate an OpenCL kernel for the SUT and launch test execution in parallel on the

GPU threads. By automating testing on the GPU, the effort involved in using GPUs is reduced, as no expert knowledge of the architecture and its programming models is necessary. In addition, the compiler-based source-to-source transformations facilitate the implementation of techniques that support additional features for compilation on the GPU. Finally, standard techniques to optimise the performance of transferring tests between main and GPU memory are implemented in ParTeCL.

The next chapter evaluates the applicability and effectiveness of this approach when applied to the testing of embedded systems.

Chapter 5

Testing Embedded Software: Evaluating Applicability and Performance

5.1 Introduction

Embedded systems are ubiquitous, featuring in consumer electronics, telecommunication systems and safety-critical systems, such as car sensors, breaking systems and medical devices. The widespread use and close daily interaction with these systems makes safety concerns a top priority when developing and approving embedded software. Testing such software is crucial for gaining confidence in its quality and reliability, and for limiting risks to users and companies. For these reasons, embedded software often needs to adhere to strict standards to ensure quality and safety [31], leading to the use of more sophisticated quality assurance, better quality measures and more test stages compared to other types of software [32]. This typically results in large test suites [171], making testing a major cost driver in development [32].

Existing solutions (Section 3.2) that focus on reducing the number of tests to be executed are not suitable for the field of embedded software as they come at the expense of fault finding effectiveness [77, 78], departing from the crucial goal of ensuring safety. Therefore, new approaches are necessary that accelerate test execution while maintaining the large number of tests.

This chapter evaluates the applicability and effectiveness of using GPUs to automatically accelerate embedded software test executions, using the approach and tools developed in Chapter 4. The evaluation uses applications from EEMBC industry-

standard benchmark suite for embedded systems [27], which is designed to represent real-world embedded systems applications. It is created and maintained by an industry consortium which includes major embedded systems companies, such as Samsung, Sony and Huawei. Performance results show that parallelising test executions on an NVidia GPU achieves a speedup of up to $4\times$ (avg. $1.4\times$), compared to parallel execution on a 16-core CPU.

The rest of this chapter is organised as follows. Section 5.2 presents the approach for analysing applicability and measuring performance. Section 5.3 describes the experimental methodology. Section 5.4 presents and analyses the results. Finally, Section 5.5 concludes.

5.2 Approach

This section outlines the empirical approach taken to evaluate the applicability and performance of using GPUs to accelerate embedded software test execution. This approach consists of three stages steps: (1) choosing representative embedded systems benchmarks, (2) analysing applicability based on them and (3) designing and executing performance experiments.

5.2.1 Embedded Systems Benchmarks

The programs chosen for the evaluation are from the Embedded Microprocessor Benchmark Consortium (EEMBC) benchmark suite. It consists of a diverse suite of 33 benchmarks, written in the C programming language, that span real-world application domains, such as automotive, digital media, networking, office and telecom [27] and are widely used both in industry and academia. A description of the individual subject programs is provided in Table 5.1. They consist of a main algorithm, a test harness, and example input data available for each benchmark.

5.2.2 Analysing Applicability

Figure 5.1 illustrates the approach taken to analyse the applicability of using GPUs to accelerate test execution for embedded software. It aims to answer three questions:

Program	Domain	Description
a2time01	automotive	Angle to time conversion
aifftr01	automotive	Fast Fourier Transform (FFT)
aifirf01	automotive	Finite Impulse Response (FIR) filter
aiifft01	automotive	Inverse Fast Fourier Transform (iFFT)
basefp01	automotive	Basic integer and floating point
bitmnp01	automotive	Bit manipulation
cacheb01	automotive	Cache “Buster”
canrdr01	automotive	Controller Area Network (CAN) remote data request
idctrn01	automotive	Inverse Discrete Cosine Transform (iDCT)
iirflt01	automotive	Infinite Impulse Response (IIR) filter
matrix01	automotive	Matrix arithmetic
pntrch01	automotive	Pointer chasing
puwmod01	automotive	Pulse width modulation
rspeed01	automotive	Road speed calculation
tblook01	automotive	Table lookup
ttsprk01	automotive	Tooth to spark
cjpeg	consumer	JPEG compress
djpeg	consumer	JPEG decompress
rgbcmy01	consumer/filters	RGB to CMYK colour conversion
rgbhpg01	consumer/filters	RGB to HPG colour conversion
rgbyiq01	consumer/filters	RGP to YIQ colour conversion
ospf	networking	Dijkstra shortest path first algorithm
pktflow	networking	Packet flow
routelookup	networking	Route lookup
bezier01	office	Bezier curve algorithm
dither01	office	Floyd-Steinberg error diffusion dithering algorithm
rotate01	office	Bitmap image rotation algorithm
text01	office	Text parsing
autcor00	telecom	Cross corr. of signals
conven00	telecom	Convolution encoder
fbital00	telecom	Bit allocation
fft00	telecom	Fast Fourier Transform (FFT)
viterb00	telecom	Viterbi decoder

Table 5.1: EEMBC benchmark programs.

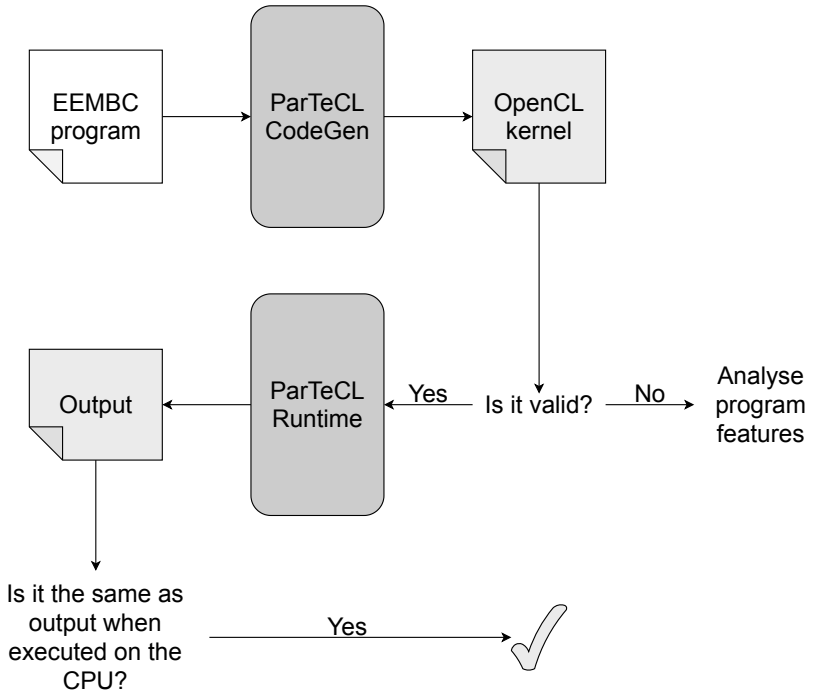


Figure 5.1: Analysing applicability. For each EEMBC program, ParTeCL CodeGen generates an OpenCL kernel, which is used by ParTeCL Runtime to execute the program on the GPU. Benchmarks, whose OpenCL kernels are invalid, i.e. cannot be successfully compiled by the OpenCL compiler, are analysed. For the rest, outputs from the GPU execution are compared to outputs from the CPU.

1. Can ParTeCL CodeGen generate valid OpenCL kernels for embedded programs? A valid kernel is one which can be successfully compiled by the OpenCL compiler.

2. What are the features present in embedded programs which prevent the generation of valid kernels?

3. Does execution on the GPU produce correct program outputs?

To answer these questions, the following steps are performed. First, ParTeCL CodeGen is used to generate an equivalent OpenCL kernel for each EEMBC program. Benchmark programs, for which ParTeCL CodeGen cannot produce valid kernels, are analysed in order to discover which program features are incompatible with OpenCL. Second, the EEMBC programs whose kernels are successfully compiled, are executed on the GPU by ParTeCL Runtime, using the input data provided with the benchmarks. Finally, the benchmark outputs from the GPU are compared to those from CPU execution, in order to empirically verify that correctness is preserved during GPU execution.

Successfully executed on the GPU Table 5.2 shows which of the EEMBC programs are successfully compiled and executed on the GPU by ParTeCL CodeGen and Runtime. In total, there are 17 out of the 33 EEMBC benchmarks which can be executed on the GPU. They consist of 11 out of 16 benchmarks from the automotive domain, all 5 benchmarks from the telecom domain and 1 out of 3 benchmarks from the networking domain. For all of them, their outputs from the GPU are exactly the same as their outputs from the CPU.

Not executed on the GPU The remaining 16 out of 33 EEMBC programs cannot be successfully compiled and/or executed on the GPU. Analysing the source code for these benchmarks reveals that this is due to 5 different features:

1. Dynamic memory allocation, which includes dynamic memory allocation with sizes not known at compile time, as well as other memory operations, such as `realloc` and `memset`.

2. File IO

3. Function pointers

4. Not enough constant memory available on the GPU

5. String literals in data structures

The specific FSMs which contain each feature are listed in Table 5.2. The most common feature is dynamic memory allocation, present in 13 of the benchmarks. The second most common feature is file IO, present in 4 of the benchmarks, all of which also use dynamic memory allocation. One benchmark (`cacheb`) contains function pointers and one benchmark (`ttsprk`) cannot be executed on the GPU, as it uses large statically allocated data tables, that do not fit into the GPU's constant memory. Finally, one benchmark (`pktflow`) cannot be executed on the GPU, as it uses string literals to initialise some of its structures. However, string literals are required by OpenCL to be in constant memory and cannot be assigned to non-constant variables on the GPU.

As all of these features are known limitations of the GPU hardware and programming models. Testing of embedded systems, and other C software, that use these features cannot be accelerated using GPUs.

Program	Domain	Success on GPU?	Reason
a2time01	automotive	✓	
aifftr01	automotive	✓	
aifirf01	automotive	✗	Dynamic memory allocation
aiifft01	automotive	✓	
basefp01	automotive	✓	
bitmnp01	automotive	✓	
cacheb01	automotive	✗	Function pointers
canrdr01	automotive	✓	
idctrn01	automotive	✓	
iirflt01	automotive	✗	Dynamic memory allocation
matrix01	automotive	✗	Dynamic memory allocation
pntrch01	automotive	✓	
puwmod01	automotive	✓	
rspeed01	automotive	✓	
tblook01	automotive	✓	
ttsprk01	automotive	✗	Not enough constant memory
cjpeg	consumer	✗	Dynamic memory allocation
djpeg	consumer	✗	Dynamic memory allocation
rgbcmy01	consumer	✗	Dynamic memory allocation, File IO
rgbhpg01	consumer	✗	Dynamic memory allocation, File IO
rgbyiq01	consumer	✗	Dynamic memory allocation, File IO
ospf	networking	✓	
pktflow	networking	✗	String literals in data structures
routelookup	networking	✗	Dynamic memory allocation
bezier01	office	✗	Dynamic memory allocation, File IO
dither01	office	✗	Dynamic memory allocation
rotate01	office	✗	Dynamic memory allocation
text01	office	✗	Dynamic memory allocation
autcor00	telecom	✓	
conven00	telecom	✓	
fbital00	telecom	✓	
fft00	telecom	✓	
viterb00	telecom	✓	

Table 5.2: Success in executing EEMBC programs on the GPU. Out of 33 programs, 17 are successfully executed on the GPU and 16 are not.

Summary Using GPUs to accelerate test execution is partially applicable to embedded systems programs. ParTeCL is able to compile and execute half of the EEMBC benchmark programs on the GPU, particularly for the automotive and telecom domains.

This demonstrates the that the scope of the approach is currently restricted by known limitations originating from the GPU hardware and programming models. Further research is necessary to address these restrictions.

5.2.3 Evaluating Performance

To evaluate the performance of GPU test execution for embedded system, a set of performance experiments are designed and executed. They involve the following steps: (1) definition of research questions, (2) generation of test inputs for the benchmark programs, (3) execution of experimental runs on the GPU and in parallel on the CPU and (4) analysis of the experimental results. The design of the experiments is described in Section 5.3, while the results are presented and analysed in Section 5.4.

5.3 Experimental Setup

This section presents the experiments carried out to evaluate the performance of parallel GPU execution of embedded system tests. Testing time on the GPU is measured and compared to the time taken to execute the same tests in parallel by a 16-core CPU. For test execution on the GPU, the following measurements are taken:

- **Input transfer time:** time taken to transfer the test inputs from main memory to GPU memory.
- **Kernel execution time:** time taken by the GPU to execute the tests.
- **Output transfer time:** time taken to transfer test outputs from GPU memory back to main memory.
- **Total GPU time:** the overall time taken by the GPU; represents the sum of input transfer, kernel execution and output transfer time.

The experiments aim to answer the following research questions:

Q1. GPU vs CPU execution. *What is the GPU performance compared to a single-thread and multi-core CPU?* For each subject program and test suites, test executions are performed on the GPU and the CPU. On the GPU, test executions are performed *without* data transfer overlap. The effect of data transfer overlap on GPU execution time is assessed in Q3. For CPU execution, experiments are

performed sequentially on 1 core, and in parallel on 2, 4, 8 and 16 cores. Times taken by the GPU and CPU are compared in order to assess GPU performance.

Q2. Kernel execution vs data transfer time. *What is the overhead of data transfer when compared to kernel execution time?* Data transfer time consists of the times to transfer test inputs and outputs between main memory and GPU memory. They are measured separately and compared to kernel execution time for each subject.

Q3. Data transfer overlap: effect on performance. *Does data transfer overlap improve GPU performance?* For each subject program, test executions are performed on the GPU *with* data transfer overlap. Total GPU time is measured and used to calculate speedup relative to a single-thread and multi-core CPU. This speedup is compared that achieved without data transfer overlap.

Q4. Correctness. *Do test outputs remain the same when tests are executed on the GPU?* For each subject program and test suite, test outputs are collected from the CPU and from the GPU and compared to check if they are an exact match.

5.3.1 Test Generation

Due to time constraints, nine out of the 17 EEMBC programs which are successfully executed on the GPU (as shown in Table 5.2.2) are chosen for the performance evaluation. They are listed in Table 5.3. These are all five benchmarks in the telecom domain and four of the programs in the automotive domain, chosen at random.

The EEMBC benchmark suite does not contain tests for the programs. To simulate large test suites, 2^{17} (131,072) unique and random test inputs are generated for each program. The number of test inputs is chosen with regards to the GPU used for the experiments, which has approx. 2^{14} threads. Having 2^{17} test inputs produces more than enough threads to saturate the GPU and allows performance observation before and after the saturation point is reached. For each subject program, experiments are carried out with test suites containing between 2^{8} (256) to 2^{17} (131,072) tests. Table 5.3 shows the input size of for each program in bytes. The smallest input size just 512 bytes per test for `conven`, whilst the largest input is 4849 bytes for `puwmod`.

5.3.2 Hardware and Measurements

The GPU architecture used for the experiments is an NVidia Tesla K40m GPU with 15860 threads, spread across 15 compute units, which operates at 745 MHz and has

Subject	Domain	Description	Input Size [bytes]	Work-group size
a2time01	automotive	Angle to time conversion	2000	128
puwmod01	automotive	Pulse width modulation	4849	32
rspeed01	automotive	Road speed calculation	2000	512
tblook01	automotive	Table lookup & interpol.	1856	512
autcor00	telecom	Cross corr. of signals	1024	128
conven00	telecom	Convolution encoder	512	32
fbital00	telecom	Bit allocation	604	128
fft00	telecom	Fast Fourier Transform	1024	32
viterb00	telecom	Viterbi decoder	688	512

Table 5.3: EEMBC programs used in the performance evaluation.

12 GB global memory, 64 KB constant memory and 50 KB local memory. For the CPU executions, an Intel(R) Xeon(R) CPU E5-2640 v3 processor with 16 cores at 2.60 GHz and 16 GB RAM is used. All the programs are compiled using the gcc compiler with the highest optimisation level (`-O3`). To measure GPU kernel execution and data transfer time, the profiling functions from the OpenCL API are used. CPU execution time is measured through the standard C function *gettimeofday*. Each experimental execution is performed 100 times and median values are reported.

Multi-core CPU Execution To provide a fair comparison between the GPU and a multi-core CPU, test execution on the CPU is parallelised using OpenMP.

5.4 Results and Analysis

This section presents and analyses the results of the experiments outlined in Section 5.3. First, GPU speedup without data transfer overlap is assessed and analysed in Q1. Then, the overhead of data transfer is measured and presented in Q2. The effect on performance of using data transfer overlap is evaluated in Q3. Section 5.4.4 presents an analysis of the differences in results among benchmarks. Finally, correctness of the testing results is evaluated in Q4.

5.4.1 Q1. GPU vs CPU Execution

This section presents the speedup achieved when executing tests on the GPU compared to a single-thread and parallel multi-core execution on the CPU. Test execution on the GPU is without data transfer overlap. The GPU time presented in this section represents the worst-case time of the approach.

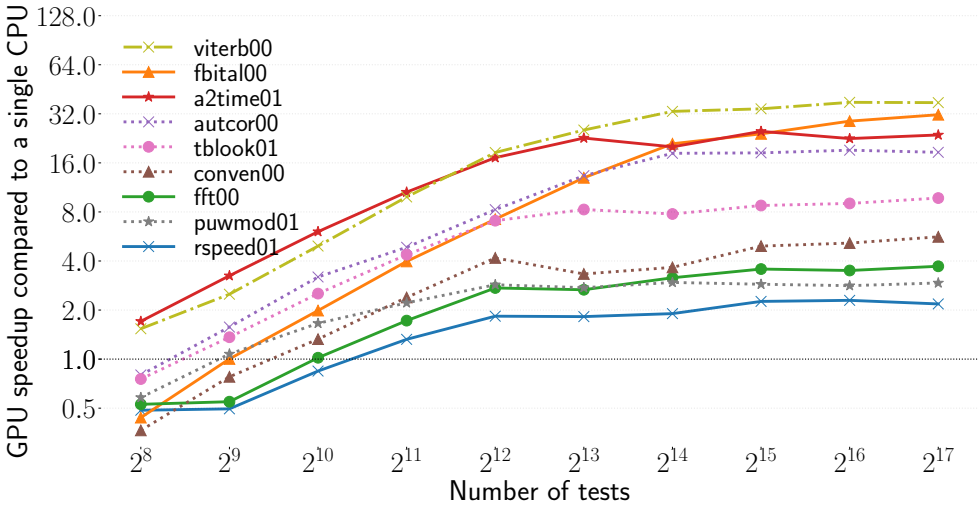


Figure 5.2: Speedup on the GPU vs single-thread execution for 10 different test suite sizes, without data transfer overlap.

GPU vs Single-thread CPU For each subject program and each of the associated test suite sizes (ranging from 2^8 to 2^{17}), two measurements are used to calculate GPU speedup: CPU time and Total GPU time. Figure 5.2 presents the GPU speedup for each subject program over the different test suite sizes. It shows that for all the programs, speedup increases linearly with increase in test suite size until a point, after which it begins to stabilise. The reason for the linear increase in speedup is because doubling the test suite size causes a proportional doubling in execution time on a single-thread CPU. However, on the GPU, execution time remains the same with increase in test suite size until a threshold size is reached. After reaching the threshold size, speedup generally remains stable when increasing the test suite size. This is because after the threshold, both CPU and GPU execution times increase at similar rates with test suite sizes, resulting in no change in speedup. For the subject programs, the threshold test suite size is 16,384 (2^{14}), which is the point at which all GPU threads are saturated. For two subject programs, `fbital` and `viterb`, increases in speedup continue to be observed up to test suite sizes of 131,072, but by approx. 20% after the threshold

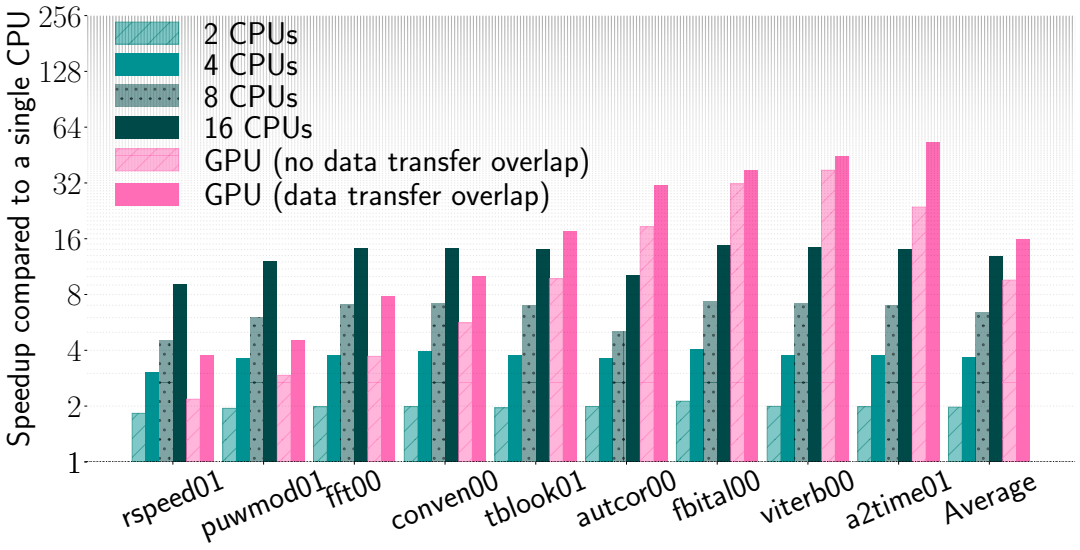


Figure 5.3: Speedup of GPU and multi-core CPU compared to a single-thread CPU, when executing the largest test suite for each benchmark.

size, rather than doubling. For these two programs, GPU execution time increases at a slightly lower rate than CPU execution time.

