# Accelerated Finite State Machine Test Execution Using GPUs

Vanya Yaneva
*School of Informatics*
*University of Edinburgh*
Edinburgh, UK
vanya.yaneva@ed.ac.uk

Arnav Kapoor
*International Institute*
*of Information Technology*
Hyderabad, India
arnav.kapoor@research.iiit.ac.in

Ajitha Rajan
*School of Informatics*
*University of Edinburgh*
Edinburgh, UK
arajan@ed.ac.uk

Christophe Dubach
*School of Informatics*
*University of Edinburgh*
Edinburgh, UK
christophe.dubach@ed.ac.uk

***Abstract*—Model-based development has emerged as a popular approach aiding automation of the software development process, where software is implemented and tested based on a model of the required system. Finite State Machines (FSMs) are a widely used model representation for a variety of systems, including control systems, signal processing and communications protocols. Ensuring that the model accurately represents the required behaviour involves the generation and execution of a large number of tests that is time consuming and expensive.**

**In this paper, we focus on test execution and propose exploiting Graphics Processing Units (GPUs) for accelerating FSM testing by executing the tests in parallel on GPU threads. Our approach includes methods to encode the FSM efficiently and optimise the layout of tests in GPU memory for fast execution. We compare speedup achieved by our approach against parallel test execution on a multi-core CPU with 16 cores. We also assess the improvement in speedup using the proposed FSM encoding and test layouts. We use large FSMs from the networking domain and a large industry FSM from Keysight, who provide electronic measurement solutions, in our evaluation. We accelerate the execution of test suites providing all-transition pair coverage for each of the FSMs. Speedup achieved is subject to characteristics of the FSM and associated tests, and is greatly improved with efficient FSM encoding and test layout in memory. We find our approach on the GPU achieves a maximum test execution speedup of $12\times$ over a 16-core CPU.**

***Index Terms*—software testing, finite state machines, gpus**

## I. Introduction

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 [24]. Note that, in this paper, we use the term 'model' to refer to a finite state machine (FSM) that is widely used to model systems in diverse areas, including sequential circuits, control systems and communication protocols [19].

The problem of testing FSMs (for state identification, state verification, conformance testing) has received a lot of attention over the past decades since brute force testing of FSMs with a large number of states and transitions is infeasible. Several techniques for automatically generating and selecting tests have been proposed in the literature. Nevertheless, even after test selection, the number of tests that needs to be executed is very high, requiring time consuming test runs.

### A. Motivating Example

The problem with time consuming FSM test runs was first brought to us by a company - Keysight Technologies [2]. Keysight provide electronic measurement solutions to the wireless communications, aerospace and semiconductor industries. Their systems are modelled using FSMs that get tested extensively. Owing to the large sizes of their FSMs and the numerous tests that need to be executed, test execution is an extremely time-consuming task, often becoming a bottleneck in the development and testing process.

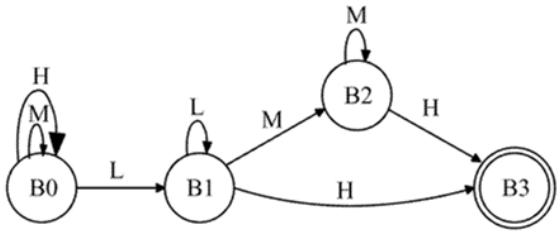


Fig. 1: Keysight example (taken from [20])

Figure 1 shows one of their FSMs, presented in [20]. This FSM is used to identify and trigger particular measurements of interest performed by a digital oscilloscope. 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, Keysight perform black-box tests, similar to those used in functional testing. They execute different 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 scale Keysight FSMs with thousands of states, depending on the input pattern they are designed to identify. Such FSMs require thousands of tests to ensure correctness and take arbitrarily long times for executing all the tests.

### B. Contributions

In this paper, we present automatic execution of functional FSM tests in parallel using graphics processing units (GPUs). We present the following contributions:

- An efficient encoding to represent finite state machines in GPU memory. We accept FSMs in a format that is both intuitive to the user and allows FSMs to be generated into valid OpenCL code. We then efficiently encode the FSM so it fits in GPU memory allowing fast access.
- An OpenCL implementation which executes FSM tests in parallel on the GPU in such a way that expert knowledge of GPU programming is not required.
- Optimisations targeting the memory layout of tests to enhance the achieved speedup on the GPU.
- Evaluation using large FSMs and assessment of the speedup achieved on the GPU when compared to parallel testing on a *multi-core* CPU.

The rest of this paper is organised as follows. Section II discusses relevant background and related work. Section III presents our approach for parallel test execution of FSMs on the GPU and the performance optimisations implemented in our framework. We discuss the research questions and experiment setup for our evaluation in Section IV. Speedup results and the effect of optimisations are presented and analysed in Section V. Section VI discusses the threats to validity of our experiment and finally, Section VII concludes.

