Groupe de travail PEQUAN (LIP6, UPMC)
Paris, France, July 7, 2011

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)

UPVD
Université de Perpignan Via Domitia

Laboratoire
d'Informatique
de Robotique
et de Microélectronique
de Montpellier
LIRMM

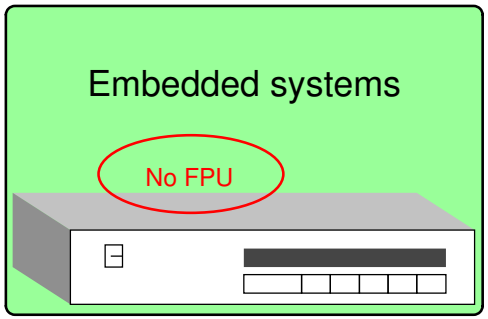
Joint work with Christophe Mouilleron (LIP, ENS Lyon)

Motivation

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

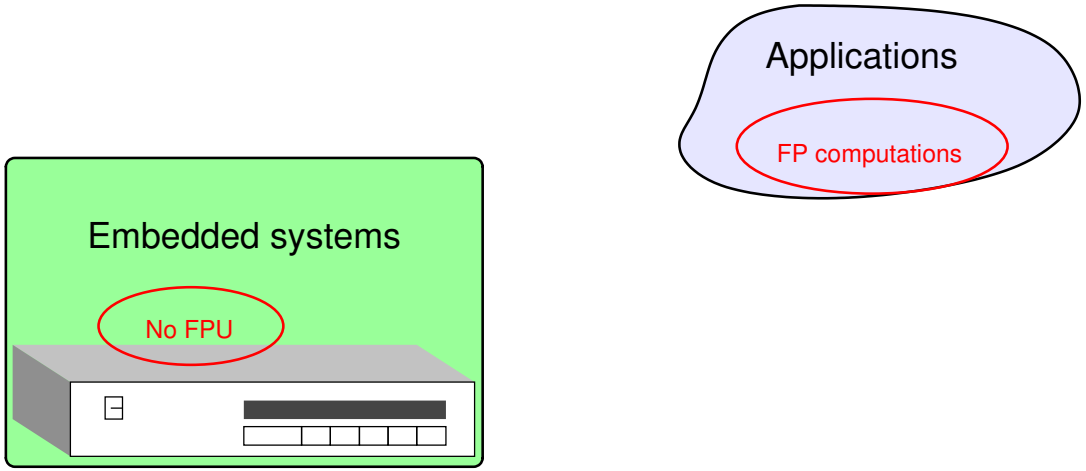
Motivation

- 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

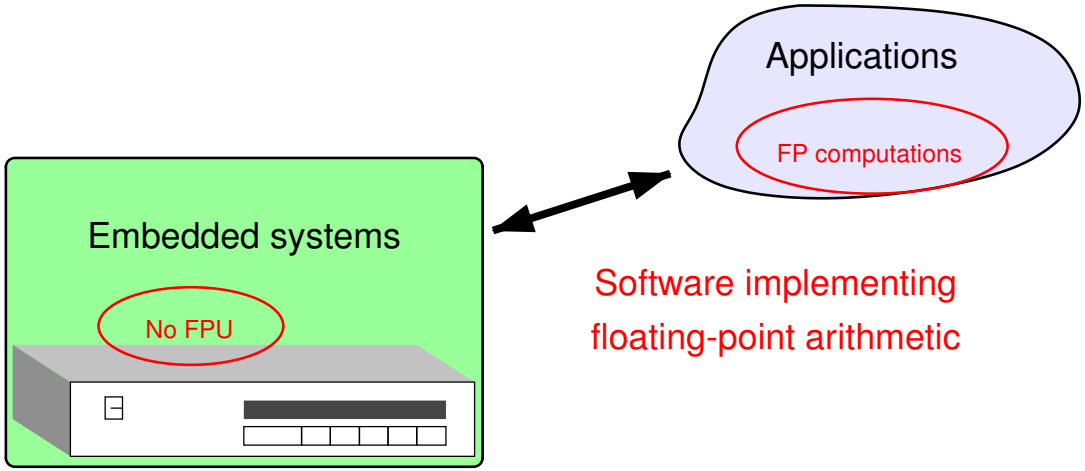
- Embedded systems are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost
- Some embedded systems do not have any FPU (floating-point unit)



- 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)

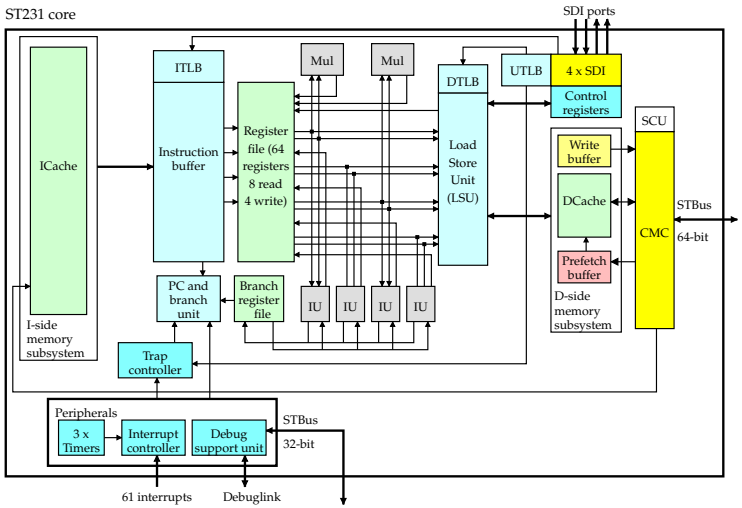


- Highly used in audio and video applications
  - demanding on floating-point computations

On the one side: the IEEE 754-2008 standard, ...

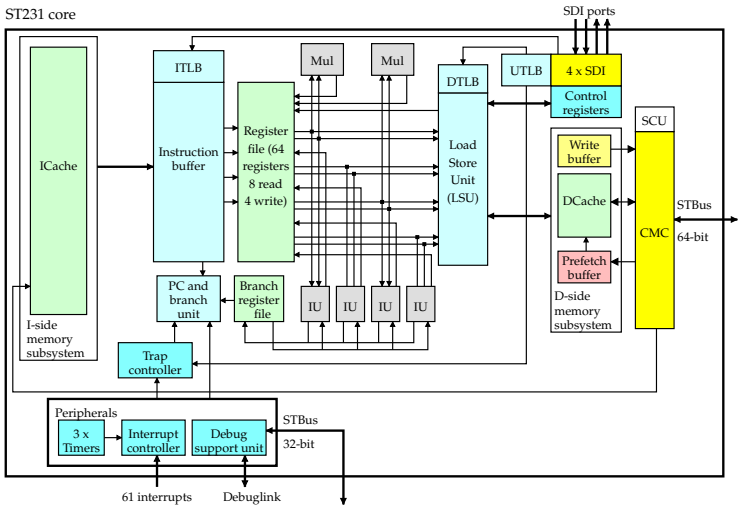
- Definition of IEEE floating-point arithmetic
  - floating-point formats: single precision, double precision, ...
  - special values: ± 0, $\pm\infty$, NaN
  - 4 rounding modes: to nearest even, upward, downward, and toward zero
  - mathematical function behavior
    - $\rightarrow$ special input (ex: $\sqrt{-0} = -0$)
    - $\rightarrow$ 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
  - $\longrightarrow$ no FPU
- Parallel execution unit
  - 4 integer ALUs
  - 2 pipelined multipliers $32 \times 32 \longrightarrow 32$
- Latencies: ALU $=$ 1 cycle / Mul $=$ 3 cycles