Figure 5.2 shows that speedup is higher for large test suite sizes. The average speedup for the maximum test suite size across all experimental programs is 9.6×. For the smallest test suite size of 256, the speedups are limited. Running on GPU starts to achieve speedup gains over the CPU when test suite sizes are larger than 2048 tests. This is not surprising, since for smaller test suite sizes, there is not enough workload to benefit from parallelisation on the GPU.

Figure 5.2 also shows that most subject programs achieve considerable speedup when large test suites are executed on the GPU. For `viterb`, `fbital`, `a2time`, `autcor` and `tblook`, GPU speedup ranges from 9.7× to 37× for the largest test suite. For `rspeed`, `puwmod` and `conven`, the speedup gain is more limited, ranging from 2.2× to 5.6×. Section 5.4.4 analyses the reasons for the varying speedup observed among different programs.

GPU vs Multi-core CPU To offer a fair comparison point, Figure 5.3 shows the speedup achieved by a multi-core CPU, using 2, 4, 8 and 16 cores, when executing the largest test suites for all subject programs. Multi-core CPU test execution is parallelised with OpenMP and speedup is calculated compared to a single-thread CPU. Figure 5.3 also shows the speedup achieved by the GPU without and with data transfer overlap.

Here, GPU speedup *without* data transfer overlap is discussed. GPU speedup *with* data transfer overlap is discussed in Section 5.4.3.

For all programs, multi-core CPU speedup scales linearly with the number of cores, achieving a maximum speedup of 13.6× (average speedup 11×) with 16 cores, across all subject programs. For 4 of the 9 programs, `autcor, fbital, viterb` and `a2time`, the GPU achieves considerably higher speedup ranging from 18.6× to 37.5× without data transfer overlap, demonstrating the benefit of using the GPU for these programs. For 5 benchmarks, `rspeed, puwmod, fft, conven` and `tblook`, the GPU is slower than the 16-core CPU without data transfer overlap. The reasons for differences across benchmarks are explored in Section 5.4.4. Average GPU speedup over all benchmarks, without data transfer overlap, is 9.6×, compared to 11× for a 16-core CPU, demonstrating the importance of optimising GPU test execution. The next sections analyse the impact of data transfer overhead and the effectiveness of using data transfer overlap to improve GPU performance.

5.4.2 Q2. Kernel Execution vs Data Transfer Time

To analyse the effect which data transfer overhead has on GPU test execution, Figure 5.4 shows the breakdown of total GPU time for all subject programs. In this section, kernel execution time and data transfer time are first analysed separately and then compared.

Kernel Execution Time. Figure 5.4 shows that for all programs, for small input sizes, kernel execution time remains the same on the GPU, due to the large number of available threads. This is expected, because as test suite sizes increase, the GPU is able solicits more threads for parallel execution of the additional test cases, up to the threshold size of approx. 16,384 (2^{14}), as identified in Q1. After reaching the threshold size, kernel execution time starts to increase linearly with test suite size.

Data Transfer Time. In contrast, data transfer is not parallelised and Figure 5.4 shows that data transfer time, including the transfer of both test inputs and outputs, increases linearly with the test suite size.

Data Transfer vs Kernel Execution Time. While the separate trends for data transfer and kernel execution times are the same across benchmarks, the proportion of total GPU time spent on each varies considerably across subject programs. For 4 benchmarks, `autcor, conven, fbital` and `viterb`, kernel execution time dominates and is up to 5× higher than data transfer time. This means that the cost of data transfer is minimal compared to the time it takes to perform test execution. For 4 other benchmarks,

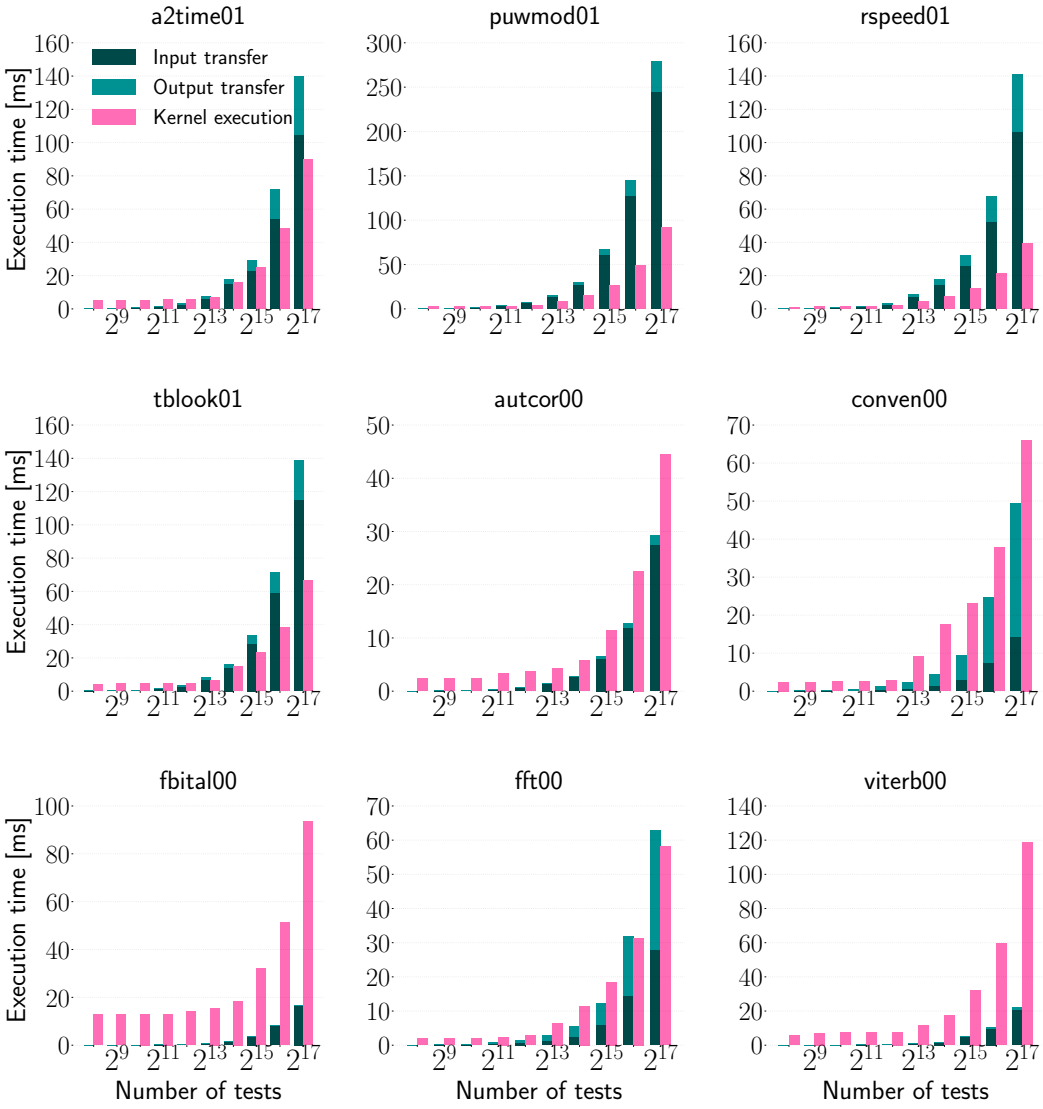


Figure 5.4: Breakdown of total GPU time into data transfer and kernel execution time for each subject program.

a2time, puwmod, rspeed and tblook, the data transfer time is larger than kernel execution time, representing a large overhead to using the GPU. This overhead could be mitigated by using an on-chip GPU or using data transfer overlap, as evaluated in the next section.

5.4.3 Q3. Data Transfer Overlap: Effect on Performance

To mitigate the effect of data transfer overhead on total GPU time, data transfer overlap is used (Section 4.5.2). Figure 5.3, shows the GPU speedup achieved with and without it for the largest test suite size for each subject program, when compared to a single-thread CPU. It shows that overlapping data transfer always improves GPU performance, bringing the largest GPU speedup from 38× (for viterb) to 53× (for a2time) and the average GPU speedup from 9.6× to 16×.

Across all programs, the GPU outperforms single-thread CPU execution. When compared to a parallel execution on a 16-core CPU, the GPU is faster for 5 out of the 9 benchmarks. Its maximum speedup compared to the 16-core CPU is 4×, while the average speedup across all benchmarks is 1.4×.

5.4.4 Analysis

This section analyses the diversity of GPU speedup observed across all benchmarks. It is based on the general observation that the more compute-intensive a program is, the higher the performance will be on the GPU. The metric used to establish the computational intensity of a benchmark is $Time\_cpu/Input\_size$, representing the time it took for the CPU to process a Byte of input data. A higher value of this metric means that more computations are executed per Byte of input data, suggesting that the benchmark is more computationally demanding.

Figure 5.5 shows the computational intensity of each benchmark. It shows that the benchmarks with the lowest value (rspeed, puwmod, fft), also have the lowest GPU speedup (Figure 5.3). Conversely, benchmarks with a high value, fbital, viterb, a2time are the ones exhibiting the largest GPU speedups.

While a2time does not have the highest computational intensity, it achieves large speedup due to the effect of data transfer overlap. Input data transfer time for a2time is comparable to kernel execution time, allowing for effective pipelining of the groups of tests, which leads to more than doubling its speedup, from 24× to 53×.

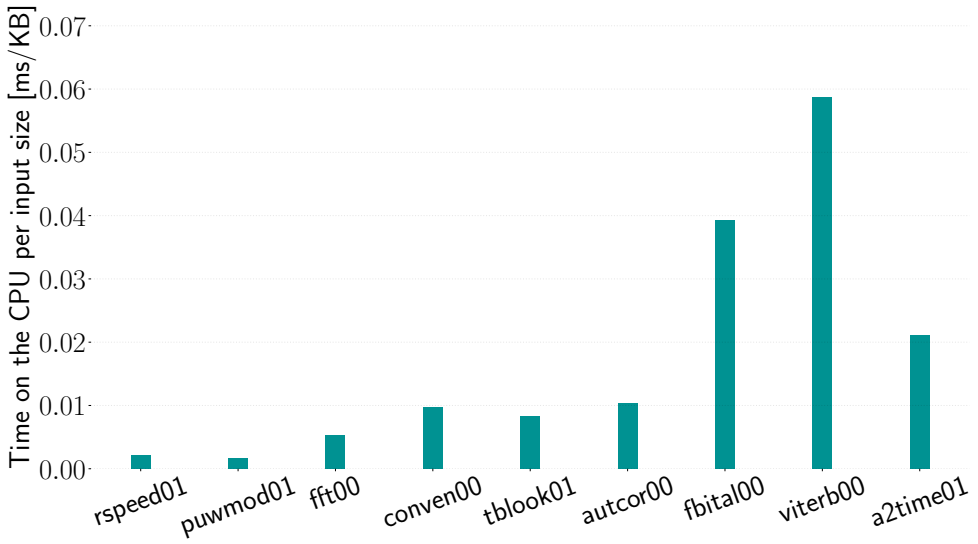


Figure 5.5: Computational intensity of benchmarks ordered by speedup achieved.

Another outlier is `conven`, which based on its computational intensity, would be expected to perform well on the GPU, but its speedup is comparable to `fft`, that has lower computational intensity. The reason for this lies in the fact that `conven` uses a temporary array of 1KB for each input test, that is accessed frequently in a loop. This array can easily fit in the large cache of the CPU, but cannot fit in the smaller cache of the GPU, providing additional benefit to CPU execution.

In summary, embedded benchmarks with high computational intensity tend to benefit more from using GPUs for accelerated test execution. Nevertheless, relative performance on the GPU is also affected by other factors, such as the ability to balance kernel execution and data transfer time when overlapping and the availability of large enough memory caches. Given the trend towards larger cache sizes on GPUs, it can be expected that embedded programs will be able to exhibit better performance on future generations of GPUs.

5.4.5 Q4. Correctness

To empirically demonstrate that using ParTeCL to execute embedded system tests on the GPU does not alter their functionality, test outputs from the CPU and GPU are collected for all benchmarks and all test suites. For all 9 subject programs, with 256 to 131,072 test inputs each, the test outputs between the CPU and GPU executions are an **exact match**. This demonstrates that using ParTeCL for test execution on the GPU *preserves correctness of program execution* for all 9 embedded system benchmarks.

5.5 Summary

This chapter evaluates the applicability and performance of accelerating test execution for embedded software on the GPU, using embedded systems benchmarks from the EEMBC benchmark suite.

To evaluate applicability, ParTeCL is used to compile and execute the benchmark programs on the GPU. This process reveals that the approach is partially applicable: 17 out of the 33 EEMBC benchmarks are successfully compiled and executed on the GPU, particularly the ones from the automotive and telecom domains. For the rest, limiting features are dynamic memory allocation, file IO, function pointers and some uses of string literals, which are not currently supported in OpenCL.

To evaluate performance, experiments are performed using 9 benchmarks from the automotive and telecom domains of EEMBC. The experiments reveal that the GPU outperforms sequential execution on the CPU by up to 53× (avg. 16× across benchmarks), and parallel execution on a 16-core CPU by up to 4× (avg. 1.4×).

Chapter 6

Testing Finite State Machine Models: Establishing Feasibility

6.1 Introduction

Model-based development is a widespread development approach in which software is implemented and verified based on a model of the required system. Finite state machines are a popular abstraction that is widely used to model a large variety of systems, including signal processing tools, communications protocols and control systems. Validating that an FSM model accurately represents the system requirements is a crucial task that involves the generation and execution of a large number of tests, which is often a costly and time-consuming process.

GPUs can be used to automatically and transparently accelerate test execution for FSM model validation, using the approach and tools presented in Chapter 4, but there are challenges specific to FSMs. This chapter, along with Chapter 7, addresses these challenges, demonstrating the feasibility using GPUs to accelerate FSM test execution. The specific challenges are:

1) Performance of the GPU kernel. FSM execution involves a large number of memory accesses, which are expensive operations on the GPU. Test sequences are read, input by input, corresponding transitions are searched in the FSM, and corresponding test outputs are written into GPU memory. This means that the memory layouts used for the FSM and tests (inputs and outputs) are crucial for GPU performance. In order to minimise kernel execution time and maximise performance, it is essential to design optimal memory layouts.

2) FSM test suite size. Test suites for large FSM models consist of long input sequences, comprising large amounts of data, impacting both performance and scalability. Performance is reduced, as transferring large test inputs and outputs between CPU and GPU memory adds considerable overhead to GPU time. Scalability is also limited, as GPUs have less memory than CPUs and may not be able to accommodate the entire test suites all at once.

The current chapter addresses the first challenge, which relates to the performance of the *GPU kernel only*. Two memory layouts for the FSM and three memory layouts for the tests are defined and implemented into separate OpenCL kernels. They are then used by ParTeCL Runtime to automatically launch parallel FSM test execution on the GPU threads. To determine the optimal memory layouts for kernel performance, they are evaluated and compared in terms of kernel execution time. Evaluation focuses on kernel execution time only, assuming that the FSM and tests are transferred to GPU memory prior to kernel execution. It demonstrates that, when using the optimal memory layouts, the GPU kernel is up to 12× faster (6.5× avg.), compared to a parallel implementation on a 16-core CPU. The second challenge is addressed in Chapter 7. Data transfer is optimised and *total GPU performance*, which includes both data transfer and kernel execution time, is evaluated and compared to a multi-core CPU.

The rest of this chapter is organised as follows. Section 6.2 presents the memory layouts for the FSMs and tests, together with further optimisations for kernel execution time and implementation details. The research questions, experimental setup and subjects FSMs used in the evaluation are presented in Section 6.3. Experimental results and analysis are discussed in Section 6.4. Finally, Section 6.5 concludes the chapter.

6.2 Approach

A test checking the behaviour of an FSM is represented as a sequence of inputs. Executing such a test involves applying the inputs in the sequence one by one, commencing at the specified starting state, transitioning through the states of the FSM, and recording the outputs associated with each transition. The test passes if the output sequence is the expected one and fails otherwise.

This section introduces the design and implementation of FSM test execution on the GPU. ParTeCL Runtime, which is introduced in Chapter 4, is extended to accept the FSM and test suite as inputs. They are read, stored in main memory and transferred to GPU memory. Two memory layouts for the FSM and three memory layouts for

the tests are designed and implemented. OpenCL kernels are implemented for each of the memory layouts, to execute the given FSM over an input sequence on the GPU. ParTeCL Runtime then uses the new OpenCL kernels to transparently launch the tests in parallel on the GPU threads.

6.2.1 Memory Layouts

This section presents two memory layouts for the FSM and three memory layouts for the test suite that are implemented in ParTeCL Runtime and evaluated in terms of their impact on GPU kernel performance.

6.2.1.1 FSM Layouts

The two FSM layouts are called *sparse* and *dense*.

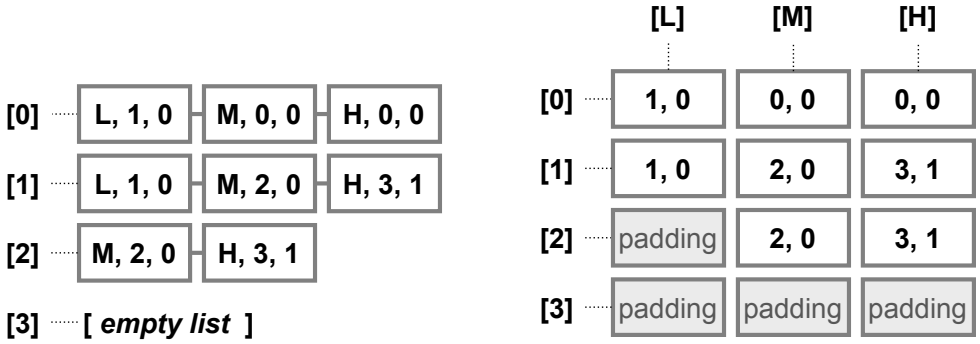
Sparse The sparse FSM layout consists of a one-dimensional array, indexed by state, in which each element is a list of (*input, next state, output*) triplets. Figure 6.1a illustrates the sparse FSM layout for the digital oscilloscope FSM, shown in Figure 2.6. Given a state s and an input a, the list of triplets corresponding to s is found in constant time and then a search is performed to find the correct triplet based on a. For example, to find the transition from state 1 on input M, first the list of triplets corresponding to state 1 is found at index 1 and then the list is searched to find the triplet containing input M.

The advantage of the sparse FSM layout is compactness. Since it uses dynamic lists, it requires memory only for the transitions present in the FSM and does not use any padding. This could be beneficial for sparse FSM matrices as it may allow them to fit into constant GPU memory. The disadvantage of the sparse FSM layout is the potentially slow execution time due to the need to perform a search to find the correct triplet in a list for each input in a test sequence.

Dense The dense FSM layout consists of a two-dimensional array, indexed by state and input, in which each element is a single (*next state, output*) tuple. Figure 6.1b illustrates the dense FSM layout for the digital oscilloscope FSM, shown in Figure 2.6. Given a state s and an input a, corresponding tuple is picked in constant time, based on the array indices.

The advantage of the dense FSM layout is the fast execution time, as for each input in a test sequence, the lookup for (next state, output) takes constant time. The

disadvantage is the need to allocate memory for every possible state/input pair in the FSM and use padding for pairs which are not present in the FSM. This could incur a large overhead in memory compared to the sparse layout, particularly for sparse FSMs. This would not be a drawback for FSMs with high density, as they require memory for approx. the same amount of transitions in both layouts, ($|S| \times |I|$).



(a) Sparse (b) Dense

Figure 6.1: The two FSM memory layouts, representing the digital oscilloscope FSM.

6.2.1.2 Test Layouts

There are three memory layouts for FSM tests, called *padded*, *padded-transposed* and *with-offsets*, illustrated in Figure 6.2. Note that during execution the same layout is used for both the test inputs and outputs.

Padded In the padded layout, tests are stored in a two-dimensional array, in which each row represents a separate test and each column represents a test input. All tests are padded with `null` bytes to the length of the longest test. Each GPU thread executes a single row of the two-dimensional array. Test execution stops when the padding `null` byte is encountered. This test layout is easy and intuitive to implement, but has high memory requirements and does not allow efficient coalesced memory accesses on the GPU. Coalesced memory accesses are introduced in Section 2.3.2.

Padded-transposed The padded-transposed layout uses the same two-dimensional array as the padded layout, but transposes it to allow coalesced memory accesses by the GPU threads. Each column represents a separate test and each row represents a test input, allowing the GPU to use coalescing to optimise global memory accesses.

With-offsets In the with-offsets layout, all tests are concatenated into a single array. An array of offsets is calculated and used by the threads to point them to the starting point of each test sequence. This layout is more compact than the other two, as it does not use padding. This means that during GPU execution, it may utilise the global memory cache more efficiently, providing performance improvement.

