Automatic Generation of Fast and Certified Code for Polynomial Evaluation

Guillaume Revy
DALI project-team
Université de Perpignan Via Domitia
LIRMM (CNRS: UMR 5506 - UM2)

Joint work with Christophe Mouilleron (LIP, ENS Lyon)
Motivation

- **Embedded systems** are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost
Motivation

- **Embedded systems** are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

- Some embedded systems **do not have any FPU** (floating-point unit)
Motivation

- **Embedded systems** are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

- Some embedded systems **do not have any FPU** (floating-point unit)

- Highly used in audio and video applications
  - demanding on **floating-point computations**
Motivation

- **Embedded systems** are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

- Some embedded systems **do not have any FPU** (floating-point unit)

- Highly used in audio and video applications
  - demanding on **floating-point computations**
On the one side: the IEEE 754-2008 standard, ...

- Definition of IEEE floating-point arithmetic
  - floating-point formats: single precision, double precision, ...
  - special values: $\pm 0$, $\pm \infty$, NaN
  - 4 rounding modes: to nearest even, upward, downward, and toward zero
  - mathematical function behavior
    - special input (ex: $\sqrt{-0} = -0$)
    - requires / recommends correct rounding

- Motivation:
  - make computations reproducible
  - and make results architecture-independent
... on the other side: the ST231 processor

- 4-issue VLIW 32-bit integer processor → no FPU
- Parallel execution unit
  - 4 integer ALUs
  - 2 pipelined multipliers $32 \times 32 \rightarrow 32$
- Latencies: $\text{ALU} = 1 \text{ cycle} / \text{Mul} = 3 \text{ cycles}$
... on the other side: the ST231 processor

- **4-issue VLIW 32-bit integer processor**
  - no FPU
- **Parallel execution unit**
  - 4 integer ALUs
  - 2 pipelined multipliers 32 × 32 → 32
- **Latencies**: ALU = 1 cycle / Mul = 3 cycles

- **VLIW (Very Long Instruction Word)**
  - instructions grouped into **bundles**
  - Instruction-Level Parallelism (ILP) explicitly exposed by the compiler
Towards the generation of fast and certified codes

- This work takes mainly part in the context of the development of FLIP
  → software support for binary32 floating-point arithmetic on integer processors

- Underlying problem: development “by hand”
  ▶ long and tedious, error prone
  ▶ new target ? new floating-point format ?
Towards the generation of fast and certified codes

- This work takes mainly part in the context of the development of FLIP
  - software support for binary32 floating-point arithmetic on integer processors

- Underlying problem: development “by hand”
  - long and tedious, error prone
  - new target ? new floating-point format ?

  \{ \text{automation and certification} \}
Towards the generation of fast and certified codes

- This work takes mainly part in the context of the development of FLIP
  → software support for binary32 floating-point arithmetic on integer processors

- Underlying problem: development “by hand”
  - long and tedious, error prone
  - new target ? new floating-point format ?

- Current challenge: tools and methodologies for the automatic generation of efficient and certified programs
  - optimized for a given format, for the target architecture
Towards the generation of fast and certified codes

- Some works on code generation and transformation:
  - code generators: hardware (FloPoCo) and software (Sollya, Metalibm)
  - code transformation for increasing numerical accuracy [Martel, 2009]

- LEMA project [Lefèvre et al., 2010]: language and library
  - design easily a generation toolchain
Towards the generation of fast and certified codes

- Some works on code generation and transformation:
  - code generators: hardware (FloPoCo) and software (Sollya, Metalibm)
  - code transformation for increasing numerical accuracy [Martel, 2009]

- LEMA project [Lefèvre et al., 2010]: language and library
  - design easily a generation toolchain

- Spiral project: hardware and software code generation for DSP algorithms

  Can we teach computers to write fast libraries?
Towards the generation of fast and certified codes

- Some works on code generation and transformation:
  - code generators: hardware (FloPoCo) and software (Sollya, Metalibm)
  - code transformation for increasing numerical accuracy [Martel, 2009]

- LEMA project [Lefèvre et al., 2010]: language and library
  - design easily a generation toolchain

- Spiral project: hardware and software code generation for DSP algorithms
  - Can we teach computers to write fast libraries?