... on the other side: the ST231 processor



- 4-issue VLIW 32-bit integer processor
  - $\longrightarrow$ no FPU
- Parallel execution unit
  - 4 integer ALUs
  - 2 pipelined multipliers $32 \times 32 \longrightarrow 32$
- Latencies: ALU $=$ 1 cycle / Mul $=$ 3 cycles

- VLIW (Very Long Instruction Word)
  - $\longrightarrow$ instructions grouped into bundles
  - $\longrightarrow$ 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 ?

  } 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 ?

  } automation and certification

- 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

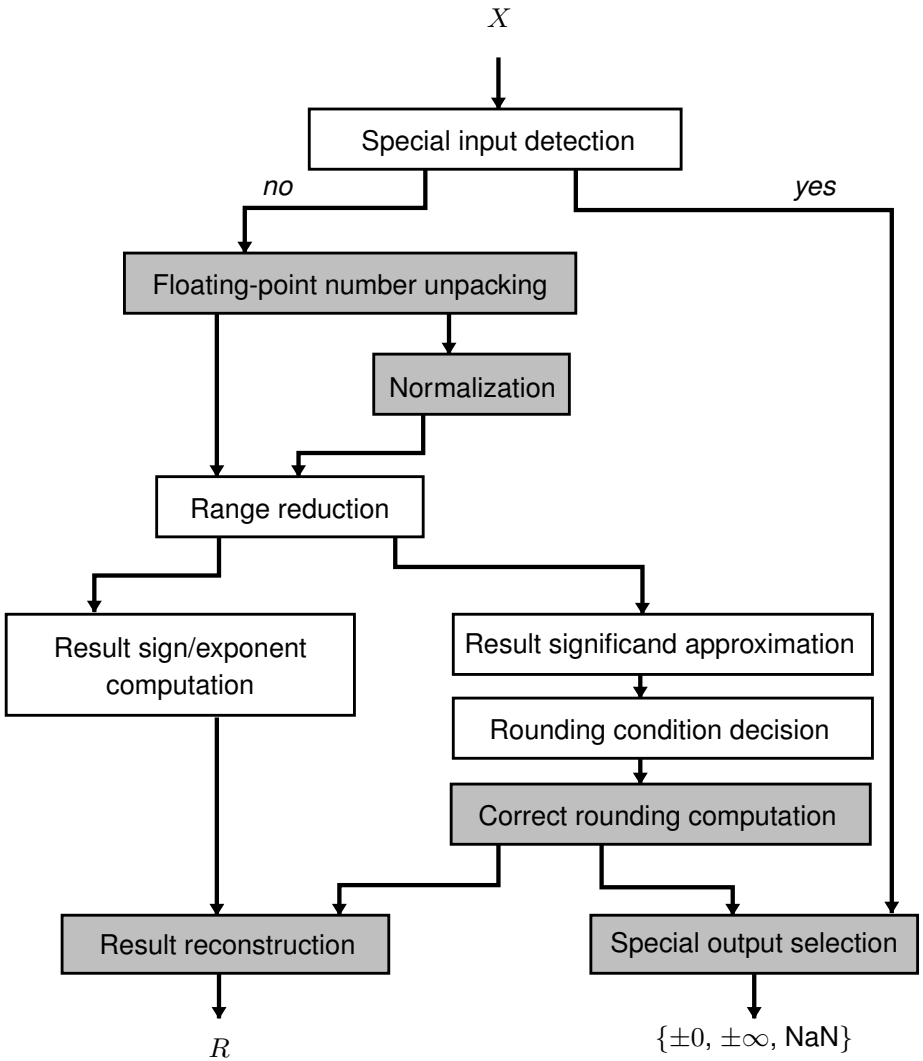
Can we teach computers to write fast libraries?

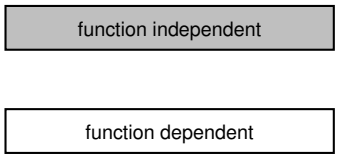
- Our tool: CGPE (Code Generation for Polynomial Evaluation)