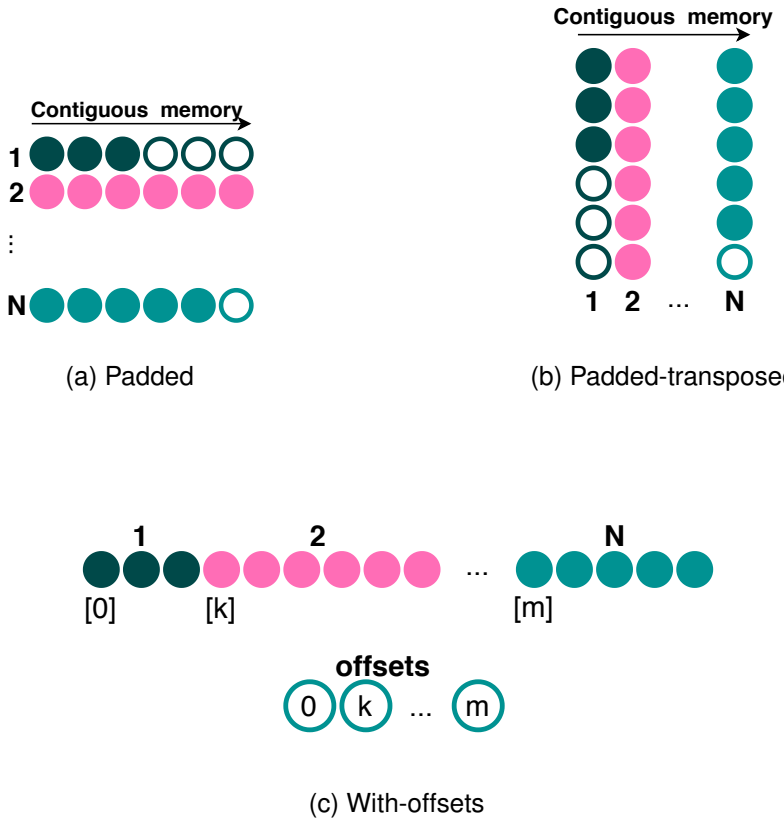


Figure 6.2: The three test memory layouts. Empty circles represent padding.

6.2.1.3 Placement in GPU Memory

Placement of the FSM and tests in the GPU memory hierarchy is also crucial for GPU performance. The GPU memory hierarchy is illustrated in Figure 2.3.

The FSM placement in memory is decided by ParTeCL Runtime during execution. The program calculates the size of the FSM and queries the GPU for the amount of space available in **constant memory** and **local memory**. When there is enough space available in constant memory, the FSM is placed into it. Constant memory is a fast read-only portion of global memory which is suitable for the FSM since test execution does not modify it. When the FSM is too large for constant memory, ParTeCL Runtime

checks if there is enough space in local memory and places the FSM there. This requires duplication of the FSM across work-groups, as each work-group has its own portion of local memory, but allows for faster memory accesses than global memory. Finally, if there is not enough local memory, the FSM is placed in global memory.

Test inputs and outputs are placed in **global memory**, where they are accessible to all threads. Each thread uses different portions of the data. Coalesced memory accesses and cache utilisation are crucial optimisations for improving GPU performance when data is stored in global memory.

6.2.2 Sorting the Test Sequences Based on Length

A straightforward and effective optimisation for GPU kernel execution time is sorting the FSM test sequences based on their length before kernel execution. This is due to the fact that, as outlined in Section 2.3, all GPU threads are organised into work-groups, which execute in lock-step with each thread executing the same instruction. This means that the whole work-group would only run as fast as the thread executing the longest test sequence. Thus, if the threads within the work-group are executing test sequences of different lengths, the threads with shorter test sequences will stay idle while waiting for the threads with longer test sequences to complete execution. However, if all threads in a work-group are running tests of similar lengths, they will all finish at the same time, freeing up resources for a new work-group to be scheduled. Sorting the tests based on their length and launching them on the same work-group would allow work-groups to execute tests of similar lengths. This would speedup the execution time not only for the *with-offsets* layout, but also for the *padded* and *padded-transposed* layouts, as each thread finishes execution when it encounters the first padding (`null`) character in the test sequence.

6.2.3 FSM Input Formats for ParTeCL

FSM definitions represent a higher level of abstraction than embedded C code. Input formats for the FSM and tests are required that are both user-friendly and intuitive to implement into ParTeCL.

The input format chosen for the FSM is *kiss2* [172]. It is a standard tabular FSM format, used primarily in the circuit design domain. It consists of a header, containing information about the FSM, and a list of transitions, stored in a text file. It is an intuitive format that is accessible to the end-user and easy to support in ParTeCL Runtime.

```
1  .i 1
2  .o 1
3  .s 4
4  .p 8
5  .r 0
6  H 0 0 0
7  M 0 0 0
8  L 0 1 0
9  H 1 3 1
10 M 1 2 0
11 L 1 1 0
12 H 2 3 1
13 M 2 2 0
```

Listing 6.1: *kiss2* input format for the digital oscilloscope FSM, shown in Figure 2.6

```
1  1 HH
2  2 HM
3  3 HL
4  4 MH
5  5 MM
6  6 ML
7  7 LH
8  8 LM
9  9 LL
10 10 LLH
11 11 LLM
12 12 LLL
13 13 LMH
14 14 LMM
15 15 LMMH
16 16 LMMM
```

Listing 6.2: Input format for the test suite for the digital oscilloscope FSM.

To illustrate it, Listing 6.1 shows the kiss2 file for the digital oscilloscope FSM, shown in Figure 2.6. Lines 1 to 5 contain the header, which holds the following information: (1) the inputs and outputs have lengths of 1 character (lines 1 and 2 respectively), the number of states in the FSM is 4 (line 3), the number of transitions is 8 (line 4) and the starting state is 0 (line 5). Lines 6 to 13 define the transitions with each line corresponding to one transition in the format: `input start_state end_state output`. For example, line 6 describes the transition which takes input H, starts in state 0, stays in state 0 and produces output 0. The output is 0 as the destination state, state 0, is a non-accepting state. Line 9 describes the transition which takes input H, starts in state 1, transitions to state 3 and produces output 1. The output is 1 as the destination state, state 3, is an accepting state.

The input format for the test suite is a text file in which each line corresponds to a separate test input. Each line contains a test index, followed by the input sequence for the FSM. Listing 6.2 illustrates an input file, containing a test suite for the digital oscilloscope FSM in Figure 2.6. This test suite is generated based on the all-transition pair coverage criterion. The test generation algorithm is discussed in more detail in Section 7.2.

6.3 Experimental Setup

This section presents the experiments carried out to evaluate the effects which the memory layouts and optimisations outlined in Section 6.2 have on the performance of the GPU kernel, when executing FSM tests in parallel. The evaluation focuses on kernel execution time only and does *not* take into account data transfer time. The effects of data transfer overhead on total GPU performance and optimisations for it are addressed in Chapter 7.

The evaluation uses 12 FSMs from the network intrusion detection domain and one industry FSM model from the signal processing domain. Full test suites for each FSM are generated, based on all-transition pair coverage. The subject FSMs, along with test generation, are presented in more detail in Section 6.3.1. For each FSM, GPU kernel execution time is measured and compared to the time taken to execute the same tests in parallel on a 16-core CPU.

The experiments aim to answer the following research questions:

Q1. GPU kernel vs multi-core CPU execution. *What is the GPU kernel performance compared to a 16-core CPU?* For each FSM, tests are executed in parallel on the GPU and on a 16-core CPU, and GPU speedup is calculated based on kernel execution time. The experiment is performed using test suite sizes ranging from 2048 to the full test suite in order to assess how speedup changes as the test suite grows. The results presented for this question are the ones achieved by the fastest GPU implementation overall, as determined in the experiments for Q2, Q3 and Q4 (*dense* FSM layout, *padded-transposed* test layout and *sorting* of the tests before execution).

Q2. Effect of FSM layout. *Is the GPU kernel performance dependent on the FSM layout in memory (Sparse vs Dense)?* The time taken by the GPU kernel and a 16-core CPU to execute the full test suites of all FSMs, using both layouts, is

measured and the GPU speedups are compared. The presented results use the *padded-transposed* test layout. Tests are *not* sorted before execution.

Q3. Effect of test layout. *Is the GPU performance dependent on the test layout in memory (Padded vs Padded-transposed vs With-offsets)?* The time taken by the GPU and 16-core CPU to execute the full test suites of all FSMs, using the three test layouts, is measured and the GPU speedups are compared. The *dense* FSM layout is used and tests are *not* sorted before execution.

Q4. Effect of test sorting. *How does the GPU performance change when the test sequences are sorted based on length?* The experiments for Q3 are repeated, but with sorted test inputs before its execution. GPU speedup for all three test layouts are compared in order to assess the effect of test sorting on performance.

6.3.1 Subject FSMs and Tests

This evaluation uses 12 subject FSMs selected at random from the l7-filter pattern set [173], which contains regular expressions used for network intrusion detection. It also uses one industry provided FSM from Keysight Technologies, used to perform transition localisation in the signal processing domain [174]. For all FSMs, full test suites are generated based on the all-transition pair coverage criteria. Table 6.1 provides a summary of the FSMs, together with their sizes and numbers of tests.

l7-filter At the time of experimentation, the l7-filter set contains 113 regular expression patterns and a sample of 12 subjects is selected at random for experimentation. FSMs are generated from the regular expressions, using the Flex tool [175]. A custom Python script is used to generate a kiss2 file for each FSM [172]. The input set for the l7-filter FSMs comprises the 256 ASCII characters.

Keysight The Keysight FSM represents a model for a transition localisation tool in communication signals [174]. The FSM takes three inputs, L, M and H, which correspond to Low, Medium and High voltage pulses. It accepts when a low or high state has been established, identifying a transition in the signal. The size of the FSM is determined by a parameter p. It defines the number of pulses necessary to establish that a signal has been sustained long enough to be considered a valid transition and not simply a glitch. In the experiments, $p = 1000$ is used as it generates an FSM of similar size to those from l7-filter, in terms of the number of transitions.

FSM	Domain	#States	#Inputs	Density	#Tests
ssl	l7-filter	34	256	83%	1,475,251
battlefield2	l7-filter	71	256	56%	1,476,796
dns	l7-filter	197	256	83%	8,533,671
aim	l7-filter	41	256	58%	1,344,963
rtp	l7-filter	28	256	95%	1,536,723
tsp	l7-filter	27	256	84%	1,162,511
yahoo	l7-filter	54	256	82%	2,627,405
ntp	l7-filter	31	256	90%	1,374,296
hotline	l7-filter	34	256	66%	1,216,433
h323	l7-filter	46	256	90%	2,241,832
halflife2	l7-filter	24	256	80%	1,088,409
counterstrike	l7-filter	30	256	85%	1,472,463
keysight	signal processing	4004	3	100%	36,027

Table 6.1: Subject FSMs used in the evaluation. 12 FSMs are from the l7-filter network intrusion detection protocols and one is a digital signal processing model provided by Keysight Technologies.

Test Generation In order to perform the evaluation using realistic tests, full test suites for each of the subject FSMs is generated, using the all-transition pair coverage criterion. FSM coverage criteria are introduced in Section 3.5. All-transition pair is chosen, because it has been shown to be a rigorous coverage criterion [138], as it ensures that events in the system are tested not only individually, but also in relation to one another. The specific algorithm used to generate the test suites is discussed in more detail in Chapter 7, which proposes an approach to optimising the test suites.

Table 6.1 shows the size of the resulting test suites for each FSM and Figure 6.3 shows the average test lengths for each test suite. Table 6.1 shows that the keysight FSM has fewer tests than the other FSMs. It has $36,072$ tests, while the average number of tests across the l7-filter FSMs is $2,129,229$. However, the average length of its tests is larger than that of the l7-filter FSMs. For keysight the average test length is $1,000$ inputs, while across the l7-filter FSMs the average test length is 11 inputs. There are two reasons for these differences: (1) the l7-filter FSMs have fewer states than keysight (avg. 53 vs $4,004$) and (2) they have a larger input set than keysight (256 vs 3). Therefore, for the l7-filter FSMs, a large volume of transition pairs originate in a small number of states, requiring a large number of shorter test sequences to traverse

them all. `keysight`, on the other hand, has the opposite - a small number of transition pairs go through a large number of states, requiring fewer but longer test sequences to cover them.

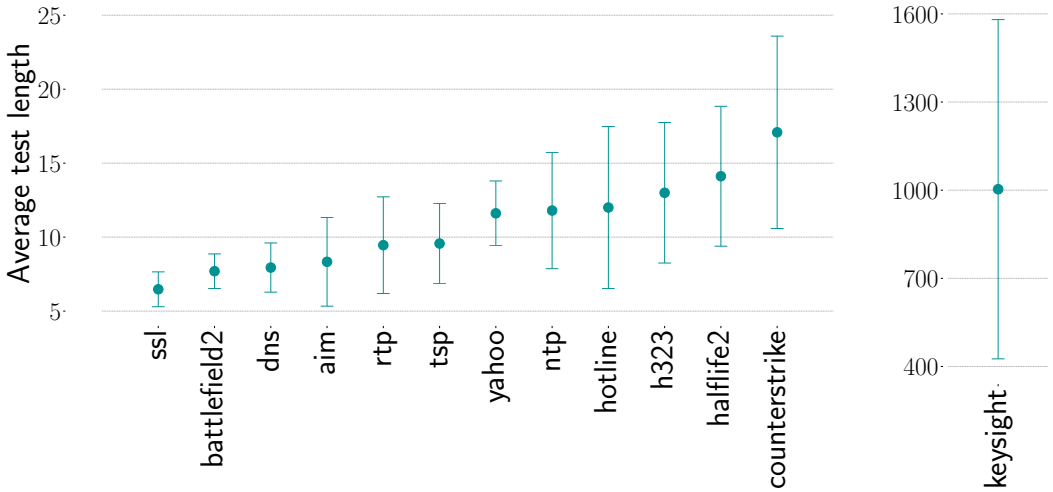


Figure 6.3: Average test lengths for the subject FSMs. Error bars show standard deviation within the test suite.

6.3.2 Hardware and Measurements

The same GPU and CPU used in Chapter 5 are used for these experiments. They are described in Section 5.3.2. Each experimental execution is performed 100 times and median values are reported.

Multi-core CPU Execution To provide a fair comparison between the GPU and a multi-core CPU, test execution on the CPU is parallelised using OpenMP. All design configurations (FSM layout, test layout and test sorting) are executed on the 16-cores CPU. The GPU speedups reported across Section 6.4 are calculated when compared to the fastest CPU execution times.

Correctness For each experiment, the testing outputs produced by the GPU are compared to those from the CPU in order to confirm that they are an **exact match** ensure that using the GPU preserves the correctness of test execution.

6.4 Results and Analysis

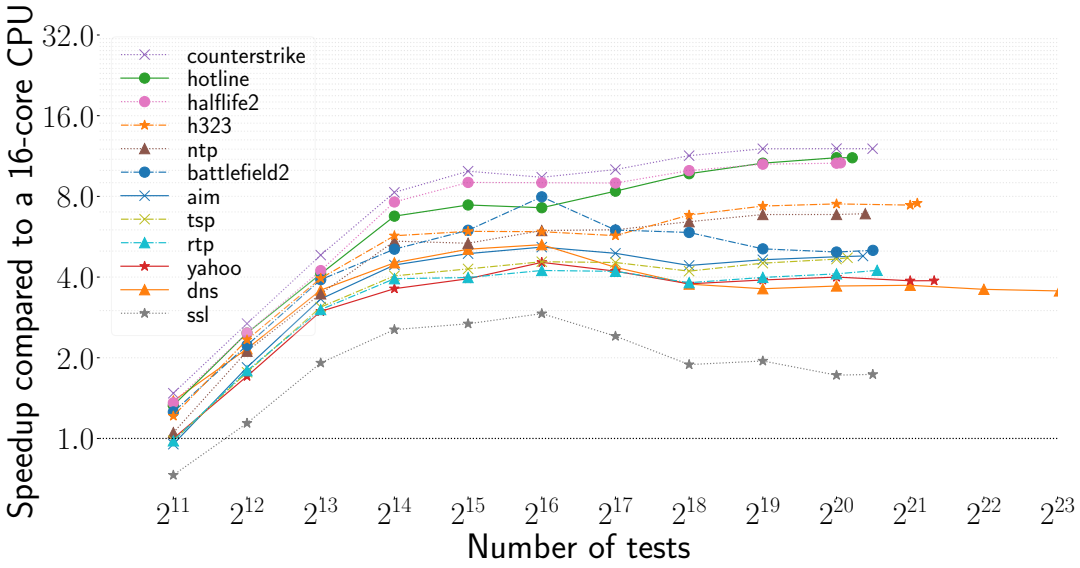
This section presents the results and analysis of the experiments described in Section 6.3. First, the overall *GPU kernel execution speedup* when compared to a 16-core CPU is presented in Q1 and then, each individual design choice is assessed in Q2, Q3 and Q4. Finally, Section 6.4.5 briefly discusses the data transfer overhead for the subject FSMs.

6.4.1 Q1. GPU Kernel vs Multi-Core CPU Execution

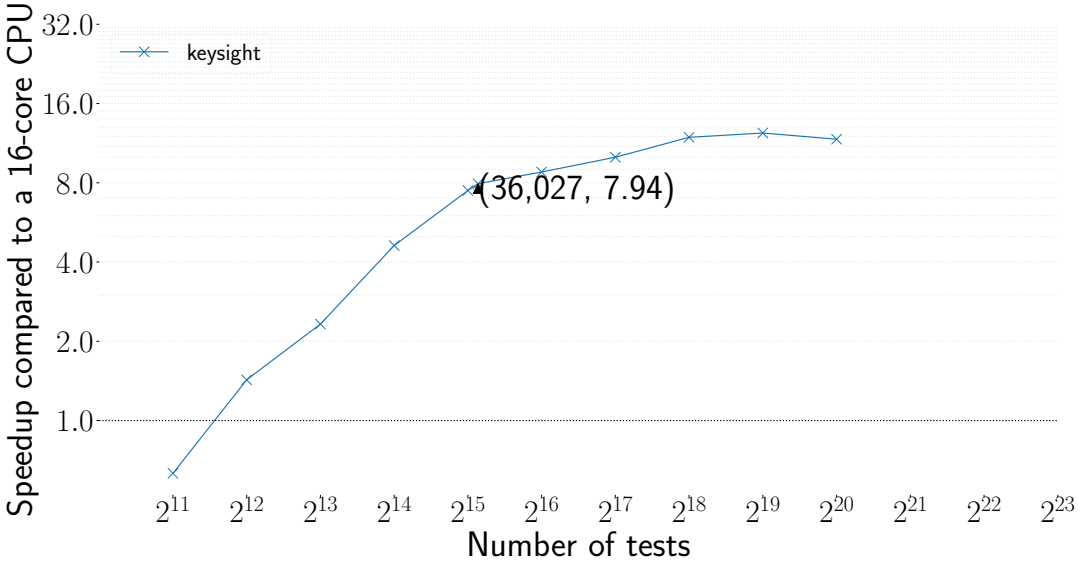
Figure 6.4 shows the speedup achieved in kernel execution time on the GPU when compared to an optimised parallel implementation on a 16-core CPU. For each FSM, speedup is presented for test suite sizes ranging from 2048 up to the maximum number of tests in the test suite. As `keysight` requires fewer tests that the l7-filter FSMs, its test suite is padded to contain 2^{20} tests, by randomly duplicating its existing tests.

GPU kernel speedup increases with the number of tests in the test suite, since the GPU is able to utilise more threads as tests are added. This continues up until the GPU's saturation point, at approx. 2^{15} tests, after which there are no more GPU threads to be utilised and the speedup remains stable. Speedup is observed for test suite sizes larger than 2^{12}, over all FSMs, and the highest speedup is achieved for each FSM is for the largest test suite size. Results for the l7-filter and Keysight FSMs are further discussed separately.

l7-filter The speedup observed for the l7-filter FSMs ranges between $1.7\times$ for `ssl` and $12\times$ for `counterstrike`. The average speedup across all FSMs is $6.4\times$. The difference in speedup across the FSMs is due to the difference in the lengths of tests. Figure 6.3 shows the average test lengths of each FSM, as well as the standard deviation across the test suite. The FSMs which achieve the highest speedup, `counterstrike`, `hotline` and `halflife2`, are also among the ones which have the longest average test lengths. Conversely the FSMs with lowest speedup, `ssl` and `dns` are among the ones with shortest average test lengths. The longer test sequences require longer execution per test both on the GPU and CPU. Since the GPU has a much higher degree of parallelism, the extra computation per thread is lower than that of each individual CPU core, which needs to execute multiple tests. This allows the GPU to execute longer test sequences much faster than the CPU, resulting in higher speedup.



(a) l7-filter



(b) Keysight

Figure 6.4: Speedup of GPU kernel execution when compared to a 16-core CPU over different test suite sizes. The presented speedups are for the fastest GPU and multi-threaded CPU implementations.