## II. Background & Related Work

### A. Finite State Machines

Finite state machines are widely used to model a variety of computer systems, including control circuits, signal processing, communications protocols and pattern matching. They are also the basis of many industry tools, including Simulink [4], IBM Rational Rhapsody [3] and Sparx Systems Enterprise Architect [1]. In this work we consider models based on *Mealy machines* that have a finite number of states and produce outputs on state transitions based on current state and inputs. Lee et al. [19] provide the following definition of an FSM.

*Definition.* A finite state machine $M$ is a quintuple

$$M = (I, O, S, \delta, \lambda) \tag{1}$$

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 $\lambda(s, a)$.

### B. GPU Architecture, Programming & Performance

GPUs are parallel accelerators, designed for graphics computations, which have been successfully employed in many areas of general purpose computing.

*a) Architecture:* GPUs consist of one or more *compute units*, each of which contains one or more *processing elements*, which execute the individual threads. The functions executed by the GPU threads are called *kernels* and each thread runs the same kernel over different input data (Single Instruction Multiple Data programming model). This makes GPUs suited to functional testing, as functional testing consists of running the same tested functionality over multiple test inputs. Our prior work showed that GPUs are well suited to parallelising test suite execution for C programs [25], [28].

*b) Programming:* GPUs require the use of specialist programming models, such as CUDA [22] and OpenCL [12]. Based on the C/C++ programming languages, they expose low-level hardware details, which require the programmer to explicitly express the parallelism in terms of the architecture.

*c) Performance:* GPUs have a memory hierarchy, outlined below. The placement of data during GPU execution can have significant impact on performance.

- **Global memory.** Large and slow, accessed by all threads in all compute units. Performance is greatly improved, when accesses are *coalesced* during execution, i.e. when threads access consecutive addresses in global memory.
- **Constant memory.** A read-only portion of global memory, which contains a special cache allowing much faster memory access.
- **Local memory.** Local to compute units, shared among the threads in a single unit.
- **Private memory.** Private to individual threads.

All threads in a compute unit share the same instruction counter, thus execution on a single compute unit is done in *lock-step* - at each step all threads execute the same instruction. In OpenCL terminology such a group of threads is called a *work-group*. If there is control-flow divergence across threads within the same work-group, divergent instructions will be serialised, negatively impacting performance. Similarly, when the workload is not balanced across the threads in a work-group, this will lead to idle threads, reducing performance.

### C. Related Work

Comprehensive review on testing of finite state machines can be found in [19] and [8]. These surveys focus on *functional testing*, also known as *conformance testing*, which is used to demonstrate that system implementation conforms to an FSM model. The FSM is used to generate test cases for the implementation, by using techniques based on the construction of *distinguishing*, *identifying* or *unique I/O sequences*. While powerful, these techniques have high computational complexity and can lead to exponentially long testing sequences, incurring significant execution costs. As a result, there is a body of research focused on generating *minimised* testing sequences [16]–[18], [27].

A practical alternative often employed in industry is generating test suites based on some type of coverage criteria of the FSM. Popular choices are *all-transition*, *all-transition pair* and *full predicate*, formalised in [23], as well as *transition tree* [5], based on the W-method introduced by Chow [9].

Briand et al. [7] present an empirical investigation into the cost and fault detection effectiveness of the four criteria. They conclude that simply using *all-transition* coverage provides weak fault detection capabilities, while *full predicate* and *transition tree* lead to huge increases in cost. In contrast, *all-transition pair* offers strong fault detection guarantees while maintaining lower costs relative to the other criteria. In this paper we use *all-transition pair* coverage to generate test suites for our experimental setup. We discuss test generation in more detail in Section IV-A.

In recent years, Hierons and Trker have been using GPUs to accelerate the *generation* of testing sequences for FSMs based on unique I/O sequences [14], state harmonised state identifiers and characterising sets [13] and distinguishing sequences [15]. As far as we are aware, there is no existing work in using GPUs to accelerate the *execution* of FSM tests.

## III. Approach

A functional 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.

The executions of individual test sequences are independent of one another, allowing them to be executed in parallel. Our approach executes each test sequence on an individual GPU thread. It involves the following steps:

1. Take the FSM and test suite as inputs.
2. Transfer the FSM and test inputs to the GPU memory.
3. Launch test execution in parallel on the GPU.
4. Transfer test outputs back to CPU memory.

We perform steps 1 and 2 as part of preparation for test execution. The speedup reported in Section V uses time taken for step 3. We also discuss the time taken for transferring the test inputs and results in steps 2 and 4 in Section V-E. The speed with which the test suite executes on the GPU is determined by a number of different factors - the characteristics of the FSM and the test suite, as well as particular design considerations in our test execution framework. In the following subsections we present these design considerations and the optimisations we used to enhance test execution speedup.