*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

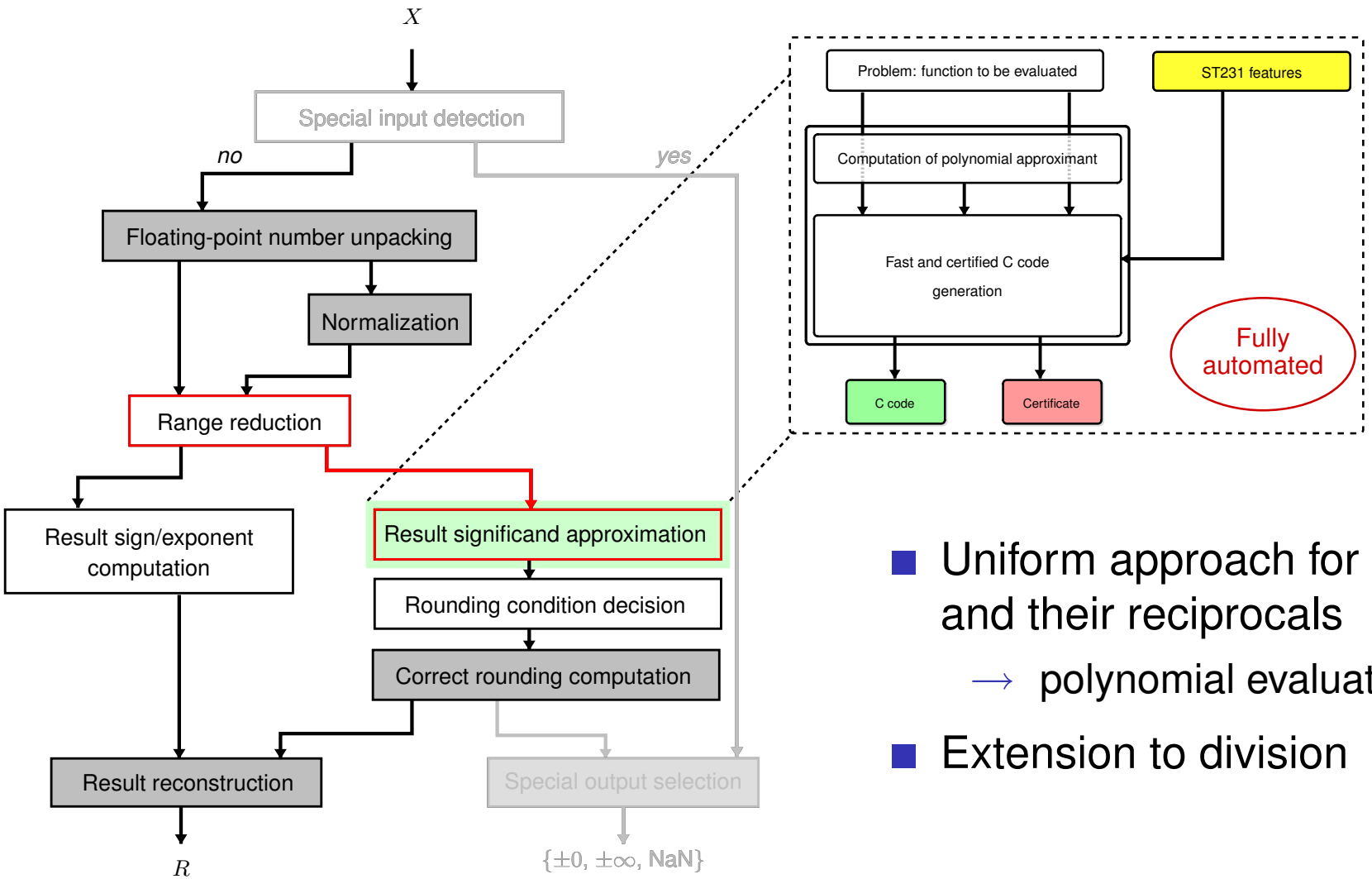




Objectives

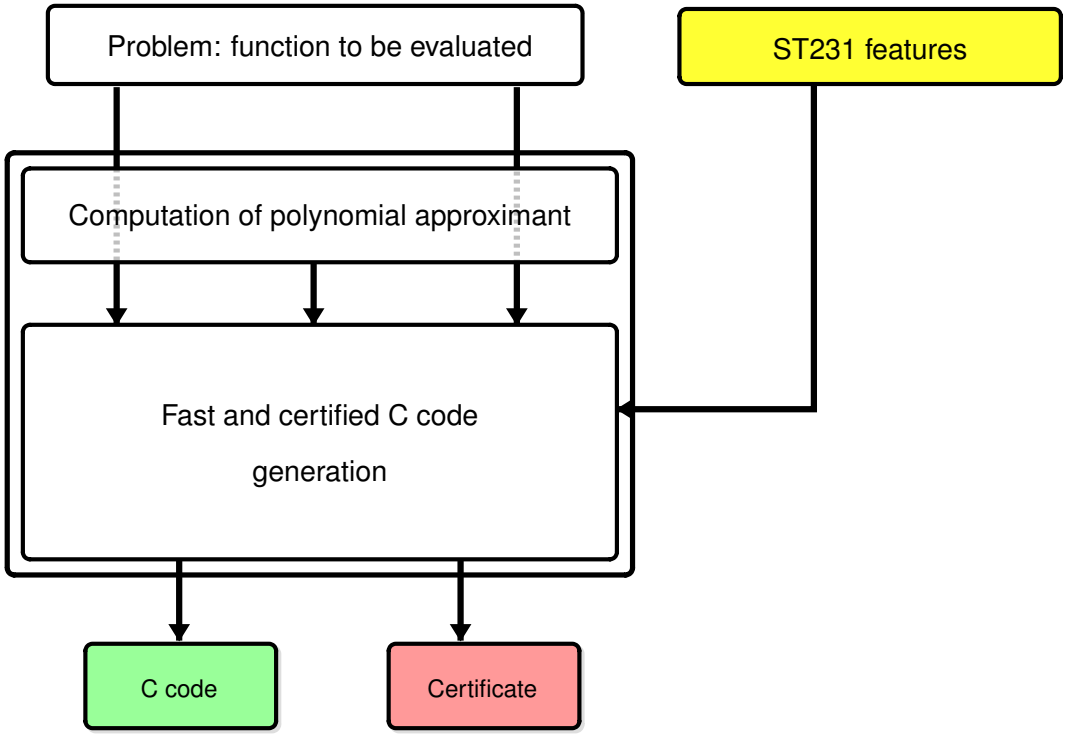
- → Low latency, correctly-rounded implementations
- → ILP exposure

Basic blocks for implementing correctly-rounded operators

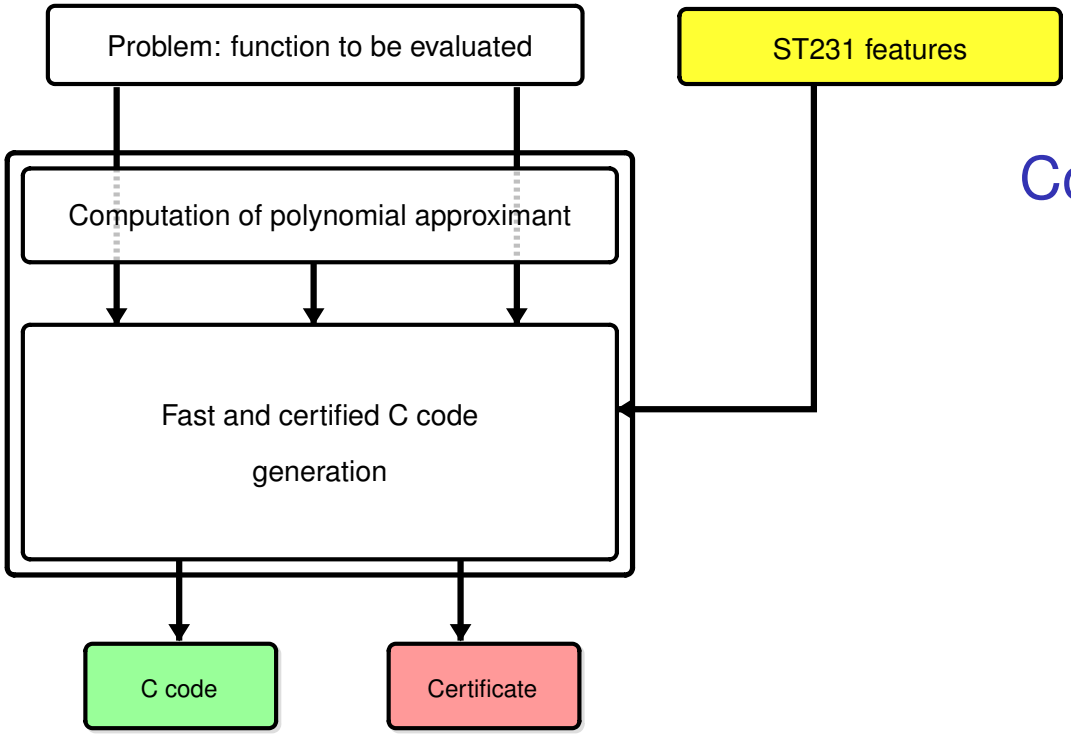


- Uniform approach for *n*th roots and their reciprocals
  - $\longrightarrow$ polynomial evaluation
- Extension to division