- Our tool: CGPE (Code Generation for Polynomial Evaluation)

  In the particular case of polynomial evaluation, can we teach computers to write fast and certified codes, for a given target and optimized for a given format?
  - adding a systematic certification step
Basic blocks for implementing correctly-rounded operators

Objectives

→ Low latency, correctly-rounded implementations

→ ILP exposure
Basic blocks for implementing correctly-rounded operators

- Floating-point number unpacking
- Normalization
- Range reduction
- Result sign/exponent computation
- Result significand approximation
- Rounding condition decision
- Correct rounding computation
- Result reconstruction
- Special input detection
- Special output selection

- Problem: function to be evaluated
- Computation of polynomial approximant
- Fast and certified C code
- ST231 features

- Uniform approach for $n$th roots and their reciprocals
  $\rightarrow$ polynomial evaluation
- Extension to division

- $R \{\pm 0, \pm \infty, \text{NaN}\}$
Flowchart for generating efficient and certified C codes

1. Problem: function to be evaluated
2. Computation of polynomial approximant
3. Fast and certified C code generation
4. ST231 features
5. C code
6. Certificate
Flowchart for generating efficient and certified C codes

Problem: function to be evaluated

Computation of polynomial approximant

Fast and certified C code generation

ST231 features

Constraints

- Accuracy of approximant and C code
Flowchart for generating efficient and certified C codes

Problem: function to be evaluated

Computation of polynomial approximant

Fast and certified C code generation

ST231 features

Constraints

- Accuracy of approximant and C code
- Low evaluation latency on ST231, ILP exposure
Flowchart for generating efficient and certified C codes

Constraints

- Accuracy of **approximant and C code**
  - Sollya

- Low evaluation latency on ST231, ILP exposure
Flowchart for generating efficient and certified C codes

Constraints

- Accuracy of approximant and C code
  - Sollya
  - interval arithmetic (MPFI), Gappa

- Low evaluation latency on ST231, ILP exposure
Flowchart for generating efficient and certified C codes

Problem: function to be evaluated

Fast and certified C code generation

Computation of polynomial approximant

ST231 features

C code Certificate

Constraints

- Accuracy of approximant and C code
  - Sollya
  - interval arithmetic (MPFI), Gappa
- Low evaluation latency on ST231, ILP exposure
  - ?
Outline of the talk

1. Background on polynomial evaluation

2. The CGPE tool

3. Experimental results

4. Conclusion
Outline of the talk

1. Background on polynomial evaluation

2. The CGPE tool

3. Experimental results

4. Conclusion
Our objective

- Compute fast and certified schemes for evaluating a polynomial
  \[ P(x, y) = \alpha + y \cdot a(x) \]
  → using only additions and multiplications
  → reducing the evaluation latency on unbounded parallelism
Our objective

- Compute fast and certified schemes for evaluating a polynomial $P(x, y) = \alpha + y \cdot a(x)$
  - using only additions and multiplications
  - reducing the evaluation latency on unbounded parallelism

- Evaluation program = main part of the full software implementation
  - dominates the cost
  - make it as efficient as possible
Our objective

- Compute fast and certified schemes for evaluating a polynomial $P(x, y) = \alpha + y \cdot a(x)$
  - using only additions and multiplications
  - reducing the evaluation latency on unbounded parallelism

- Evaluation program = main part of the full software implementation
  - dominates the cost
  - make it as efficient as possible

Two families of algorithms

- algorithms with coefficient adaptation: Knuth and Eve (60’s), Paterson and Stockmeyer (1973), ...
  - ill-suited in the context of fixed-point arithmetic
- algorithms without coefficient adaptation
Our objective

- Compute fast and certified schemes for evaluating a polynomial
  \[ P(x, y) = \alpha + y \cdot a(x) \]
  → using only additions and multiplications
  → reducing the evaluation latency on unbounded parallelism

- Evaluation program = main part of the full software implementation
  → dominates the cost
  ↔ make it as efficient as possible

