ParTeCL: Parallel Testing Using OpenCL

ParTeCL: Parallel Testing Using OpenCL*

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

Ajitha Rajan
University of Edinburgh, UK
arajan@staffmail.ed.ac.uk

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

ABSTRACT

With the growing complexity of software, the number of test cases needed for effective validation is extremely large. Executing these large test suites is expensive and time consuming, putting an enormous pressure on the software development cycle. In previous work, we proposed using Graphics Processing Units (GPUs) to accelerate test execution by running test cases in parallel on the GPU threads. However, the complexity of GPU programming poses challenges to the usability and effectiveness of the proposed approach.

In this paper we present ParTeCL - a compiler-assisted framework to automatically generate GPU code from sequential programs and execute their tests in parallel on the GPU. We show feasibilitiy and performance achieved when executing test suites for 9 programs from an industry standard benchmark suite on the GPU. ParTeCL achieves an average speedup of 16× when compared to a single CPU for these benchmarks.

CCS CONCEPTS

•Software and its engineering →Software testing and debugging; Source code generation; •Computer systems organization →*Embedded software;*

KEYWORDS

Functional testing, GPUs, Embedded software, Compilers, Automated testing

ACM Reference format:
Vanya Yaneva, Ajitha Rajan, and Christophe Dubach. 2017. ParTeCL: Parallel Testing Using OpenCL. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17-DEMOS),* 4 pages.
DOI: 10.1145/3092703.3098227

1 INTRODUCTION

Testing to ensure the software meets its requirements is a notoriously hard and time consuming process, often representing 50% of the cost of software development [2, 6]. As the scale and complexity of software increases, the number of tests needed for effective validation becomes extremely large, slowing down development and hindering programmer productivity with time consuming test runs.

To combat this problem, industry is moving towards distributing test execution among multiple machines, executing them concurrently to reduce execution time of the entire test suite. This approach, however, is costly in terms of resources, infrastructure, maintenance and energy consumed. Present day commodity parallel accelerators, such as Graphics Processing Units (GPUs), offer enormous computing power while also being cheap, easily available and energy efficient. A single GPU offers thousands of parallel threads with the potential to execute a large number of test cases concurrently. However, GPUs are notoriously hard to program and require significant expertise and a thorough understanding of the hardware and programming model to unlock their potential.

We plan to address these problems in the context of *test execution* using our *ParTeCL* framework. ParTeCL has the following **goals**,

(1) Increase the ***usability and feasibility*** of GPUs for test execution.
(2) Increase the ***performance and effectiveness*** with compiler optimisations that analyse the tests and the program.

Our recently accepted paper [10] in the main research track of ISSTA 2017 presents empirical evaluations of our approach and discusses the performance and effectiveness of using GPUs for test execution. In this paper, we tackle the feasibility and ease of use challenge by designing a framework that allows test cases to be *automatically* launched on the GPU without requiring any GPU programming knowledge and improving supported program features.

Users. ParTeCL's envisioned users are software engineers and testers of systems with large numbers of functional tests. ParTeCL does not require software engineers to have GPU programming knowledge.

2 RELATED WORK

There has been no work in the past examining the use of GPUs to accelerate test execution and our paper [7] was the first in exploring this possibility. The approach in [7], however, *manually* transforms the program and tests to run on the GPU. This approach is incomplete in tackling GPU limitations with respect to ease of programming, unsupported program features, and performance optimizations.

GPU programming poses many challenges for the developer, both in terms of programmability and performance. The use of low-level programming models, such as CUDA and OpenCL, requires familiarity with the architecture in order to write correct parallel code, and effective optimizations in order to reach the full performance potential of the GPU. Previous research addresses these challenges by proposing high-level programming frameworks, compilers and code generation tools. For instance, [8, 9] introduce and evaluate a framework, which automatically generates low-level OpenCL code from high-level parallel primitives. Another example is SYCL [3], which provides a high level-abstraction of OpenCL to allow programmers to write GPU code in standard C++.