### A. Placement of FSM and Tests in GPU Memory

The FSM and tests are placed in the following locations in the GPU memory hierarchy:

- The FSM is needed by all tests, thus it is necessary to be placed in global memory. However, since accesses to global memory are slow and the tests do not modify the FSM, we try to place it in the read-only constant memory when it is small enough to fit. In Section V-B we observe that all of the FSMs used in our experiments fit in constant memory.
- Test inputs and outputs are transferred to/from GPU memory in two arrays, which are shared across threads. Therefore, they are placed in global memory, but each thread reads/writes to different portions of the arrays. In order to mitigate the performance penalty of accessing large test arrays in global memory, we consider different test layouts in Section III-C and evaluate them in Section V-C.

### B. FSM Layout in Memory

Consider the FSM as a two-dimensional matrix, in which each row represents a *state* and each column represents an *input* from the input set. Then each element of the matrix is a tuple (*next state*, *output*), which encodes a transition of the FSM. 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 matrix indicates that there are many state/input pairs for which transitions are not defined, while a dense matrix indicates that there is a transition for most state/input pairs.

We consider two possible layouts for the FSM, which we call *sparse* and *dense*.

*1) Sparse FSM Layout:* 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. Given a state $s$ and an input $a$, we pick the list of triplets corresponding to $s$ in constant time and then perform a search to find the correct triplet based on $a$. Figure 2 illustrates the sparse FSM layout for the motivating example shown in Figure 1.

The sparse FSM layout has the advantage of using memory only for the transitions present in the FSM since it does not use any padding. It could be beneficial for sparse FSM matrices, as it can encode them compactly and allow them to fit into constant GPU memory. The downside of using the sparse FSM layout is the potentially slow execution time - for each input in a test sequence, finding the (next state, output) tuple requires a potentially expensive search for the matching input in the list of triplets for the current state.

| | |
|---|---|
| B0 | (L, B1, 0), (M, B0, 0), (H, B0, 0) |
| B1 | (L, B1, 0), (M, B2, 0), (H, B3, 1) |
| B2 | (M, B2, 0), (H, B3, 1) |

Fig. 2: Sparse FSM layout for the motivating example shown in Figure 1.

*2) Dense FSM Layout:* 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. Given a state $s$ and an input $a$, we pick the corresponding tuple in constant time. Figure 3 illustrates the dense FSM layout for the motivating example shown in Figure 1.

The benefit of using the dense FSM layout is the fast execution time, as for each input in a test sequence, the lookup

for (next state, output) happens in constant time. On the other hand, the drawback is the need to allocate memory for every possible state/input pair in the FSM in the form of padding. This could incur a large overhead in memory space compared to the sparse layout, particularly for sparse FSM matrices.

| | L | M | H |
|---|---|---|---|
| B0 | (B1, 0) | (B0, 0) | (B0, 0) |
| B1 | (B1, 0) | (B2, 0) | (B3, 1) |
| B2 | padding | (B2, 0) | (B3, 1) |
| B3 | padding | padding | padding |

Fig. 3: Dense FSM layout for the motivating example shown in Figure 1, indexed by state and input.

Note that for a dense FSM matrix, the size of the sparse FSM layout would be the same as that for the dense FSM layout or even greater. This is the case because a dense matrix would require memory for the same amount of transitions ($|S| \times |I|$) in both layouts, but in the sparse layout each transition needs to encode the input as well as the next state and output. In the case of a sparse matrix, the benefit of fast memory accesses, provided the sparse layout fits in constant memory, may be offset by the extra time needed to perform a search for each input in the test sequence.

### C. Test Layout in Memory

Each test in the FSM test suite is a sequence of inputs. Generally, input sequences representing tests are of different lengths and their layout in memory affects the GPU execution time. We consider three possible memory layouts, which we call *padded*, *padded-transposed* and *with-offsets*. Note that during execution the same layout is used for both the test inputs and outputs.

*1) Padded Test Layout:* In this layout, tests are laid out in a two-dimensional array, in which each row is a separate test and each column is a test input. All tests are padded with `null` bytes to the length of the longest test sequence. Each GPU thread executes a single row of the two-dimensional array. Test execution stops when the padding `null` byte is encountered. While easy and intuitive to implement, this test layout has high memory requirements and does not allow efficient coalesced memory accesses on the GPU.

*2) Padded-transposed Test Layout:* This layout uses the same two-dimensional array as the padded layout, but transposes it to *allow coalesced memory access*. In this way each column represents a separate test and each row is a test input and the GPU can use coalescing to optimise global memory access for speed.

*3) With-offsets Test Layout:* In this layout, all tests are concatenated into a single array. In order for each thread to know where its test starts, an array of offsets is calculated and used by the threads. While this layout does not allow coalesced memory accesses, it is more compact than the other two, as there is no padding involved, and may utilise the global memory cache more effectively.

### D. Sorting the Test Sequences Based on Length

As outlined in Section II-B, all GPU threads are organised into work-groups, which execute in lock-step (each thread executes 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 need to wait for the longer ones 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.

To achieve work-groups executing tests of similar lengths, a straight forward approach is to sort the test sequences based on their length and launch them on the same work-groups. This should 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.