Flowchart for generating efficient and certified C codes



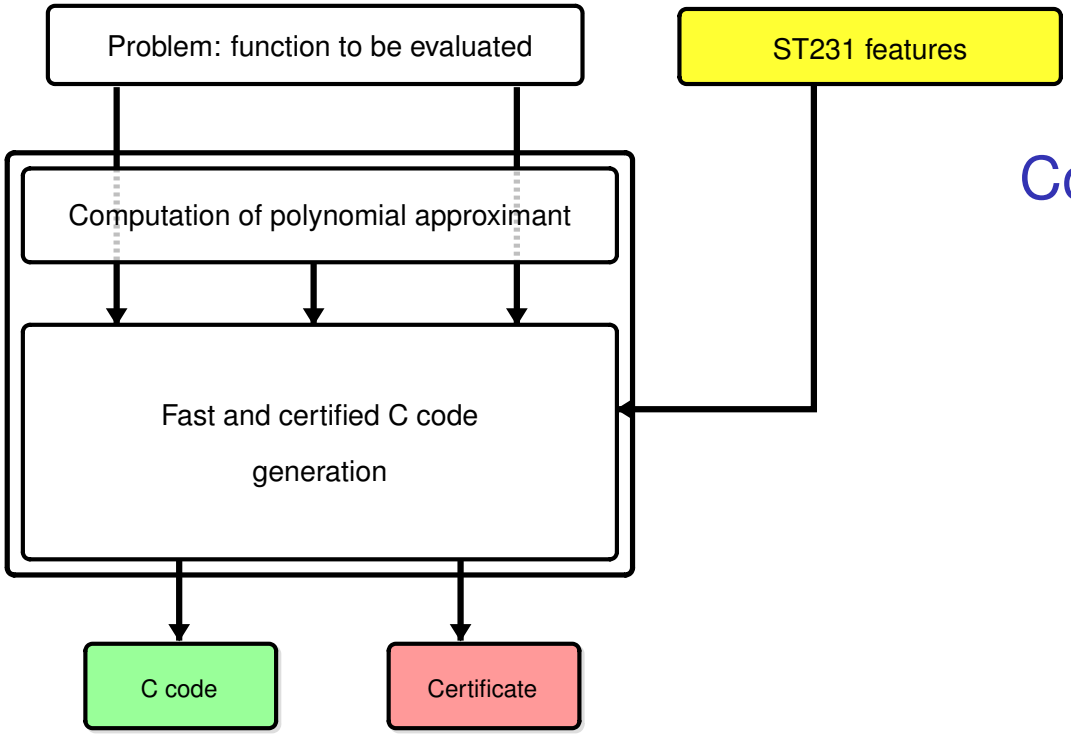
Flowchart for generating efficient and certified C codes



Constraints

- Accuracy of approximant and C code

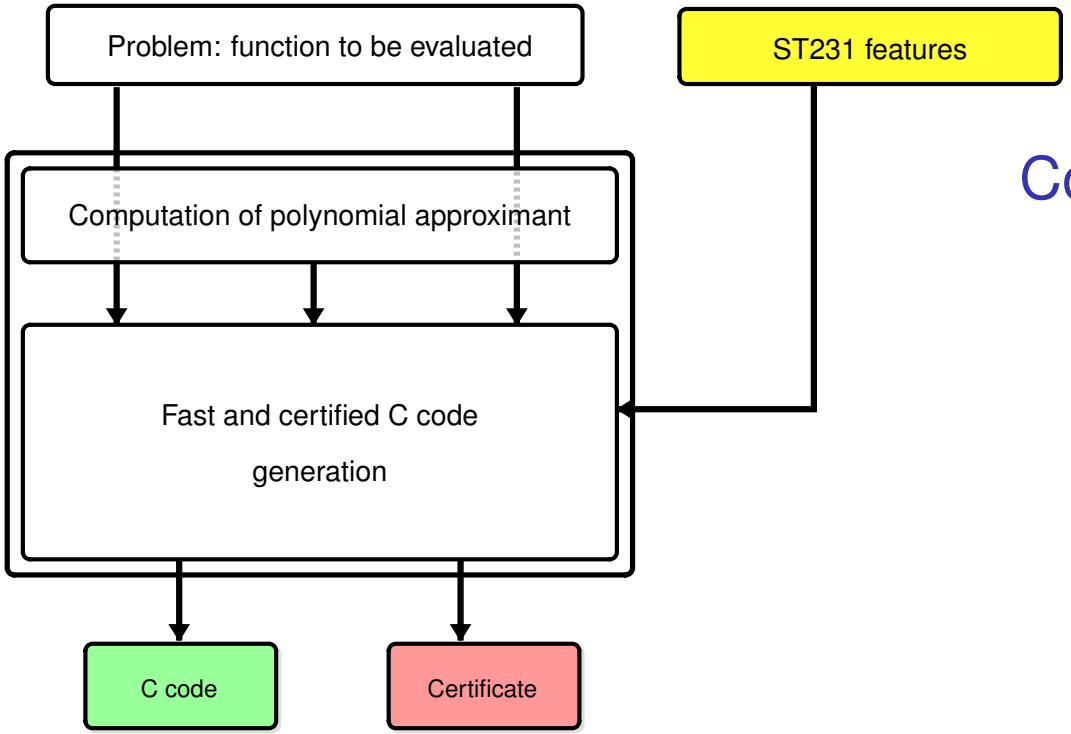
Flowchart for generating efficient and certified C codes



Constraints

- Accuracy of approximant and C code
- Low evaluation latency on ST231, ILP exposure