These existing tools and frameworks provide high-level mechanisms to both discover and express parallelism for the GPU. They are, however, not suited for our purposes since the need in our approach is *not* identification of parallelism, as that is inherent in test execution - the test cases are directly mapped to the GPU threads. The need lies in a code generation tool that will take the CPU program and transform it into an OpenCL kernel, without affecting the core program functionality, while also launching it with a different test case on each GPU thread. None of the existing tools can provide this capability. In the next sections, we describe the design and implementation of our framework that addresses these needs.

*This work was supported by grant $EP/L01503X/1$ from EPSRC.

ISSTA'17-DEMOS, Santa Barbara, CA, USA
2017. 978-1-4503-5076-1/17/07...$15.00
DOI: 10.1145/3092703.3098227

3 DESIGN & IMPLEMENTATION

ParTeCL consists of two systems, illustrated in Figure 1.

1) ParTeCL CodeGen - a code generation tool: It translates the tested C program to an OpenCL kernel. It also generates data structures and functions, used by the runtime, to transfer test cases and results between the CPU and GPU memories.

2) ParTeCL Runtime - a test execution system: It uses the OpenCL kernel generated by ParTeCL CodeGen to launch the test cases on the GPU in the following steps: (1) Read the test cases and transfer them to the GPU memory. (2) Build the generated OpenCL kernel and launch it in parallel on the GPU threads. (3) Transfer the testing results back to the CPU for inspection.

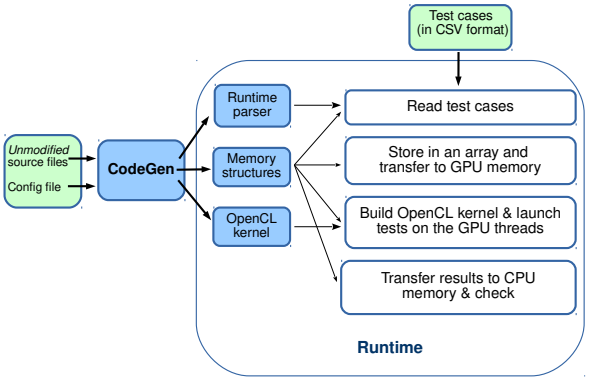

Figure 1: ParTeCL: automated test execution on the GPU.

We describe the above two systems in Section 3.1 using a simple use case program. Section 3.2 presents code transformations supported by ParTeCL, and Section 3.3 discusses the implementation of the two systems.

3.1 Use Case

To help describe ParTeCL, we use a simple C program, shown in Figure 2. The example program reads a string and a character through standard input, `str` and `ch`, and counts the number of occurances of the character in the string. It prints the results to standard output. A test case for this program would consist of values for the string and the character, and the expected result would be a value for the variable `occurs`.

User inputs required by ParTeCL are discussed in Section 3.1.1. We describe the workings of the two systems, ParTeCL CodeGen and ParTeCL Runtime, using the example program, in Sections 3.1.2 and 3.1.3 respectively.

3.1.1 User Inputs. To launch the test cases for the program using ParTeCL, the user needs to supply three inputs: the unmodified source code for the tested program, a configuration file and the test cases.

Configuration File. Describes the test case inputs and results for the tested program. It is used by ParTeCL CodeGen to generate the data structures which are used to transfer test inputs and results between the CPU and GPU memories. Figure 3 presents the configuration file for the example program. The configuration shows that the program takes two inputs, a string and a character, through the standard input, and produces a single integer result, which corresponds to the `occurs` variable. ParTeCL supports the use of both built-in and custom data types, as well as pointers and arrays for the test case inputs and results.

Figure 3 also shows the data structures generated by ParTeCL CodeGen: `struct partecl_input` for the test case inputs and `struct partecl_result` for test case results.

Finally, the configuration is used to generate a parser for the ParTeCL Runtime, which is used to read values for the test cases and assign them to members of the generated memory structures.

Test Cases. ParTeCL assumes that the test cases are provided in a CSV (Comma Separated Value) file where, (1) each row corresponds to a test case, (2) first column contains the id of the test case, (3) subsequent columns contain the input variables, in the order in which they are given in the configuration file.