### E. Implementation

Our approach is implemented for the GPU using the OpenCL programming model. The implementation takes two inputs: (1) an FSM, specified in the *kiss2* format [26] and (2) its test suite specified in a text file, each test sequence on a new line. It then transparently launches the tests in parallel on the GPU threads, requiring no other input or GPU knowledge from the programmer.

We implemented each of the design decisions discussed in this section and evaluated them using the experiments outlined in Section IV. In order to compare GPU execution time to that of a multi-core CPU, we implemented equivalent designs using the C programming language and parallelised test execution on the CPU using OpenMP.

## IV. Experiment

We evaluate the effectiveness of using GPUs to accelerate the execution of FSM tests using FSMs derived from the l7-filter intrusion detection patterns [21], as well as an industry FSM model provided by Keysight [11]. We generate full test suites for each FSM based on the all-transition pair coverage. Our evaluation focuses on GPU execution time and not on data transfer time. We study the effects of the different design considerations outlined in Section III by answering the following research questions:

**Q1. GPU vs multi-core CPU execution.** *What is the GPU performance compared to a 16-core CPU?* For each FSM, we executed its test suite in parallel on the GPU threads using our approach and on a 16-core CPU, then compared their execution time. We performed the experiment using different test suite sizes, ranging from 2048 to the full test suite, in order to assess how the speedup changes as the test suite grows. We present the results achieved by the fastest GPU implementation, as determined in our experiments for Q2, Q3 and Q4 (*dense*

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

TABLE I: Subject FSMs used in our experiments.

FSM layout, *padded-transposed* test layout and *sorting* of the tests before execution).

**Q2. Effect of FSM layout.** *Is the GPU performance dependent on the FSM layout in memory (Sparse vs Dense)?* We measured the time taken by the GPU and 16-core CPU to execute the *full test suites* of all FSMs, using both layouts, and compared the GPU speedups. We used the *padded* test layout for the l7-filter FSMs and both *padded* and *padded-transposed* for the Keysight FSM. We *did not* sort the tests 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)?* We measured the time taken by the GPU and 16-core CPU to execute the *full test suites* of all FSMs, using the three test layouts, and compared the GPU speedups in execution time. We used the *dense* FSM layout and *did not* sort the tests before execution.

**Q4. Effect of sorting the tests.** *How does the GPU performance change when the test sequences are sorted based on length?* To answer this question, we repeated the experiment for Q3, but sorted the tests for each FSM before its execution. We compare the GPU speedup for all three test layouts to assess the effect of sorting tests, based on their length.

### A. Subject FSMs & Tests

To evaluate our approach, we use 12 subject FSMs from the network intrusion detection domain and 1 industry provided FSM from Keysight, used in the signal processing domain. For all FSMs we generate full test suites based on the all-transition pair coverage criteria. Table I provides a summary of the FSMs together with their sizes and numbers of tests.

*1) l7-filter:* 12 subject FSMs are taken from the Linux layer 7 filter (l7-filter) [21] pattern set, which contains multiple regular expressions used in network intrusion detection. The set of inputs comprises of the full set of ASCII characters. We use the Flex tool [10] to convert the patterns into FSMs and a custom Python script to generate a file for each FSM in the kiss2 format [26].

*2) Keysight:* The Keysight FSM represents a model of transition localisation in communication signals [11]. The FSM takes three possible inputs, L, M and H, which correspond to Low, Medium and High voltage pulses and 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 $p$, which defines when a pulse has been sustained long enough to be considered a valid transition and not a glitch in the signal. In our experiments, we used $p = 1000$ as it generates an FSM of similar size ($|S| \times |I|$) to our other subject FSMs.

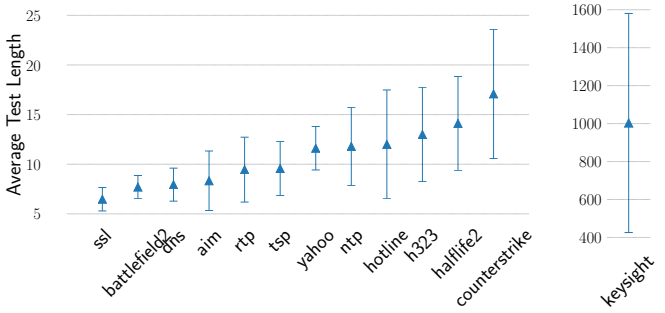


Fig. 4: Average lengths of the tests of the FSMs used in our evaluation. Error bars show standard deviation.

*3) Test Generation:* In order to evaluate out approach using realistic tests, we used the all-transition pair coverage criteria to generate full test suites for each of the FSMs used in our study. 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. All-transition pair has been shown to be a rigorous coverage criterion [7], as it ensures that events in the system are tested not only individually, but also in relation to one another.

