Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks

Mohamed Amine Najahi

7ème Rencontres Arithmétiques de l'Informatique Mathématique (RAIM2015)
Rennes, 7-9 april 2015

Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks

Amine Najahi

Univ. Perpignan Via Domitia, DALI project-team
Univ. Montpellier 2, LIRMM, UMR 5506
CNRS, LIRMM, UMR 5506

Which arithmetic for computational tasks?

Floating-point computations

Fixed-point computations

Which arithmetic for computational tasks?

Floating-point computations

- ☺ Easy and fast to implement
- ☺ Easily portable [IEEE754]

Fixed-point computations

- ☹ Tedious and time consuming to implement
  - $>$ 50% of design time [Wil98]

Which arithmetic for computational tasks?

Floating-point computations

- ☺ Easy and fast to implement
- ☺ Easily portable [IEEE754]
- ☹ Requires dedicated hardware
- ☹ Slow if emulated in software

Fixed-point computations

- ☹ Tedious and time consuming to implement
  - $>50\%$ of design time [Wil98]
- ☺ Relies only on integer instructions
- ☺ Efficient

Which arithmetic for computational tasks?

Floating-point computations

- ☺ Easy and fast to implement
- ☺ Easily portable [IEEE754]
- ☹ Requires dedicated hardware
- ☹ Slow if emulated in software

Fixed-point computations

- ☹ Tedious and time consuming to implement
  - $> 50\%$ of design time [Wil98]
- ☺ Relies only on integer instructions
- ☺ Efficient

Embedded systems targets

μ-controllers

DSPs

FPGAs

→ have efficient integer instructions

- Fixed-point arithmetic is well suited for embedded systems

Which arithmetic for computational tasks?

Floating-point computations

- ☺ Easy and fast to implement
- ☺ Easily portable [IEEE754]
- ☹ Requires dedicated hardware
- ☹ Slow if emulated in software

Fixed-point computations

- ☹ Tedious and time consuming to implement
  - $>50\%$ of design time [Wil98]
- ☺ Relies only on integer instructions
- ☺ Efficient

Embedded systems targets

μ-controllers

DSPs

FPGAs

$\longrightarrow$ have efficient integer instructions

- Fixed-point arithmetic is well suited for embedded systems

But, how to make it easy, fast, and numerically safe to use by non-expert programmers?

The DEFIS approach

- DEFIS (ANR, 2011-2015)

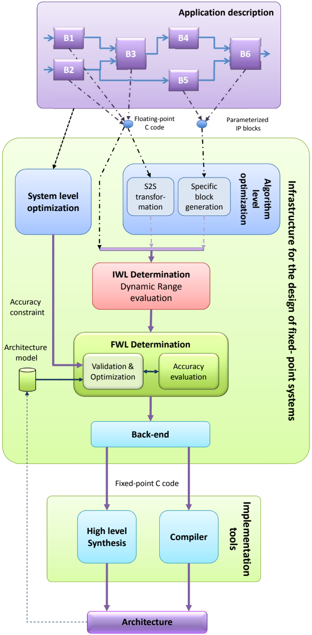
  Goal: develop techniques and tools to automate fixed-point programming

The DEFIS approach

- DEFIS (ANR, 2011-2015)

  Goal: develop techniques and tools to automate fixed-point programming

- Combines conversion and IP block synthesis
  - Ménard *et al.* (CAIRN, Univ. Rennes) [MCCS02]:
    - automatic float-to-fix conversion
  - Didier *et al.* (PEQUAN, Univ. Paris) [LHD14]:
    - code generation for the linear filter IP block

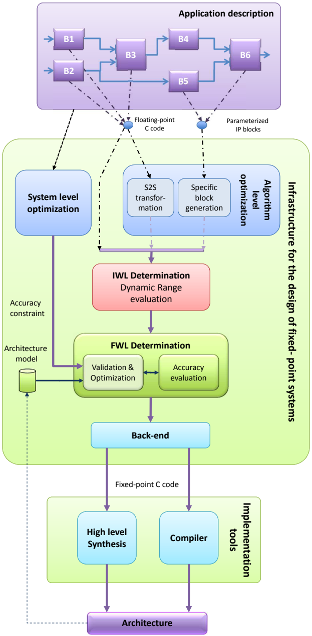


The DEFIS approach

- DEFIS (ANR, 2011-2015)

  Goal: develop techniques and tools to automate fixed-point programming

- Combines conversion and IP block synthesis
  - Ménard *et al.* (CAIRN, Univ. Rennes) [MCCS02]:
    - automatic float-to-fix conversion
  - Didier *et al.* (PEQUAN, Univ. Paris) [LHD14]:
    - code generation for the linear filter IP block
  - Our approach (DALI, Univ. Perpignan):
    - certified fixed-point synthesis for:
      - **Fine grained IP blocks:** dot-products, polynomials, ...
      - **High level IP blocks:** matrix multiplication, triangular matrix inversion, Cholesky decomposition

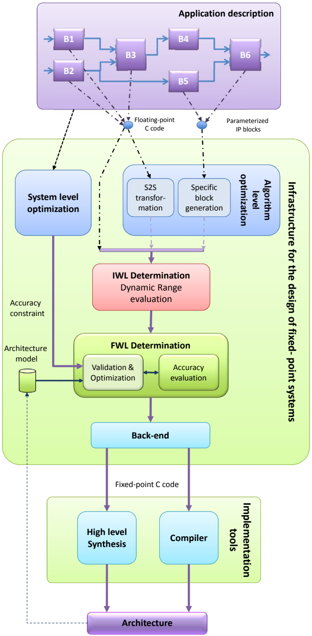


The DEFIS approach

- DEFIS (ANR, 2011-2015)

  Goal: develop techniques and tools to automate fixed-point programming

- Combines conversion and IP block synthesis
  - Ménard *et al.* (CAIRN, Univ. Rennes) [MCCS02]:
    - automatic float-to-fix conversion
  - Didier *et al.* (PEQUAN, Univ. Paris) [LHD14]:
    - code generation for the linear filter IP block
  - Our approach (DALI, Univ. Perpignan):
    - certified fixed-point synthesis for:
      - **Fine grained IP blocks:** dot-products, polynomials, ...
      - **High level IP blocks:** matrix multiplication, triangular matrix inversion, Cholesky decomposition



- **Long term objective:** code synthesis for matrix inversion

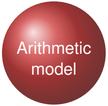
Our road-map

How to generate certified fixed-point code for matrix inversion?

Our road-map

How to generate certified fixed-point code for matrix inversion?

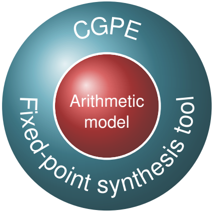
1. Specify an arithmetic model
   - Contributions:
     - formalization of $\sqrt{}$ and $/$



Our road-map

How to generate certified fixed-point code for matrix inversion?

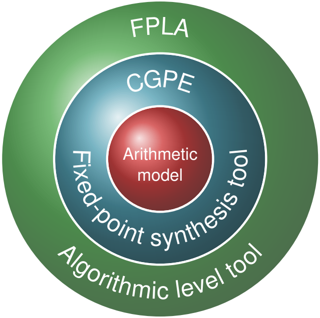
1. Specify an arithmetic model
   - Contributions:
     - formalization of $\sqrt{}$ and $/$
2. Build a synthesis tool, CGPE, for fine grained IP blocks:
   - it adheres to the arithmetic model
   - Contributions:
     - implementation of the arithmetic model



Our road-map

How to generate certified fixed-point code for matrix inversion?

