ANR MetaLibm kick-off meeting
Lyon, 22 January, 2014

Automatic Synthesis of Fast and Certified Code for Polynomial Evaluation

through the example of the CGPE tool

Guillaume Revy

Équipe-projet DALI, Univ. Perpignan Via Domitia
LIRMM, CNRS: UMR 5506 - Univ. Montpellier 2

UPVD
Université de Perpignan Via Domitia

Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier

Context of CGPE

- This work takes mainly part in the context of the development of FLIP
  - software support for binary32 floating-point arithmetic on integer processors
- In this talk, we will focus on polynomial evaluation
  - it frequently appears as a building block of some mathematical operator implementation, typically in FLIP
- Current challenge: tools and methodologies for the automatic synthesis of fast and certified programs
  - optimized for a given format, for the target architecture

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

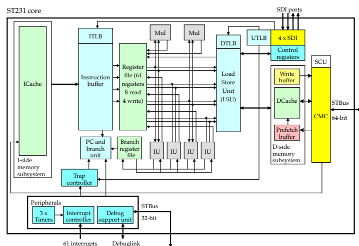
Definition of IEEE floating-point arithmetic

- 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
  - ⇝ special input (ex: $\sqrt{-0} = -0$)
  - ⇝ requires / recommends correct rounding

Motivation:

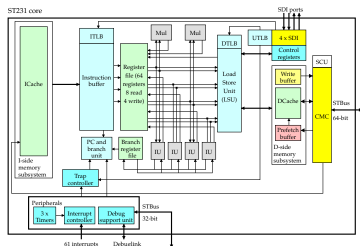
- make computations reproducible
- and make results architecture-independent

... 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: ALU = 1 cycle / Mul = 3 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 $32 \times 32 \rightarrow 32$
- 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

Our objective

- Compute fast and certified schemes for evaluating a polynomial, such as

$$P(x,y) = \alpha + y \cdot a(x)$$

  - using only additions and multiplications
  - reducing the evaluation latency on unbounded parallelism

Our objective

- Compute fast and certified schemes for evaluating a polynomial, such as

$$P(x,y) = \alpha + y \cdot a(x)$$

  - 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 fast as possible

Our objective

- Compute fast and certified schemes for evaluating a polynomial, such as

$$P(x,y) = \alpha + y \cdot a(x)$$

  - 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 fast as possible

- Two families of algorithms
  - algorithms with coefficient adaptation: Knuth and Eve (1964), Paterson and Stockmeyer (1973), ...
    - $\rightsquigarrow$ ill-suited in the context of fixed-point arithmetic
  - algorithms without coefficient adaptation

Our objective

- Compute fast and certified schemes for evaluating a polynomial, such as

$$P(x,y) = \alpha + y \cdot a(x)$$

  - 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 fast as possible

- Two families of algorithms
  - algorithms with coefficient adaptation: Knuth and Eve (1964), Paterson and Stockmeyer (1973), ...
    - $\rightsquigarrow$ ill-suited in the context of fixed-point arithmetic
  - algorithms without coefficient adaptation

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

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

$\rightsquigarrow$ 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	98543690848554380947490522591191672		

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	1055157310305502607244946	23	
9	86298937651093314877	190070917121184028045719056344	46	
10	2449381767217281163362301	98543690848554380947490522591191672	98	

- Two well-known special cases
  - the number of evaluation schemes for x^n

$$w_n \sim \frac{\eta \xi^n}{n^{3/2}} \quad \text{or} \quad \begin{cases} \xi \approx 2.48325 \\ \eta \approx 0.31877 \end{cases}$$

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

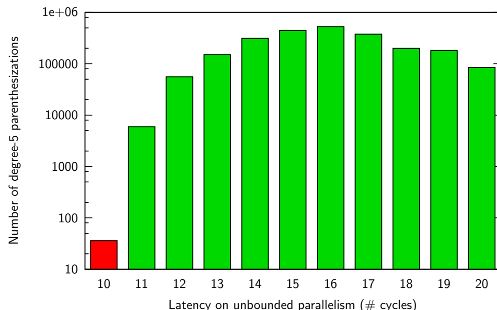
- Two well-known special cases
  - the number of evaluation schemes for x^n