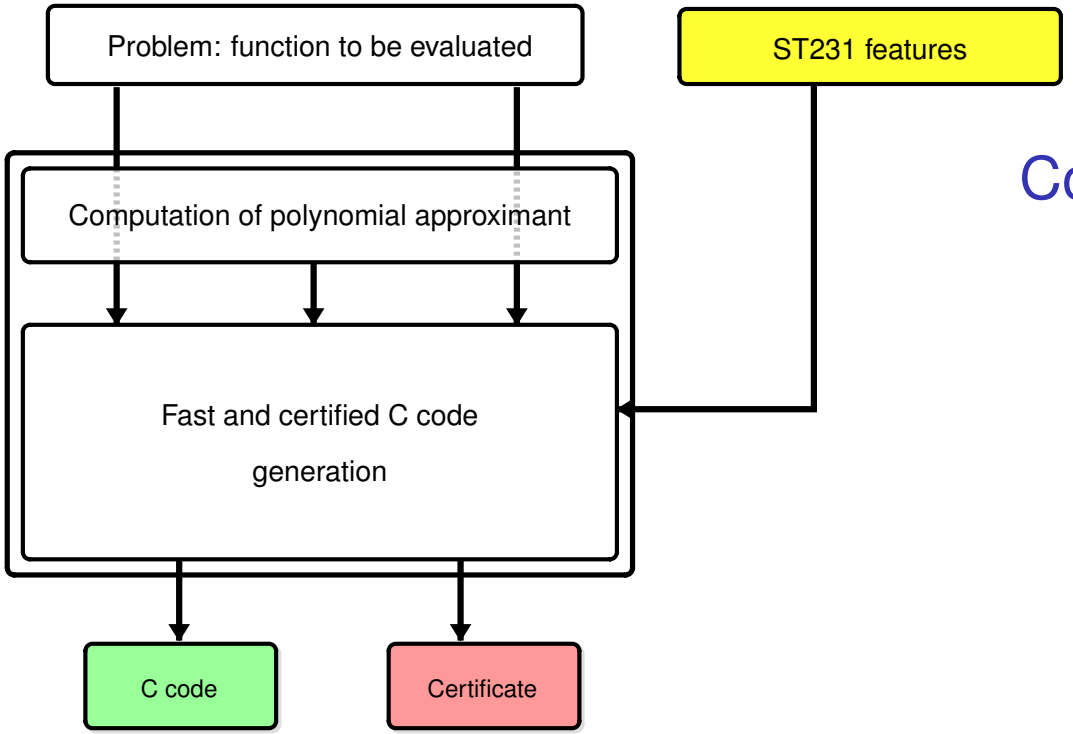
Flowchart for generating efficient and certified C codes



Constraints

- Accuracy of approximant and C code
  - Sollya
- Low evaluation latency on ST231, ILP exposure

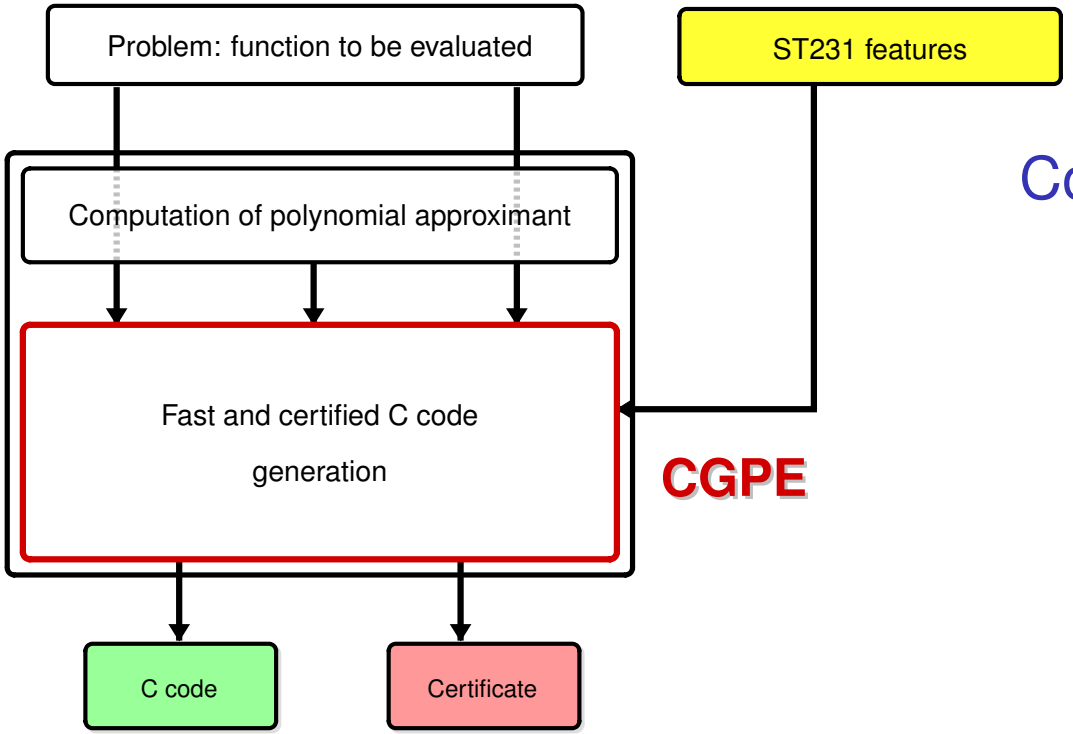
Flowchart for generating efficient and certified C codes



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



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

Our objective

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

Our objective

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

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

Our objective

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

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

- Two families of algorithms
  - algorithms with coefficient adaptation: Knuth and Eve (60's), Paterson and Stockmeyer (1973), ...
    - $\longrightarrow$ ill-suited in the context of fixed-point arithmetic
  - algorithms without coefficient adaptation

Our objective

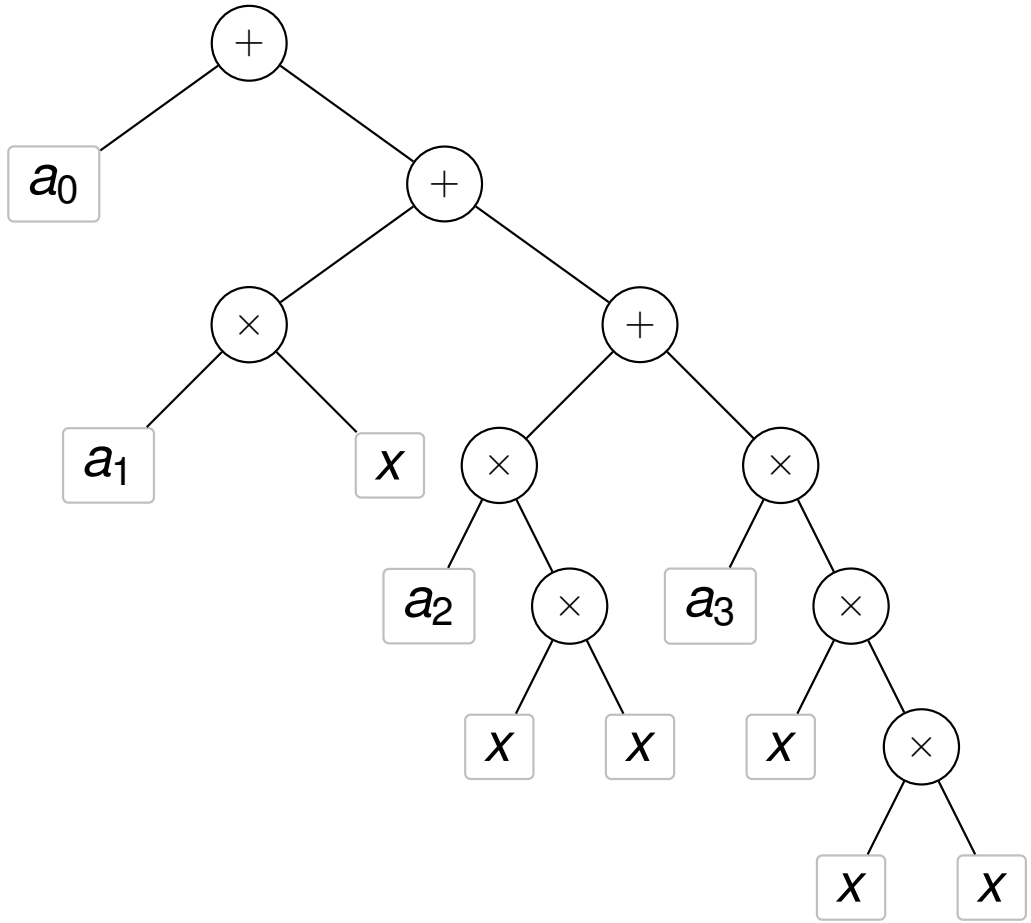
- Compute fast and certified schemes for evaluating a polynomial $P(x, y) = \alpha + y \cdot a(x)$
  - $\longrightarrow$ using only additions and multiplications
  - $\longrightarrow$ reducing the evaluation latency on unbounded parallelism