For the example in Figure 2, a sample CSV file with 5 test cases is presented below,

```
1 <"Tests are important." <"t"
2 <"" <"a"
3 <"bbbbbbbbbbbbbbbbb" <"b"
4 <"0 + 0 = 0" <"0"
5 <"Hello, World!" <"!"
```

where the values for inputs `str` and `ch` for test case 1 are "Tests are important." and "t" respectively, for test case 2 they are an empty string and "a". The '<' symbol denotes that both these inputs are read through standard input in the program being tested. ParTeCL Runtime also supports custom data structures and arrays for the test case inputs. Within the CSV, users can also choose to provide test case data in separate files.

3.1.2 ParTeCL CodeGen. In addition to using the configuration to generate the memory structures and test case parser, ParTeCL CodeGen also generates an OpenCL kernel which executes the tested program on the GPU.

Figure 2 shows the kernel generated for the example program. ParTeCL CodeGen changes the signature to the main function, which now takes two arguments, (1) the test inputs; values for which are initialised by the CPU, and (2) the test results, which will be calculated by the kernel. It uses the memory structures `partecl_input` and `partecl_result`, generated by ParTeCL CodeGen. On lines $9-11$, each GPU thread, identified by its global id (`idx`), selects a different test case for execution (`input_gen`) as well as a different test result, in which to record its outputs (`result_gen`). On lines 19 and 21, ParTeCL CodeGen has replaced reading of the input parameters from standard input with reading from the input value for test case `input_gen`. It also replaces calls to the standard library functions `fgets` and `fgetc` with our custom OpenCL implementation of those functions, which is found in `cl-stdio.h`. Finally, the tool adds an assignment of the results of the test, variable `occurs`, to `result_gen`.

It is important to note that ParTeCL CodeGen does not change the core algorithm of the program, only the input/output interface, thus ensuring that the tested functionality remains the same.

3.1.3 ParTeCL Runtime. Once ParTeCL CodeGen generates the code necessary for launching the tests on the GPU, ParTeCL Runtime executes them in the following steps:

(1) Reading test cases. ParTeCL Runtime uses the parser generated by ParTeCL CodeGen to read the values of the test cases from the user supplied CSV file and stores them in an array of type `sruct partecl_input`.

(2) Kernel build and launch. The array with test case inputs is transferred to GPU memory, from where each

Listing 1: Original C program.

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]){
4    char str[1000], ch;
5
6    printf("Enter a string: ");
7    fgets(str, 1000, stdin);
8    printf("Enter a character: ");
9    ch = fgetc(stdin);
10
11   char* str_ptr = str;
12   int occurs = 0;
13   while(*str_ptr != '\0'){
14     if(*str_ptr == ch){
15       occurs++;
16     }
17     str_ptr++;
18   }
19
20   printf("'%c' occurs %d times.\n", ch, occurs);
21 }
```

Listing 2: Automatically generated OpenCL kernel.

```
1  #include "structs.h"
2  #include "cl-stdio.h"
3  //#include <stdio.h>
4
5  kernel void main_kernel(
6    global struct partecl_input* inputs,
7    global struct partecl_result* results)
8  {
9    int idx = get_global_id(0);
10   struct partecl_input input_gen = inputs[idx];
11   global struct partecl_result *result_gen = &results[idx];
12   int argc = input_gen.argc;
13   result_gen->test_case_num = input_gen.test_case_num;
14   int stdin_count_gen = 0;
15
16   char str[1000], ch;
17
18   /*printf("Enter a string: ");*/
19   fgets(str, 1000, input_gen.stdin1, &stdin_count_gen);
20   /*printf("Enter a character: ");*/
21   ch = fgetc(input_gen.stdin2, &stdin_count_gen);
22
23   char* str_ptr = str;
24   int occurs = 0;
25   while(*str_ptr != '\0'){
26     if(*str_ptr == ch){
27       occurs++;
28     }
29     str_ptr++;
30   }
31
32   /*printf("'%c' occurs %d times.\n", ch, occurs);*/
33   result_gen->occurs = occurs;
34 }
```

Figure 2: Example of converting a C program into an OpenCL GPU kernel using ParTeCL CodeGen.

Listing 3: Configuration file.