1. Specify an arithmetic model
   - Contributions:
     - formalization of $\sqrt{}$ and $/$
2. Build a synthesis tool, CGPE, for fine grained IP blocks:
   - it adheres to the arithmetic model
   - Contributions:
     - implementation of the arithmetic model
3. Build a second synthesis tool, FPLA, for algorithmic IP blocks:
   - it generates code using CGPE
   - Contributions:
     - trade-off implementations for matrix multiplication
     - code synthesis for Cholesky decomposition and triangular matrix inversion



Outline of the talk

1. An arithmetic model for fixed-point code synthesis

2. An implementation of the arithmetic model: the CGPE tool

3. Fixed-point code synthesis for linear algebra basic blocks

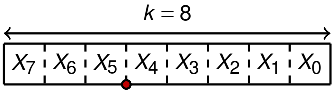
Outline of the talk

1. An arithmetic model for fixed-point code synthesis
2. An implementation of the arithmetic model: the CGPE tool
3. Fixed-point code synthesis for linear algebra basic blocks

Fixed-point arithmetic numbers

A fixed-point number x is defined by two integers:

- X the k-bit integer representation of x
- f the implicit scaling factor of x

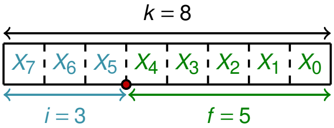


⇝ The value of x is given by $x = \dfrac{X}{2^f} = \displaystyle\sum_{\ell=-f}^{k-1-f} X_{\ell+f} \cdot 2^{\ell}$

Fixed-point arithmetic numbers

A fixed-point number x is defined by two integers:

- ▹ X the k-bit integer representation of x
- ▹ f the implicit scaling factor of x



⇝ The value of x is given by $x = \dfrac{X}{2^f} = \displaystyle\sum_{\ell=-f}^{k-1-f} X_{\ell+f} \cdot 2^{\ell}$

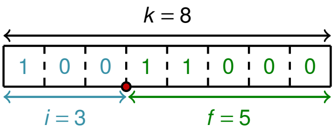
Notation

A fixed-point number with i bits of integer part and f bits of fraction part is in the $\mathbf{Q}_{i.f}$ format

Fixed-point arithmetic numbers

A fixed-point number x is defined by two integers:

- ▹ X the k-bit integer representation of x
- ▹ f the implicit scaling factor of x



⇝ The value of x is given by $x = \frac{X}{2^f} = \sum_{\ell=-f}^{k-1-f} X_{\ell+f} \cdot 2^\ell$

Notation

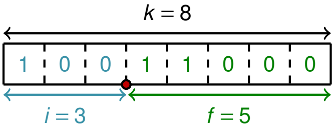
A fixed-point number with i bits of integer part and f bits of fraction part is in the $\mathbf{Q}_{i.f}$ format

- Example:
  - x in $\mathbf{Q}_{3.5}$ and $X = (1001\,1000)_2 = (152)_{10}$ ⟶ $x = (100.11000)_2 = (4.75)_{10}$

Fixed-point arithmetic numbers

A fixed-point number x is defined by two integers:

- ▹ X the k-bit integer representation of x
- ▹ f the implicit scaling factor of x



⇝ The value of x is given by $x = \dfrac{X}{2^f} = \sum_{\ell=-f}^{k-1-f} X_{\ell+f} \cdot 2^{\ell}$

Notation

A fixed-point number with i bits of integer part and f bits of fraction part is in the $\mathbf{Q}_{i.f}$ format

- Example:
  - x in $\mathbf{Q}_{3.5}$ and $X = (1001\,1000)_2 = (152)_{10}$ ⟶ $x = (100.11000)_2 = (4.75)_{10}$

How to compute with fixed-point numbers?

An interval arithmetic based model

- For each coefficient or variable v, we keep track of 2 intervals $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$
- Our model assumes a fixed word-length k

$\mathbf{Val}(v)$ is the range of v

$\mathbf{Err}(v)$ encloses the rounding error of computing v

An interval arithmetic based model

- For each coefficient or variable v, we keep track of 2 intervals $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$
- Our model assumes a fixed word-length k

$\mathbf{Val}(v)$ is the range of v

- the format $\mathbf{Q}_{i.f}$ of v is deduced from $\mathbf{Val}(v) = [\underline{\mathbf{v}}, \overline{\mathbf{v}}]$
  - $i = \lceil \log_2 (\max(|\underline{\mathbf{v}}|, |\overline{\mathbf{v}}|)) \rceil + \alpha$
  - $f = k - i$

$$\alpha = \begin{cases} 1, & \text{if } \bmod(\log_2(\overline{\mathbf{v}}), 1) \neq 0, \\ 2, & \text{otherwise} \end{cases}$$

$\mathbf{Err}(v)$ encloses the rounding error of computing v

- a bound ϵ on rounding errors is deduced from $\mathbf{Err}(v) = [\underline{\mathbf{e}}, \overline{\mathbf{e}}]$
  - $\epsilon = \max(|\underline{\mathbf{e}}|, |\overline{\mathbf{e}}|)$

An interval arithmetic based model

- For each coefficient or variable v, we keep track of 2 intervals $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$
- Our model assumes a fixed word-length k

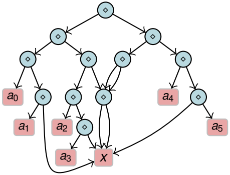
$\mathbf{Val}(v)$ is the range of v

- the format $\mathbf{Q}_{i.f}$ of v is deduced from $\mathbf{Val}(v) = [\underline{\mathbf{v}}, \overline{\mathbf{v}}]$
  - $i = \left\lceil \log_2\left(\max\left(|\underline{\mathbf{v}}|, |\overline{\mathbf{v}}|\right)\right)\right\rceil + \alpha$
  - $f = k - i$

$$\alpha = \begin{cases} 1, & \text{if } \bmod\left(\log_2(\overline{\mathbf{v}}), 1\right) \neq 0, \\ 2, & \text{otherwise} \end{cases}$$

$\mathbf{Err}(v)$ encloses the rounding error of computing v

- a bound ϵ on rounding errors is deduced from $\mathbf{Err}(v) = [\underline{\mathbf{e}}, \overline{\mathbf{e}}]$
  - $\epsilon = \max\left(|\underline{\mathbf{e}}|, |\overline{\mathbf{e}}|\right)$



An interval arithmetic based model

- For each coefficient or variable v, we keep track of 2 intervals $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$
- Our model assumes a fixed word-length k

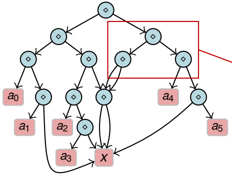
$\mathbf{Val}(v)$ is the range of v

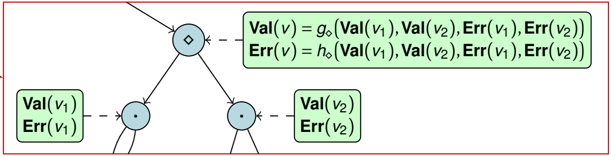
- the format $\mathbf{Q}_{i.f}$ of v is deduced from $\mathbf{Val}(v) = [\underline{\mathbf{v}}, \overline{\mathbf{v}}]$
  - $i = \lceil \log_2(\max(|\underline{\mathbf{v}}|, |\overline{\mathbf{v}}|)) \rceil + \alpha$
  - $f = k - i$

$$\alpha = \begin{cases} 1, & \text{if } \operatorname{mod}(\log_2(\overline{\mathbf{v}}), 1) \neq 0, \\ 2, & \text{otherwise} \end{cases}$$

$\mathbf{Err}(v)$ encloses the rounding error of computing v

- a bound ϵ on rounding errors is deduced from $\mathbf{Err}(v) = [\underline{\mathbf{e}}, \overline{\mathbf{e}}]$
  - $\epsilon = \max(|\underline{\mathbf{e}}|, |\overline{\mathbf{e}}|)$

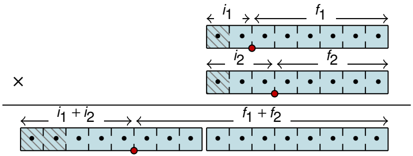




How to propagate $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ for $\diamond \in \{+, -, \times, \ll, \gg, \sqrt{}, /\}$?