Keysight For `keysight`, the GPU achieves speedup of 7.9× for its full test suite (36,027 tests) when compared to a 16-core CPU. This speedup seems lower than expected, when compared to the l7-filter FSMs, considering the longer test sequences of `keysight`. There are two reasons for this:

1. `keysight` has only 36,072 tests in its test suite - not enough to completely utilise the GPU. Figure 6.4 shows that padding the test suite to 2^{20} tests enables the GPU to achieve higher speedup of up to 12.4×.

2. The longer testing sequences can successfully utilise the high degree of parallelism available on the GPU, resulting in high speedups. Nevertheless, when executing a test, each step of the FSM traversal involves expensive test input/output reads and writes from/to global memory. Each of these data accesses is more time consuming on the GPU than the CPU. As `keysight`'s testing sequences are 2 orders of magnitude longer than those of the l7-filter FSMs, the cumulative effect of global memory accesses has a negative impact on GPU speedup.

6.4.2 Q2. Effect of FSM Layout

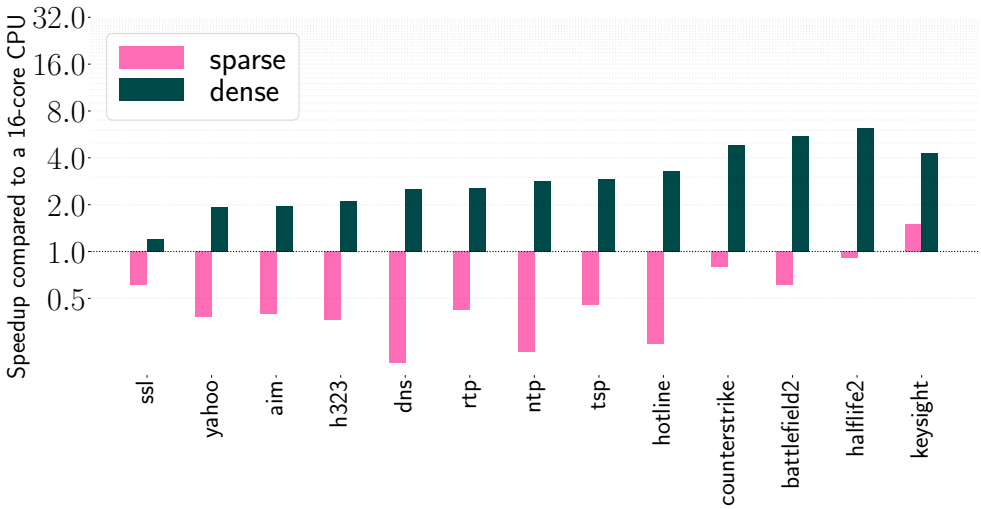


Figure 6.5: GPU kernel speedup, when compared to a 16-core CPU, for the *sparse* and *dense* FSM layouts. The presented values are for the full test suite for each FSM. The test layout is *padded-transposed* and tests are *not* sorted before execution.

Figure 6.5 shows a comparison of the GPU speedup achieved when using the two FSM layouts, sparse and dense, for the full test suites of each FSM. With sparse FSM layout, GPU speedup for the l7-filter FSMs is consistently lower than 1, meaning that

the GPU is slower than the 16-core CPU. For `keysight`, it is 1.5×. With dense FSM layout, GPU speedup is considerably higher for all FSMs. For the l7-filter FSMs it ranges between 1.2× and 6.2× and for `keysight` it is 4.3×. For all subject FSMs, the dense FSM layout outperforms the sparse FSM layout. Furthermore, only with the dense FSM layout is the GPU faster than a 16-core CPU.

There are two reasons for the differences between GPU performance with the two FSM layouts, related to their characteristics outlined in Section 6.2.1.1. First, the GPU is able to perform faster lookups for the next state and output for a given input with the dense FSM layout. This effect is multiplied over long test sequences, since lookups are necessary for each input in the test sequence. Second, the density for all subject FSMs is high. Table 6.1 shows that across all FSMs it ranges between 56% and 100% with an average of 81%. This means the amount of memory taken by the sparse layout would be close to that of the dense layout, removing the benefit of using the sparse layout. In addition, all subject FSMs fit into the constant memory available on the GPU with both layouts, allowing the dense layout to take advantage of fast access to constant memory.

6.4.3 Q3. Effect of Test Layout

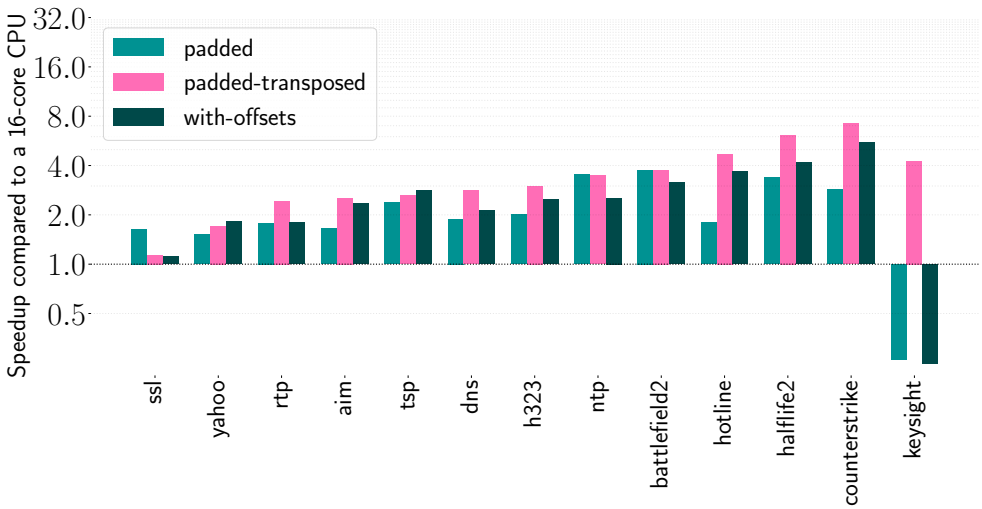


Figure 6.6: GPU kernel speedup, when compared to a 16-core CPU, for the *padded*, *padded-transposed* and *with-offsets* test layouts. The results are for the full test suite for each FSM. The FSM layout is *dense* and tests are *not* sorted before execution.

Figure 6.6 shows the GPU kernel speedup achieved when using the three test layouts, padded, padded-transposed and with-offsets, for the full test suites of each subject FSM. This section analyses the results for the l7-filter FSMs and `keysight` separately.

I7-filter Figure 6.6 shows a variation in performance across the different FSMs and test layouts. In some cases, for FSMs with shorter test inputs (`yahoo` and `tsp`), the highest GPU speedup is achieved with the with-offsets test layout (up to 2.8× for `tsp`). In contrast, for FSMs with longer test inputs (`h323, hotline, halflife2` and `counterstrike`), padded-transposed achieves the highest speedup (up to 7.2× for `counterstrike`).

There are two factors which contribute to the efficiency of the test layout. These are the ability of the GPU to (1) use the global memory cache and (2) to perform coalesced memory accesses. They explain why with-offsets can be the fastest test layout for FSMs with short tests, while padded-transposed is the fastest test layout for FSMs with long tests. With-offsets provides a more compact test representation, which fits easily into the global memory cache for FSMs with short test sequences. As test sequences grow, they no longer fit in the GPU cache and the ability to perform coalesced memory access becomes more beneficial. This is provided by the padded-transposed test layout and explains the higher speedup achieved with it for the FSMs with long test suites.

Keysight Figure 6.6 shows that for `keysight`, the GPU kernel speedup with the padded and with-offsets test layouts is less than 1 - the GPU is slower than the 16-core CPU. In contrast, with the padded-transposed layout, GPU speedup improves considerably, reaching a value of 4.2×. This large difference is due to the long test sequences in `keysight`'s test suite. Figure 6.3 shows that the average length of its test inputs is approx. 1000. Each input is a value encoded as a character. This means that the average test takes approx. 1MB of memory, making it impossible to fit into the GPU's global memory cache.

With the padded and with-offsets layouts, at every input traversal step, every GPU thread is performing two expensive memory accesses (reading test input and writing test output) without the help of the GPU cache, resulting in an inefficient GPU computation. This effect is multiplied by a factor of 1000 with the long test sequences, resulting in poor GPU performance for these layouts. Performance is dramatically improved with the padded-transposed test layout, which allows memory coalescing. With it, for every input, separate GPU threads perform accesses to consecutive addresses in memory. This allows the GPU architecture to combine memory accesses into a single efficient memory transaction across a work-group, resulting in improved performance.

6.4.4 Q4. Effect of Test Sorting

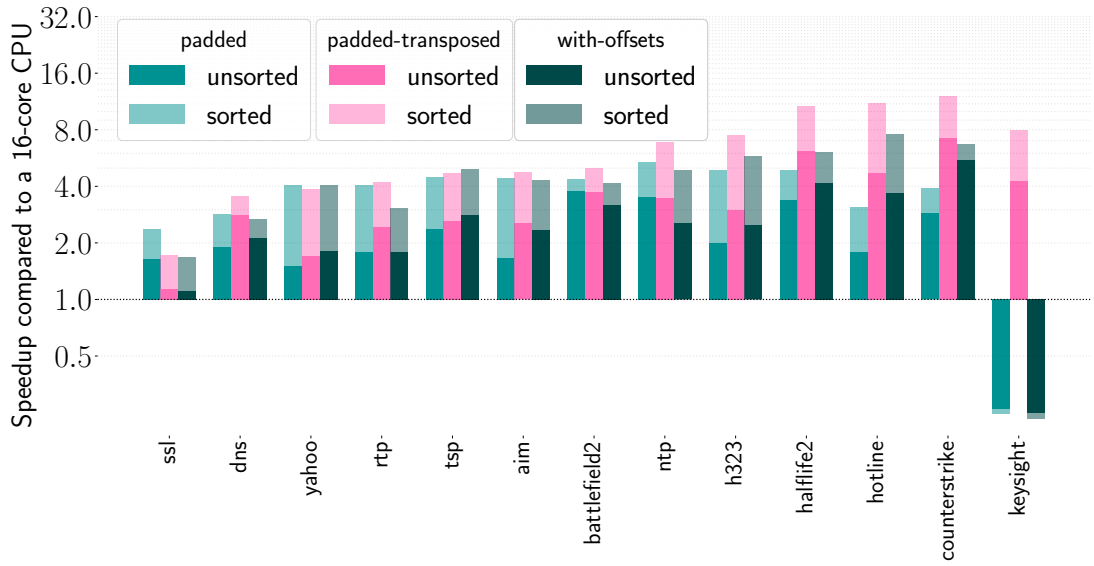


Figure 6.7: GPU kernel speedup, when compared to a 16-core CPU, *sorted* and *unsorted* test suites, for each test layout. Tests are sorted based on the length of the testing sequences. The results are for the full test suite for each FSM. The FSM layout is *dense*.

Figure 6.7 shows the effect of sorting the tests based on length on GPU kernel speedup. Sorting the tests improves speedup across all FSMs and test layouts. For FSMs with long test inputs (`counterstrike`, `hotline`, `h323`, `ntp` and `keysight`) the improvement is by a factor of approx. 2. This brings the maximum GPU speedup to 12× (for `counterstrike`), with an average across all FSMs of 6.5×.

These results are expected, based on the discussion in Section 6.2.2. Sorting the tests prior to execution ensures that all threads within a work-group have tests of similar lengths and finish executing them at the same time, immediately freeing resources for another work-group to be scheduled. This leads to less time spent by threads being idle and improves performance.

6.4.5 Assessing Data Transfer Overhead

The results presented so far are for GPU kernel execution only. This Section discusses the overhead of data transfer time. This is the time taken to transfer the test inputs and outputs between main memory and GPU memory. It is a well known limitation of GPU architecture that data transfer time is slow due to the high latency of the interface and can have significant negative impact on GPU performance.

FSM	Overhead without overlap	Overhead with overlap
ssl	64%	9%
battlefield2	86%	30%
dns	75%	16%
aim	80%	22%
rtp	80%	13%
tsp	81%	20%
yahoo	78%	16%
ntp	87%	34%
hotline	91%	47%
h323	85%	28%
halflife2	90%	50%
counterstrike	90%	54%
keysight	92%	56%

Table 6.2: Data transfer overhead as % of total GPU time for l7-filter FSMs.

Table 6.2 shows the overhead of data transfer as a fraction of the total GPU time (data transfer and kernel execution time combined) for the fastest GPU kernel implementation. It shows that for all FSMs, data transfer incurs a large overhead, ranging from 64% (for `ssl`) to 92% (for `keysight`). Unsurprisingly, `ssl` and `keysight`, the FSMs with the shortest and longest tests, respectively, have the lowest and highest overhead, respectively.

This high degree of data transfer overhead is explained by the fact that the approach presented in this chapter optimises the kernel execution time. This results in it becoming only a small proportion of total GPU time and data transfer becoming the dominant factor. Therefore, optimising data transfer is an important next step to improving total GPU speedup.

Optimisation. It is possible to mitigate the impact of large data transfer overhead by using data transfer overlap, as described in Section 4.5.2. Table 6.2 shows the reduction in overhead achievable by overlapping data transfer with kernel execution. As most of the total time on the GPU is spent in data transfer, this optimisation would lead to better overall performance on the GPU for all FSMs.

6.5 Summary

This chapter establishes the feasibility of accelerating test execution for finite state machines on the GPU, using the approach and tools presented in Chapter 4. To allow FSM test execution, ParTeCL Runtime is extended to accept user-friendly formats for the input FSM and its tests. In addition, two memory layouts for the FSM and three memory layouts for the test suites are defined, implemented into separate OpenCL kernels and evaluated based on their impact on the performance of *GPU kernel execution*. The performance evaluation uses 13 subject FSMs from the network intrusion detection and signal processing domains. It reveals that the GPU kernel is up to 12× faster (average of 6.5× across all FSMs) when compared to a 16-core CPU and that it is fastest for FSMs with long test sequences. In addition, the experiments reveal that using the *dense* FSM layout results in better GPU performance across all FSMs. With regards to test layout, the results show that *with-offsets* tends to perform better on the GPU for FSMs with short test sequences, while *padded-transposed* is more suitable for FSMs with long sequences. Finally, the evaluation demonstrates that sorting the tests based on their length before execution on the GPU considerably improves its performance.

This chapter focused on the performance of the GPU kernel only. The next chapter addresses the challenges that large FSM test suites pose to *total GPU performance* and to the scalability of the approach.

Chapter 7

Testing Finite State Machine Models: Evaluating Performance and Scale

7.1 Introduction

Chapter 6 demonstrates that GPUs can be used to automatically execute tests for FSM model validation. It evaluates and establishes the optimal memory layouts for the FSM and tests which achieve the fastest kernel execution times, but leaves unaddressed challenges, related to the large sizes of FSM test suites. These challenges are:

Data transfer overhead. Chapter 6 optimises kernel performance, after the FSM and tests are transferred to GPU memory, but does not address time spent in transferring test inputs and outputs between main memory and GPU memory. It is well known that data transfer time adds a large overhead to total GPU time, which may severely limit the achieved performance. This is a significant challenge for FSM validation, as complex FSMs have large test suites with long test sequences.

Scalability with FSM size. The evaluation in Chapter 6 uses 13 FSMs, all of which fit into GPU memory together with their test suites, but without optimisations this approach will not scale to larger FSMs whose test suites do not fit in GPU memory.

Test input generation. The FSM test inputs used in Chapters 6 and 7 are generated based on the *all-transition pair* coverage criterion for state machines. The algorithm used for test generation ensures that the test suites meet the coverage criterion, but does not attempt to optimise the size of the test suite. As a result, test suites contain distinct tests that are redundant and do not contribute to coverage. This means that they cover only transition pairs that are already covered by other tests and could be removed without impacting coverage. Redundant tests add to the size of the test suite

and exacerbate the remaining challenges. Removing them could reduce transfer time of the test suite and can be used in addition to data transfer optimisations on the GPU.

This chapter expands the work in Chapter 6 by addressing the above challenges. Data transfer overlap, presented in Chapter 4, is applied to FSM test execution to minimise data transfer overhead and improve scalability. Test suite reduction, which removes redundant tests from the test suite, is applied and its effectiveness is analysed in order to determine if the large test suits are necessary for coverage. Thorough empirical evaluation, using 15 large FSMs from the Snort [176] network intrusion detection protocol, is performed. Total GPU performance, comprising both data transfer and kernel execution time, is measured, analysed and compared to that of a 16-core CPU.

The evaluation results show that with the optimisations presented in this chapter, the GPU is up to 9.3× faster (average 4.5×) than a 16-core CPU in test execution for FSM model validation. This is *total GPU speedup*, including both data transfer and kernel execution times, which is achieved completely automatically with the use of ParTeCL Runtime. Test suite reduction results show that for large FSMs with high density, test suites generated based on all-transition pair coverage have a negligible number of redundant tests, only up to 0.4% across all subject FSMs. This demonstrates the need for large test suites for large FSMs and confirms the importance of finding other effective techniques to accelerate execution.

The rest of this chapter is organised as follows. Section 7.2 presents the optimisation approaches: (1) for splitting FSM test inputs into groups to use data transfer overlap (2) for removing redundant test inputs from the test suites. Section 7.3 describes the experimental setup and subject FSMs used for evaluation, while the results and analysis are presented in Section 7.4. Finally, Section 7.5 summarises this chapter's findings.

7.2 Approach

Two techniques are used improve the performance and scalability of using GPUs to accelerate FSM testing. The first is using data transfer overlap to pipeline test transfers and kernel execution, presented in Section 4.5.2. Section 7.2.1 describes the application of this strategy to FSM input sequences. The second technique is applying test suite reduction - finding and removing redundant input sequences from the test suites. To present test suite reduction, Section 7.2.2 first describes the algorithm used for test input generation based on all-transition pair coverage and Section 7.2.3 then presents the test suite reduction method.

7.2.1 Data Transfer Overlap: Dynamic Splitting of Tests Into Groups

ParTeCL Runtime is used to perform dynamic splitting of the test suites into test groups of equal sizes for data transfer overlap. The size of the groups is an input parameter provided by the user. Figure 7.1 illustrates FSM tests before and after they are split into groups. Tests are read into main memory and stored in the padded layout (Section 6.2.1.2). Tests are sorted based on their length, following the performance results in Section 6.4.4. Each test is padded to the length of the longest test. ParTeCL Runtime goes over the tests one by one and adds them to the current group, until the size of the group becomes equal to the required group size. At this point, a new group is started and this process is continued until all tests are assigned into a group.

As a result, some groups have a larger number of shorter tests and other groups have a smaller number of longer tests, but the size of each group in Bytes is roughly equal, maintaining the data transfer time between groups. In addition, memory is saved in padding, as tests within each group need to be padded only to the length of the longest test in the respective group and not the longest test in the whole test suite. This is illustrated in Figure 7.1b.

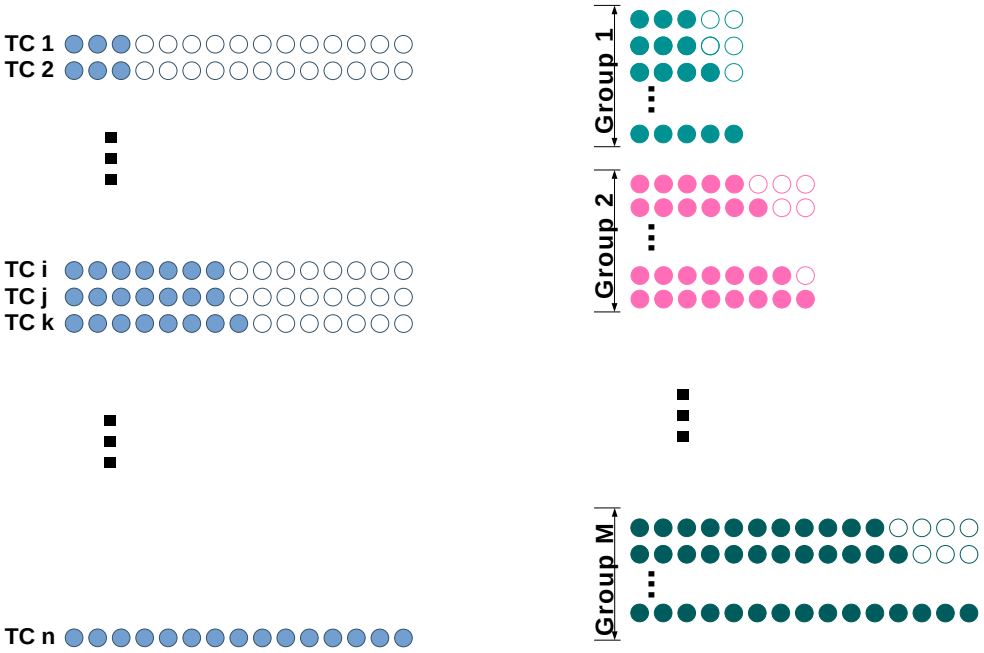
Each group is then transposed before transferring to GPU memory to utilise the padded-transposed test layout, following performance results in Section 6.4.3.

Choosing the Size of the Test Groups With respect to performance, data transfer overlap is most effective when data transfer and kernel execution time per test group are balanced. Therefore, it is important to choose the size of the test group in a way which achieves optimal performance. In order to allow benchmarking to choose the optimal group size for each FSM, ParTeCL Runtime takes the group size as an input argument, in KBytes. It then divides the tests into groups based on the supplied group size.