```
stdin: char* stdin1
stdin: char stdin2
result: int occurs variable: occurs
```

Listing 4: Generated data structures - file structs.h.

```
typedef struct partecl_input{
  int test_case_num;
  int argc;
  char* stdin1;
  char stdin2;
} partecl_input;

typedef struct partecl_result{
  int test_case_num;
  int occurs;
} partecl_result;
```

Figure 3: Configuration file and generated data structures for the example program.

GPU thread reads its respective test case and executes it, as described in Section 3.1.2. The ParTeCL Runtime builds and launches the OpenCL kernel generated by CodGen, using the standard OpenCL API.

(3) Results validation. Once the GPU kernel has executed, we transfer results back to the host where the results are validated against the golden output. Any difference observed is recorded and presented to the user.

3.2 Code Transformations

While generating the OpenCL kernel, ParTeCL CodeGen performs code transformations for C features, which are not readily supported by the OpenCL standard.

Global scope variables. OpenCL does not support assignment to global scope variables. ParTeCL CodeGen moves them to local scope by moving their declarations into the kernel function, `main_kernel`, and passing them as arguments to any functions using them. By using pointers, the tool ensures that any changes made to their values would be visible to the all the other functions.

Standard input and output. A value for every standard input needs to be supplied as part of the test case, as is shown in our use case. ParTeCL CodeGen then replaces references to the standard input with references to the memory structures containing the test cases. Standard output is commented out by ParTeCL CodeGen by default, but the user can choose to save it to the testing results using an option in the configuration file. In those cases, the tool replaces it with a write to the generated `test_result` structure.

Command line arguments. Similar to standard input, values for command line arguments should be supplied as part of the test case input. ParTeCL CodeGen replaces references to the command line arguments with references to the corresponding values in the generated `partecl_input` structure.

Standard library calls. The OpenCL standard does not support calls to the C Standard Library. We have implemented a small subset of Standard Library functions in OpenCL, namely functions in `ctype.h`, `string.h` and `stdio.h`. We took inspiration from **uClibc** [1], a very small C standard library typically used for embedded systems. In our future work, we plan to add OpenCL implementations of other standard library functions as the need for them arises. The code for this project is hosted on https://github.com/wyaneva/clclibc.

3.3 Implementation

ParTeCL CodeGen is implemented in C++14, using the Clang LibTooling library [4]. It uses LibTooling's AST Matchers to perform sequential compiler passes, which find and transform the relevant portions of the original program. ParTeCL Runtime is implemented in standard C, using the OpenCL API to perform all the GPU related operations. The source code for the two systems, along with instructions to build and execute them, can be found at https://github.com/wyaneva/partecl-codegen and https://github.com/wyaneva/partecl-runtime.

4 EVALUATION

We check whether ParTeCL meets its goals by evaluating its performance, correctness and usability.

Performance and effectiveness. In our work in [10], we evaluate the performance of GPU test acceleration using ParTeCL on C programs from the embedded systems domain. We use 9 benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC), which provides a diverse suite of benchmarks organised into categories that span numerous real-world applications, namely automotive, digital media, networking, office automation and telecom, among others [5].

Figure 4 shows the speedup achieved on the GPU using ParTeCL when executing test suites of size 131K test cases over each of the benchmarks. Our experiment in [10] shows that this test suite size saturates the GPU threads, resulting in highest speedup. To optimise speedup, ParTeCL Runtime provides the option of transferring test cases between the CPU and GPU memory in chunks, overlapping data transfer and test case execution. Figure 4 shows the speedups achieved on the GPU with and without data transfer overlap. We compare GPU speedups to those achieved by a multi-core CPU with 2, 4 and 8 cores (tests distributed using OpenMP).