To generate the test suites, we implemented the algorithm presented in [6] using Python scripts. Table I contains the size

of the resulting test suites for each FSM and Figure 4 shows the average test lengths for each test suite. We see that the `keysight` FSM has much fewer tests than the rest (36072 vs avg. 2129229 across the l7-filter FSMs), but the average length of its tests is much larger (1000 inputs vs avg. 11). There are two reasons for these differences: (1) `keysight` has a lot more states than the l7-filter FSMs (4004 vs avg. 53) and (2) the l7-filter FSMs have a lot more possible inputs 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. On the other hand, for `keysight` we have the opposite situation - a large number of states, but with a small number of transition pairs going through each of them, requiring fewer but longer test sequences to traverse them.

### B. Hardware & Measurements

In our experiments, we use an NVidia Tesla K40m GPU with 15860 threads, spread across 15 compute units. The GPU operates at 745 MHz and has 12 GB global memory, 64 KB constant memory and 50 KB local memory. For the CPU comparison we use an Intel(R) Xeon(R) CPU E5-2640 v3 processor with 16 cores at 2.60 GHz and 16 GB RAM. All the programs were compiled with GCC with the highest optimization level (`-O3`). To measure GPU execution and data transfer time, we use the profiling functions contained in the OpenCL API. For CPU execution time, we use the standard C function *gettimeofday*. For each experiment we perform 100 runs and report median values.

*a) Multi-core CPU execution:* In order to provide fair comparison between the GPU and a multi-core CPU, we parallelised test execution on the CPU using OpenMP and executed all design configurations (FSM layout, test layout and test sorting) using 16 cores available on our system. Across the paper, we report speedup calculated when compared to the fastest CPU execution times.

*b) Correctness:* For each experiment, we compare the testing outputs produced by the GPU to those from the CPU and confirm that they are an **exact match** to ensure that our GPU optimisations preserve the correctness of test execution.

## V. Results & Analysis

In this section we discuss the results and analysis of the experiments described in Section IV. First, we present the *total GPU execution speedup* when compared to the multi-core CPU in Q1 and then, we assess each individual design choice in Q2, Q3 and Q4.

### A. Q1. GPU Execution Speedup vs Multi-core CPU

Figure 5 shows the speedup achieved in test execution time on the GPU when compared to an optimised parallel implementation on a 16-core CPU. For each FSM we present the GPU speedup for test suite sizes ranging from 2048 up to the maximum number of tests in the test suite. As `keysight` requires significantly fewer tests that the l7-filter FSMs, we padded its test suite to contain $2^{20}$ tests by randomly duplicating its existing tests.

We find that GPU speedup increases as the number of tests in the test suite increases, since the GPU is able to utilise more threads as tests are added. This continues up until the GPU's saturation point after which there are no more GPU threads to be utilised and the speedup remains stable (approx. $2^{18}$ tests). Speedup is observed for test suite sizes larger than $2^{13}$ (approx. 8K), over all FSMs, and the highest speedup is achieved for the largest test suite size. We discuss results for the l7-filter and Keysight FSMs separately.

*1) 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 4 shows the average test lengths of each FSM, as well as the standard deviation across the test suite. We see that 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.

*2) Keysight:* For `keysight`, we observe that the GPU achieves speedup of $7.9\times$ for its full test suite (36027 tests) when compared to a 16-core CPU. This speedup seems lower than expected, considering the much longer test sequences of `keysight`. There are two reasons for this:

1) `keysight` has only 36072 tests in its test suite - not enough to completely utilise the GPU. Figure 5 shows that padding the test suite to $2^{20}$ tests enables the GPU to achieve higher speedup of up to $12.4\times$.
2) As we discussed in Section V-A1, 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 much 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 the GPU execution speedup.

### B. Q2. Effect of FSM Layout

Figure 6 shows a comparison of the GPU speedup achieved using the two FSM layouts - sparse and dense, for the full test suites of each FSM.

As we are only comparing FSM layout, we use only *padded* test layout for the l7-filter FSMs. For `keysight`, the padded

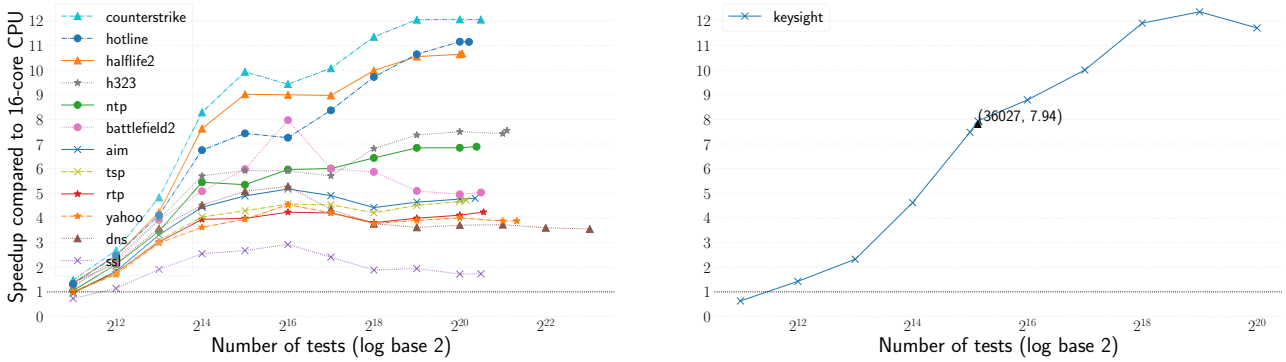


Fig. 5: Speedup in GPU execution time when compared to a 16-core CPU over different test suite sizes. Results presented use the fastest GPU and multi-threaded CPU implementations.