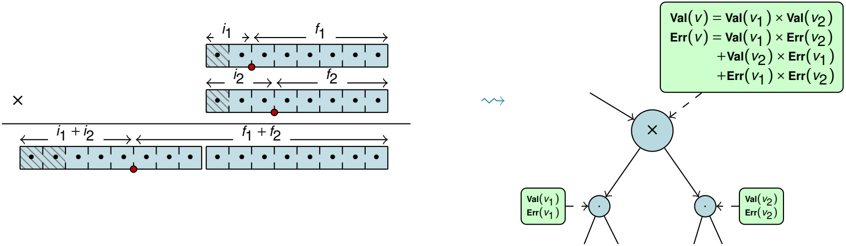
Fixed-point multiplication

- The output format of a $\mathbf{Q}_{i_1.f_1} \times \mathbf{Q}_{i_2.f_2}$ is $\mathbf{Q}_{i_1+i_2.f_1+f_2}$



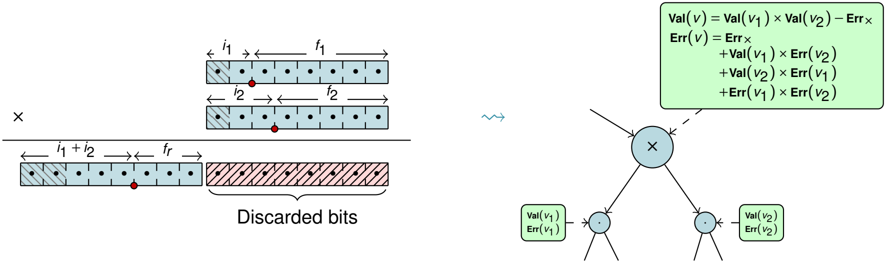
Fixed-point multiplication

- The output format of a $\mathbf{Q}_{i_1.f_1} \times \mathbf{Q}_{i_2.f_2}$ is $\mathbf{Q}_{i_1+i_2.f_1+f_2}$



Fixed-point multiplication

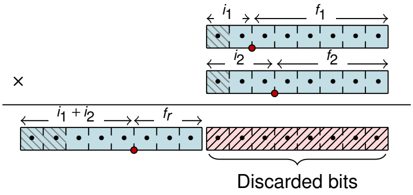
- The output format of a $\mathbf{Q}_{i_1.f_1} \times \mathbf{Q}_{i_2.f_2}$ is $\mathbf{Q}_{i_1+i_2.f_1+f_2}$
- But, doubling the word-length is costly

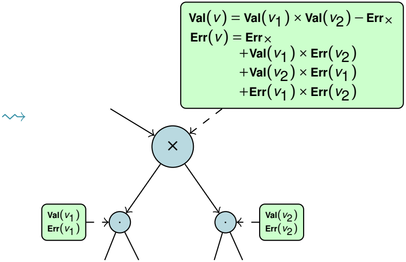


- $\mathbf{Err}_\times = \left[0, 2^{-f_r} - 2^{-(f_1+f_2)}\right]$

Fixed-point multiplication

- The output format of a $\mathbf{Q}_{i_1.f_1} \times \mathbf{Q}_{i_2.f_2}$ is $\mathbf{Q}_{i_1+i_2.f_1+f_2}$
- But, doubling the word-length is costly



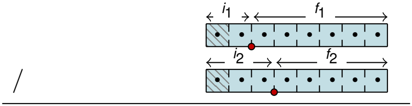


- $\mathbf{Err}_\times = \left[0, 2^{-f_r} - 2^{-(f_1+f_2)}\right]$
- This multiplication is available on integer processors and DSPs

```
int32_t mul (int32_t v1, int32_t v2){
  int64_t prod = ((int64_t) v1) * ((int64_t) v2);
  return (int32_t) (prod >> 32);
}
```

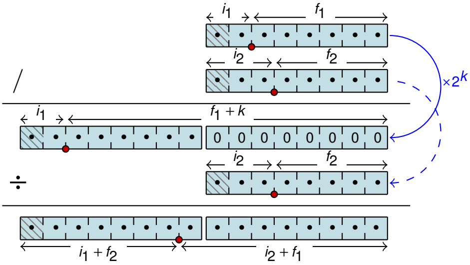
Our new fixed-point division

- The output integer part of $\mathbf{Q}_{i_1.f_1}/\mathbf{Q}_{i_2.f_2}$ may be as large as $i_1 + f_2$



Our new fixed-point division

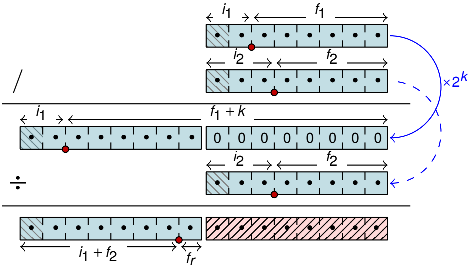
- The output integer part of $\mathbf{Q}_{i_1.f_1}/\mathbf{Q}_{i_2.f_2}$ may be as large as $i_1 + f_2$



- $\mathbf{Err}_{/} = [-2^{i_2+f_1}, 2^{i_2+f_1}]$

Our new fixed-point division

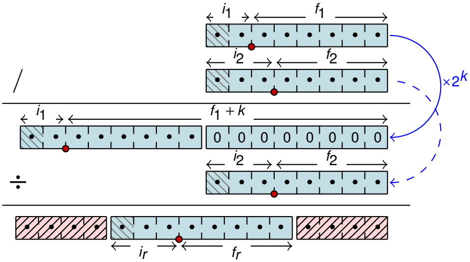
- The output integer part of $\mathbf{Q}_{i_1.f_1}/\mathbf{Q}_{i_2.f_2}$ may be as large as $i_1 + f_2$
- But, doubling the word-length is costly



- $\mathbf{Err}_/ = [-2^{f_r}, 2^{f_r}]$

Our new fixed-point division

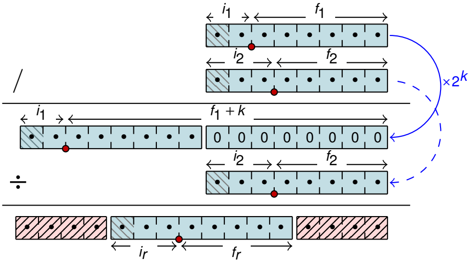
- The output integer part of $\mathbf{Q}_{i_1.f_1}/\mathbf{Q}_{i_2.f_2}$ may be as large as $i_1 + f_2$
- But, doubling the word-length is costly
- How to obtain sharper a error bounds on $\mathbf{Err}_{/}$?