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

  - the number of evaluation schemes for $\sum_{i=0}^{n} a_i$ 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

Overview of CGPE

- Goal of CGPE: automate the design of fast and certified C codes for evaluating univariate or bivariate polynomials in fixed-point arithmetic
  - by using unsigned fixed-point arithmetic only
  - by using the target architecture features (as much as possible)

- Remarks on CGPE
  - fast $\rightsquigarrow$ that reduce the evaluation latency on a given target
  - certified $\rightsquigarrow$ for which we can bound the error entailed by the evaluation within the given target's arithmetic

Global architecture of CGPE

- Input of CGPE

```
cgpe --degree="[8,1]" --xml-input=cgpe-test1.xml --coefs="[1000000001111111]"             \
     --latency=lowest --gappa-certificate --mpfi-error-bound --output --register=32      \
     --schedule="[4,2]" --max-kept=5 --operators="[11111111111111111:03333333000333330]"   \
     --optimized-search="[1,2]" --mul=3 --add=1
```

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. some architectural constraints: operator cost, parallelism level, ...

```
<polynomial>
  <coefficient value="0x00000020" sign="0" integer_width="2" fraction_width="30" width="32"/>
  <coefficient value="0x80000000" sign="0" integer_width="1" fraction_width="31" width="32"/>
  <coefficient value="0x40000000" sign="0" integer_width="1" fraction_width="31" width="32"/>
  <coefficient value="0x10000000" sign="1" integer_width="1" fraction_width="31" width="32"/>
  <coefficient value="0x07fe93e4" sign="0" integer_width="1" fraction_width="31" width="32"/>
  <coefficient value="0x04eef694" sign="1" integer_width="1" fraction_width="31" width="32"/>
  <coefficient value="0x032d6643" sign="0" integer_width="1" fraction_width="31" width="32"/>
  <coefficient value="0x01c6cebd" sign="1" integer_width="1" fraction_width="31" width="32"/>
  <coefficient value="0x00aebe7d" sign="0" integer_width="1" fraction_width="31" width="32"/>
  <coefficient value="0x00200000" sign="1" integer_width="1" fraction_width="31" width="32"/>
  <variable inf="0x00000000" sup="0xfffffe00" sign="0" integer_width="0" fraction_width="32"
            precision="23" width="32"/>
  <variable inf="0x80000000" sup="0xb504f334" sign="0" integer_width="1" fraction_width="31"
            width="32" delay="2"/>
  <error value="25081373483158693012463053528118040380976733198921b-191" strict="false" type="absolute"/>
</polynomial>
```

Global architecture of CGPE (cont'd)