In the evaluation experiments in Section 7.3, GPU test execution for each FSM and test suite size is performed over a range of test group sizes. The range starts from 64 KB and continuously doubles the size of the test groups up to 524 MB[1]. For each FSM and test suite size, the lowest total GPU time[2] is presented. Thus, the results shown in Section 7.4 use the optimal test group size for each FSM and test suite size.

[1] In other words, the test group sizes are 64 KB, 128 KB, 256 KB and so on up to 524 MB.

[2] Total GPU time is the overall time taken by the GPU, including data transfer and kernel execution time, as defined in Section 7.3



(a) Tests before they are split into groups. All tests are sorted based on length and padded to the length of the longest test in the suite.

(b) Tests after they are split into groups. Tests are still sorted based on length, but padded only to the length of the longest test within their group.

Figure 7.1: Splitting of FSM tests into groups for data transfer overlap. Empty circles represent padding.

7.2.2 Generating FSM Test Inputs

The FSM test input sequences used in this study are generated based on the all-transition pair coverage criterion to serve as a representation of a realistic test suite. All-transition pair is chosen for this work, because it has been shown to be a rigorous coverage criterion with strong fault finding capabilities [138].

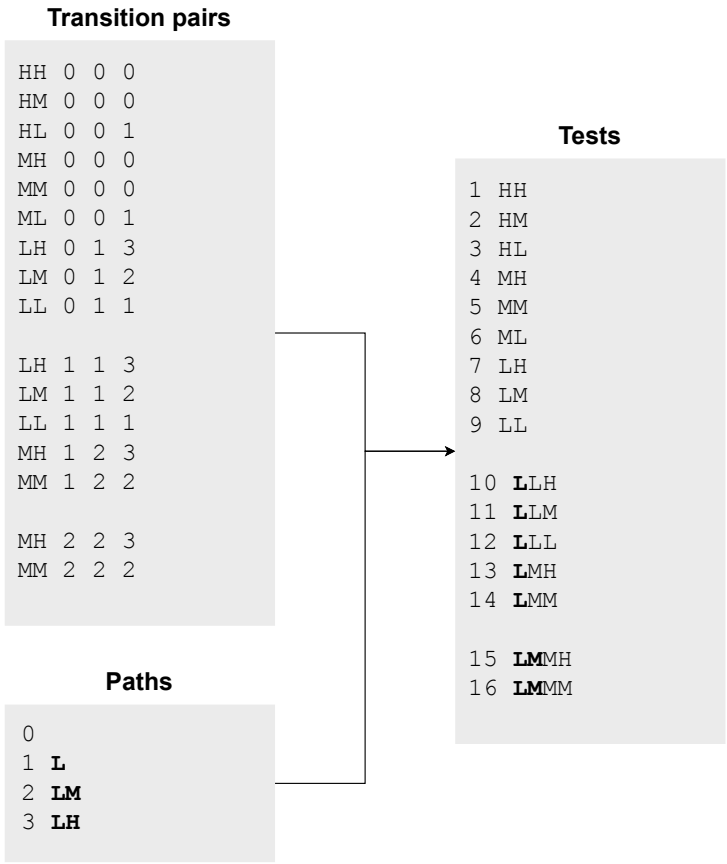


Figure 7.2: Test input generation for the digital oscilloscope FSM shown in Figure 2.6. The transition pairs are characterised by their transition inputs, start state, middle state and end state. The FSM has 16 transition pairs and 16 tests, each of which covers a separate transition pair.

All-transition pair coverage requires that for each pair of adjacent transitions in the FSM, the test suite contains a test which traverses it in sequence. Two transitions are adjacent when one of them enters and the other exits the same state. To generate the FSM test suites, a three step approach is used. First, the full list of transition pairs is generated for the FSM. Then, Dijkstra's algorithm [177] is used to generate the shortest path to each state, from the FSM's starting state. Finally, each transition pair

is concatenated to the shortest path to its start state, resulting in a test sequence which starts in the FSM's start state and ends with the transition pair.

Figure 7.2 illustrates this process for the digital oscilloscope FSM, shown in Figure 2.6 in Section 2.5.2. The FSM has 4 states and 8 transitions, which result in 16 transition pairs. Each transition pair is characterised by its two transition inputs, start state, middle state and end state. Figure 7.2 shows that the first 9 transition pairs start in state B0, the next 5 transition pairs start in state B1 and the last two transition pairs start in state B3. It also shows the shortest paths to each of the four states. Concatenating each transition pair to the shortest path to its start state generates a test sequences that covers that transition pair. Therefore, there are 16 tests, each of which covers a separate transition pair. In this way, the generated test suite provides full transition pair coverage for the FSM.

7.2.3 Test Suite Reduction

The test generation algorithm ensures that the test suite meets the coverage criterion, but does not necessarily lead to the smallest possible test suite. Thus, there could be redundant tests in the test suites that cover only transition pairs already covered by other tests. This is illustrated in the example shown in Figure 7.2. One redundant test in this example is test 8, `LM`. It covers only one transition pair, which starts in state 0, uses transition `L` to move to state 1 and then uses transition `M` to move to state 2. But this transition pair is covered by other tests, such as tests 15 and 16, which cover additional transition pairs. Therefore, test 8 is redundant. Removing redundant tests would reduce the size of the test suites, making them faster to transfer and execute, while maintaining coverage. It could compliment other approaches to improving the performance and scalability of GPU test acceleration.

To remove redundant tests, Algorithm 1 is designed and implemented. The key observation behind it is that shorter tests are more likely to be redundant than longer tests. Since all tests start from the same starting state, only longer tests are able to cover transition pairs that are far from the starting state. Shorter tests, on the other hand, are more likely to be redundant, since they might comprise part of the path of a longer test. The algorithm requires three inputs: (1) the FSM tests, sorted by length in decreasing order, (2) the FSM transition pairs, each of which consists of the inputs for the two transitions, as well as start state, middle state and end state, where the transitions meet, and (3) the start state for the FSM. The algorithm iterates over all tests, marking all

ALGORITHM 1

Test suite reduction algorithm. It finds and removes redundant tests from an FSM test suite. Redundant tests are tests which cover *only* transition pairs that are already covered by other tests.

Require: T - tests, sorted by length, decreasing order
  TP - transition pairs in the FSM
  $start\_state$ - the start state of the FSM
  for each t in T **do**
    $visits\_new\_tp \leftarrow false$
    $current\_state \leftarrow start\_state$
    for each input in the test sequence of t **do**
      find corresponding transition pair tp in TP
      if tp has not been visited **then**
        mark tp as visited
        $visits\_new\_tp \leftarrow true$
      end if
      $current\_state \leftarrow tp.middle\_state$
    end for
    if $visits\_new\_tp$ is $false$ **then**
      remove t from T
    end if
  end for

transition pairs that a test covers as *visited*. At each step, it also checks if the test covers transition pairs that have not been marked as *visited* by previous tests. If it does not, then the test is redundant and removed from the test suite.

Implementation Both test generation and test suite reduction are implemented as stand-alone programs in C.

7.3 Experimental Setup

This section presents the experiments carried out to evaluate the performance of FSM test execution on the GPU and the effectiveness of the optimisations outlined in Section 7.2. The evaluation uses 15 large FSMs from the Snort [176] network intrusion detection protocols. For each FSM, testing time on the GPU is measured and compared to the time taken to execute the same tests in parallel on a 16-core CPU. For test execution on the GPU, the following measurements are taken:

- **Data transfer time:** time taken to transfer the FSM and tests from main memory to GPU memory, plus time taken to transfer the test outputs back from GPU memory to main memory.
- **Kernel execution time:** time taken by the GPU to execute the tests.
- **Total GPU time:** the overall time taken by the GPU; it comprises the sum of data transfer and kernel execution time.

The experiments aim to answer the following research questions:

Q1. Speedup in total GPU time, without data transfer overlap. *What is the speedup in total GPU time compared to a 16-core CPU, without data transfer overlap?* For all FSMs, test suites are executed in parallel on the GPU and on a 16-core CPU, and GPU speedup is calculated based on total GPU time. No data transfer overlap is performed and this speedup is used as a baseline for comparison to the results for Q2. The experiments are performed using different test suite sizes, ranging from $2048 (2^{11})$ to the full test suite, in order to assess how the speedup changes as the size of the test suite increases. To measure the overhead of data transfer time, separate measurements for data transfer, kernel execution and total GPU time are taken and compared.

Q2. Data transfer overlap: effect on performance. *Does data transfer overlap improve GPU speedup?* For all FSMs, test suites are executed by overlapping data transfer with kernel execution. The speedup of total GPU time (over a 16-core CPU) *with* data transfer overlap is compared to the speedup *without* data transfer overlap.

Q3. Data transfer overlap: effect on scalability. *Does data transfer overlap allow the execution of larger test suites?* For each FSM, the number of tests which can fit into GPU memory with data transfer overlap is compared to that without data transfer overlap.

Q4. Test suite reduction. *Can the number of tests the test suite be reduced by removing redundant tests?* For each FSM, test suite reduction is performed by removing redundant tests with respect to all-transition pair coverage, as described in Section 7.2.3, and measuring the percentage reduction in test suite size.

7.3.1 Subject FSMs and Tests

This evaluation uses 15 subject FSMs from the Snort [176] Community ruleset for network intrusion detection. The ruleset contains regular expressions representing network intrusion detection protocols. The Flex tool [175] is used to convert the regular expressions into FSMs, along with a custom Python script to generate a file for each FSM in the kiss2 format [172]. The set of possible inputs for each of the FSMs comprises the 256 ASCII characters.

At the time of experimentation, the Snort ruleset contains 821 regular expressions. All of them are converted into FSMs and a sample of 15 subject FSMs is chosen for experimentation in such a way that they represent a wide range of sizes in terms of number of states (ranging between 204 and $2,169$) and transitions (ranging between 48K and 547K). Table 7.1 provides a summary of the subject FSMs, showing their sizes. All FSMs are chosen to be larger than the ones used in Chapter 6.

Test Generation A full test suite for each FSM is generated, based on the all-transition pair coverage criterion, using the approach outlined in Section 7.2.2. Table 7.1 shows the number of tests, while Figure 7.3 shows the average length of the tests for each FSM. The sizes of the test suites range between 12M and 138M tests and the test suites vary in terms of the average length of tests, ranging between 100 and 544 inputs.

FSM	#States ($\|S\|$)	#Transitions	Density	#Tests
`fsm42`	204	48,079	94%	12M
`fsm247`	215	50,610	94%	13M
`fsm239`	221	50,617	91%	13M
`fsm216`	269	64,524	95%	15M
`fsm263`	273	63,764	93%	15M
`fsm265`	275	63,766	92%	15M
`fsm261`	548	129,578	93%	33M
`fsm286`	445	109,304	97%	28M
`fsm291`	445	109,304	97%	28M
`fsm287`	446	109,305	97%	28M
`fsm37`	515	126,510	97%	32M
`fsm770`	1,565	379,565	95%	96M
`fsm213`	2,031	509,562	98%	129M
`fsm382`	2,169	546,994	99%	138M
`fsm268`	2,169	546,994	99%	138M

Table 7.1: Subject FSMs used in the evaluation. All FSMs are from the Snort Community ruleset for network intrusion detection.

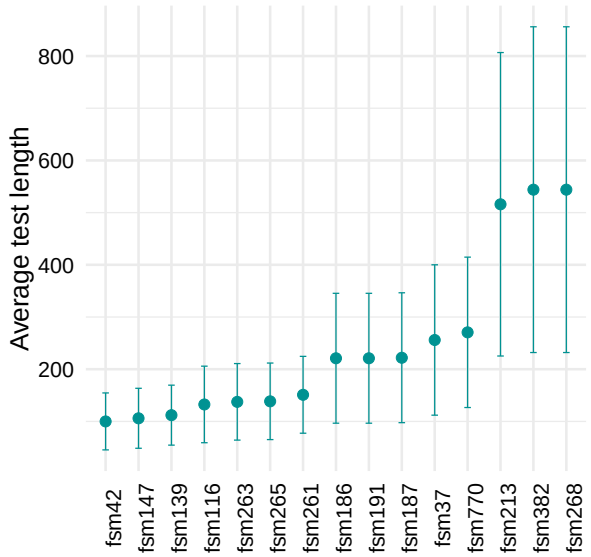


Figure 7.3: Average test lengths for FSMs used in evaluation. Error bars show standard deviation within the test suite.

FSM Density FSM density is defined in Section 2.5. It is the percentage of transitions present in the FSM out of the maximum number of transitions possible for the FSM. In this case, for all subject FSMs there are 256 possible input characters for each state, thus 256 possible transitions. Therefore, the maximum number of transitions in any of the subject FSMs is $|S| * 256$, where S is the set of states. Table 7.1 shows the densities of the subject FSMs. All of them have high densities between 91% and 99%, which indicates that for most states a large number of transitions are present and as a result, there is a high degree of branching and path divergence within the FSMs.

7.3.2 Hardware and Measurements

The same GPU and CPU used in Chapter 5 are used for these experiments. They are described in Section 5.3.2. Each experimental execution is performed 11 times and median values are reported.

Multi-core CPU Execution Test execution on the CPU is parallelised using OpenMP and FSM tests are executed in parallel on the 16 cores.

Correctness For each experiment, the testing outputs produced by the GPU are compared to those from the CPU in order to confirm that they are an **exact match** ensure that using the GPU preserves the correctness of test execution.

7.4 Results and Analysis

This section presents the results and analysis of the experiments described in Section 7.3. First, the speedup for total GPU time compared to a 16-core CPU without data transfer overlap is evaluated in Q1. Then, the effectiveness of data transfer overlap is assessed in Q2 and Q3. Finally, the effectiveness of test suite reduction is evaluated in Q4.

7.4.1 Q1. Speedup in Total GPU Time, No Data Transfer Overlap.

Figure 7.4 shows the speedup achieved by the GPU for total GPU time when compared to a parallel test execution on a 16-core CPU for different test suite sizes, without data transfer overlap. For all FSMs, the GPU is consistently faster than the 16-core CPU with a maximum speedup of 5.6× and an average speedup across all FSMs of 2.9×.

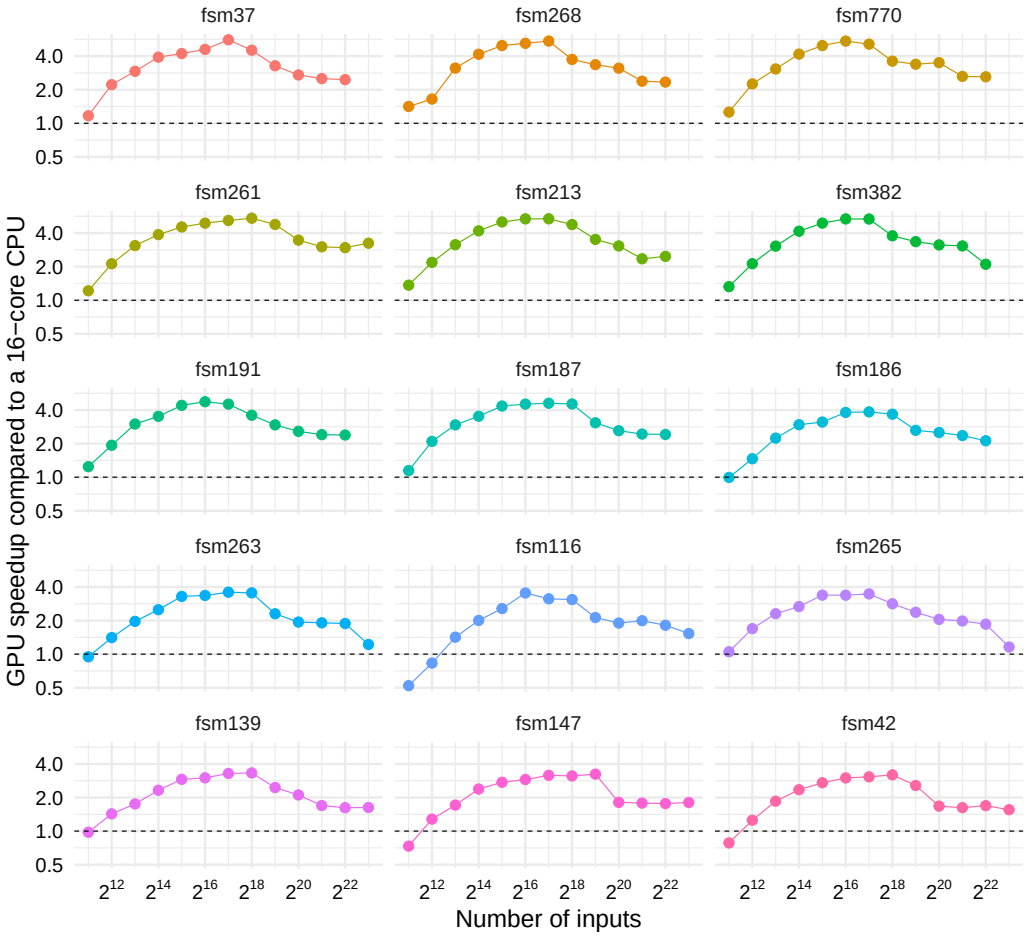


Figure 7.4: Speedup in total GPU time when compared to a 16-core CPU over different test suite sizes, without data transfer overlap.

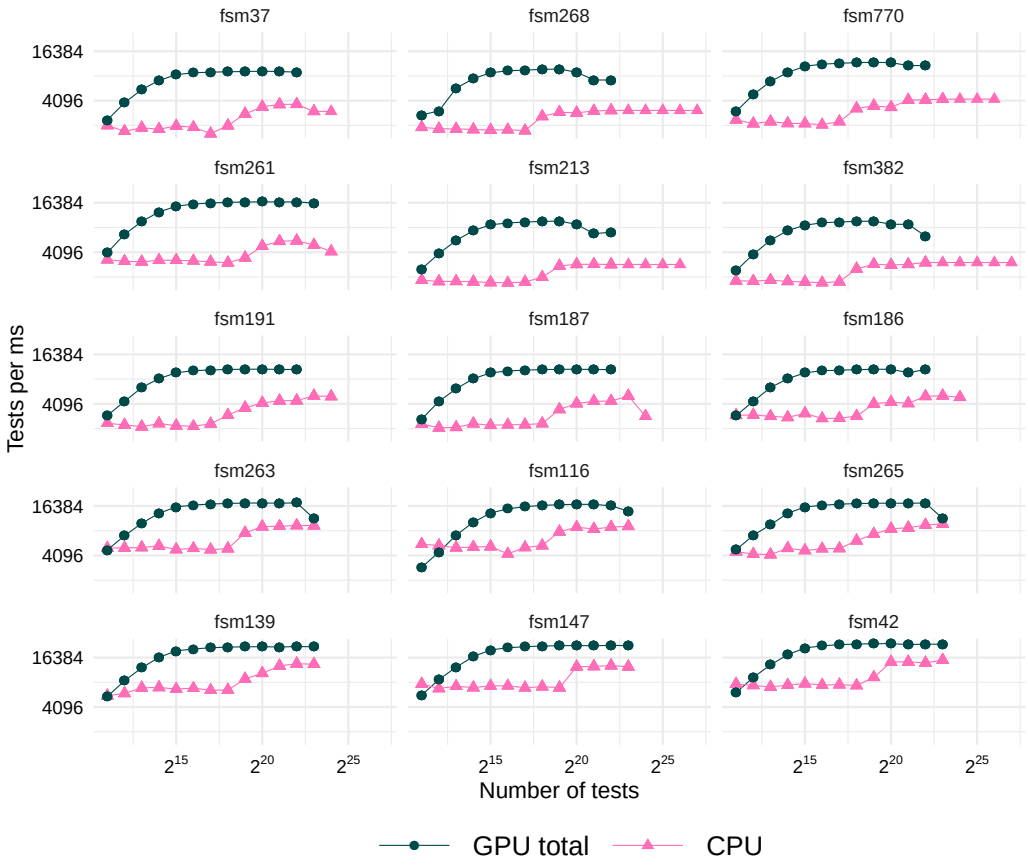


Figure 7.5: Efficiency of the GPU and CPU measured as number of tests executed per millisecond.

Speedup Trend Figure 7.4 shows that for all FSMs, speedup increases with the sizes of the test suite to up to approx. 2^{18} tests, after which there is a dip. To aid analysing this trend, Figure 7.5 illustrates the ways in which execution times change both on the CPU and the GPU as the number of tests increases. It shows the efficiency of the GPU and CPU, measured as *number of tests executed per millisecond*. For the GPU, efficiency increases with the size of the test suite for all FSMs up to 2^{15} tests, after which it remains the same. This is expected, because up to 2^{15} tests, the GPU has free threads to utilise, after which all of its resources are saturated and its efficiency is constant. In contrast, on the CPU all cores are utilised for all test suite sizes and therefore the efficiency of the CPU remains the same as more tests are added, up to approx. 2^{18} tests. These trends explain the increase of GPU speedup up to approx. 2^{18} tests, which is shown in Figure 7.4.