- $\mathbf{Err}_{/} = [-2^{f_r}, 2^{f_r}]$
  - ☺ sharper bound
  - ☹ risk of overflow at run-time

Our new fixed-point division

- The output integer part of $\mathbf{Q}_{i_1.f_1}/\mathbf{Q}_{i_2.f_2}$ may be as large as $i_1 + f_2$
- But, doubling the word-length is costly
- How to obtain sharper a error bounds on $\mathbf{Err}_{/}$?



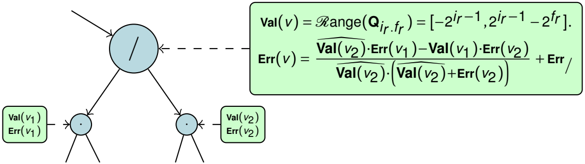
- $\mathbf{Err}_{/} = \left[-2^{f_r}, 2^{f_r}\right]$
  - ☺ sharper bound
  - ☹ risk of overflow at run-time

How to decide of the output format of division?

- A large integer part
  - ✓ prevents overflow
  - ✗ loose error bounds and loss of precision
- A small integer part
  - ✗ may cause overflow
  - ✓ sharp error bounds and more accurate computations

The propagation rule and implementation of division

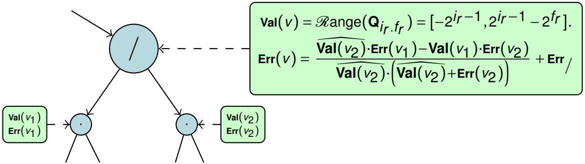
- Once the output format decided $\mathbf{Q}_{i_r.f_r}$



- $\widehat{\mathbf{Val}(v_2)} = \dfrac{\mathbf{Val}(v_1)}{\widehat{\mathbf{Val}(v)} + \mathbf{Err}_{/}} \cap \mathbf{Val}(v_2)$ and $\widehat{\mathbf{Val}(v)} = [-2^{i_r-1}, -2^{-f_r}] \cup [2^{-f_r}, 2^{i_r-1} - 2^{f_r}]$

The propagation rule and implementation of division

- Once the output format decided $\mathbf{Q}_{i_r.f_r}$



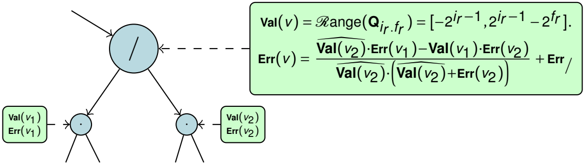
- $\widehat{\mathbf{Val}(v_2)} = \dfrac{\mathbf{Val}(v_1)}{\widehat{\mathbf{Val}(v)} + \mathbf{Err}_/} \cap \mathbf{Val}(v_2)$ and $\widehat{\mathbf{Val}(v)} = [-2^{i_r-1}, -2^{-f_r}] \cup [2^{-f_r}, 2^{i_r-1} - 2^{f_r}]$

```
int32_t div (int32_t V1, int32_t V2, uint16_t eta)
{
  int64_t t1 = ((int64_t)V1) << eta;
  int64_t V  = t1 / V2;


  return (int32_t) V;
}
```

The propagation rule and implementation of division

- Once the output format decided $\mathbf{Q}_{i_r.f_r}$



- $\widehat{\mathbf{Val}(v_2)} = \frac{\mathbf{Val}(v_1)}{\widehat{\mathbf{Val}(v)} + \mathbf{Err}_{/}} \cap \mathbf{Val}(v_2)$ and $\widehat{\mathbf{Val}(v)} = [-2^{i_r-1}, -2^{-f_r}] \cup [2^{-f_r}, 2^{i_r-1} - 2^{f_r}]$

```
int32_t div (int32_t V1, int32_t V2, uint16_t eta)
{
  int64_t t1 = ((int64_t)V1) << eta;
  int64_t V  = t1 / V2;
  CGPE_ASSERT((((V & 0xFFFFFFFF80000000ll) == 0xFFFFFFFF80000000ll)
            || ((V & 0xFFFFFFFF80000000ll) == 0)));
  return (int32_t) V;
}
```

- Additional code to check for run-time overflows

The division format trade-off: case of inverting 2×2 matrices

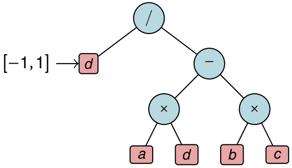
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$

The division format trade-off: case of inverting 2×2 matrices

- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

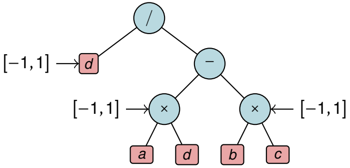
The division format trade-off: case of inverting 2×2 matrices

- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$



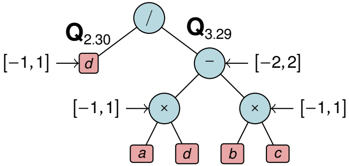
The division format trade-off: case of inverting 2×2 matrices

- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$



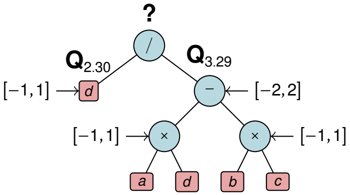
The division format trade-off: case of inverting 2×2 matrices

- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$



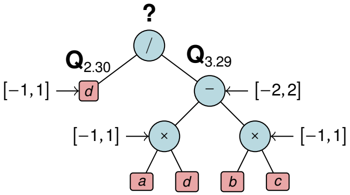
The division format trade-off: case of inverting 2×2 matrices

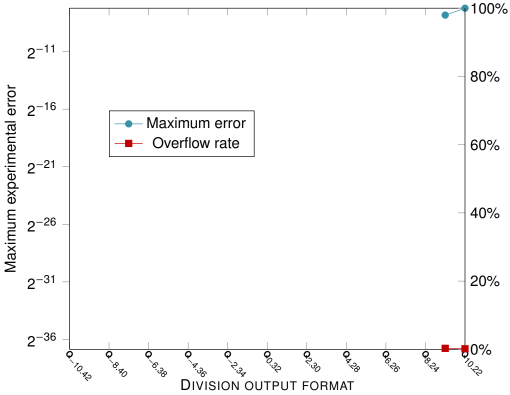
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$



The division format trade-off: case of inverting 2×2 matrices

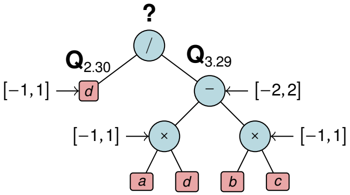
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

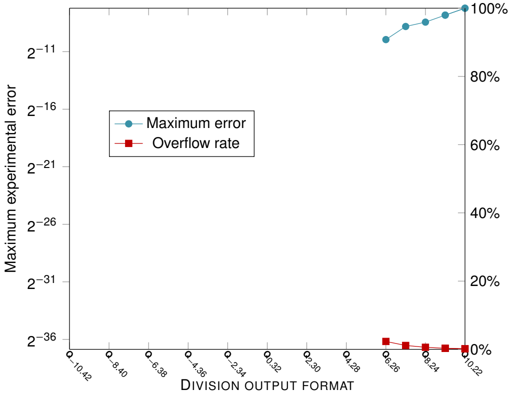




The division format trade-off: case of inverting 2×2 matrices

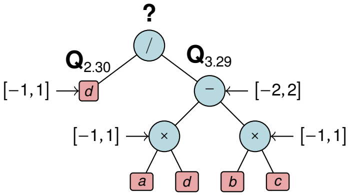
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