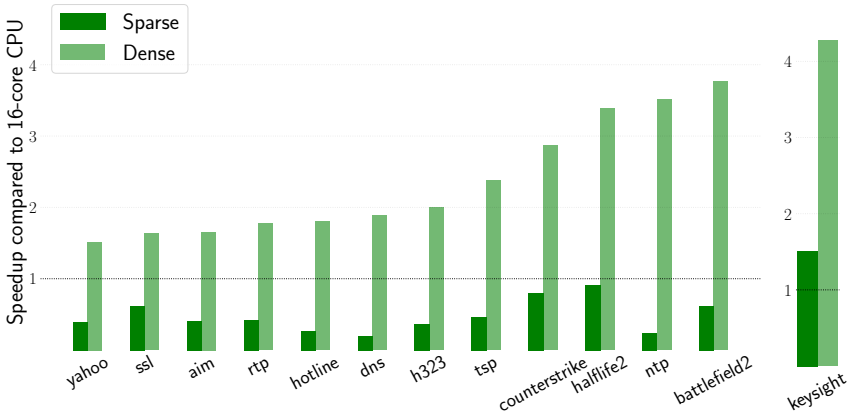


Fig. 6: Difference in GPU execution time speedup (when compared to a 16-core CPU) between the *sparse* and *dense* FSM layouts. The presented values are for the *full test suite* for each FSM. The test layout is *padded* for l7-filter and *padded-transposed* for the Keysight FSM. Tests are *not* sorted before execution.

test layout did not yield a speedup and we also performed an experiment using the *padded-transposed* test layout in order to be able to provide a comparison between sparse and dense FSM layouts. We discuss results for the l7-filter FSMs and `keysight` separately and then provide a common analysis.

*1) l7-filter:* In Figure 6, we observe that for the sparse FSM layout, the speedup for the GPU is consistently lower than 1 - that is, the GPU is slower than than the 16-core CPU. Switching to the dense representation improves the GPU speedup significantly, achieving values between $1.5\times$ and $3.8\times$ across the FSMs.

*2) Keysight:* Figure 6 shows that for `keysight`, when using the padded test representation, the GPU performs worse than the CPU with both the sparse and the dense FSM layouts. This is due to the high inefficiency of the padded test layout for long test sequences. We discuss this in detail in Section V-C2. Owing to the high inefficiency of the test layout, resulting in very slow GPU execution, the FSM layout has little impact on the performance.

Figure 6 also shows that the GPU speedup is significantly improved when switching to padded-transposed test layout, both for the sparse and dense FSM layouts, allowing us to compare them. As with the l7-filter FSMs, the dense layout achieves better GPU speedup - $4.3\times$ vs $1.5\times$ for the sparse.

*3) Analysis:* To understand the reasons for the difference between the two FSM layouts, we consider the characteristics, outlined in Section III-B. When using the dense layout, our approach is able to lookup the next state and output for a given input in constant time, saving considerable execution time for each input in each test sequence when compared to the sparse layout. In addition, in Table I, we observe that the density of our subject FSMs is very high - from $56\%$ for `battlefield2` to $100\%$ for `keysight` with an average of $81\%$. This means that for each FSM, the amount of memory taken by the sparse layout would be close to that of the dense layout, providing no benefit in choosing it. Finally, the large amount of constant memory available on modern GPUs was enough for all of our subject FSMs to fit into it, allowing our approach to take advantage of the fast access to it even with the dense FSM layout.

### C. Q3. Effect of Test Layout

Figure 7 shows a comparison of the GPU speedup achieved using the three test layouts - padded, padded-transposed and with-offsets, for the full test suites of each FSM. We discuss results for the l7-filter FSMs and `keysight` separately.

*1) l7-filter:* In Figure 7, we see that there is a variation in performance across the different FSMs and test layouts. Generally, for the FSMs with shorter test inputs (`yahoo, tsp, rtp` and `tsp`) the highest GPU speedup is achieved by the with-offsets test layout (up to $3\times$ for `tsp`). In contrast, for the FSMs with longer test inputs (`hotline, h323, halflife2` and `counterstrike`, padded-transposed achieves the highest speedup (up to $7.8\times$ for `counterstrike`).