- Two families of algorithms
  - algorithms with coefficient adaptation: Knuth and Eve (60’s), Paterson and Stockmeyer (1973), ...
    → ill-suited in the context of fixed-point arithmetic
  - algorithms without coefficient adaptation
Classical evaluation schemes

- Naive evaluation
  - 3 additions
  - 5 multiplications
  - latency: 12 cycles
Background on polynomial evaluation

Classical evaluation schemes

- Horner’s rule

  → 3 additions
  → 3 multiplications
  → latency: 12 cycles

⊕ optimal in terms of multiplication number
  [Pan, 1966], [Borodin, 1971],

⊕ fully sequential
Classical evaluation schemes

- Second-order Horner’s rule
  - 3 additions
  - 4 multiplications
  - latency: 11 cycles

  + some ILP exposure
  + subparts evaluated in a fully sequential way: at most 2 ways used
Classical evaluation schemes

- Estrin’s rule
  - 3 additions
  - 4 multiplications
  - latency: 8 cycles
  - “divide and conquer” strategy
  - more ILP exposure
Definition of evaluation schemes

Mathematical expression: \( a_0 + a_1 \cdot x + a_2 \cdot x^2 \)

Common parenthesizations:
\[
(a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x))
\]
Definition of evaluation schemes

Mathematical expression

\[ a_0 + a_1 \cdot x + a_2 \cdot x^2 \]

Common parenthesized expressions

\[ (a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x)) \]

Feasible parenthesized expressions

\[ (a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x)) \]
\[ (a_0 + (a_1 \cdot x)) + ((a_2 \cdot x) \cdot x) \]
\[ a_0 + ((a_1 \cdot x) + ((a_2 \cdot x) \cdot x)) \]
\[ a_0 + ((a_1 + (a_2 \cdot x)) \cdot x) \]
\[ ((a_1 + (a_2 \cdot x)) \cdot x) + a_0 \]

Commutativity, associativity, distributivity
Definition of evaluation schemes

Mathematical expression

\[ a_0 + a_1 \cdot x + a_2 \cdot x^2 \]

Common *parenthesizations*

\[ (a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x)) \]

Feasible *parenthesizations*

\[ \ldots \]

Evaluation schemes

\[
\begin{align*}
(a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x)) & \\
(a_0 + (a_2 \cdot (x \cdot x))) + (a_1 \cdot x) & \\
a_0 + ((a_1 \cdot x) + (a_2 \cdot (x \cdot x))) & \\
a_0 + (a_1 + (a_2 \cdot x)) \cdot x
\end{align*}
\]

\[
\begin{align*}
(a_0 + (a_1 \cdot x)) + ((a_2 \cdot x) \cdot x) & \\
(a_0 + ((a_2 \cdot x) \cdot x)) + (a_1 \cdot x) & \\
a_0 + ((a_1 \cdot x) + ((a_2 \cdot x) \cdot x))) & \\
a_0 + ((a_1 \cdot x) + ((a_2 \cdot x) \cdot x)))
\end{align*}
\]
Remarks on polynomial evaluation

- There are several other schemes for evaluating a polynomial $a(x)$
  - can be adapted for bivariate polynomial $P(x, y) = \alpha + y \cdot a(x)$

- Constant number of $+$, while number of $\times$ is non-constant
  - reducing the latency $\Leftrightarrow$ increasing the number of $\times$ to expose ILP
  - trade-off latency / number of multiplications
Remarks on polynomial evaluation

- There are several other schemes for evaluating a polynomial $a(x)$
  - can be adapted for bivariate polynomial $P(x, y) = \alpha + y \cdot a(x)$

- Constant number of $+$, while **number of $\times$ is non-constant**
  - reducing the latency $\iff$ increasing the number of $\times$ to expose ILP
  - trade-off latency / number of multiplications

- Evaluation error
  - different theoretical error bounds
  - difference between numerical quality in practice [Revy, 2006]
Remarks on polynomial evaluation

- There are several other schemes for evaluating a polynomial $a(x)$
  - can be adapted for bivariate polynomial $P(x, y) = \alpha + y \cdot a(x)$

- Constant number of $+$, while number of $\times$ is non-constant
  - reducing the latency $\iff$ increasing the number of $\times$ to expose ILP
  - trade-off latency / number of multiplications