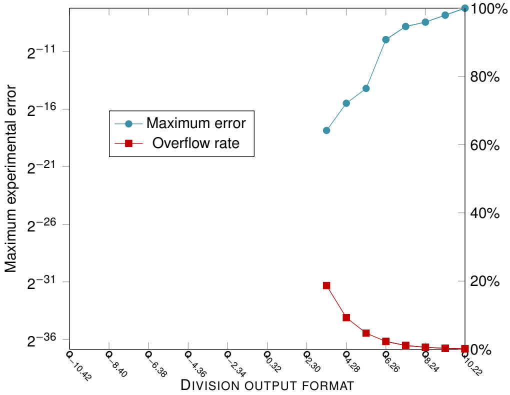




The division format trade-off: case of inverting 2×2 matrices

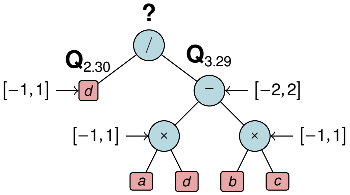
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

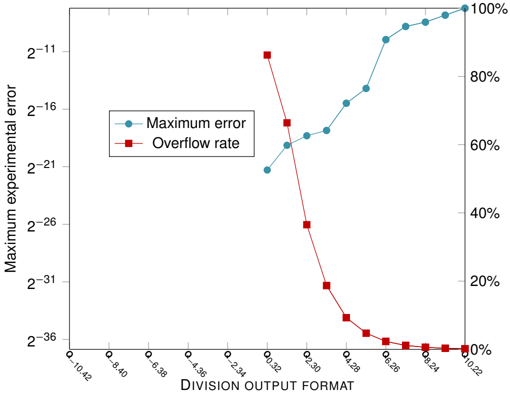




The division format trade-off: case of inverting 2×2 matrices

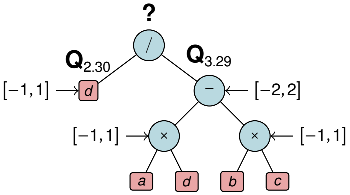
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

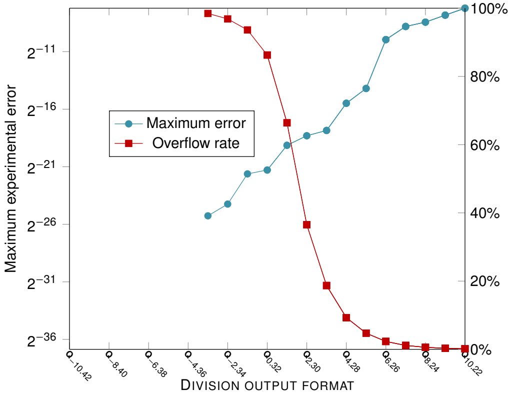




The division format trade-off: case of inverting 2×2 matrices

- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

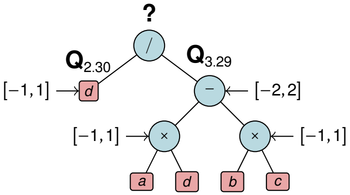


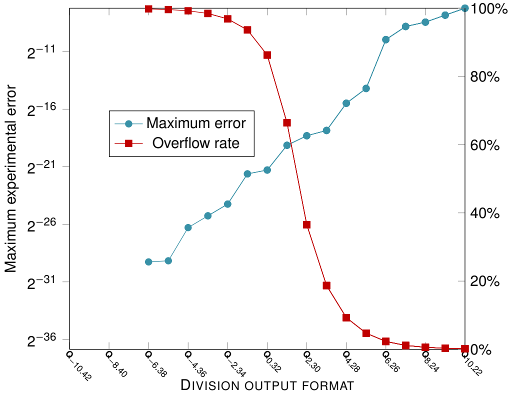


The division format trade-off: case of inverting 2×2 matrices

- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$

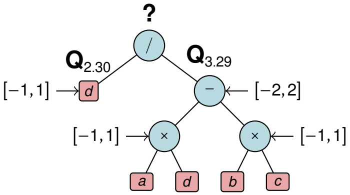
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

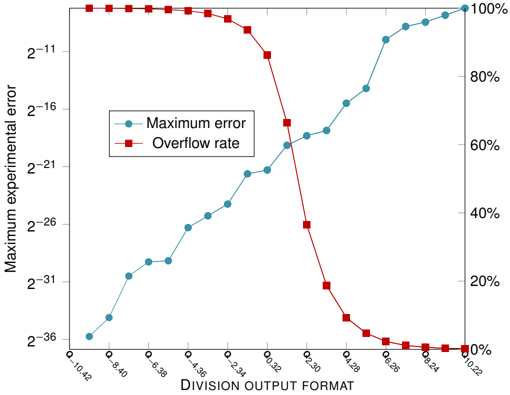




The division format trade-off: case of inverting 2×2 matrices

- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

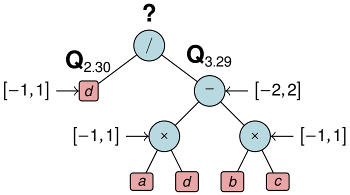


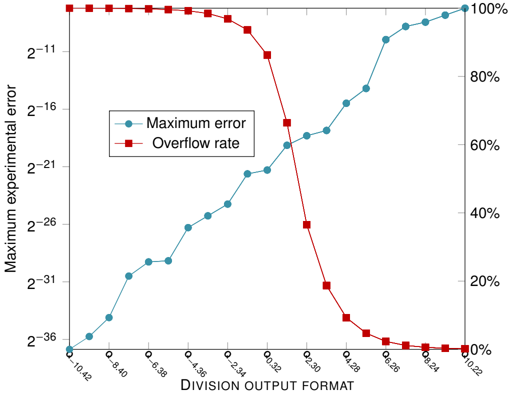


The division format trade-off: case of inverting 2×2 matrices

- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$

- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$





Outline of the talk

1. An arithmetic model for fixed-point code synthesis

2. An implementation of the arithmetic model: the CGPE tool

3. Fixed-point code synthesis for linear algebra basic blocks

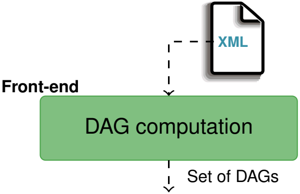
The CGPE tool

- CGPE (*Code Generation for Polynomial Evaluation*): initiated by Revy [MR11]
  - synthesizes fixed-point code for polynomial evaluation

The CGPE tool

- CGPE (*Code Generation for Polynomial Evaluation*): initiated by Revy [MR11]
  - synthesizes fixed-point code for polynomial evaluation

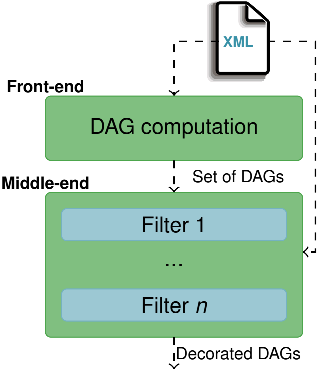
1. Computation step $\rightsquigarrow$ front-end
   - computes evaluation schemes $\rightsquigarrow$ DAGs



The CGPE tool

- CGPE (*Code Generation for Polynomial Evaluation*): initiated by Revy [MR11]
  - synthesizes fixed-point code for polynomial evaluation

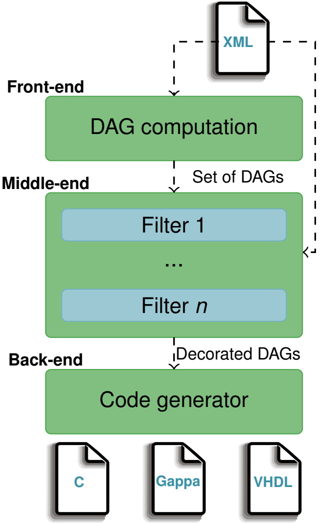
1. Computation step ⇝ front-end
   - computes evaluation schemes ⇝ DAGs
2. Filtering step ⇝ middle-end
   - applies the arithmetic model
   - prunes the DAGs that do not satisfy different criteria:
     - latency ⇝ scheduling filter
     - accuracy ⇝ numerical filter
     - ...