To understand the differences, we consider the two factors which contribute to the efficiency of the test layout - the ability of the GPU to (1) use the global memory cache and (2) to perform coalesced memory accesses. With-offsets provides a more compact test representation, which fits easily into the global memory cache. This is the case in particular for shorter test sequences which explains the better speedup achieved by with-offsets for those FSMs. However, as test sequences grow, the effect of the GPU cache diminishes and the ability to perform coalesced memory access becomes more beneficial. This is provided by the padded-transposed representation and explains the higher speedup achieved with it for the FSMs with longer test suites.

*2) Keysight:* We see in Figure 7 that for the padded and with-offsets test layouts, the GPU has a speedup value less than 1, implying that it is slower than the CPU. In contrast, for the padded-transposed test layout, the GPU speedup improves significantly, reaching a value of $4.2\times$. This inefficiency is due to the long test sequences in `keysight`'s test suite. As seen in Figure 4, the average length of the 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 cache. As the padded and with-offsets layouts do not provide coalesced memory accesses, 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 extremely inefficient GPU computation. This is dramatically improved when we switch to padded-transposed test layout. At every input, each GPU thread performs reads/writes from/to consecutive addresses in memory. Due to these coalesced memory accesses the GPU architecture performs a single efficient memory transaction across a work-group for each memory access, resulting in much better GPU performance.

### D. Q4. Effect of Sorting the Tests

Figure 8 shows the effect on GPU speedup of sorting the tests before execution. We observe improvement of speedup across all FSMs and across all test layouts. For the FSMs with long test inputs (`counterstrike`, `hotline`, `h323`, `ntp` and `keysight`) this improvement is significant, by a factor of approx. 2. This brings the maximum GPU speedup across FSMs to $12\times$ (for `counterstrike`).

This is as expected, based on our discussion in Section III-D. Sorting the tests prior to execution ensures that all threads within a work-group have tests of similar lengths and finish at the same time, immediately freeing resources for another work-group to be scheduled.

Two exceptions are the padded and with-offsets test layouts of `keysight`, which do not improve as we sort the tests. The reason for this is that for `keysight` these test layouts are extremely inefficient on the GPU, leading to worse performance when compared to the 16-core CPU. Sorting the test sequences is not enough to mitigate this effect.

### E. Assessing Data Transfer Time

In this Section we evaluate the time taken to transfer the tests from the CPU to GPU and the time needed to transfer the test execution results back to the CPU. It is well known that data transfer between CPU and GPU is slow due to high latency of the interface. This limitation maybe less of an issue in next generation GPUs that are projected to have larger memory size and bandwidth and in systems with integrated CPU and GPU memory (heterogeneous system architectures).

Table II shows the overhead incurred from data transfer as a fraction of the total GPU time (data transfer and execution time combined). The table shows that for all FSMs data transfer incurs a significant overhead, ranging from $64\%$ (for `ssl`) to $92\%$ (for `keysight`). It is not surprising that `ssl` has the lowest overhead for transferring tests, as it is the FSM with shortest test sequences. Similarly, we expect `keysight` to be the FSM with highest overhead, as it is has the longest tests.

The high degree of data transfer overhead is explained by the fact that our approach focuses on the acceleration of the GPU execution time. This results in the execution time becoming only a small proportion of the total time on the GPU with data transfer becoming the dominant factor.

*Optimisation.* It is possible to mitigate the effect of high transfer time using pipelining to overlap GPU execution and data movement. Large test suites can be split into several smaller groups of tests. While one group of tests executes on the GPU, we can safely start the data transfer for the next group of tests and keep feeding the GPU enough data to process so as to maximize its utilization.

Table II shows the reduction in overhead achievable by overlapping the data with kernel execution on the GPU. As most of the total time on the GPU is spent in data transfer, overlapping data transfer always leads to better overall performance on the GPU, reducing the overhead to values in the range of $9\%$ (for `ssl`) to $56\%$ (for `keysight`).

Optimisation through pipelining would help reduce the time needed for data transfer. It is, however, worth noting that transferring the test suite and FSM to the GPU can be done prior to commencing test execution and the overhead need not be incurred as part of execution.

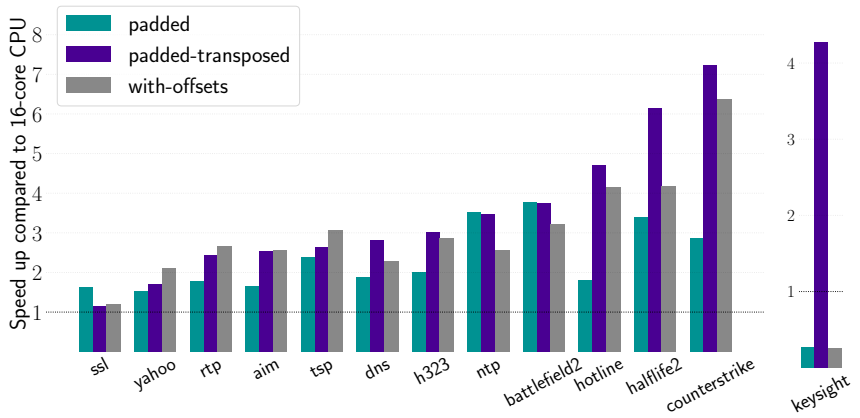


Fig. 7: Difference in GPU execution time speedup (when compared to a 16-core CPU) between the *padded*, *padded-transposed* and *with-offsets* test layouts. The presented values are for the *full test suite* for each FSM. The FSM layout is *dense* and tests are *not* sorted before execution.