- Evaluation error
  - different theoretical error bounds
  - difference between numerical quality in practice [Revy, 2006]

$\leadsto$ We need a tool for exploring the space of evaluation schemes.
### How many schemes for evaluating a polynomial?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\mu_n \rightarrow a(x)$</th>
<th>$\mu'_n \rightarrow \alpha + y \cdot a(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>481</td>
</tr>
<tr>
<td>3</td>
<td>163</td>
<td>88384</td>
</tr>
<tr>
<td>4</td>
<td>11602</td>
<td>57363910</td>
</tr>
<tr>
<td>5</td>
<td>2334244</td>
<td>122657263474</td>
</tr>
<tr>
<td>6</td>
<td>1304066578</td>
<td>829129658616013</td>
</tr>
<tr>
<td>7</td>
<td>1972869433837</td>
<td>17125741272619781635</td>
</tr>
<tr>
<td>8</td>
<td>8012682343669366</td>
<td>1055157310305502607244946</td>
</tr>
<tr>
<td>9</td>
<td>86298937651093314877</td>
<td>190070917121184028045719056344</td>
</tr>
<tr>
<td>10</td>
<td>2449381767217281163362301</td>
<td>98543690848554380947490522591191672</td>
</tr>
</tbody>
</table>
How many schemes for evaluating a polynomial?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\mu_n \rightarrow a(x)$</th>
<th>$\mu'_n \rightarrow \alpha + y \cdot a(x)$</th>
<th>$w_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>481</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>163</td>
<td>88384</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>11602</td>
<td>57363910</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2334244</td>
<td>122657263474</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1304066578</td>
<td>829129658616013</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1972869433837</td>
<td>17125741272619781635</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>8012682343669366</td>
<td>1055157310305502607244946</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>86298937651093314877</td>
<td>190070917121184028045719056344</td>
<td>46</td>
</tr>
<tr>
<td>10</td>
<td>2449381767217281163362301</td>
<td>98543690848554380947940522591191672</td>
<td>98</td>
</tr>
</tbody>
</table>

Two well-known special cases

- the number of evaluation schemes for $x^n$ [Wedderburn, Etherington]

$$w_n \sim \eta \xi^n \frac{n}{n^{3/2}} \quad \text{or} \quad \begin{cases} 
\xi \approx 2.48325 \\
\eta \approx 0.31877 
\end{cases} \quad \text{[Otter, 1948]},$$
How many schemes for evaluating a polynomial?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\mu_n \rightarrow a(x)$</th>
<th>$\mu'_n \rightarrow \alpha + y \cdot a(x)$</th>
<th>$w_n$</th>
<th>$(2n - 1)!!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>481</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>163</td>
<td>88384</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>11602</td>
<td>57363910</td>
<td>2</td>
<td>105</td>
</tr>
<tr>
<td>5</td>
<td>2334244</td>
<td>122657263474</td>
<td>3</td>
<td>945</td>
</tr>
<tr>
<td>6</td>
<td>1304066578</td>
<td>829129658616013</td>
<td>6</td>
<td>10395</td>
</tr>
<tr>
<td>7</td>
<td>1972869433837</td>
<td>17125741272619781635</td>
<td>11</td>
<td>135135</td>
</tr>
<tr>
<td>8</td>
<td>8012682343669366</td>
<td>1055157310305502607244946</td>
<td>23</td>
<td>2027025</td>
</tr>
<tr>
<td>9</td>
<td>86298937651093314877</td>
<td>190070917121184028045719056344</td>
<td>46</td>
<td>34459425</td>
</tr>
<tr>
<td>10</td>
<td>2449381767217281163362301</td>
<td>98543690848554380947490522591191672</td>
<td>98</td>
<td>654729075</td>
</tr>
</tbody>
</table>

- Two well-known special cases
  - the number of evaluation schemes for $x^n$ [Wedderburn, Etherington]
    
    $w_n \sim \frac{\eta \xi^n}{n^{3/2}}$ or
    
    $\begin{cases} 
    \xi \approx 2.48325 \\
    \eta \approx 0.31877
    \end{cases}$ [Otter, 1948],

  - the number of evaluation schemes for $\sum_{i=1}^{n} a_i \text{est} (2n - 1)!! \sim \sqrt{2} \left( \frac{2n}{e} \right)^n$. 