- Output of CGPE

```
// Operator latency:  1-cycle addition/subtraction
//                    3-cycle pipelined multiplication

// Remark: mul ~~> 32 x 32 -> 32 bits

uint32_t func_0x1a52b10(uint32_t T, uint32_t S)
{
  uint32_t r0  = T >> 2;                  // (+) Q[1.31]
  uint32_t r1  = 0x80000000 + r0;         // (+) Q[1.31]
  uint32_t r2  = mul(S, r1);              // (+) Q[2.30]
  uint32_t r3  = 0x00000020 + r2;         // (+) Q[2.30]
  uint32_t r4  = mul(T, T);               // (+) Q[0.32]
  uint32_t r5  = mul(S, r4);              // (+) Q[1.31]
  uint32_t r6  = mul(T, 0x07fe93e4);      // (+) Q[1.31]
  uint32_t r7  = 0x10000000 - r6;         // (-) Q[1.31]
  uint32_t r8  = mul(r5, r7);             // (-) Q[2.30]
  uint32_t r9  = r3 - r8;                 // (+) Q[2.30]
  uint32_t r10 = mul(r4, r4);             // (+) Q[0.32]
  uint32_t r11 = mul(S, r10);             // (+) Q[1.31]
  uint32_t r12 = mul(T, 0x032d6643);      // (+) Q[1.31]
  uint32_t r13 = 0x04eef694 - r12;        // (-) Q[1.31]
  uint32_t r14 = mul(T, 0x00aebe7d);      // (+) Q[1.31]
  uint32_t r15 = 0x01c6cebd - r14;        // (-) Q[1.31]
  uint32_t r16 = r4 >> 11;                // (-) Q[1.31]
  uint32_t r17 = r15 + r16;               // (-) Q[1.31]
  uint32_t r18 = mul(r4, r17);            // (-) Q[1.31]
  uint32_t r19 = r13 + r18;               // (-) Q[1.31]
  uint32_t r20 = mul(r11, r19);           // (-) Q[2.30]
  uint32_t r21 = r9 - r20;                // (+) Q[2.30]
  return r21;
}

// Optimal latency of 13 cycles on the ST231:
// ~~> 4-issue 32-bit VLIW integer processor
// ~~> with at most 2 multiplications per cycle
```

Listing 1   C code.

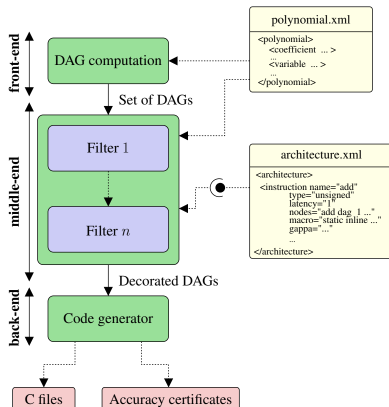
```
## Coefficients and variables definition
a0 = fixed<-30,dn>(0x00000020p-30);
a1 = fixed<-31,dn>(0x80000000p-31);
a2 = fixed<-31,dn>(0x40000000p-31);
          ...
a8 = fixed<-31,dn>(0x00aebe7dp-31);
a9 = fixed<-31,dn>(0x00200000p-31);

T = fixed<-23,dn>(var0);
S = fixed<-31,dn>(var1);

CertifiedBound =
25081373483158693012463053528118040380976733198921b-191;

## Evaluation scheme
r0 fixed<-31,dn>= T * a2;        Mr0 = T * a2;
r1 fixed<-31,dn>= a1 + r0;       Mr1 = a1 + Mr0;
          ...
r21 fixed<-30,dn>= r9 - r20;     Mr21 = Mr9 - Mr20;

## Results
{
  (
       var0 in [0x00000000p-32,0xfffffe00p-32]
    /\ T - MT in [0,0]
    /\ var1 in [0x80000000p-31,0xb504f334p-31]
    /\ S - MS in [0,0]
    ->
       r0 in [0,8388607b-24]
    /\ r0 - Mr0 in ?
          ...
    /\ r21 in [200710843b-28,2277750317b-30]
    /\ |r21 - Mr21| - CertifiedBound <= 0
    /\ CertifiedBound in ?
    /\ r21 - Mr21 in ?
  )
}
```

Listing 2   Piece of Gappa certificate.

Global architecture of CGPE (cont'd)

- Architecture of CGPE $\approx$ architecture of a compiler
  - it proceeds in three main steps

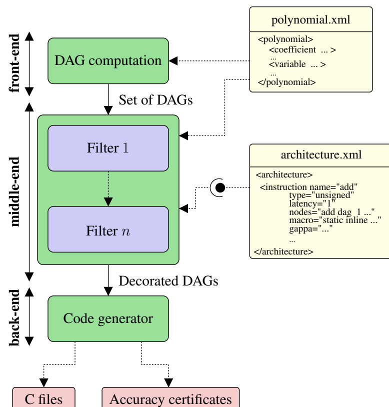
1. Computation step $\rightsquigarrow$ front-end
   - computes schemes reducing the evaluation latency on unbounded parallelism $\rightsquigarrow$ DAG
   - considers only the cost of $\oplus$ and $\otimes$