In addition, CPU efficiency increases for test suite sizes greater than 2^{18} tests, corresponding to the dip in GPU speedup observed in Figure 7.4. The likely reason for this is a hardware optimisation, called *automatic frequency scaling*. It is a power saving optimisation, which dynamically reduces or increases the processor frequency based on the workload. When executing larger test suites, the CPU frequency is increased, providing a boost to its performance. Therefore, the dip in GPU speedup is not due to the loss of efficiency in the GPU, but due to an improvement in performance on the CPU for larger test suites.

Scalability to Large Test Suites In Figure 7.5 shows that for 9 FSMs the GPU does not execute the largest test suite sizes. These are fsm37, fsm268, fsm770, fsm261, fsm213, fsm382, fsm191, fsm187 and fsm186. This is because the larger test suites do not fit into GPU memory, illustrating the scalability limitations of a naive approach. This problem is address with the use of data transfer overlap and evaluated in Q3 (Section 7.4.3).

Data Transfer Overhead Figure 7.4 shows the speedup of *total GPU time*, which includes both data transfer and kernel execution time on the GPU. For comparison, Figure 7.6 shows the speedup of the GPU kernel, when data transfer is not taken into account. GPU kernel speedup is up to 45.1× compared to the 16-core CPU, with an average of 20× across FSMs. This is an order of magnitude higher than total GPU speedup presented in Figure 7.4, demonstrating the large overhead of data transfer and its strong effect on GPU speedup.

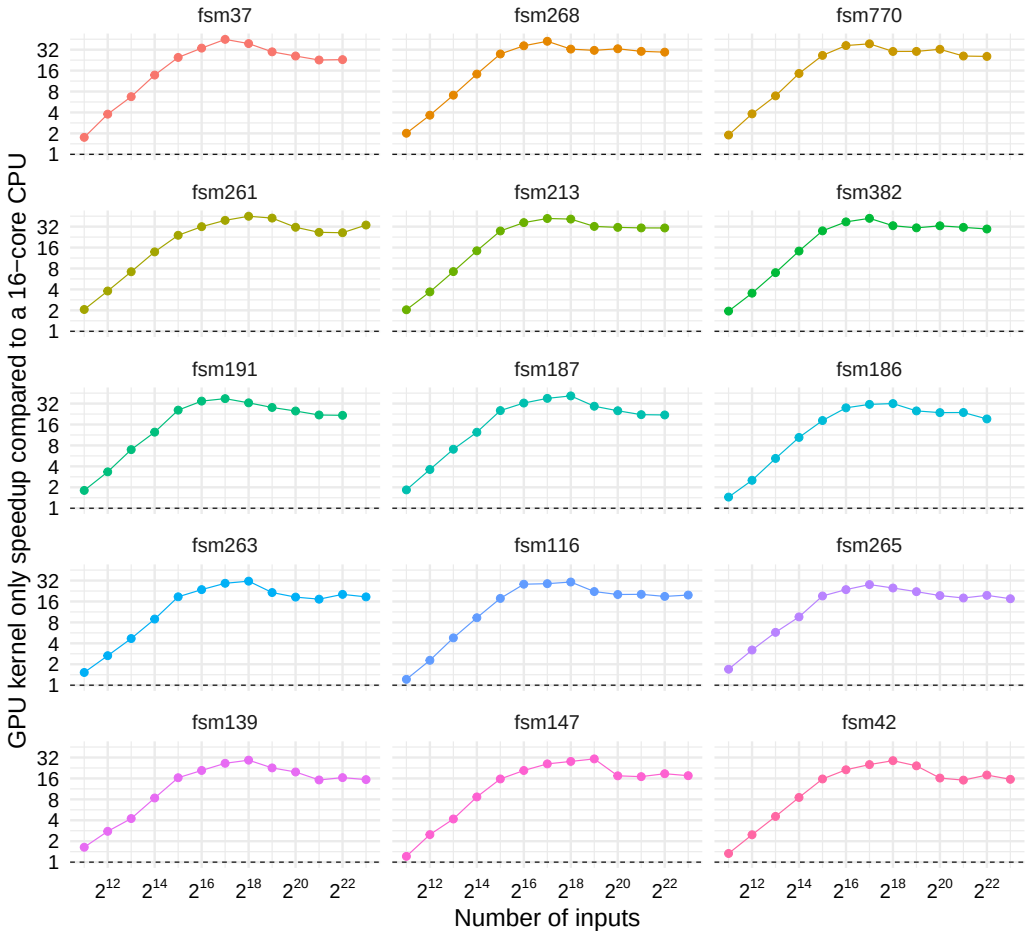


Figure 7.6: Speedup in *GPU kernel time only* when compared to a 16-core CPU over different test suite sizes.

To further analyse data transfer overhead, Figure 7.7 shows the breakdown of total GPU time into data transfer and kernel execution times. The pattern is similar for all FSMs. The smaller the test suite sizes, the smaller the data transfer overhead. For 2^{11} tests only approx. 20% of the time is spent in data transfer and 80% is spent in kernel execution. As the size of the test suite grows and the GPU threads get saturated after 2^{15} tests, the proportions gradually change until approx. 90% of the execution time is spent in data transfer, while only about 10% is spent in kernel execution. The reason for this is that unlike kernel execution, data transfer is not parallelised at all. Therefore, for large test suites, data transfer represents a significant overhead, which explains the order of magnitude difference between the total GPU and kernel speedups.

This demonstrates the importance of optimising data transfer, as even small reductions in its time could lead to considerable gains in speedup for total GPU execution. This is addressed using data transfer overlap and evaluated in Q2.

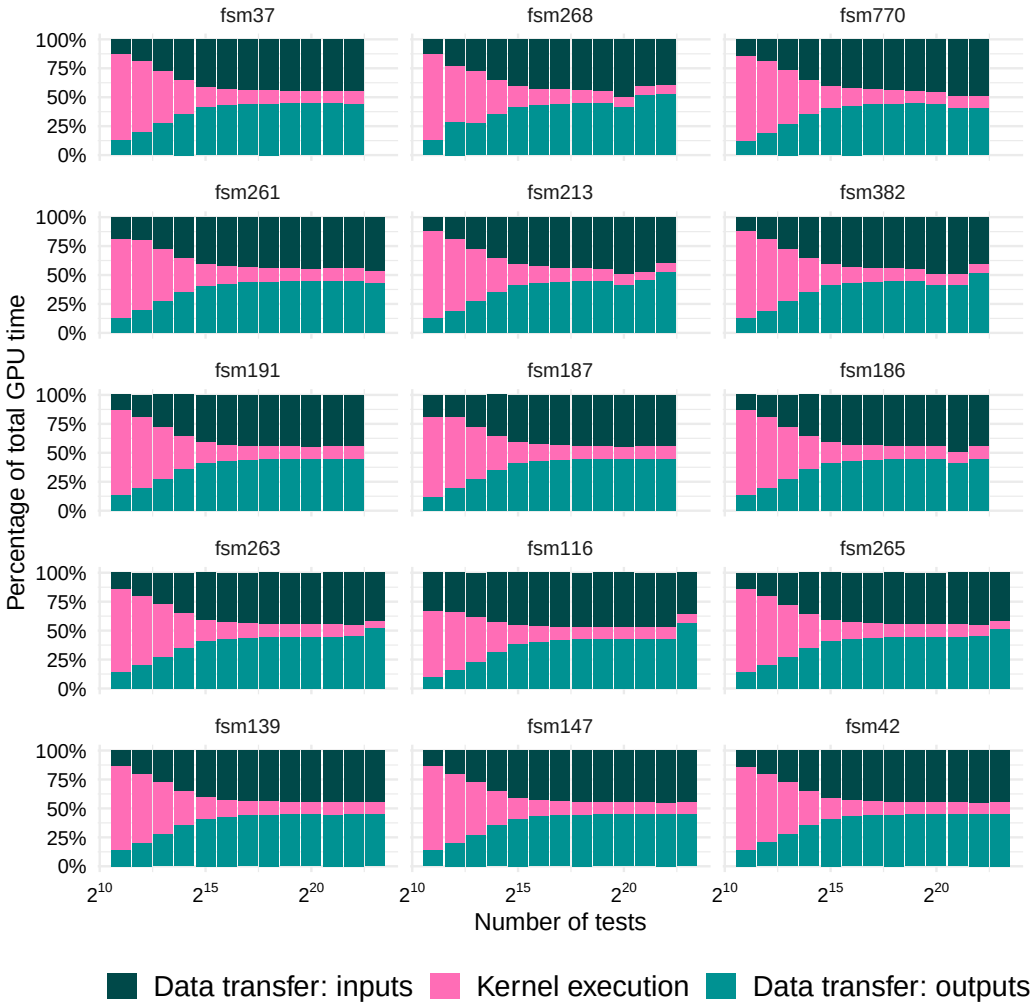


Figure 7.7: Breakdown of total GPU time into data transfer and kernel execution times.

7.4.2 Q2. Data Transfer Overlap: Effect on Performance

Figure 7.8 shows the speedup achieved in total GPU time, compared to the 16-core CPU, with data transfer overlap. It also shows GPU speedup without data transfer overlap to aid comparison. For all FSMs, GPU speedup in both cases is similar for small test suite size of up to approx. 2^{14} tests. This is not surprising, as in both cases, up to this point, the GPU threads are not saturated and the GPU is not working at its full capacity. In addition, as seen in Figure 7.7, small test suites comprise only a small portion of total GPU time even without data transfer overlap. In contrast, for test suites larger than 2^{14}, after all GPU threads are employed, there is improvement in total GPU speedup with data transfer overlap versus without, across all FSMs. The maximum speedup with data transfer overlap is $9.3\times$ and the average speedup across all FSMs is $4.5\times$. On average, the increase in speedup compared to execution without data transfer overlap is 58.95%.

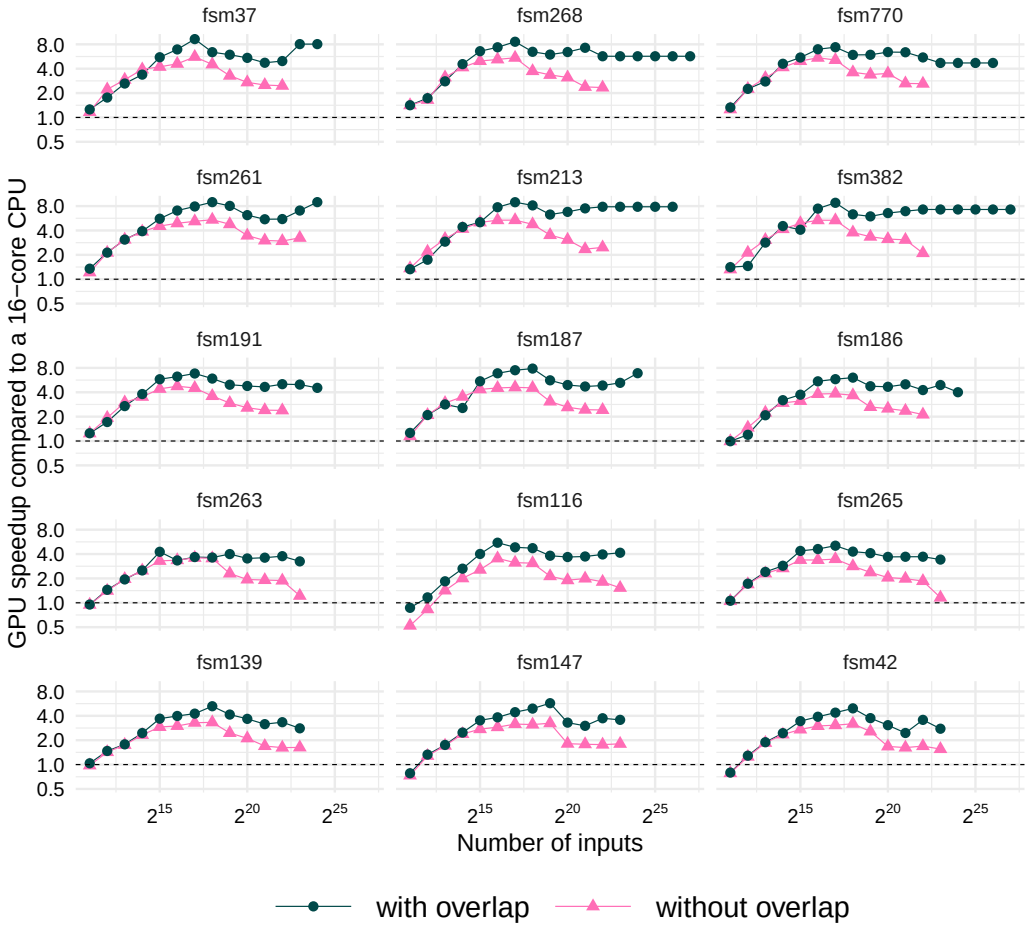


Figure 7.8: Comparison between speedup in total GPU time, with and without data transfer overlap, with different test suite sizes. Speedup is over a 16-core CPU.

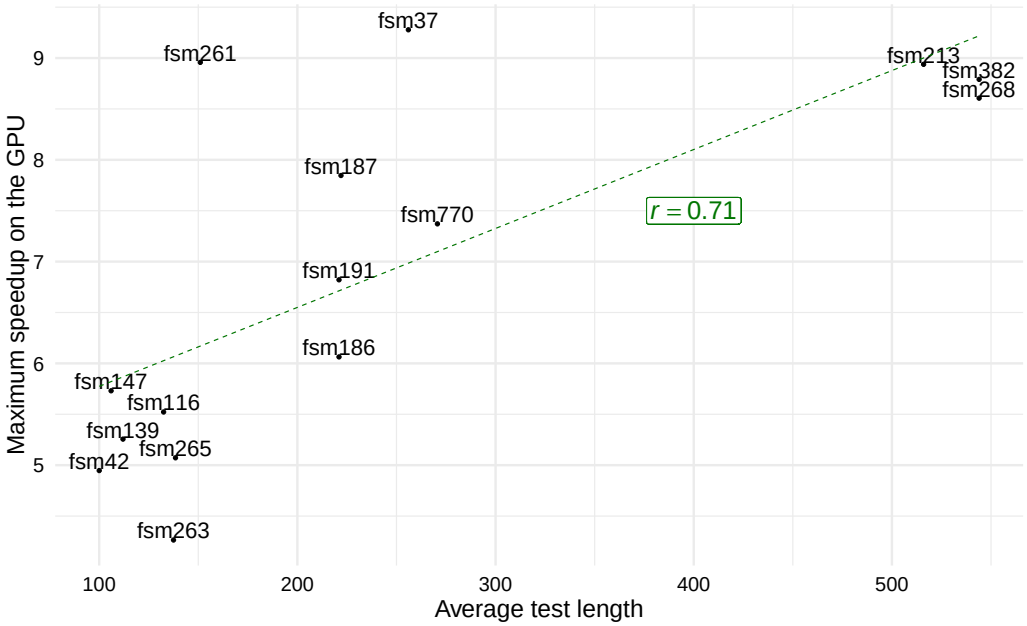


Figure 7.9: Correlation between the average test length in the test suite and the maximum GPU speedup. Pearson correlation coefficient is 0.71.

Difference in Speedup Across FSMs The maximum GPU speedup across all FSMs ranges between 4.3× (for fsm263) and 9.3× (for fsm37), with an average of 6.9×. The reason for the difference across FSMs lies in the length of the tests in their test suites. Analysis of evaluation results in Section 6.4.1 suggest that a positive correlation exists between the average length of tests in the test suite and the maximum GPU speedup for a given FSM. Figure 7.9 shows that there is indeed a strong positive correlation between the two, with a Pearson correlation coefficient of 0.71, for the subject FSMs used in this evaluation. The longer the tests in the test suites, the higher the speedup achieved by the GPU. This is because longer test sequences require longer execution per test both on the GPU and CPU. As the GPU has a higher degree of parallelism, the extra computation for the longer tests per thread is lower than that of each individual CPU core, which needs to execute multiple tests. This allows the GPU to execute longer test sequences faster than the CPU, resulting in higher speedup.

7.4.3 Q3. Data Transfer Overlap: Effect on Scalability

Table 7.2 shows the number of tests that can be executed on the GPU without data transfer overlap and with data transfer overlap. Without data transfer overlap, the full test suites for six of the FSMs can be executed, while for the remaining nine FSMs, only up to approx. 2^{22} tests can be executed. When executing the full test suites for these FSMs,

GPU execution is failing with OpenCL's *CL_MEM_OBJECT_ALLOCATION_FAILURE* error, indicating that the GPU is running out of memory. This is because for these FSMs, the full test suites are large, requiring a large amount of memory to accommodate both the test inputs and outputs, which is not available on the GPU. In particular, the GPU used for the experiments has 12 GB of global memory. For the first six FSMs in Table 7.2, the combined sizes of test inputs and outputs vary between 1.36 GB and 4.50 GB, while for the remaining nine FSMs, they vary between 14.8 GB and 145.52 GB, not fitting into the global memory of the GPU.

FSM	#Tests	#Exec on GPU no overlap?	#Exec. on GPU with overlap?
fsm42	12M (2^{23})	2^{23} (✓)	2^{23} (✓)
fsm147	13M (2^{23})	2^{23} (✓)	2^{23} (✓)
fsm139	13M (2^{23})	2^{23} (✓)	2^{23} (✓)
fsm116	16M (2^{23})	2^{23} (✓)	2^{23} (✓)
fsm263	16M (2^{23})	2^{23} (✓)	2^{23} (✓)
fsm265	16M (2^{23})	2^{23} (✓)	2^{23} (✓)
fsm261	33M (2^{24})	2^{23} (✗)	2^{24} (✓)
fsm186	28M (2^{24})	2^{22} (✗)	2^{24} (✓)
fsm191	28M (2^{24})	2^{22} (✗)	2^{24} (✓)
fsm187	28M (2^{24})	2^{22} (✗)	2^{24} (✓)
fsm37	32M (2^{24})	2^{22} (✗)	2^{24} (✓)
fsm770	96M (2^{26})	2^{22} (✗)	2^{26} (✓)
fsm213	129M (2^{26})	2^{22} (✗)	2^{26} (✓)
fsm382	138M (2^{27})	2^{22} (✗)	2^{27} (✓)
fsm268	138M (2^{27})	2^{22} (✗)	2^{27} (✓)

Table 7.2: Number of tests executed on the GPU with and without data transfer overlap.

In contrast, when data transfer overlap is used, the full test suites for all FSMs can be executed on the GPU, as shown in Table 7.2. This is because data transfer overlap removes the need to store the full test suite in GPU memory. Instead, the test suite is split into groups which are swapped in and out of GPU memory as tests are being executed. For larger test suites, there are more groups, but the amount of memory required on the GPU at any one time is always the same - equal to the size of the group. Thus, data transfer overlap makes testing on the GPU scalable to larger FSMs with larger test suites than is possible without it.

In addition, the same limitation can be observed for the CPU. Even though modern CPUs tend to have more memory than GPUs, it is finite. For FSMs whose tests do not fit into CPU memory, the same strategy can by employed by reading tests in groups from the hard disk.

7.4.4 Q4. Test Suite Reduction

Figure 7.10 shows the reduction in the number of tests after removing redundant tests from the test suites of the subject FSMs, while maintaining all-transition pair coverage. Across all FSMs, the reduction is negligible, only up to approx. 0.4%. The reason for this lies in the high density of the subject FSMs, which is shown in Table 7.1. All of them have densities over 90%. Since high density indicates a high degree of branching and path divergence in the FSM, there is little overlap among the paths covered by the tests and thus, only a small number of redundant tests.

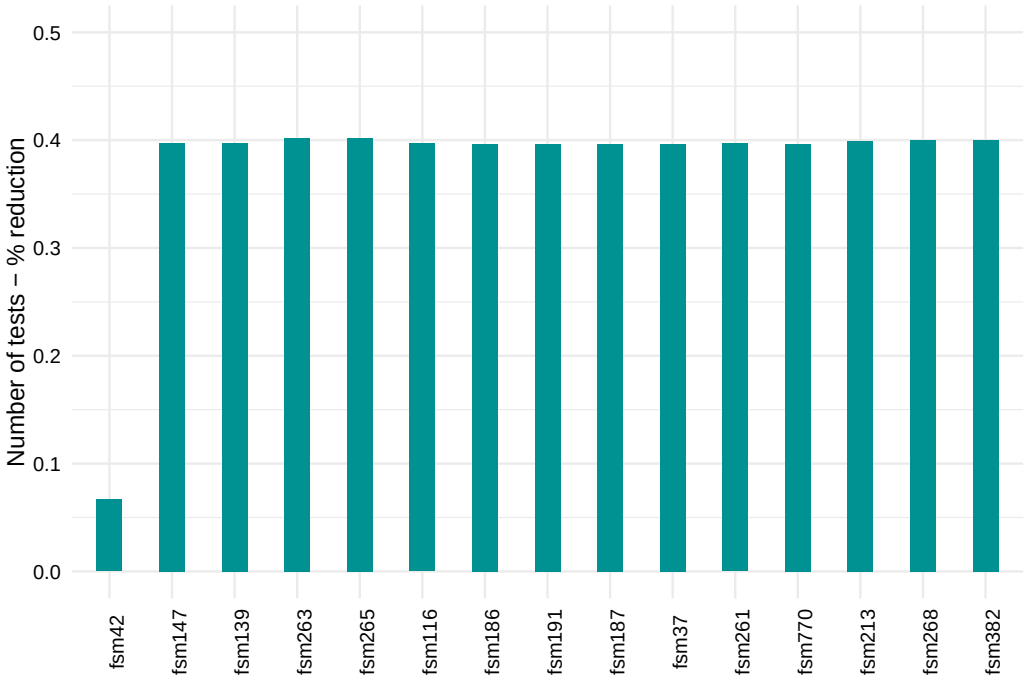


Figure 7.10: Percentage change in the number of tests after test suite reduction for all subject FSMs.