Guillaume Revy (Groupe de travail PEQUAN – July 7, 2011)
Schemes of low evaluation latency

What is the latency of degree-5 evaluation schemes?

Total number of schemes: 2334244

\[ \Rightarrow \text{minimal latency for degree-5 univariate polynomial: 10 cycles} \]

\[ \Rightarrow \text{number of schemes of minimal latency: 36} \]
Outline of the talk

1. Background on polynomial evaluation

2. The CGPE tool

3. Experimental results

4. Conclusion
Overview of CGPE and related works

- Goal of CGPE [Mouilleron and Revy, 2011]: automate the design of fast and certified C codes for evaluating univariate/bivariate polynomials
  - in fixed-point arithmetic
  - by using the target architecture features (as much as possible)

- Remarks:
  - \textit{fast} = that reduces the evaluation latency on a given target
  - \textit{certified} = we can bound the error entailed by the evaluation within the given target’s arithmetic
Overview of CGPE and related works

■ Some related works

▶ [Cheung et al., 2005] and [Lee and Villasenor, 2009]: methodology for implementing automatically mathematical function in a given precision
  ⊖ based on small degree polynomial evaluation using Horner’s → no ILP

▶ [Harrison et al., 1999]: method for generating optimal evaluation scheme to evaluate univariate polynomials on Itanium® using fma
  ⊖ ST231 has only addition and multiplication, but no fma

▶ [Green, 2002]: brute force method for generating polynomial evaluation schemes using at best SIMD instructions of the processor of PlayStation® 2
  ⊖ objective: generation at compile-time → brute force method is unfeasible
Global architecture of CGPE

Input of CGPE

1. polynomial coefficients and variables: value intervals, fixed-point format, ...
2. set of criteria: maximum error bound and bound on latency (or the lowest)
3. delay of one of the variable
4. some architectural constraints: operator cost, parallelism, ...
Global architecture of CGPE

- **Input of CGPE**
  1. polynomial coefficients and variables: value intervals, fixed-point format, ...
  2. set of criteria: maximum error bound and bound on latency (or the lowest)
  3. delay of one of the variable
  4. some architectural constraints: operator cost, parallelism, ...

- **CGPE works in two steps:**
  1. **computation** of evaluation schemes: reducing evaluation latency on unbounded parallelism and exposing as much ILP as possible
  2. **selection** among the generated schemes, according to different criteria:
     - evaluation using only unsigned fixed-point arithmetic
     - scheduling feasible on ST231
     - evaluation error bound satisfying the required error bound

- **At the end:** CGPE automatically writes C codes with accuracy certificates
Heuristics in DAG set computation

- Determination of the minimal target latency on unbounded parallelism
  - gives a good estimation of the best evaluation latency of the polynomial on the target architecture
  - takes some problem parameters (operator costs, delay, ...) into account
Heuristics in DAG set computation

- Determination of the minimal target latency on unbounded parallelism
  - gives a good estimation of the best evaluation latency of the polynomial on the target architecture
  - takes some problem parameters (operator costs, delay, ...) into account

- Non exhaustive computation of evaluation schemes
  - elimination of the schemes that do not satisfy latency constraint
  - limitation to some splittings: evaluation of high and low parts separately
  - restriction to $N$ schemes at each step of the computation
Heuristics in DAG set computation

- Determination of the minimal target latency on unbounded parallelism
  - gives a good estimation of the best evaluation latency of the polynomial on the target architecture
  - takes some problem parameters (operator costs, delay, ...) into account

- Non exhaustive computation of evaluation schemes
  - elimination of the schemes that do not satisfy latency constraint
  - limitation to some splittings: evaluation of high and low parts separately
  - restriction to $N$ schemes at each step of the computation

- At the end of the computation: set of DAG evaluating the input polynomial and satisfying the latency constraint on unbounded parallelism.
Filters for adding numerical constraints