Global architecture of CGPE (cont'd)

- Architecture of CGPE ≈ architecture of a compiler
  - it proceeds in three main steps

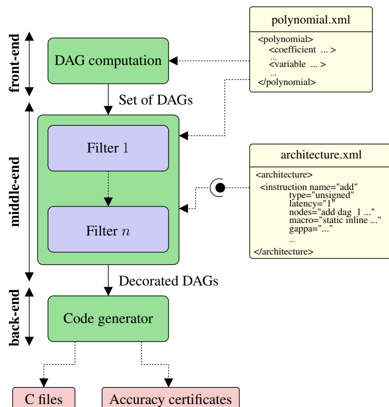
1. Computation step ⇝ front-end
   - computes schemes reducing the evaluation latency on unbounded parallelism ⇝ DAG
   - considers only the cost of ⊕ and ⊗
2. Filtering step ⇝ middle-end
   - prunes the DAGs that do not satisfy different criteria:
     - latency ⇝ scheduling filter,
     - accuracy ⇝ numerical filter, ...



Global architecture of CGPE (cont'd)

- Architecture of CGPE $\approx$ architecture of a compiler
  - it proceeds in three main steps

1. Computation step $\rightsquigarrow$ front-end
   - computes schemes reducing the evaluation latency on unbounded parallelism $\rightsquigarrow$ DAG
   - considers only the cost of $\oplus$ and $\otimes$
2. Filtering step $\rightsquigarrow$ middle-end
   - prunes the DAGs that do not satisfy different criteria:
     - latency $\rightsquigarrow$ scheduling filter,
     - accuracy $\rightsquigarrow$ numerical filter, ...
3. Generation step $\rightsquigarrow$ back-end
   - generates C codes and Gappa 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, like 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, like 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 and backends

- **Filters** implemented so far
  - arithmetic operator choice, scheduling on a simplified target model *(fully available)*
  - instruction selection, wordlength optimization (for fixed-point code implementation) *(under development)*
  - evaluation error bound checking using Gappa or MPFI *(fully available)*

- **Backends** implemented so far
  - software in fixed point arithmetic: (un)signed C type or ac-fixed type
  - hardware in fixed-point arithmetic: VHDL (heavily relying on FloPoCo)
  - accuracy certificate: Gappa or MPFI
  - for debugging purpose: binary32 or exact evaluation using GMP
  - and also: GraphViz/DOT or XML description (DEFIS output)

Improvements of current approach

- Precomputation of the splittings leading to the minimal latency on unbounded parallelism:
  - minimal latency for free
  - no need for some extra heuristic to limit the number of splittings
- Move parts of the numerical constraints from the filter to the computation step:
  - more guarantees on the generated schemes
  - in particular, we can prove the constant-sign part and get a certified error bound

$\rightsquigarrow$ Gain on the synthesis time of $\approx 15\%$ on average

Example of CGPE usage

- Let $P(x,y) = \alpha + y \cdot a(x)$ be a bivariate polynomial that approximates the function

$$2^{-25} + y \cdot \sqrt{1+x}, \quad \text{over} \quad [0, 1-2^{-23}] \times \{1, \sqrt{2}\}.$$

  with $a(x)$ a degree-8 univariate polynomial

  - coefficients ans variables stored in 32-bit words
  - data computed using Sollya, and stored in a XML input file

- How to evaluate $P(x,y)$ on the ST231 processor, with an evaluation error no greater than ε, with

$$\varepsilon \approx 2^{-26.89}?$$

Example of CGPE usage (cont'd)

DEMO

To conclude...

CGPE now freely available for download under CeCILL v2 licence.

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

Please do not hesitate to have a try...