Fig. 8: Difference in GPU execution time speedup (when compared to a 16-core CPU) between the sorted and unsorted test suites, for each of the test layouts. The presented values are for the *full test suite* for each FSM. The FSM layout is *dense*.

## VI. Threats to Validity

We see two threats to the validity of our experiment based on the FSMs and test generation technique used. The first one is that we use FSMs from one particular domain in our study (intrustion detection patterns) along with an industrial FSM modeling transition localisation in communication signals. Generalising our results to FSMs in other domains requires further investigation and empirical evaluation. Second, we generate test suites for FSMs based on the all-transition pair coverage criteria. However, there are several other coverage criteria and test generation techniques that will potentially yield different test suites. Impact of test generation techniques on the speedup achieved with our approach requires further investigation. We plan to conduct an extensive evaluation addressing these questions in our future work.

## VII. Conclusions

We presented a novel approach which accelerates the execution of functional tests of finite state machines by running tests in parallel on the GPU threads. We considered different design choices for the encoding of the FSM and its test inputs in order to maximise the speedup achieved on the GPU. We evaluated our approach using 13 subject FSMs from the network intrusion detection and signal processing domains. For each of the 13 subject FSMs we generate full test suites based on the all-transition pair coverage criteria. Our study makes the following findings:

- Our approach achieved a maximum speedup of $12\times$ when compared to a 16-core CPU.
- GPUs generally achieve higher speedups for FSMs with long test sequences.
- Using a dense FSM layout results in higher GPU speedup, particularly for FSMs represented as dense matrices.
- When considering test layout, with-offsets tends to perform better for FSMs with shorter test sequences, while padded-transposed is more suitable to FSMs with long sequences.
- Sorting the test sequences before execution on the GPU

| FSM | Overhead without pipeline | Overhead with pipeline |
|---|---|---|
| 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-source | 90% | 54% |
| keysight | 92% | 56% |

TABLE II: Data transfer overhead as a % of the total time taken by the GPU.

improves the achieved speedup.

To generalise these findings, further study should extend this approach to FSMs from other industry domains, using different test generation criteria.

## ACKNOWLEDGMENT

We would like to thank Keysight for providing us with the case study used in this paper and for their help in understanding its finite state machine. This work was supported by grant $EP/L01503X/1$ for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism http://pervasiveparallelism.inf.ed.ac.uk/ from the UK Engineering and Physical Sciences Research Council (EPSRC).

## REFERENCES

[1] Enterprise architect, sparx systems. http://www.sparxsystems.com/. Accessed: 2017-12-03.

[2] Keysight technologies, company website. https://www.keysight.com/. Accessed: 2017-12-03.

[3] Rational rhapsody, ibm. http://www-03.ibm.com/software/products/en/ratirhapfami. Accessed: 2017-12-03.

[4] Simulink, mathworks. https://uk.mathworks.com/products/simulink.html. Accessed: 2017-12-03.

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

[6] R Blanco, J. G. Fanjul, and Tuya J. Test case generation for transition-pair coverage using scatter search. *International Journal of Software Engineering and It's Applications*, 4(4), 2010.

[7] 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.

[8] 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.

[9] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[10] W. Estes. Flex: A fast scanner generator. https://github.com/westes/flex.

[11] 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.

[12] The Khronos Group. Opencl, 2018.

[13] R. M. Hierons and U. C. Trker. Parallel algorithms for generating harmonised state identifiers and characterising sets. *IEEE Transactions on Computers*, 65(11):3370–3383, Nov 2016.

[14] R. M. Hierons and U. C. Trker. Parallel algorithms for testing finite state machines:generating uio sequences. *IEEE Transactions on Software Engineering*, 42(11):1077–1091, Nov 2016.

[15] R. M. Hierons and U. C. Trker. Parallel algorithms for generating distinguishing sequences for observable non-deterministic fsms. *ACM Trans. Softw. Eng. Methodol.*, 26(1):5:1–5:34, July 2017.

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

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

[18] 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.

[19] 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.

[20] A.R. Lehane, A.J.A. Kirkham, and L.A. Barford. Digital triggering using finite state machines, March 24 2016. US Patent App. 14/957,491.

[21] J. Levandovski, E. Sommer, and M. Strait. Application layer packet classifier for linux. http://l7-filter.sourceforge.net/.

[22] NVidia. Cuda programming guide, 2018.

[23] 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–, Washington, DC, USA, 1999. IEEE Computer Society.

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

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

[26] 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.

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

[28] 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*, pages 35–45. ACM, 2017.