1. Arithmetic operator choice
   ▶ ensure that all intermediate variables are of constant sign
   ▶ avoid an extra cost due to sign handling / gain 1 bit of accuracy
Filters for adding numerical constraints

1. Arithmetic operator choice
   - ensure that all intermediate variables are of constant sign
   - avoid an extra cost due to sign handling / gain 1 bit of accuracy

2. Scheduling on a simplified model of the target (like the ST231)
   - constraints of architecture: cost of operators, instruction bundling, ...
   - delays on variables
Filters for adding numerical constraints

1. Arithmetic operator choice
   - ensure that all intermediate variables are of constant sign
   - avoid an extra cost due to sign handling / gain 1 bit of accuracy

2. Scheduling on a simplified model of the target (like the ST231)
   - constraints of architecture: cost of operators, instruction bundling, ...
   - delays on variables

3. Evaluation error bound checking
   - straightforward polynomial evaluation program
   - “C code certification” using Gappa
     ~ we can bound the evaluation error in integer arithmetic
Outline of the talk

1. Background on polynomial evaluation

2. The CGPE tool

3. Experimental results

4. Conclusion
## Timings for certified code generation

<table>
<thead>
<tr>
<th></th>
<th>$x^{1/2}$</th>
<th>$x^{-1/2}$</th>
<th>$x^{1/3}$</th>
<th>$x^{-1/3}$</th>
<th>$\log_2 (1 + x)$</th>
<th>$\frac{1}{\sqrt{1 + x^2}}$</th>
<th>$\frac{\exp(1 + x)}{1 + x}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Degree ($d_x, d_y$)</strong></td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(6,0)</td>
<td>(7,0)</td>
<td>(10,0)</td>
</tr>
<tr>
<td><strong>Target / Minimal latency</strong></td>
<td>13 / 13</td>
<td>13 / ?</td>
<td>16 / 16</td>
<td>16 / 16</td>
<td>10 / 11</td>
<td>10 / 11</td>
<td>13 / 13</td>
</tr>
<tr>
<td><strong>Achieved latency</strong></td>
<td>13</td>
<td>14</td>
<td>16</td>
<td>16</td>
<td>11</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td><strong>Scheme computation</strong></td>
<td>195ms</td>
<td>73ms</td>
<td>26s</td>
<td>25s</td>
<td>17ms</td>
<td>10ms</td>
<td>40ms</td>
</tr>
<tr>
<td></td>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
</tr>
<tr>
<td><strong>Arithmetic operator choice</strong></td>
<td>3ms</td>
<td>3ms</td>
<td>7ms</td>
<td>11ms</td>
<td>1ms</td>
<td>2ms</td>
<td>3ms</td>
</tr>
<tr>
<td></td>
<td>[35]</td>
<td>[29]</td>
<td>[30]</td>
<td>[26]</td>
<td>[2]</td>
<td>[12]</td>
<td>[27]</td>
</tr>
<tr>
<td><strong>Scheduling checking</strong></td>
<td>16s</td>
<td>1m33s</td>
<td>43ms</td>
<td>439ms</td>
<td>2ms</td>
<td>64ms</td>
<td>49s</td>
</tr>
<tr>
<td><strong>Certification (Gappa)</strong></td>
<td>10s</td>
<td>1s</td>
<td>27s</td>
<td>27s</td>
<td>230ms</td>
<td>1s</td>
<td>7s</td>
</tr>
<tr>
<td><strong>Total time ($\approx$)</strong></td>
<td>27s</td>
<td>1m35s</td>
<td>55s</td>
<td>53s</td>
<td>1s</td>
<td>2s</td>
<td>57s</td>
</tr>
</tbody>
</table>
## Timings for certified code generation