The CGPE tool

- CGPE (*Code Generation for Polynomial Evaluation*): initiated by Revy [MR11]
  - synthesizes fixed-point code for polynomial evaluation

1. Computation step ⇝ front-end
   - computes evaluation schemes ⇝ DAGs
2. Filtering step ⇝ middle-end
   - applies the arithmetic model
   - prunes the DAGs that do not satisfy different criteria:
     - latency ⇝ scheduling filter
     - accuracy ⇝ numerical filter
     - ...
3. Generation step ⇝ back-end
   - generates C codes and Gappa accuracy certificates



Code synthesis for an IIR filter using CGPE

- Low-pass Butterworth filter with cutoff frequency $0.3 \cdot \pi$:

$$y[k] = \sum_{i=0}^{3} b_i \cdot u[k-i] - \sum_{i=1}^{3} a_i \cdot y[k-i]$$

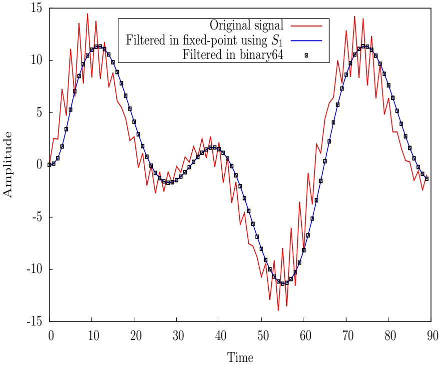
```
<dotproduct inf="0xb1e91685" sup="0x4e16e97b" integer_width="6" fraction_width="26" width="32">
  <coefficient name="b0" value="0x65718e3b" integer_width="-3" fraction_width="35" width="32"/>
    ...
  <variable name="y3" inf="0xb1e91685" sup="0x4e16e97b" integer_width="6" fraction_width="26" width="32"/>
</dotproduct >
```

Code synthesis for an IIR filter using CGPE

- Low-pass Butterworth filter with cutoff frequency $0.3 \cdot \pi$:

$$y[k] = \sum_{i=0}^{3} b_i \cdot u[k-i] - \sum_{i=1}^{3} a_i \cdot y[k-i]$$

```
<dotproduct inf="0xb1e91685" sup="0x4e16e97b" integer_width="6" fraction_width="26" width="32">
  <coefficient name="b0" value="0x65718e3b" integer_width="-3" fraction_width="35" width="32"/>
    ...
  <variable name="y3" inf="0xb1e91685" sup="0x4e16e97b" integer_width="6" fraction_width="26" width="32"/>
</dotproduct>
```

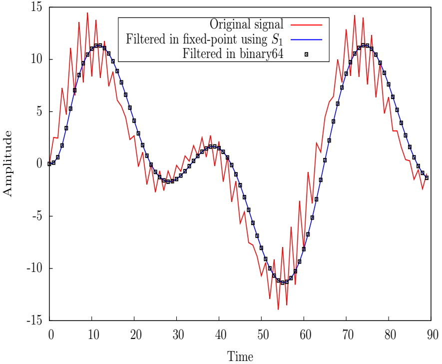


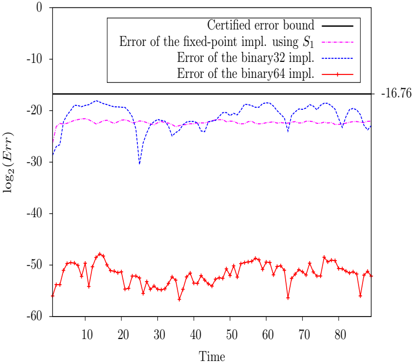
Code synthesis for an IIR filter using CGPE

- Low-pass Butterworth filter with cutoff frequency $0.3 \cdot \pi$:

$$y[k] = \sum_{i=0}^{3} b_i \cdot u[k-i] - \sum_{i=1}^{3} a_i \cdot y[k-i]$$

```
<dotproduct inf="0xb1e91685" sup="0x4e16e97b" integer_width="6" fraction_width="26" width="32">
  <coefficient name="b0" value="0x65718e3b" integer_width="-3" fraction_width="35" width="32"/>
    ...
  <variable name="y3" inf="0xb1e91685" sup="0x4e16e97b" integer_width="6" fraction_width="26" width="32"/>
</dotproduct>
```





Code synthesis for an IIR filter using CGPE

- Low-pass Butterworth filter with cutoff frequency $0.3 \cdot \pi$:

$$y[k] = \sum_{i=0}^{3} b_i \cdot u[k-i] - \sum_{i=1}^{3} a_i \cdot y[k-i]$$

```
int32_t filter( int32_t u0    /*Q5.27*/ , int32_t u1    /*Q5.27*/ ,
                int32_t u2    /*Q5.27*/ , int32_t u3    /*Q5.27*/ ,
                int32_t y1    /*Q6.26*/ , int32_t y2    /*Q6.26*/ ,
                int32_t y3    /*Q6.26*/ )
{                                                //Formats  Err
  int32_t r0  = mul(0x4a5cdb26, y1);             //Q8.24   [-2^{-24},0]
  int32_t r1  = mul(0xa6eb5908, y2);             //Q7.25   [-2^{-25},0]
  int32_t r2  = mul(0x4688a637, y3);             //Q5.27   [-2^{-27},0]
  int32_t r3  = mul(0x65718e3b, u0);             //Q2.30   [-2^{-30},0]
  int32_t r4  = mul(0x65718e3b, u3);             //Q2.30   [-2^{-30},0]
  int32_t r5  = r3 + r4;                         //Q2.30   [-2^{-29},0]
  int32_t r6  = r5 >> 2;                         //Q4.28   [-2^{-27.6781},0]
  int32_t r7  = mul(0x4c152aad, u1);             //Q4.28   [-2^{-28},0]
  int32_t r8  = mul(0x4c152aad, u2);             //Q4.28   [-2^{-28},0]
  int32_t r9  = r7 + r8;                         //Q4.28   [-2^{-27},0]
  int32_t r10 = r6 + r9;                         //Q4.28   [-2^{-26.2996},0]
  int32_t r11 = r10 >> 1;                        //Q5.27   [-2^{-25.9125},0]
  int32_t r12 = r2 + r11;                        //Q5.27   [-2^{-25.3561},0]
  int32_t r13 = r12 >> 2;                        //Q7.25   [-2^{-24.3853},0]
  int32_t r14 = r1 + r13;                        //Q7.25   [-2^{-23.6601},0]
  int32_t r15 = r14 >> 1;                        //Q8.24   [-2^{-23.1798},0]
  int32_t r16 = r0 + r15;                        //Q8.24   [-2^{-22.5324},0]
  int32_t r17 = r16 << 2;                        //Q6.26   [-2^{-22.5324},0]
  return r17;
}
```

Outline of the talk

1. An arithmetic model for fixed-point code synthesis
2. An implementation of the arithmetic model: the CGPE tool
3. Fixed-point code synthesis for linear algebra basic blocks

A strategy to synthesize code for matrix inversion

Let M be a matrix of fixed-point variables,

to generate certified code that inverts $M' \in M$ a symmetric positive definite, we need to:

