COMPILER-ASSISTED TEST ACCELERATION ON GPUS FOR EMBEDDED SOFTWARE

VANYA YANEVA

Ajitha Rajan, Christophe Dubach

ISSTA 2017
10 July 2017
Santa Barbara, CA
EMBEDDED SOFTWARE IS EVERYWHERE

- ITS SAFETY AND CORRECTNESS ARE CRUCIAL
- FUNCTIONAL TESTING IS CRITICAL
FUNCTIONAL TESTING CAN BE EXTREMELY TIME CONSUMING
FUNCTIONAL TESTING CAN BE EXTREMELY TIME CONSUMING
FUNCTIONAL TESTING CAN BE EXTREMELY TIME CONSUMING

TESTING IS AN IDEAL CANDIDATE FOR PARALLELISATION
CPU SERVERS

- Expensive
- Do not scale easily as test suites grow
- Can be extremely underutilised
CPU SERVERS

- Expensive
- Do not scale easily as test suites grow
- Can be extremely underutilised

GPUS

- Cheap and widely available
- Large-scale parallelism, thousands of threads
- SIMD architecture suited to functional testing
EXECUTE TESTS IN PARALLEL ON THE GPU THREADS

EXECUTE TESTS IN PARALLEL ON THE GPU THREADS

INTRODUCING PARTECL

- Test cases (CSV format)
- Unmodified source files
- Config file

ParTeCLCodeGen

OpenCL

ParTeCL Runtime

Execution on the GPU
Example:

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

int c;

int addc(int a, int b){
    return a + b + c;
}

int main(int argc, char* argv[]){
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    c = 3;
    int sum = addc(a, b);
    printf("%d + %d + %c = %d\n", a, b, c, sum);
}
```

Configuration:

- input: int a 1
- input: int b 2
- result: int sum variable: sum

Test cases:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example:

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

int c;

int addc(int a, int b){
    return a + b + c;
}

int main(int argc, char* argv[]){
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    c = 3;
    int sum = addc(a, b);
    printf("%d + %d + %c = %d\n", a, b, c, sum);
}
```

OpenCL:

```c
#include "structs.h"

int addc(int a, int b, int *c){
    return a + b + (*c);
}

kernel void main_kernel(
    global struct test_input* inputs,
    global struct test_result* results){

    int idx = get_global_id(0);
    struct test_input input_gen = inputs[idx];
    global struct test_result *result_gen = &results[idx];

    int argc = input_gen.argc;
    result_gen->test_case_num = input_gen.test_case_num;
    int c;
    int a = input_gen.a;
    int b = input_gen.b;
    c = 3;
    int sum = addc(a, b, &c);
    /*printf("%d + %d + %c = %d\n", a, b, c, sum);*/
    result_gen->sum = sum;
}```
CODE TRANSFORMATIONS

- global scope variables
- command line arguments
- standard in/out
- standard library (partial support): clClibc
Read test cases:
\[ \text{INPUT}[] = \{\text{test case 1} \ldots \text{test case n}\} \]

Transfer \[\text{INPUT}[]\] to GPU memory

Build and launch \textit{tested program} on the GPU threads

\[
\text{OUTPUT}[\text{th_id}] = \text{program}( \text{INPUT}[\text{th_id}] )
\]

Transfer \[\text{OUTPUT}[]\] to CPU memory

\text{Automatically generated OpenCL}
CHALLENGES

Usability ✔
Scope ✔
Performance ?

Unmodified source files
Config file
ParTeCLCodeGen
OpenCL
ParTeCLRuntime
Execution on the GPU
Test cases (CSV format)
EVALUATION

1. Speedup against CPU
2. Data transfer overhead
3. Comparison to a multi-core CPU
4. Correctness
EXPERIMENT

- **Subjects:** EEMBC - Industry-standard benchmark suite for embedded software
- **Hardware:** GPU - NVidia Tesla K40m; CPU - Intel Xeon, 8 cores
- **Test suite size:** 130K
SPEEDUP AGAINST CPU

The diagram shows the speedup of various GPU tasks compared to a single CPU as the number of tests increases. The x-axis represents the number of tests in logarithmic scale, while the y-axis shows the GPU speedup. Different tasks are represented by colored lines and markers.

- viterb00
- fbital00
- a2time01
- autcor00
- tblook01
- conven00
- fft00
- puwmod01
- rspeed01
DATA TRANSFER OVERHEAD
DATA TRANSFER OVERHEAD

The diagram shows the speedup compared to a single CPU for various benchmarks under two conditions: GPU (no data transfer overlap) and GPU (data transfer overlap). The benchmarks are rspeed01, puwmmod01, fft00, conven00, tblock01, autcor00, fbital00, viterb00, a2time01, and the average. The speedup is measured on a logarithmic scale, with the y-axis representing the speedup ratio.
COMPARISON TO A MULTI-CORE CPU

![Diagram showing speedup compared to a single CPU for different numbers of CPUs and GPUs with and without data transfer overlap. The x-axis represents different benchmarks: rspeed01, puwmod01, fft00, conven00, tblock01, autcor00, fbital00, viterb00, a2time01, and Average. The y-axis represents speedup ranging from 1 to 50. The chart includes bars for 2 CPUs, 4 CPUs, 8 CPUs, GPU (no data transfer overlap), and GPU (data transfer overlap).]
<table>
<thead>
<tr>
<th>CHALLENGES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability</strong></td>
</tr>
<tr>
<td><strong>Scope</strong></td>
</tr>
<tr>
<td><strong>Performance</strong></td>
</tr>
</tbody>
</table>
For all 9 benchmarks, testing results from the GPU are an exact match to the testing results from the CPU.
SUMMARY

- Automatic GPU code generation
- Automatic test execution on the GPU threads
- Speedup of up to 53x (avg 16x) on EEMBC benchmarks
- Correct testing results
SUMMARY

• Automatic GPU code generation
• Automatic test execution on the GPU threads
• Speedup of up to 53x (avg 16x) on EEMBC benchmarks
• Correct testing results

FUTURE WORK

• Extend evaluation & scope
• Analyse & improve performance
# THANKS

<table>
<thead>
<tr>
<th>ParTeCL CodeGen</th>
<th>github.com/wyaneva/partecl-codegen</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParTeCL Runtime</td>
<td>github.com/wyaneva/partecl-runtime</td>
</tr>
<tr>
<td>clClibc</td>
<td>github.com/wyaneva/clClibc</td>
</tr>
</tbody>
</table>
C FEATURES

• **Out of the box:**
  - pure functions, function calls, double precision (for OpenCL 1.2)

• **With transformations:**
  - standard in/out
  - global scope variables
  - standard library calls (partial support)

• **Unsupported (yet):**
  - dynamic memory allocation
  - file I/O
  - recursion