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

- Two families of algorithms
  - algorithms with coefficient adaptation: Knuth and Eve (60's), Paterson and Stockmeyer (1973), ...
    - $\longrightarrow$ ill-suited in the context of fixed-point arithmetic
  - algorithms without coefficient adaptation

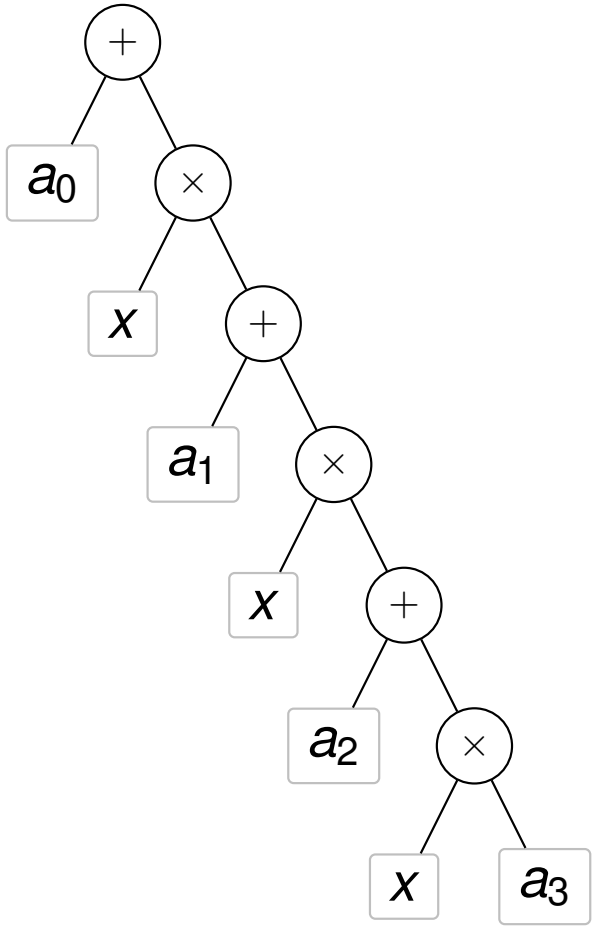
Classical evaluation schemes

- Naive evaluation
  - ↪ 3 additions
    5 multiplications
  - ↪ latency: 12 cycles



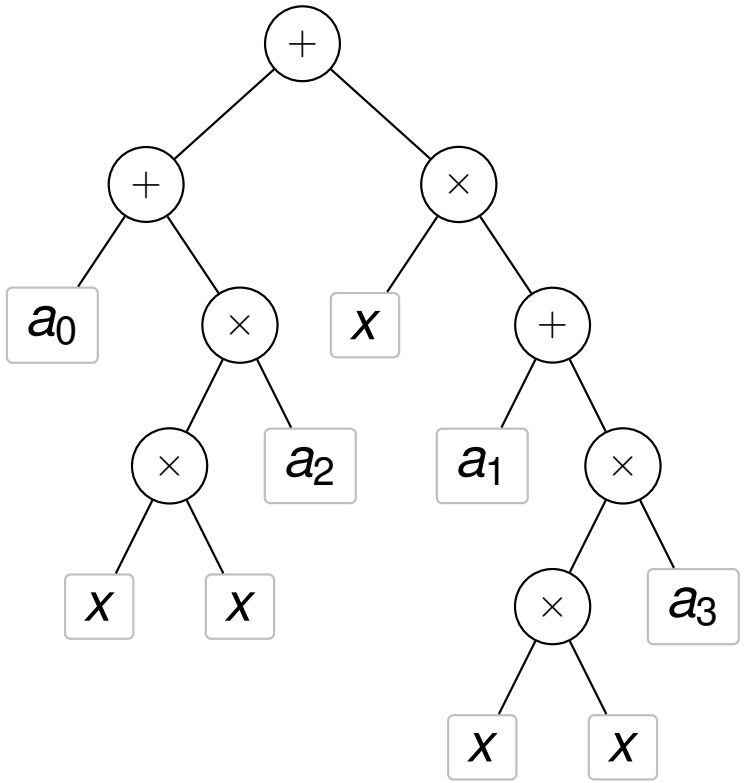
Classical evaluation schemes

- Horner's rule
  - ↪ 3 additions
    3 multiplications
  - ↪ latency: 12 cycles
  - ⊕ optimal in terms of multiplication number
    [Pan, 1966], [Borodin, 1971],
  - ⊖ fully sequential



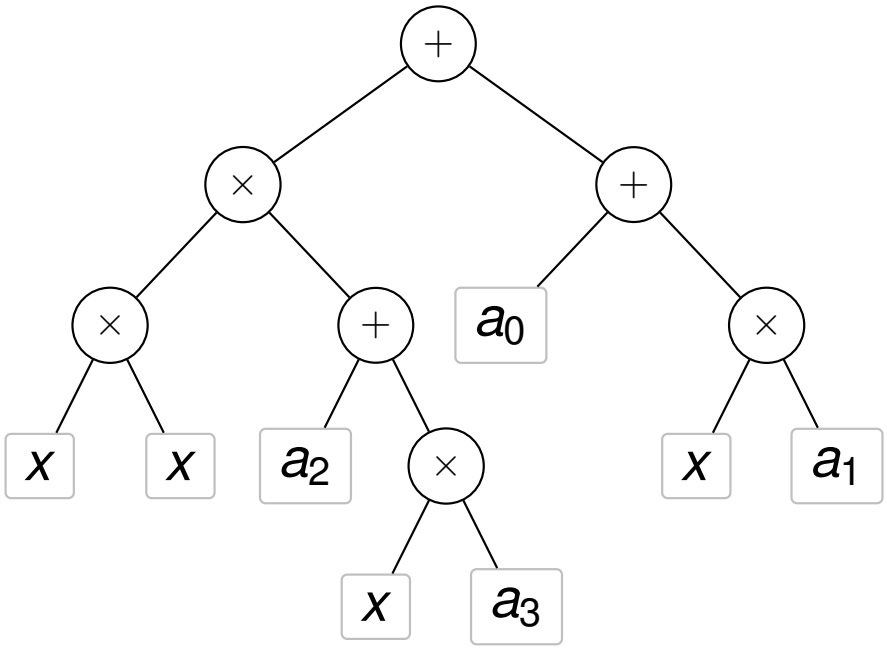
Classical evaluation schemes

- 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 $a_0 + a_1 \cdot x + a_2 \cdot x^2$ 1