1. Generate certified code to compute B a lower triangular s.t. $M' = B \cdot B^T$
2. Generate certified code to compute $N = B^{-1}$
3. Generate certified code to compute $M'^{-1} = N^T \cdot N$

A strategy to synthesize code for matrix inversion

Let M be a matrix of fixed-point variables,

to generate certified code that inverts $M' \in M$ a symmetric positive definite, we need to:

1. Generate certified code to compute B a lower triangular s.t. $M' = B \cdot B^T$
2. Generate certified code to compute $N = B^{-1}$
3. Generate certified code to compute $M'^{-1} = N^T \cdot N$

The basic blocks we need to include in our tool-chain

- Certified code synthesis for Cholesky decomposition

A strategy to synthesize code for matrix inversion

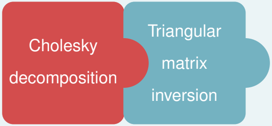
Let M be a matrix of fixed-point variables,

to generate certified code that inverts $M' \in M$ a symmetric positive definite, we need to:

1. Generate certified code to compute B a lower triangular s.t. $M' = B \cdot B^T$
2. Generate certified code to compute $N = B^{-1}$
3. Generate certified code to compute $M'^{-1} = N^T \cdot N$

The basic blocks we need to include in our tool-chain

- Certified code synthesis for Cholesky decomposition
- Certified code synthesis for triangular matrix inversion



A strategy to synthesize code for matrix inversion

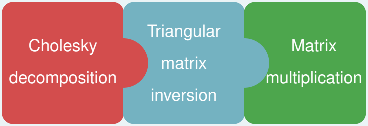
Let M be a matrix of fixed-point variables,

to generate certified code that inverts $M' \in M$ a symmetric positive definite, we need to:

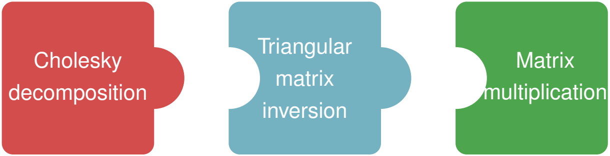
1. Generate certified code to compute B a lower triangular s.t. $M' = B \cdot B^T$
2. Generate certified code to compute $N = B^{-1}$
3. Generate certified code to compute $M'^{-1} = N^T \cdot N$

The basic blocks we need to include in our tool-chain

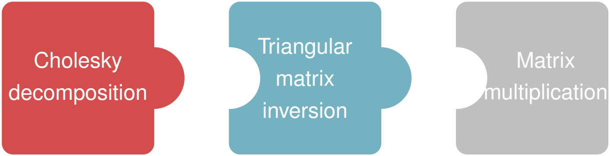
- Certified code synthesis for Cholesky decomposition
- Certified code synthesis for triangular matrix inversion
- Certified code synthesis for matrix multiplication



Linear algebra basic blocks



Linear algebra basic blocks



Cholesky decomposition and triangular matrix inversion

Cholesky decomposition

$$b_{i,j} = \begin{cases} \sqrt{c_{i,i}} & \text{if } i = j \\ \dfrac{c_{i,j}}{b_{j,j}} & \text{if } i \neq j \end{cases}$$

$$\text{with } c_{i,j} = m_{i,j} - \sum_{k=0}^{j-1} b_{i,k} \cdot b_{j,k}$$

Triangular matrix inversion

$$n_{i,j} = \begin{cases} \dfrac{1}{b_{i,i}} & \text{if } i = j \\ \dfrac{-c_{i,j}}{b_{i,i}} & \text{if } i \neq j \end{cases}$$

$$\text{where } c_{i,j} = \sum_{k=j}^{i-1} b_{i,k} \cdot n_{k,j}$$

Cholesky decomposition and triangular matrix inversion

Cholesky decomposition

$$b_{i,j} = \begin{cases} \sqrt{c_{i,i}} & \text{if } i = j \\ \dfrac{c_{i,j}}{b_{j,j}} & \text{if } i \neq j \end{cases}$$

$$\text{with } c_{i,j} = m_{i,j} - \sum_{k=0}^{j-1} b_{i,k} \cdot b_{j,k}$$

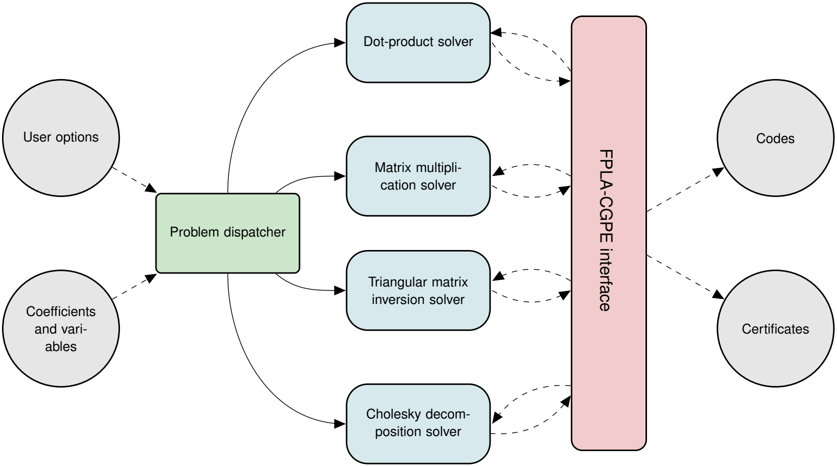
Triangular matrix inversion

$$n_{i,j} = \begin{cases} \dfrac{1}{b_{i,i}} & \text{if } i = j \\ \dfrac{-c_{i,j}}{b_{i,i}} & \text{if } i \neq j \end{cases}$$

$$\text{where } c_{i,j} = \sum_{k=j}^{i-1} b_{i,k} \cdot n_{k,j}$$

Dependencies of the coefficient $b_{4,2}$ in the decomposition and inversion of a 6×6 matrix.

FPLA (Fixed-Point Linear Algebra)

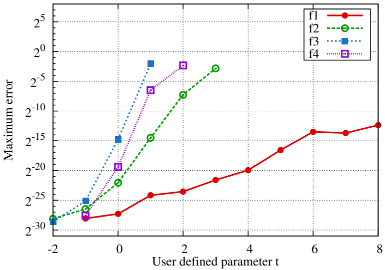


Impact of the output format of division

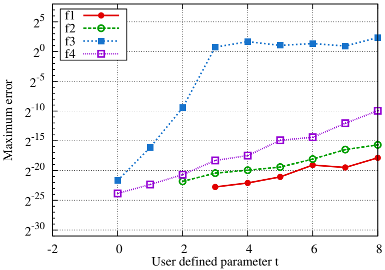
Different functions to set the output format of division

1. $f_1(i_1, i_2) = t$,
2. $f_2(i_1, i_2) = \min(i_1, i_2) + t$,
3. $f_3(i_1, i_2) = \max(i_1, i_2) + t$,
4. $f_4(i_1, i_2) = \lfloor (i_1 + i_2)/2 \rfloor + t$,

i_1 and i_2: integer parts of the numerator and denominator and $t \in [-2, 8]$



(a) Cholesky 5×5.