To confirm this, the correlation between FSM density and test suite reduction is calculated and shown in Figure 7.11. For the purpose of having a wider range of FSM densities, 13 additional FSMs from the Snort ruleset with densities ranging between 2% and 84% are chosen, noted as *additional FSMs* in Figure 7.11. For all of them, a full

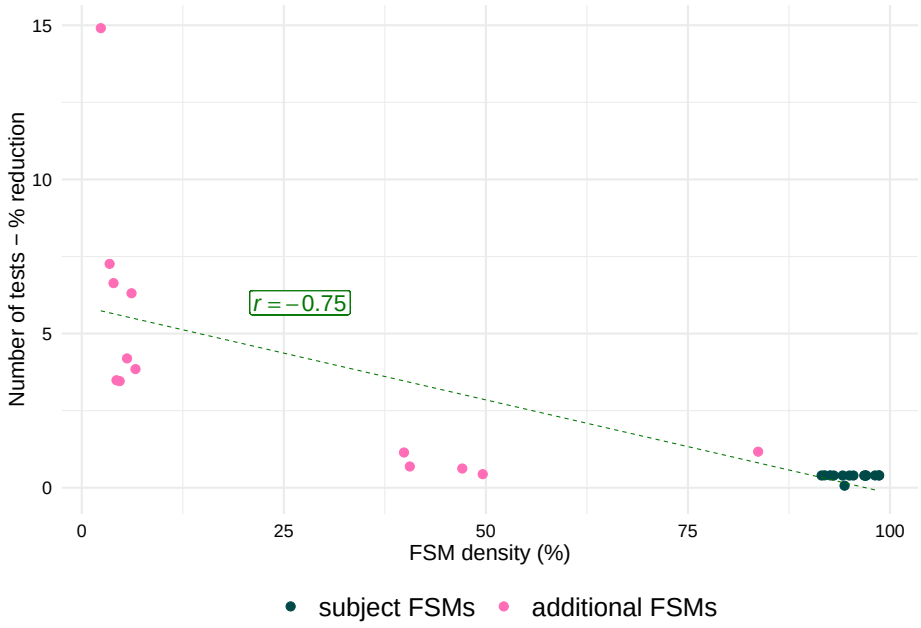


Figure 7.11: Correlation between the density of the FSM and the % reduction in number of tests. Pearson correlation coefficient is -0.75.

test suite based on all-transition pair coverage is generated and then test reduction is performed. Figure 7.11 shows that there is a strong negative correlation between FSM density and test suite reduction, with a Pearson correlation coefficient of -0.75; the higher the density of the FSM, the lower the reduction.

This shows that for large FSMs with high density test suite reduction is not effective. Such FSMs need large test suites to satisfy rigorous coverage criteria. This confirms the importance of developing alternative techniques, such as using GPUs, for the acceleration of test execution.

7.5 Summary

This chapter presents approaches to improve the performance and scalability of using GPUs to accelerate test execution for large FSMs. It extends the work presented in Chapter 6 by considering the time taken to transfer data between main memory and GPU memory and mitigating its impact on total GPU performance and scalability by (1) using data transfer overlap to pipeline data transfer time with GPU kernel execution, and (2) performing test suite reduction for large test suites, while maintaining coverage.

To assess the effectiveness of these techniques, an extensive empirical evaluation is performed using 15 large FSMs from the Snort ruleset for network intrusion detection.

In contrast to Chapter 7 which focuses the performance of the GPU kernel only, this chapter evaluates total GPU performance, which includes both kernel execution and data transfer. The results reveal that by using data transfer overlap, the GPU outperforms a parallel test execution by a 16-core CPU by up to 9.3×, average 4.5× across all subject FSMs. This an average performance improvement of 58.95%, achieved by data transfer overlap. In addition, data transfer overlap improves scalability to large FSMs with more than 2K states and 500K transitions, by splitting large test suites that would otherwise not fit in GPU memory, into smaller groups.

With respect to test suite reduction, for the 15 subject FSMs, the maximum percentage reduction in the number of tests is only 0.4%. This is owing to the high density of the subject FSMs, which is more than 90% for all of them. Analysis of the results reveals that the percentage of redundant tests in the test suite is inversely correlated with the density of the FSM. For these reasons, all of the tests in the test suites of these FSMs are needed to maintain coverage, limiting the effectiveness of test suite reduction and requiring alternative approaches to test execution speedup, such as using GPU hardware.

Chapter 8

Conclusion

This thesis presents a novel approach to accelerate software test execution by parallelising it using GPU architectures. It addresses challenges related to the usability, scope, performance and scalability of the approach and demonstrates that GPUs can effectively reduce test execution time in certain contexts. Chapter 4 presents the underlying approach and tools used throughout this thesis in the form of ParTeCL - a fully automated testing framework, which transforms the SUT into GPU source code and launches test execution in parallel on the GPU threads. Chapter 5 evaluates the applicability and performance of the approach when applied to the embedded systems domain, using programs from the EEMBC benchmark suite. Chapter 6 demonstrates the feasibility of the approach when applied to the testing of FSM models by defining and evaluating suitable memory layouts for the FSM and its test suite. Finally, Chapter 7 extends the work in Chapter 6 by exploring techniques to improve the performance and scalability of the approach for FSM test execution.

This chapter concludes the work presented in this thesis. Section 8.1 summarises the main contributions, Section 8.2 presents a critical analysis of the work and Section 8.3 discusses future research directions. Finally, Section 8.4 provides concluding remarks.

8.1 Contributions

This thesis makes three main contributions, representing first steps towards the adoption of GPU architectures for accelerated test execution.

Automated GPU Test Execution

Usability represents a serious challenge to the adoption of GPUs in software engineering, as writing GPU software requires specialist architecture knowledge and the use of niche low-level programming models. To address this challenge, Chapter 4 presents ParTeCL, an automated testing framework for the GPU, which targets sequential C programs. It uses compiler-based source-to-source transformations to generate OpenCL kernels for the SUT and launches test execution in parallel on the GPU threads.

By automating the entire testing process on the GPU, ParTeCL removes the need to manually write GPU code, improving the usability of the approach and making it accessible to all software engineers. Furthermore, automation is crucial for the adoption of any novel method that improves regression test execution, as it ensures that it can be reused with minimal effort in regular test runs [178]. Finally, ParTeCL forms the basis for the research in this thesis, as it is framework in which the scope performance and scalability optimisations are implemented and evaluated.

Accelerated Embedded System and FSM Testing

Chapters 5 and 6 evaluate the applicability of using GPUs for test executions for two types of applications. The first is C programs from the embedded systems domain. Chapter 5 shows that through the use of ParTeCL, 17 out of 33 applications from the EEMBC benchmark suite can be successfully tested on the GPU. These are applications primarily from the automotive and telecommunications domains. This is achieved through the use of compiler-based source-to-source transformations inside ParTeCL, targeting global scope variables, command line arguments, standard input and output and standard library calls, which are not readily available in OpenCL. Chapter 5 also analyses features within the rest of the EEMBC programs, which prevent them from being compiled for the GPU and limit the scope of the approach. These features include dynamic memory operations, file IO, function pointers and some data structures.

The second type of application is FSM models. Chapter 6 demonstrates the feasibility of using ParTeCL to automatically accelerate FSM test execution by defining and implementing two memory layouts for the FSM and three memory layouts for the test suites. It evaluates them using 12 FSMs from the network intrusion detection and 1 FSM from the signal processing domains, establishing which implementation lead to best GPU kernel performance for test execution. Chapter 7 improves the scalability of the approach to large FSMs by using techniques implemented in ParTeCL.

Performance Optimisations and Analysis

Throughout this thesis, GPU performance for test execution is optimised and analysed. First, Chapter 4 implements two standard techniques for the optimisation of data transfer between main memory and GPU memory: (1) using DMA controllers and (2) overlapping data transfer and kernel execution on the GPU. Then, Chapter 5 provides performance analysis for embedded system testing, showing that, with the standard optimisations, the GPU outperforms a 16-core CPU by up to 4× (avg. 1.4×). Chapter 6 analyses the performance of *GPU kernel execution* for FSM testing, in order to discover the optimal memory layouts for the FSM and test suite. Finally, Chapter 7 analyses *total GPU performance* for FSM test execution, which includes both kernel execution and data transfer time. It demonstrates that, with optimisations, the GPU is up to 9.3× (avg. 4.5×) faster when executing FSM tests than a 16-core CPU.

These performance results demonstrate that GPUs can achieve better performance for test execution when compared to parallel execution on a multi-core CPU.

8.2 Critical Analysis

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

Limited Scope of C Programs

Chapter 5 analyses the applicability of the approach to sequential C programs from the embedded systems domain. 16 out of the 33 programs in the EEMBC benchmark suite cannot be compiled for execution on the GPU, due to using features that are unsupported by the GPU architecture and programming models. Such features are dynamic memory allocation, file I/O, function pointers and some data structures. Of these, dynamic memory allocation is the one present in most applications (13 out of 16). Another feature which is not supported on the GPU is recursion.

It is unclear how prevalent these features are in most C programs, but some of them, such as dynamic memory allocation and recursion, are likely to occur frequently in other applications. Therefore, these GPU limitations considerably restrict the current scope of the approach and its applicability in real-world settings. Section 4.4 discusses the unsupported features and briefly outlines ideas for potential solutions, but further work is necessary to explore them.

Scalability to Large C Programs

The C programs from the EEMBC benchmark suite used in Chapter 5 are small, each consisting of a single core algorithm, and they are fast to execute. While representative of the types of computations in the embedded system domain, they are not representative of complex real-world C programs with multiple functions. Thus, the current evaluation does not demonstrate how well the approach will scale to larger C programs.

Scalability to larger programs could be limited by GPU memory. Limited constant memory is the reason for one EEMBC benchmark, `ttsprk01`, to fail to execute on the GPU in Section 5.2.2. GPUs tend to have less memory than CPUs and have no access to a hard disk. When executing tests in parallel, GPU memory is shared by all instances of the SUT, further limiting the amount of memory available per test. On the other hand, using integrated GPUs which have access to main memory might alleviate these issues. Furthermore, as GPU architectures evolve their memory tends to grow, and this limitation may be less of a problem in the future. Nevertheless, further evaluation with larger C programs is needed to analyse the scalability of the approach.

Validity of Embedded Software Testing on the GPU

A key characteristic of embedded software is its close relationship with the hardware on which it executes. Embedded software is typically developed and executed on specific target platforms using distinct tool-chains. GPUs represent a very different environment to the ones in which an embedded SUT will operate, posing a threat to the validity of the testing results produced on the GPU. For example, bugs in the GPU acceleration tool-chain could lead to tests passing when they should actually fail or vice versa. Similarly, bugs in the embedded system tool-chain could lead to the deployed system failing, but will be missed during testing when GPU acceleration is used. Differences can also arise due to divergence in implementation-defined and undefined behaviours between the GPU and target platforms.

This problem can be avoided by occasionally executing the full test suite in the target environment. Therefore, GPUs are not suitable as the sole method for testing embedded software. They can be used to accelerate regular testing during development, together with less frequent test executions on the target hardware with the target tool-chain.

Generality of FSMs

Evaluations in Chapters 6 and 7 demonstrate that GPUs are well suited to parallelising test executions for validating FSM models. All subject FSMs are successfully tested on the GPU, achieving better performance than a 16-core CPU. While they represent a range of sizes (from 24 to 2,169 states) and densities (from 56% to 99%), all of them, except for one, are from a single domain - network intrusion detection protocols. Therefore, it remains unclear if the performance results in these chapters are representative for FSMs in other domains. Generalising the results requires further empirical evaluation.

Performance Dependence on the GPU Architecture

Only one type of GPU architecture, the NVidia Tesla K40m, is used for all experiments in this thesis. Nevertheless, performance is known to vary across GPU architectures, as they differ in their hardware characteristics. For example, optimisations performed in this work rely on the availability of specific GPU hardware components - a DMA controller, for low-latency data transfers, and a dual copy engine, for overlapping data transfers of test inputs and outputs (Figure 4.4b). To find out how much of an effect different GPU architectures have on the performance of test execution, further experiments using alternative GPU architectures are necessary.

8.3 Future Work

All of the issues discussed in Section 8.2 represent interesting problems for future work. This section presents additional directions for future research.

Performance Impact of Work-group Sizes

GPU performance is dependent on selecting the optimal value for the work-group size (Section 1.2.3). The performance experiments presented in Chapter 5 are executed using a range of work-group sizes and the fastest results are presented. For Chapters 6 and 7, the work-group size value is fixed at 256, in order to reduce experimentation effort, but different values can be expected to lead to different performance results.

In future work, measuring and analysing the effect of the work-group size on GPU test execution performance could provide useful insight into how important it is to select an appropriate value. An interesting next step could be exploring if existing dynamic approaches for work-group size tuning [21, 22] could be integrated into the testing process.

Parallel Testing of Separate Program Units

In the approach presented in this thesis, the whole GPU is dedicated to the testing of a single SUT, which can be a whole program or a subprogram. However, a system's test suite may consist of several test suites for individual units or modules. These could be compiled into separate OpenCL kernels and launched simultaneously on different compute units on GPUs which allow partitioning, using OpenCL's *clCreateSubDevices* [179]. Different scheduling strategies will need to be explored in order to optimise performance, depending on the program and test data.

Hybrid Test Acceleration. In a related research direction, a hybrid approach could be developed, in which the test for some functions are executed on the GPU, while others are simultaneously executed on a multi-core CPU. This could be particularly useful as a way to address challenges related to the limited support for C features. In large programs consisting of multiple functions, those which use unsupported features could be tested on the CPU, while others are simultaneously tested on the GPU. In addition, partitioning test execution in this way could also provide a way to support larger applications, addressing the scalability challenge.

Using Other Heterogeneous Architectures

Modern computer hardware is increasingly heterogeneous, parallel and aimed at acceleration. OpenCL is an open standard that offers functional portability across architectures. Based on the OpenCL programming model, the tools implemented in this thesis could be extended and used to explore the applicability of other types of heterogeneous architectures to the task of accelerating software test execution.

These could include GPU architectures with different hardware and software capabilities, such as integrated graphics cards and unified memory GPUs. Integrated graphics cards are GPUs that are built into the processor and share main memory with the CPU. A common example are the Intel Graphics processors [180], which are widely available in laptops and desktops. While they are typically less powerful than a dedicated GPU, using them does not require moving data before and after GPU computation. This could have significant performance advantages for parallel test execution, which uses data in the form of large test suites, and could compensate for slowdowns in kernel performance. Unified memory GPUs [181] have a single unified virtual address space for the CPU and GPU memory. Its purpose is to simplify GPU programming by providing a single pointer to the allocated memory, which can be accessed by both the CPU and GPU. The

system software then transparently and efficiently transfers the data between the CPU and GPU memories when it is needed, making use of data locality and memory page faulting for automatic synchronisations.

Assessing the performance that can be achieved for test execution on these types of GPU architectures would be a useful addition to the research in this thesis.

Beyond C Source Code

The emergence of new GPU programming models, frameworks and compilers opens up the possibility of using the GPU to accelerate testing for programming languages beyond C. A promising option is SYCL [19] - a programming model which allows using the C++ programming language to write GPU programs. It provides support for standard C++ features and libraries and allows the programmers to write the host and kernel applications together in a single source file. The SYCL compiler then cross-compiles the source code for the CPU and GPU, using OpenCL as an intermediate language for the kernel. In future work, SYCL's capabilities could be explored to assess if C++ programs, or parts of them, can be translated into OpenCL kernels to be used by ParTeCL Runtime for parallel test execution on the GPU.

Integration with Existing Testing Frameworks

Extending the ParTeCL tool-chain to integrate test suites that are implemented in existing testing frameworks, such as GoogleTest [29], is an important part of future work. It will help future empirical evaluations that are more representative of real-world testing scenarios. It will also allow the approach to be adopted with little effort into existing automated test environments.

8.4 Concluding Remarks

Software testing is a crucial but costly and time-consuming part of the software development process. This thesis proposes a novel approach to speeding up expensive test executions by leveraging the high degree of parallelism available in GPU architectures.

Performance results are promising but there is much work to be done to enable the adoption of this approach in real-world software testing. The main barriers are still the limited support for certain program features and limited GPU memory. As GPU architectures and programming models continue to evolve, challenges related to these

limitations may not pose the same restrictions in the future. Furthermore, with the emergence of new heterogeneous architectures, related and hybrid approaches could also prove successful in addressing some of these issues.

Despite its challenges, or perhaps because of them, heterogeneous computing is a fast evolving and exciting area. It is the author's hope that the work presented in this thesis will inspire and serve as a basis for future research in the possibilities that heterogeneous parallelism could bring to software testing.

Bibliography

[1] Mary Jean Harrold. Testing: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000.

[2] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[3] Wilson Bissi, Adolfo Gustavo Serra Seca Neto, and Maria Claudia Figueiredo Pereira Emer. The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, 74:45–54, 2016. doi:10.1016/j.infsof.2016.02.004.

[4] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437, 2016. doi:10.1145/2970276.2970358.

[5] M. Shahin, M. Ali Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017. doi:10.1109/ACCESS.2017.2685629.

[6] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100, 2019. doi:10.1109/ICSE-SEIP.2019.00018.

[7] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242. IEEE, 2017. doi:10.1109/ICSE-SEIP.2017.16.

[8] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *STVR*, 22(2):67–120, 2012. doi:10.1002/stvr.430.

[9] Deepak Garg and Amitava Datta. Parallel execution of prioritized test cases for regression testing of web applications. In *Proceedings of the Thirty-Sixth Australasian Computer Science Conference-Volume 135*, pages 61–68, 2013.

[10] Yanbing Yu, James Jones, and Mary Jean Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 201–210. IEEE, 2008.

[11] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.

[12] J. Candido, L. Melo, and M. d'Amorim. Test suite parallelization in open-source projects: A study on its usage and impact. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 838–848, 2017. doi:10.1109/ASE.2017.8115695.

[13] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 325–336, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1142473.1142511.

[14] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, 2009. doi:10.1109/IPDPS.2009.5160984.

[15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng.

TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2016. arXiv:1603.04467.

[16] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans. Math. Softw.*, 43(4), January 2017. doi:10.1145/3017994.

[17] Khronos Group. OpenCL - open standard for parallel programming of heterogeneous systems, 2020. https://www.khronos.org/opencl/ (visited on 19/05/2020).

[18] NVidia. CUDA programming guide, 2020. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 19/05/2020).

[19] Khronos Group. SYCL - C++ single-source heterogeneous programming for acceleration offload, 2020. https://www.khronos.org/sycl/ (visited on 19/05/2020).

[20] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, page 195–204, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1356058.1356084.

[21] C. Nugteren and V. Codreanu. CLTune: A generic auto-tuner for OpenCL kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 195–202, 2015. doi:10.1109/MCSoC.2015.10.

[22] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. Autotuning OpenCL workgroup size for stencil patterns, 2015. arXiv:1511.02490.

[23] Vanya Yaneva, Ajitha Rajan, and Christophe Dubach. ParTeCL: parallel testing using OpenCL. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 384–387, New York, NY, USA, 2017. ACM. doi:10.1145/3092703.3098227.

[24] Vanya Yaneva, Ajitha Rajan, and Christophe Dubach. Compiler-assisted test acceleration on GPUs for embedded software. In *Proceedings of the 26th ACM*

SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, pages 35–45, New York, NY, USA, 2017. ACM. doi:10.1145/3092703.3092720.

[25] V. Yaneva, A. Kapoor, A. Rajan, and C. Dubach. Accelerated finite state machine test execution using GPUs. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 109–118, 2018.

[26] V. Yaneva, A. Rajan, and C. Dubach. GPU acceleration of FSM test execution: Improving scale and performance. In *Journal of Software Testing, Verification & Reliability (under review)*, 2020.

[27] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5):18–29, 2009. doi:10.1109/MM.2009.74.

[28] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley India Pvt. Limited, 2008. URL: https://books.google.co.uk/books?id=7x10CgAAQBAJ.

[29] Google. GoogleTest - Google testing and mocking framework, 2020. https://github.com/google/googletest (visited on 07/09/2020).

[30] Khronos Group. The OpenCL specification, version 1.2, 2012. https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf (visited on 10/09/2020).

[31] Chris Hobbs. *Embedded software development for safety-critical systems*. CRC Press, Boca Raton, FL, 2016.

[32] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4), 2009.

[33] Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. Chapter three - on testing embedded software. volume 101 of *Advances in Computers*, pages 121 – 153. Elsevier, 2016. doi:10.1016/bs.adcom.2015.11.005.