We found that ParTeCL achieves significant speedup ranging from 18× to 53×. We found the magnitude of speedup is related to the computational intensity of the benchmark being executed. Benchmarks that exhibit a high compute-intensity tend to give high speedup when executing tests on the GPU. Average GPU speedup over all benchmarks is 16× as opposed to 6× for an 8-core CPU.

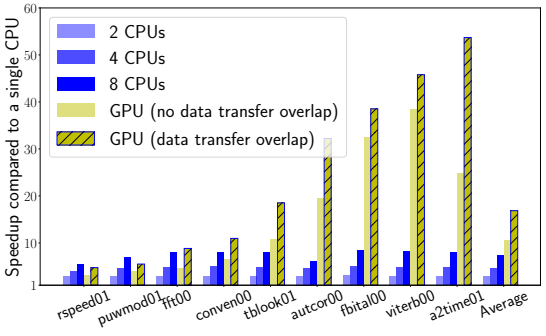

Figure 4: Speedup of GPU and multi-core CPUs over single CPU core. Test suite size 2^{17}

Correctness. For each subject program, we collected the test case outputs from the CPU and GPU executions across all test suites. We found that for all 9 subject programs, the test case outputs between the CPU and GPU executions were an **exact match**. We can safely conclude that our framework for executing tests on the GPU *preserves correctness of program execution* for all 9 embedded system benchmarks and test suites in our experiment.

Usability and feasibility. A preliminary usability study with six programmers, with no prior GPU programming knowledge, was performed to assess ParTeCL's ease of use. All programmers were asked to write tests for a simple C application, similar to the one in the use case in Section 3, and execute them on the GPU using ParTeCL. They were also asked to rate different aspects of using ParTeCL, shown in Table 1. Overall, users found the testing process with ParTeCL clear and easy to follow, demonstrating that ParTeCL successfully abstracts away the GPU programming details.

Step in the testing process	Rating
Writing test cases in the CSV format	4.00
Writing the configuration file	3.41
Running ParTeCL CodeGen	4.83
Building ParTeCL Runtime	4.85
Running the test cases	4.33
Overall process	**3.91**

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

5 CONCLUSION

ParTeCL provides the capability to leverage the computational power of GPUs for parallel execution of functional tests over C programs, without any prior GPU programming knowledge. The tool uses compilation techniques to automatically (1) generate an OpenCL kernel for the tested program, (2) provide transformations for C features which are not readily supported on the GPU, and (3) launch the tests in parallel on GPU threads. Performance evaluation on 9 embedded systems benchmarks shows that ParTeCL achieves test execution speedups of up to 53× on the GPU when compared to a single CPU. Usability evaluation demonstrates that ParTeCL is easy to use and requires no GPU programming knowledge. Future work will focus on extending ParTeCL to support further program features in OpenCL, including dynamic memory allocation, system calls and recursion.

REFERENCES

[1] E Andersen. 2004. uClibc website. (2004).
[2] Mary Jean Harrold. 2000. Testing: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. ACM.
[3] Ronan Keryell, Ruyman Reyes, and Lee Howes. 2015. Khronos SYCL for OpenCL: a tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*. ACM, 24.
[4] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.
[5] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. 2009. A Benchmark Characterization of the EEMBC Benchmark Suite. *IEEE Micro* 29, 5 (2009), 18–29.
[6] Ajitha Rajan. 2009. *Coverage metrics for requirements-based testing*. Ph.D. Dissertation. University of Minnesota.
[7] Ajitha Rajan, Subodh Sharma, Peter Schrammel, and Daniel Kroening. 2014. Accelerated test execution using GPUs. In *ACM/IEEE ASE'14*. 97–102.
[8] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. 2016. Performance Portable GPU Code Generation for Matrix Multiplication. In *GPGPU*. ACM Press, New York, 22–31. https://doi.org/10.1145/2884045.2884046
[9] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of ICFP 2015*. ACM, New York, 205–217. https://doi.org/10.1145/2784731.2784754
[10] Vanya Yaneva, Ajitha Rajan, and Christophe Dubach. To Appear 2017. Compiler-Assisted Test Acceleration on GPUs for Embedded Software. In *Proceedings of ISSTA 2017*.