Approximate Computing with Runtime Code Generation on Resource-Constrained Embedded Devices

WAPCO
HiPEAC conference 2016

Damien Couroussé Caroline Quéva Henri-Pierre Charles
Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA-LIST, MINATEC Campus, F-38054 Grenoble, France
2016-01-20
an approximate overview of

Approximate Computing

Approximate Hardware
- Resilience to (hw) errors
- Beyond the bounds of safe operating modes

Approximate Software
- Results analysis in presence of approximation
- New programming paradigms

Implementation of approximate-aware components
Solutions for precision-aware implementations:

- **Generic implementation over every possible value of p**
  - Easy to implement. Not efficient
- **Static versionning**
  - Almost zero runtime overhead. Big memory consumption
- **Runtime code generation**
  - Extra overhead for runtime code generation. Memory lightweight. Greater flexibility
Pitch: some code optimisations are not accessible to static compilers

- Unknown data
- Sometimes, the hardware is also unknown, at least partially

Delay code optimisations at runtime

- Constant propagation, elimination of dead code,
- Strength reduction,
- Loop unrolling,
- Instruction scheduling,
- etc.

Drive code performance by runtime-changing constraints

- Bounds: power / energy / execution time
- Heterogeneous cores: accelerators, specialized instructions
overview of deGoal

deGoal

- Portable DSL
- Complex variables (registers): vector support, dynamically sized, typed
- Mix runtime data & binary code
- Extendable to domain-specific instructions

Results

- Auto-adaptative dynamic libraries
- Portable runtime optimization
- Multiple performance metrics:
  - Generated code is smaller and faster
  - Code generators are smaller and faster

Simple program example: vector addition

```c
void gen_vector_add(void *buffer, int vec_len, int val)
{
    #[ Begin buffer Prelude vec_addr
    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
}
```

Program memory:

```
ldr r1, [r0]
add r1, #1
str r1, [r0]
add r0, #4
ldr r2, [r0]
add r2, #1
str r2, [r0]
add r0, #4
```
Self-optimising library

Application

Runtime code generator

A* A

B* B

C* C
use case: floating point multiplication

- MSP430: 512 bytes of RAM only!
- Floating-point multiplications on MSP430
  - Standard library function: ~1000 cycles per invocation
  - Micro-controllers lack dedicated HW support for arithmetic computing
  - Linear function often used to convert sensor value to user value
- Approximation of precision p using mantissa truncation
**performance metrics**

$t_{\text{ref}}$ : execution time of libm’s multiplication routine

$t_{\text{gen}}$ : execution time of code generation

$t_{\text{approx}}$ : execution time of the generated approximate function

- **speedup**:

  \[ s = \frac{t_{\text{approx}}}{t_{\text{ref}}} \]

- **overhead recovering**:

  \[ N = \frac{t_{\text{gen}}}{t_{\text{ref}} - t_{\text{approx}}} \]
Precision-aware implementation, no data specialisation

reference version

```c
float fmul (float M, float X) {
    return (M*X);
}
```

approximate version

```c
set_precision(p); /* p in [1;24] */
float fmul (float M, float X) {
    return (M*X);
}
```
Precision-aware implementation, no data specialisation

- Reference implementation: libm for msp430
- Our generated implementation:
  Precision $p$ in [1; 24]

$p=12$, speedup $\sim x1.5$
overhead recovery $\sim 8$

$p=24$, (similar to libm)
no performance improvement, high overhead recovery
Precision-aware implementation, with data specialisation

```c
set_precision(p); /* p in [1;24] */
f = compile_fmul(M);
float fmul (float M, float X)
{
    return f(X);   /* return M*X */
}
```
Precision-aware implementation, with data specialisation

- Reference implementation: libm for msp430
- Our generated implementation: Precision $p$ in $[1; 24]$

<5 executions only to pay off code generation in the worst case

$p=24$, (similar to libm)
Speedup > 6x,
> 7x 80% of the time
Provide the average developer with approximate-aware components

Runtime code specialisation for approximate-aware applications

> 7x faster with data specialisation in our use case

  Approximation can improve the speedup up to 10x

Follow-up work: drive code generation by higher level tools for approximation analysis

!! ??

Approximate Computing on Resource-Constrained Embedded Devices with Runtime Code Generation
damien.courrousse@cea.fr
Approaches for code specialization

**Static code versionning** (e.g. C++ Templates)

- static compilation
- runtime: select executable
- memory footprint ++
- limited genericity
- runtime blindness

**Runtime code generation**, with deGoal
A *compillette* is an ad hoc code generator, targeting one executable

- fast code generation
- memory footprint ---
- data-driven code generation

**Dynamic compilation**
(JITs, e.g. Java Hotspot)

- overhead ++
- memory footprint ++
- not designed for data dependant code-optimisations

Intermediate Representation
## deGoal supported architectures

<table>
<thead>
<tr>
<th>ARCHITECTURE</th>
<th>STATUS</th>
<th>FEATURES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM32</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ARM Cortex-A, Cortex-M [Thumb-2, VFP, NEON]</td>
<td>✓</td>
<td>SIMD, [IO/OoO]</td>
</tr>
<tr>
<td>STxP70 [including FPx] (STHORM / P2012)</td>
<td>✓</td>
<td>SIMD, VLIW (2-way)</td>
</tr>
<tr>
<td>K1 (Kalray MPPA)</td>
<td>✓</td>
<td>SIMD, VLIW (5-way)</td>
</tr>
<tr>
<td>PTX (Nvidia GPUs)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MIPS</td>
<td>⌚</td>
<td>32-bits</td>
</tr>
<tr>
<td>MSP430 (TI microcontroller)</td>
<td>✓</td>
<td>Up to &lt; 1kB RAM</td>
</tr>
</tbody>
</table>

**CROSS CODE GENERATION supported**

(e.g. generate code for STxP70 from an ARM Cortex-A)

[IO/OoO]: Instruction scheduling for in-order and out-of-order cores