Common *parenthesizations* $(a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x))$ 1

Definition of evaluation schemes

Mathematical expression $a_0 + a_1 \cdot x + a_2 \cdot x^2$ 1

Common *parenthesizations* $(a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x))$ 1

commutativity
associativity
distributivity

Feasible *parenthesizations* $(a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x))$ 160

$(a_0 + (a_1 \cdot x)) + ((a_2 \cdot x) \cdot x)$

$a_0 + ((a_1 \cdot x)) + ((a_2 \cdot x) \cdot x))$

$a_0 + ((a_1 + (a_2 \cdot x)) \cdot x)$

$((a_1 + (a_2 \cdot x)) \cdot x) + a_0 \quad \ldots$

Definition of evaluation schemes

Mathematical expression $a_0 + a_1 \cdot x + a_2 \cdot x^2$ 1

Common *parenthesizations* $(a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x))$ 1

commutativity
associativity
distributivity

Feasible *parenthesizations* $[\ldots]$ 160

$+$ and $\times$ are only commutative in fixed-point arithmetic

Evaluation schemes

$(a_0 + (a_1 \cdot x)) + (a_2 \cdot (x \cdot x))$
$(a_0 + (a_2 \cdot (x \cdot x))) + (a_1 \cdot x)$
$a_0 + ((a_1 \cdot x) + (a_2 \cdot (x \cdot x)))$
$a_0 + (a_1 + (a_2 \cdot x)) \cdot x$
$(a_0 + (a_1 \cdot x)) + ((a_2 \cdot x) \cdot x))$
$(a_0 + ((a_2 \cdot x) \cdot x)) + (a_1 \cdot x)$
$a_0 + ((a_1 \cdot x) + ((a_2 \cdot x) \cdot x)))$

7

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 $\Leftrightarrow$ 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 $\Leftrightarrow$ 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]

$\rightsquigarrow$ We need a tool for exploring the space of evaluation schemes.

How many schemes for evaluating a polynomial?

n	$\mu_n \to a(x)$	$\mu'_n \to \alpha + y \cdot a(x)$		
1	1	10		
2	7	481		
3	163	88384		
4	11602	57363910		
5	2334244	122657263474		
6	1304066578	829129658616013		
7	1972869433837	17125741272619781635		
8	8012682343669366	1055157310305502607244946		
9	86298937651093314877	190070917121184028045719056344		
10	2449381767217281163362301	985436908485543809474905225911916 72		

How many schemes for evaluating a polynomial?

n	$\mu_n \to a(x)$	$\mu'_n \to \alpha + y \cdot a(x)$	w_n	
1	1	10	1	
2	7	481	1	
3	163	88384	1	
4	11602	57363910	2	
5	2334244	122657263474	3	
6	1304066578	829129658616013	6	
7	1972869433837	17125741272619781635	11	
8	8012682343669366	10551573103055026072449 46	23	
9	86298937651093314877	19007091712118402804571905 6344	46	
10	2449381767217281163362301	98543690848554380947490522591191672	98	

- Two well-known special cases
  - the number of evaluation schemes for x^n [Wedderburn, Etherington]

$$w_n \sim \frac{\eta \xi^n}{n^{3/2}} \text{ or } \begin{cases} \xi \approx 2.48325 \\ \eta \approx 0.31877 \end{cases} \text{ [Otter, 1948],}$$

How many schemes for evaluating a polynomial?

n	$\mu_n \to a(x)$	$\mu'_n \to \alpha + y \cdot a(x)$	w_n	$(2n-1)!!$
1	1	10	1	1
2	7	481	1	3
3	163	88384	1	15
4	11602	57363910	2	105
5	2334244	122657263474	3	945
6	1304066578	829129658616013	6	10395
7	1972869433837	17125741272619781635	11	135135
8	8012682343669366	10551573103055026072449 46	23	2027025
9	86298937651093314877	19007091712118402804571905 6344	46	34459425
10	2449381767217281163362301	98543690848554380947490522591191672	98	654729075

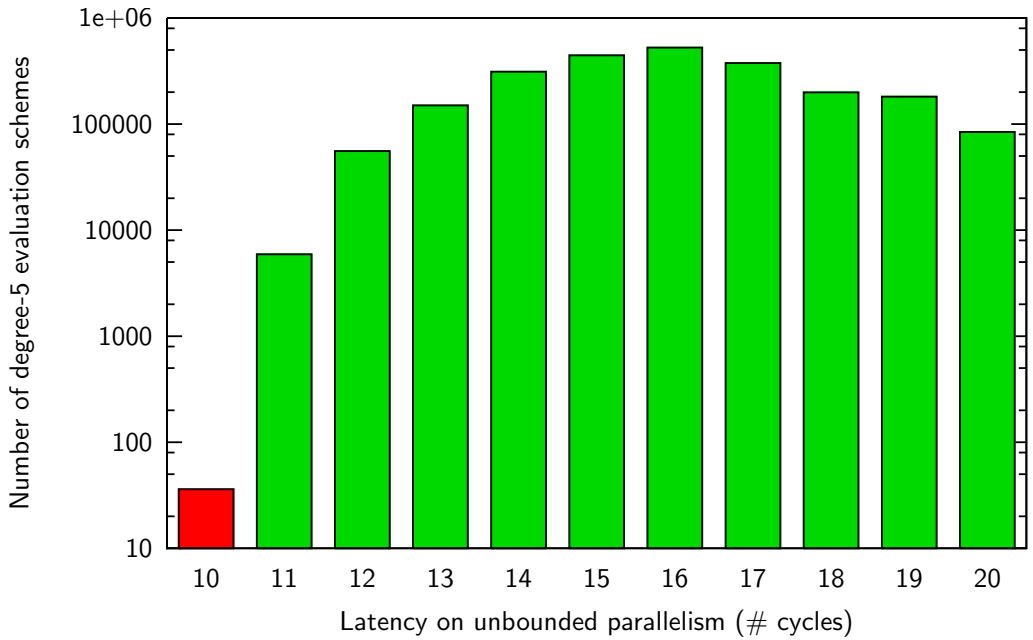
- Two well-known special cases
  - the number of evaluation schemes for x^n [Wedderburn, Etherington]

$$w_n \sim \frac{\eta \xi^n}{n^{3/2}} \text{ or } \begin{cases} \xi \approx 2.48325 \\ \eta \approx 0.31877 \end{cases} \text{ [Otter, 1948],}$$

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

Schemes of low evaluation latency

- What is the latency of degree-5 evaluation schemes?