<table>
<thead>
<tr>
<th>Degree ((d_x,d_y))</th>
<th>(x^{1/2})</th>
<th>(x^{-1/2})</th>
<th>(x^{1/3})</th>
<th>(x^{-1/3})</th>
<th>(\log_2(1+x))</th>
<th>(\frac{1}{\sqrt{1+x^2}})</th>
<th>(\frac{\exp(1+x)}{1+x})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target / Minimal latency</td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(6,0)</td>
<td>(7,0)</td>
<td>(10,0)</td>
</tr>
<tr>
<td>Achieved latency</td>
<td>13 / 13</td>
<td>13 / ?</td>
<td>16 / 16</td>
<td>16 / 16</td>
<td>10 / 11</td>
<td>10 / 11</td>
<td>13 / 13</td>
</tr>
<tr>
<td>Scheme computation</td>
<td>195ms</td>
<td>73ms</td>
<td>26s</td>
<td>25s</td>
<td>17ms</td>
<td>10ms</td>
<td>40ms</td>
</tr>
<tr>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
<td>[50]</td>
</tr>
<tr>
<td>Arithmetic operator choice</td>
<td>3ms</td>
<td>3ms</td>
<td>7ms</td>
<td>11ms</td>
<td>1ms</td>
<td>2ms</td>
<td>3ms</td>
</tr>
<tr>
<td>[35]</td>
<td>[29]</td>
<td>[30]</td>
<td>[26]</td>
<td>[2]</td>
<td>[12]</td>
<td>[27]</td>
<td></td>
</tr>
<tr>
<td>Scheduling checking</td>
<td>16s</td>
<td>1m33s</td>
<td>43ms</td>
<td>439ms</td>
<td>2ms</td>
<td>64ms</td>
<td>49s</td>
</tr>
<tr>
<td>Certification (Gappa)</td>
<td>10s</td>
<td>1s</td>
<td>27s</td>
<td>27s</td>
<td>230ms</td>
<td>1s</td>
<td>7s</td>
</tr>
<tr>
<td>Total time ((\approx))</td>
<td>27s</td>
<td>1m35s</td>
<td>55s</td>
<td>53s</td>
<td>1s</td>
<td>2s</td>
<td>57s</td>
</tr>
</tbody>
</table>

1. Impact of the target latency on the first step of the generation
Timings for certified code generation

<table>
<thead>
<tr>
<th>Degree ($d_x,d_y$)</th>
<th>$x^{1/2}$</th>
<th>$x^{-1/2}$</th>
<th>$x^{1/3}$</th>
<th>$x^{-1/3}$</th>
<th>$\log_2(1 + x)$</th>
<th>$\frac{1}{\sqrt{1 + x^2}}$</th>
<th>$\frac{\exp(1 + x)}{1 + x}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(8,1)</td>
<td>(9,1)</td>
<td>(6,0)</td>
<td>(7,0)</td>
<td>(10,0)</td>
<td></td>
</tr>
<tr>
<td>Target / Minimal latency</td>
<td>13 / 13</td>
<td>13 / ?</td>
<td>16 / 16</td>
<td>16 / 16</td>
<td>10 / 11</td>
<td>10 / 11</td>
<td>13 / 13</td>
</tr>
<tr>
<td>Achieved latency</td>
<td>13</td>
<td>14</td>
<td>16</td>
<td>16</td>
<td>11</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>Scheme computation</td>
<td>195ms</td>
<td>73ms</td>
<td>26s</td>
<td>25s</td>
<td>17ms</td>
<td>10ms</td>
<td>40ms</td>
</tr>
<tr>
<td>[50] [50]</td>
<td>[50] [50]</td>
<td>[50] [50]</td>
<td>[50] [50]</td>
<td>[50] [50]</td>
<td>[50] [50]</td>
<td>[50] [50]</td>
<td>[50] [50]</td>
</tr>
<tr>
<td>Arithmetic operator choice</td>
<td>3ms</td>
<td>3ms</td>
<td>7ms</td>
<td>11ms</td>
<td>1ms</td>
<td>2ms</td>
<td>3ms</td>
</tr>
<tr>
<td>[35] [29]</td>
<td>[30] [26]</td>
<td>[30] [26]</td>
<td>[2] [12]</td>
<td>[27]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling checking</td>
<td>16s</td>
<td>1m33s</td>
<td>43ms</td>
<td>439ms</td>
<td>2ms</td>
<td>64ms</td>
<td>49s</td>
</tr>
<tr>
<td>Certification (Gappa)</td>
<td>10s</td>
<td>1s</td>
<td>27s</td>
<td>27s</td>
<td>230ms</td>
<td>1s</td>
<td>7s</td>
</tr>
<tr>
<td>Total time ($\approx$)</td>
<td>27s</td>
<td>1m35s</td>
<td>55s</td>
<td>53s</td>
<td>1s</td>
<td>2s</td>
<td>57s</td>
</tr>
</tbody>
</table>