[34] Peter Liggesmeyer and Mario Trapp. Trends in embedded software engineering. *IEEE software*, 26(3), 2009. doi:10.1109/MS.2009.80.

[35] Mathworks. Simulink, 2020. https://uk.mathworks.com/products/simulink.html (visited on 19/05/2020).

[36] IBM. Rational Rhapsody, 2020. https://www.ibm.com/products/systems-design-rhapsody (visited on 19/05/2020).

[37] Sparx Systems. Enterprise Architect, 2020. http://www.sparxsystems.com/ (visited on 19/05/2020).

[38] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, Aug 1996. doi:10.1109/5.533956.

[39] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[40] Keysight Technologies. Company website, 2020. https://www.keysight.com/ (visited on 19/05/2020).

[41] A.R. Lehane, A.J.A. Kirkham, and L.A. Barford. Digital triggering using finite state machines, 2016. US Patent App. 14/957,491. URL: https://www.google.ch/patents/US20160085223.

[42] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. In *Proceedings. Conference on Software Maintenance 1990*, pages 302–310, 1990.

[43] Jeff Offutt, Jie Pan, and Jeffrey M Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the 12th International Conference on Testing Computer Software*, pages 111–123. ACM Press New York, 1995.

[44] Tsong Yueh Chen and Man Fai Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, 1996. doi:10.1016/S0020-0190(96)00135-4.

[45] Martina Marré and Antonia Bertolino. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, 29(11):974–984, 2003.

[46] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. *ACM SIGSOFT Software Engineering Notes*, 31(1):35–42, 2005.

[47] Dennis Jeffrey and Neelam Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on software Engineering*, 33(2):108–123, 2007.

[48] K. L. Fischer. A test case selection method for the validation of software maintenance modifications. 1977.

[49] Mary Jean Harrold and ML Souffa. An incremental approach to unit testing during maintenance. In *1988 Conference on Software Maintenance*, pages 362–367. IEEE Computer Society, 1988.

[50] A-B Taha, Stephen M Thebaut, and S-S Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *[1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference*, pages 527–534. IEEE, 1989.

[51] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *ICSM*, volume 92, pages 299–308. Citeseer, 1992.

[52] Gregg Rothermel and Mary Jean Harrold. A safe, efficient algorithm for regression test selection. In *1993 Conference on Software Maintenance*, pages 358–367. IEEE, 1993.

[53] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2):77–109, 2000.

[54] Mary Jean Harrold, James A Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S Alexander Spoon, and Ashish Gujarathi. Regression test selection for Java software. *ACM Sigplan Notices*, 36(11):312–326, 2001.

[55] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. *ACM SIGSOFT Software Engineering Notes*, 29(6):241–251, 2004.

[56] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. Regression test selection techniques: A survey. *Informatica*, 35(3), 2011.

[57] Rafaqut Kazmi, Dayang N. A. Jawawi, Radziah Mohamad, and Imran Ghani. Effective regression test case selection: A systematic literature review. *ACM Comput. Surv.*, 50(2), May 2017. doi:10.1145/3057269.

[58] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, page 140–150, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1273463.1273483.

[59] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings The Eighth International Symposium on Software Reliability Engineering*, pages 264–274, 1997.

[60] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, 33(4):225–237, 2007.

[61] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. Time-aware test-case prioritization using integer linear programming. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 213–224, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1572272.1572297.

[62] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. Coverage-based test case prioritisation: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 302–311. IEEE, 2013.

[63] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 179–188, 1999. doi:10.1109/ICSM.1999.792604.

[64] Yuen Tak Yu and Man Fai Lau. Fault-based test suite prioritization for specification-based testing. *Information and Software Technology*, 54(2):179–202, 2012.

[65] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, TH Tse, and Tsong Yueh Chen. How well does test case prioritization integrate with statistical fault localization? *Information and Software Technology*, 54(7):739–758, 2012.

[66] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189. IEEE, 2013.

[67] R Krishnamoorthi and SA Sahaaya Arul Mary. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information and Software Technology*, 51(4):799–808, 2009.

[68] Miso Yoon, Eunyoung Lee, Mikyoung Song, Byoungju Choi, et al. A test case prioritization through correlation of requirement and risk. *Journal of Software Engineering and Applications*, 5(10):823, 2012.

[69] Hema Srikanth, Charitha Hettiarachchi, and Hyunsook Do. Requirements based test prioritization using risk factors: An industrial study. *Information and Software Technology*, 69:71–83, 2016.

[70] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74 – 93, 2018. doi:https://doi.org/10.1016/j.infsof.2017.08.014.

[71] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 235–245, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2635868.2635910.

[72] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 483–493. IEEE Press, 2015. doi:10.1109/ICSE.2015.66.

[73] W Eric Wong, Joseph R Horgan, Saul London, and Aditya P Mathur. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4):347–369, 1998.

[74] W Eric Wong, Joseph R Horgan, Aditya P Mathur, and Alberto Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. *Journal of Systems and Software*, 48(2):79–89, 1999.

[75] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie 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)*, pages 34–43. IEEE, 1998.

[76] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[77] D. George and M. E. Heimdahl. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings. 19th International Conference on Automated Software Engineering*, pages 176–185, Los Alamitos, CA, USA, sep 2004. IEEE Computer Society. doi:10.1109/ASE.2004.10057.

[78] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 435–445, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2568225.2568271.

[79] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 57–68, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1572272.1572280.

[80] Colin JW Kushneryk and Paul D Barnett. Parallel test execution, April 26 2012. US Patent App. 12/911,739.

[81] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. Parallel test generation and execution with Korat. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 135–144, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1287624.1287645.

[82] Ritesh K Gupta, Santosh K Janumahanthi, MVGV Nagesh, Venkata R Somisetty, Praveen Thota, and Vikram K Vb. End to end testing automation and parallel test execution, May 12 2015. US Patent 9,032,373.

[83] T. Parveen and S. Tilley. When to migrate software testing to the cloud? In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 424–427, 2010.

[84] Scott Tilley and Tauhida Parveen. HadoopUnit: test execution in the cloud. In *Software Testing in the Cloud*, SpringerBriefs in Computer Science, pages 37–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012 edition, 2012.

[85] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. 2004.

[86] Apache Software Foundation. Hadoop, 2020. `https://hadoop.apache.org` (visited on 31/08/2020).

[87] Lian Yu, Wei-Tek Tsai, Xiangji Chen, Linqing Liu, Yan Zhao, Liangjie Tang, and Wei Zhao. Testing as a service over cloud. In *2010 Fifth IEEE International Symposium on Service Oriented System Engineering*, pages 181–188. Ieee, 2010.

[88] Alessio Gambi, Sebastian Kappler, Johannes Lampel, and Andreas Zeller. CUT: automatic unit testing in the cloud. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 364–367, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3092703.3098222`.

[89] Shin Yoo, Mark Harman, and Shmuel Ur. Highly scalable multi objective test suite minimisation using graphics cards. In *Proceedings of the Third International Conference on Search Based Software Engineering*, SSBSE'11, page 219–236, Berlin, Heidelberg, 2011. Springer-Verlag. `doi:10.1007/978-3-642-23716-4_20`.

[90] Shin Yoo, Mark Harman, and Shmuel Ur. GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards. *Empirical Software Engineering*, 18(3):550–593, 2013. `doi:10.1007/s10664-013-9247-y`.

[91] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and software Technology*, 43(14):833–839, 2001. doi:10.1016/S0950-5849(01)00189-6.

[92] Zheng Li, Yi Bian, Ruilian Zhao, and Jun Cheng. A fine-grained parallel multi-objective test case prioritization on GPU. In *International Symposium on Search Based Software Engineering*, pages 111–125. Springer, 2013. doi:10.1007/978-3-642-39742-4_10.

[93] Ahmet Celik, Sreepathi Pai, Sarfraz Khurshid, and Milos Gligoric. Bounded exhaustive test-input generation on GPUs. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 2017. doi:10.1145/3133918.

[94] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. *ACM SIGSOFT Software Engineering Notes*, 27(4):123–133, 2002. doi:10.1145/566171.566191.

[95] Ajitha Rajan, Subodh Sharma, Peter Schrammel, and Daniel Kroening. Accelerated test execution using GPUs. In *ACM/IEEE ASE'14*, pages 97–102, 2014.

[96] V. Garousi, M. Felderer, Ç. M. Karapıçak, and U. Yılmaz. What we know about testing embedded software. *IEEE Software*, 35(4):62–69, 2018. doi:10.1109/MS.2018.2801541.

[97] Mirko Conrad, Ines Fey, and Sadegh Sadeghipour. Systematic model-based testing of embedded automotive software. *Electronic Notes in Theoretical Computer Science*, 111:13–26, 2005.

[98] F. Böhr. Model-based statistical testing of embedded systems. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 18–25, 2011. doi:10.1109/ICSTW.2011.11.

[99] H. Lei and Y. Wang. A model-driven testing framework based on requirement for embedded software. In *2016 11th International Conference on Reliability, Maintainability and Safety (ICRMS)*, pages 1–6, 2016. doi:10.1109/ICRMS.2016.8050100.

[100] C. P. Vudatha, S. Nalliboena, S. K. Jammalamadaka, B. K. K. Duvvuri, and L. S. S. Reddy. Automated generation of test cases from output domain and

critical regions of embedded systems using genetic algorithms. In *2011 2nd National Conference on Emerging Trends and Applications in Computer Science*, pages 1–6, 2011. doi:10.1109/NCETACS.2011.5751411.

[101] Chengyu Zhang, Yichen Yan, Hanru Zhou, Yinbo Yao, Ke Wu, Ting Su, Weikai Miao, and Geguang Pu. Smartunit: Empirical evaluations for automated unit testing of embedded software in industry. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 296–305, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3183519.3183554.

[102] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. A model-based regression test selection approach for embedded applications. *SIGSOFT Softw. Eng. Notes*, 34(4):1–9, July 2009. doi:10.1145/1543405.1543413.

[103] S. Biswas, R. Mall, and M. Satpathy. Task dependency analysis for regression test selection of embedded programs. *IEEE Embedded Systems Letters*, 3(4):117–120, 2011. doi:10.1109/LES.2011.2173293.

[104] Swarnendu Biswas, Rajib Mall, and Manoranjan Satpathy. A regression test selection technique for embedded software. *ACM Trans. Embed. Comput. Syst.*, 13(3), December 2013. doi:10.1145/2539036.2539043.

[105] Matthew H. Netkow and Dennis Brylow. Xest: An automated framework for regression testing of embedded software. In *Proceedings of the 2010 Workshop on Embedded Systems Education*, WESE '10, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1930277.1930284.

[106] Patrick Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics*, pages 138–156. Springer, 2001. doi:10.1007/3-540-44577-3_10.

[107] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. volume 2566, pages 85–108, 2002. doi:10.1007/3-540-36377-7_5.

[108] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. *SIGPLAN Not.*, 38(5):196–207, May 2003. doi:10.1145/780822.781153.

[109] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, April 2006. doi:10.1145/1218063.1217943.

[110] Clang Static Analyser website, 2021. https://clang-analyzer.llvm.org/ (visited on 04/04/2021).

[111] Facebook Infer website, 2021. https://fbinfer.com/ (visited on 04/04/2021).

[112] Coverity website, 2021. https://scan.coverity.com/ (visited on 04/04/2021).

[113] GrammaTech website, 2021. https://www.grammatech.com/ (visited on 04/04/2021).

[114] Ajitha Rajan. *Coverage metrics for requirements-based testing*. PhD thesis, University of Minnesota, 2009.

[115] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986. doi:10.1145/5397.5399.

[116] G. Semmel and G. H. Walton. Developing and validating thousands of executable finite state machines. In *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, volume 6, pages 2837–2848 vol.6, 2001. doi:10.1109/AERO.2001.931304.

[117] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, page 25–36, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1146238.1146242.

[118] A. Rajan, M. W. Whalen, and M. P. E. Heimdahl. Model validation using automatically generated requirements-based tests. In *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, pages 95–104, 2007. doi:10.1109/HASE.2007.57.

[119] Edward F Moore et al. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[120] FC Hennine. Fault detecting experiments for sequential circuits. In *1964 Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110. IEEE, 1964.

[121] D. Lee and M. Yannakakis. Testing finite-state machines: state identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994.

[122] Robert M. Hierons and Uraz Cengiz Türker. Incomplete distinguishing sequences for finite state machines. *The Computer Journal*, 58(11):3089–3113, 06 2015. doi:10.1093/comjnl/bxv041.

[123] Robert M. Hierons and Uraz Cengiz Türker. Distinguishing sequences for distributed testing: Adaptive distinguishing sequences. *The Computer Journal*, 59(8):1186–1206, 08 2016. doi:10.1093/comjnl/bxw004.

[124] Robert M. Hierons and Uraz Cengiz Türker. Distinguishing sequences for distributed testing: Preset distinguishing sequences. *The Computer Journal*, 60(1):110–125, 01 2017. doi:10.1093/comjnl/bxw069.

[125] Kshirasagar Naik. Efficient computation of unique input/output sequences in finite state machines. *IEEE/ACM transactions on networking*, 5(4):585–599, 1997.

[126] Qiang Guo, Robert M. Hierons, Mark Harman, and Karnig Derderian. Computing unique input/output sequences using genetic algorithms. In *In Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES'2003), volume 2931 of LNCS*, pages 169–184. Springer, 2004.

[127] Karnig Derderian, Robert M Hierons, Mark Harman, and Qiang Guo. Automated unique input output sequence generation for conformance testing of FSMs. *The Computer Journal*, 49(3):331–344, 2006.

[128] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering*, (3):178–187, 1978. doi:10.1109/TSE.1978.231496.

[129] S. Fujiwara, G. v Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 06 1991. doi:10.1109/32.87284.

[130] Gang Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, 1994.

[131] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. Timed Wp-method: Testing real-time systems. *IEEE Transactions on Software Engineering*, 28(11):1023–1038, 11 2002. doi:10.1109/TSE.2002.1049402.

[132] H. Ural, Xiaolin Wu, and Fan Zhang. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(1):93–99, Jan 1997. doi:10.1109/12.559807.

[133] R. M. Hierons and H. Ural. Reduced length checking sequences. *IEEE Transactions on Computers*, 51(9):1111–1117, Sep 2002. doi:10.1109/TC.2002.1032630.

[134] R. M. Hierons and H. Ural. Optimizing the length of checking sequences. *IEEE Transactions on Computers*, 55(5):618–629, May 2006. doi:10.1109/TC.2006.80.

[135] Guy-Vincent Jourdan, Hasan Ural, Hüsnü Yenigün, and Ji Chao Zhang. Lower bounds on lengths of checking sequences. *Formal Aspects of Computing*, 22(6):667–679, Nov 2010. doi:10.1007/s00165-009-0135-6.

[136] A. J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, ICECCS '99, pages 119–129, Washington, DC, USA, 1999. IEEE Computer Society. doi:10.1109/ICECCS.1999.802856.

[137] Robert V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[138] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on Statechart. In *Proceedings. 26th International Conference on Software Engineering*, pages 86–95, May 2004. doi:10.1109/ICSE.2004.1317431.

[139] R. M. Hierons and U. C. Türker. Parallel algorithms for testing finite state machines: Generating UIO sequences. *IEEE Transactions on Software Engineering*, 42(11):1077–1091, Nov 2016. doi:10.1109/TSE.2016.2539964.

[140] K. El-Fakih, R. M. Hierons, and U. C. Turker. $\mathcal{K}$-branching UIO sequences for partially specified observable non-deterministic FSMs. *IEEE Transactions on Software Engineering*, pages 1–1, 2019. doi:10.1109/TSE.2019.2911076.

[141] R. M. Hierons and U. C. Türker. Parallel algorithms for generating harmonised state identifiers and characterising sets. *IEEE Transactions on Computers*, 65(11):3370–3383, Nov 2016. doi:10.1109/TC.2016.2532869.

[142] R. M. Hierons and U. C. Türker. Parallel algorithms for generating distinguishing sequences for observable non-deterministic FSMs. *ACM Trans. Softw. Eng. Methodol.*, 26(1):5:1–5:34, July 2017. doi:10.1145/3051121.

[143] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967. doi:10.1145/321406.321418.

[144] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4):1250010, dec 2012. doi:10.1142/S0129626412500107.

[145] Tobias Grosser and Torsten Hoefler. Polly-ACC transparent compilation to heterogeneous hardware. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2925426.2926286.

[146] Muthu Manikandan 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)*, volume 6011 LNCS, pages 244–263. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-11970-5_14.

[147] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013. doi:10.1145/2400682.2400713.

[148] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019. doi:10.1109/CGO.2019.8661197.

[149] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proceedings of ICFP 2015*, pages 205–217, New York, 2015. ACM. doi:10.1145/2784731.2784754.

[150] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. Performance portable GPU code generation for matrix multiplication. In *GPGPU*, pages 22–31, New York, USA, 2016. ACM Press. doi:10.1145/2884045.2884046.

[151] M. Steuwer, T. Remmelg, and C. Dubach. Lift: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85, 2017.

[152] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 100–112, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3168824.

[153] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. Position-dependent arrays and their application for high performance code generation. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, FHPNC 2019, page 14–26, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3331553.3342614.

[154] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. NOVA: a functional language for data parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, page 8–13, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2627373.2627375.

[155] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[156] Patricia Suriana, Andrew Adams, and Shoaib Kamil. Parallel associative reductions in Halide. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 281–291. IEEE, 2017.

[157] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004. doi:10.1016/j.parco.2003.12.002.

[158] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - a portable skeleton library for high-level GPU programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1176–1182, 2011.

[159] Cedric Nugteren and Henk Corporaal. Introducing Bones: a parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 1–10, New York, NY, USA, 2012. ACM. doi:10.1145/2159430.2159431.

[160] Cedric Nugteren and Henk Corporaal. Bones: An automatic skeleton-based C-to-CUDA compiler for GPUs. *ACM Trans. Archit. Code Optim.*, 11(4):35:1–35:25, December 2014. doi:10.1145/2665079.

[161] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[162] V. Yaneva. ParTeCL-CodeGen, 2020. https://github.com/wyaneva/partecl-codegen (visited on 03/06/2020).

[163] V. Yaneva. ParTeCL-Runtime, 2020. https://github.com/wyaneva/partecl-runtime (visited on 03/06/2020).

[164] Erik Andersen. uClibc website, 2012. https://www.uclibc.org/ (visited on 19/05/2020).

[165] V. Yaneva. clClibc, 2020. https://github.com/wyaneva/clclibc (visited on 04/06/2020).

[166] Yanhong A Liu and Scott D Stoller. From recursion to iteration : what are the optimizations? *Proceedings of PEPM'00: the ACM SIGPLAN 2000 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 73–82, 2000. doi:10.1145/328690.328700.

[167] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Operating Systems Principles (SOSP)*, pages 233–248. ACM, 2011.

[168] Pete Cooper, Uwe Dolinsky, Alastair F Donaldson, Andrew Richards, Colin Riley, and George Russell. Offload–automating code migration to heterogeneous multicore systems. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 337–352. Springer, 2010. doi:10.1007/978-3-642-11515-8_25.

[169] Mehmet E. Belviranli, Farzad Khorasani, Laxmi N. Bhuyan, and Rajiv Gupta. CuMAS: Data transfer aware multi-application scheduling for shared GPUs. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2925426.2926271.

[170] B. Bastem, D. Unat, W. Zhang, A. Almgren, and J. Shalf. Overlapping data transfers with computation on GPU with tiles. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 171–180, 2017. doi:10.1109/ICPP.2017.26.

[171] Jacson Rodrigues Barbosa, ME Delamaro, JC Maldonado, and AMR Vincenzi. Software testing in critical embedded systems: a systematic review of adherence

to the DO-178B standard. In *International Conference on Advances in System Testing and Validation Lifecycle*, pages 126–130, 2011.

[172] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. SIS: a system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/2010.html.

[173] J. Levandovski, E. Sommer, and M. Strait. Application layer packet classifier for Linux, 2009. http://l7-filter.sourceforge.net/ (visited on 19/05/2020).

[174] Y. Fang, A. A. Chien, A. Lehane, and L. Barford. Performance of parallel prefix circuit transition localization of pulsed waveforms. In *2016 IEEE International Instrumentation and Measurement Technology Conference Proceedings*, pages 1–6, May 2016. doi:10.1109/I2MTC.2016.7520365.

[175] W. Estes. Flex: A fast scanner generator, 2020. https://github.com/westes/flex (visited on 19/05/2020).

[176] Snort. Network intrusion detection and prevention system, 2020. http://snort.org/ (visited on 19/05/2020).

[177] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[178] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, K. Petersen, and M. V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42, 2012. doi:10.1109/IWAST.2012.6228988.

[179] Khronos Group. OpenCL 1.2 - clCreateSubDevices(), 2020. https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/clCreateSubDevices.html (visited on 08/09/2020).

[180] Intel. Intel's next generation integrated graphics architecture – Intel graphics media accelerator X3000, whitepaper, 2006. https://www.intel.com/Assets/PDF/whitepaper/313343.pdf (visited on 04/04/2021).

[181] NVidia. NVidia Tesla P100, whitepaper, 2016. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (visited on 04/04/2021).