- Total number of schemes: 2334244
  - ⇝ minimal latency for degree-5 univariate polynomial: 10 cycles
  - ⇝ number of schemes of minimal latency: 36

Outline of the talk

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:
  - $\rightsquigarrow$ fast = that reduces the evaluation latency on a given target
  - $\rightsquigarrow$ 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
   - straightline polynomial evaluation program
   - “C code certification” using Gappa
     - $\rightsquigarrow$ we can bound the evaluation error in integer arithmetic

Outline of the talk

Timings for certified code generation

	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$\log_2(1+x)$	$\frac{1}{\sqrt{1+x^2}}$	$\frac{\exp(1+x)}{1+x}$
Degree (d_x, d_y)	(8,1)	(9,1)	(8,1)	(9,1)	(6,0)	(7,0)	(10,0)
Target / Minimal latency	13 / 13	13 / ?	16 / 16	16 / 16	10 / 11	10 / 11	13 / 13
Achieved latency	13	14	16	16	11	11	13
Scheme computation	195ms [50]	73ms [50]	26s [50]	25s [50]	17ms [50]	10ms [50]	40ms [50]
Arithmetic operator choice	3ms [35]	3ms [29]	7ms [30]	11ms [26]	1ms [2]	2ms [12]	3ms [27]
Scheduling checking	16s [11]	1m33s [1]	43ms [30]	439ms [24]	2ms [1]	64ms [5]	49s [5]
Certification (Gappa)	10s [11]	1s [1]	27s [30]	27s [24]	230ms [1]	1s [5]	7s [4]
Total time ($\approx$)	27s	1m35s	55s	53s	1s	2s	57s

Timings for certified code generation

	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$\log_2(1+x)$	$\frac{1}{\sqrt{1+x^2}}$	$\frac{\exp(1+x)}{1+x}$
Degree (d_x, d_y)	(8,1)	(9,1)	(8,1)	(9,1)	(6,0)	(7,0)	(10,0)
Target / Minimal latency	13 / 13	13 / ?	16 / 16	16 / 16	10 / 11	10 / 11	13 / 13
Achieved latency	13	14	16	16	11	11	13
Scheme computation	195ms [50]	73ms [50]	26s [50]	25s [50]	17ms [50]	10ms [50]	40ms [50]
Arithmetic operator choice	3ms [35]	3ms [29]	7ms [30]	11ms [26]	1ms [2]	2ms [12]	3ms [27]
Scheduling checking	16s [11]	1m33s [1]	43ms [30]	439ms [24]	2ms [1]	64ms [5]	49s [5]
Certification (Gappa)	10s [11]	1s [1]	27s [30]	27s [24]	230ms [1]	1s [5]	7s [4]
Total time ($\approx$)	27s	1m35s	55s	53s	1s	2s	57s

1. Impact of the target latency on the first step of the generation

Timings for certified code generation

	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$\log_2(1+x)$	$\frac{1}{\sqrt{1+x^2}}$	$\frac{\exp(1+x)}{1+x}$
Degree (d_x, d_y)	(8,1)	(9,1)	(8,1)	(9,1)	(6,0)	(7,0)	(10,0)
Target / Minimal latency	13 / 13	13 / ?	16 / 16	16 / 16	10 / 11	10 / 11	13 / 13
Achieved latency	13	14	16	16	11	11	13
Scheme computation	195ms [50]	73ms [50]	26s [50]	25s [50]	17ms [50]	10ms [50]	40ms [50]
Arithmetic operator choice	3ms [35]	3ms [29]	7ms [30]	11ms [26]	1ms [2]	2ms [12]	3ms [27]
Scheduling checking	16s [11]	1m33s [1]	43ms [30]	439ms [24]	2ms [1]	64ms [5]	49s [5]
Certification (Gappa)	10s [11]	1s [1]	27s [30]	27s [24]	230ms [1]	1s [5]	7s [4]
Total time ($\approx$)	27s	1m35s	55s	53s	1s	2s	57s

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

	$x^{1/2}$	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$\log_2(1+x)$	$\frac{1}{\sqrt{1+x^2}}$	$\frac{\exp(1+x)}{1+x}$
Degree (d_x, d_y)	(8,1)	(9,1)	(8,1)	(9,1)	(6,0)	(7,0)	(10,0)
Target / Minimal latency	13 / 13	13 / ?	16 / 16	16 / 16	10 / 11	10 / 11	13 / 13
Achieved latency	13	14	16	16	11	11	13
Scheme computation	195ms [50]	73ms [50]	26s [50]	25s [50]	17ms [50]	10ms [50]	40ms [50]
Arithmetic operator choice	3ms [35]	3ms [29]	7ms [30]	11ms [26]	1ms [2]	2ms [12]	3ms [27]
Scheduling checking	16s [11]	1m33s [1]	43ms [30]	439ms [24]	2ms [1]	64ms [5]	49s [5]
Certification (Gappa)	10s [11]	1s [1]	27s [30]	27s [24]	230ms [1]	1s [5]	7s [4]
Total time ($\approx$)	27s	1m35s	55s	53s	1s	2s	57s

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

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.

Borodin, A. (1971).

Horner's rule is uniquely optimal.

Theory of machines and computations, pages 45–58.

Cheung, R. C. C., Lee, D.-U., Mencer, O., Luk, W., and Cheung, P. Y. K. (2005).

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.

Green, R. (2002).

Faster Math Functions.

Tutorial at Game Developers Conference.

Harrison, J., Kubaska, T., Story, S., and Tang, P. (1999).

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

Intel Technology Journal, 1999-Q4:1–7.

Lee, D.-U. and Villasenor, J. D. (2009).

Optimized Custom Precision Function Evaluation for Embedded Processors.

IEEE Transactions on Computers, 58(1):46–59.

Lefèvre, V., Théveny, P., de Dinechin, F., Jeannerod, C.-P., Mouilleron, C., Pfannholzer, D., and Revol, N. (2010).

LEMA: towards a language for reliable arithmetic.

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

Martel, M. (2009).

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

In Journal of Formal Methods in System Design, volume 35, pages 265–278. Springer.

Mouilleron, C. and Revy, G. (2011).

Automatic Generation of Fast and Certified Code for Polynomial Evaluation.

In Proc. of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20), Tuebingen, Germany.

Otter, R. (1948).

The number of trees.

The Annals of Mathematics, 49(3):pp. 583–599.

Pan, V. Y. (1966).

Methods of Computing Values of Polynomials.

Russian Mathematical Surveys, 21(1):105–136.

Revy, G. (2006).

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

Master's thesis, École normale supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France.