1. Impact of the target latency on the first step of the generation
2. What may dominate the cost: scheduling and certification using Gappa
### Timings for certified code generation

<table>
<thead>
<tr>
<th>Degree ((d_x,d_y))</th>
<th>(x^{1/2} )</th>
<th>(x^{-1/2} )</th>
<th>(x^{1/3} )</th>
<th>(x^{-1/3} )</th>
<th>(\log_2(1+x))</th>
<th>(\frac{1}{\sqrt{1+x^2}})</th>
<th>(\frac{\exp(1+x)}{1+x})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target / Minimal latency</td>
<td>13 / 13</td>
<td>16 / 16</td>
<td>10 / 11</td>
<td>13 / 13</td>
<td>11</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>Achieved latency</td>
<td>13</td>
<td>16</td>
<td>11</td>
<td>11</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheme computation</td>
<td>195ms</td>
<td>26s</td>
<td>17ms</td>
<td>10ms</td>
<td>40ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic operator choice</td>
<td>3ms</td>
<td>7ms</td>
<td>1ms</td>
<td>2ms</td>
<td>3ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling checking</td>
<td>16s</td>
<td>43ms</td>
<td>2ms</td>
<td>64ms</td>
<td>49s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Certification (Gappa)</td>
<td>10s</td>
<td>27s</td>
<td>230ms</td>
<td>1s</td>
<td>7s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total time ((\approx))</td>
<td>27s</td>
<td>55s</td>
<td>1s</td>
<td>2s</td>
<td>57s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Impact of the target latency on the first step of the generation
2. What may dominate the cost: scheduling and certification using Gappa
3. Optimality of some generated codes, in terms of evaluation latency
Outline of the talk

1. Background on polynomial evaluation

2. The CGPE tool

3. Experimental results

4. Conclusion
Conclusions

- Code generation for fast and certified polynomial evaluation
  - in fixed-point arithmetic
  - methodologies and tools to automate polynomial evaluation implementation
  - heuristics and techniques for generating quickly fast and certified C codes
  - implemented in the tool CGPE (Code Generation for Polynomial Evaluation)

  http://cgpe.gforge.inria.fr/

- Speed-up significantly the development time of mathematical library
  - CGPE: allows to write and certify automatically \(\approx 50\%\) of the codes of FLIP
Current work and perspectives

■ Current work
  ▶ precomputation in order to help the DAG set computation in choosing the appropriate splittings: the ones leading to DAGs with optimal latency on unbounded parallelism
  ▶ earlier DAG elimination, by checking accuracy during generation step
Current work and perspectives

- **Current work**
  - precomputation in order to help the DAG set computation in choosing the appropriate splittings: the ones leading to DAGs with optimal latency on unbounded parallelism
  - earlier DAG elimination, by checking accuracy during generation step

- **Perspectives**
  - extend CGPE to handle floating-point arithmetic,
  - make CGPE more general to tackle other problems, like evaluation of a polynomial at a matrix point.
Horner’s rule is uniquely optimal.
Theory of machines and computations, pages 45–58.

Automating custom-precision function evaluation for embedded processors.
In CASES’05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, pages 22–31, New York, NY, USA. ACM.

Faster Math Functions.
Tutorial at Game Developers Conference.

The computation of transcendental functions on the IA-64 architecture.

Optimized Custom Precision Function Evaluation for Embedded Processors.

LEMA: towards a language for reliable arithmetic.

ACM Communications in Computer Algebra, 44:41–52.

Enhancing the Implementation of Mathematical Formulas for Fixed-Point and Floating-Point Arithmetics.


Otter, R. (1948).
The number of trees.
Methods of Computing Values of Polynomials.  

Analyse et implantation d’algorithmes rapides pour l’évaluation polynomiale sur les nombres flottants.  