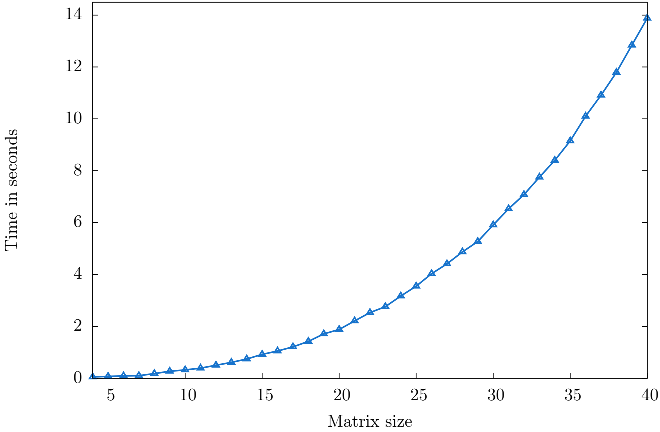


(b) Triangular 10×10.

Maximum errors with various functions used to determine the output formats of division.

How fast is generating triangular matrix inversion codes?

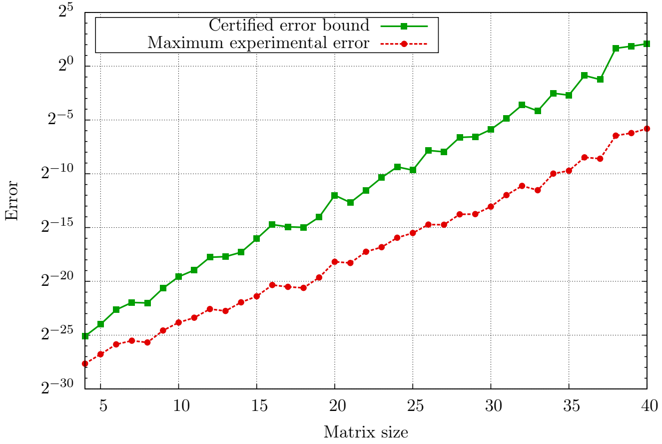
- We use $f_4(i_1, i_2) = \lfloor (i_1 + i_2)/2 \rfloor + 1$ to set the output format of division



Generation time for the inversion of triangular matrices of size 4 to 40.

How fast is generating triangular matrix inversion codes?

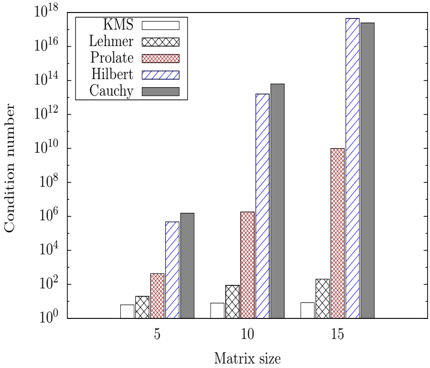
- We use $f_4(i_1, i_2) = \lfloor (i_1 + i_2)/2 \rfloor + 1$ to set the output format of division



Error bounds and experimental errors for the inversion of triangular matrices of size 4 to 40.

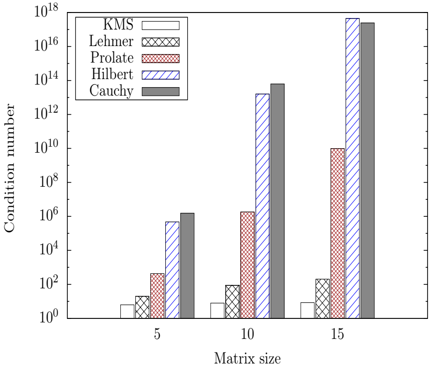
Decomposing some well known matrices

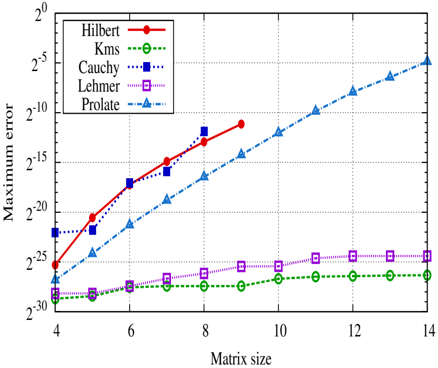
- 2 ill-conditioned matrices: Hilbert and Cauchy
- 2 well-conditioned matrices: KMS and Lehmer



Decomposing some well known matrices

- 2 ill-conditioned matrices: Hilbert and Cauchy
- 2 well-conditioned matrices: KMS and Lehmer





- Ill-conditioned matrices tend to overflow more often
  - similar behaviour in floating-point arithmetic
- The decompositions of KMS and Lehmer are highly accurate

Conclusions and perspectives

Contributions

- Formalization and implementation of an arithmetic model
  - allows certification
  - handles $\sqrt{}$ and $/$

Conclusions and perspectives

Contributions

- Formalization and implementation of an arithmetic model
  - allows certification
  - handles $\sqrt{}$ and $/$
- Adaptation of the CGPE tool to the model:
  - generates code for fine grained expressions
  - instruction selection

Conclusions and perspectives

Contributions

- Formalization and implementation of an arithmetic model
  - allows certification
  - handles $\sqrt{}$ and $/$
- Adaptation of the CGPE tool to the model:
  - generates code for fine grained expressions
  - instruction selection
- Development of FPLA:
  - automated and certified code synthesis for linear algebra basic block
    - → Cholesky decomposition and triangular matrix inversion: study of divisions' impact

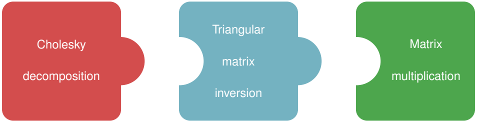
Conclusions and perspectives

Contributions

- Formalization and implementation of an arithmetic model
  - allows certification
  - handles $\sqrt{}$ and $/$
- Adaptation of the CGPE tool to the model:
  - generates code for fine grained expressions
  - instruction selection
- Development of FPLA:
  - automated and certified code synthesis for linear algebra basic block
    - → Cholesky decomposition and triangular matrix inversion: study of divisions' impact

Perspectives

- Integrate the matrix inversion flow



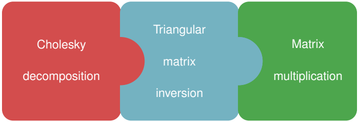
Conclusions and perspectives

Contributions

- Formalization and implementation of an arithmetic model
  - allows certification
  - handles $\sqrt{}$ and $/$
- Adaptation of the CGPE tool to the model:
  - generates code for fine grained expressions
  - instruction selection
- Development of FPLA:
  - automated and certified code synthesis for linear algebra basic block
    - → Cholesky decomposition and triangular matrix inversion: study of divisions' impact

Perspectives

- Integrate the matrix inversion flow



MERCI

[Wil98] H. Keding, M. Willems, M. Coors, and H. Meyr.
Fridge: a fixed-point design and simulation environment.

[IEEE754] IEEE 754.
IEEE Standard for Floating-Point Arithmetic.

[MCCS02] Daniel Menard, Daniel Chillet, François Charot, and Olivier Sentieys.
Automatic floating-point to fixed-point conversion for DSP code generation.

[IBMK10] Ali Irturk, Bridget Benson, Shahnam Mirzaei, and Ryan Kastner.
GUSTO: An Automatic Generation and Optimization Tool for Matrix Inversion Architectures.

[LHD12] Benoit Lopez, Thibault Hilaire, and Laurent-Stéphane Didier.
Sum-of-products evaluation schemes with fixed-point arithmetic, and their application to IIR filter implementation.

[FRC03] Claire F. Fang, Rob A. Rutenbar, and Tsuhan Chen.
Fast, accurate static analysis for fixed-point finite-precision effects in dsp designs.

[MRS12] Daniel Ménard, Romuald Rocher, Olivier Sentieys, Nicolas Simon, Laurent-Stéphane Didier, Thibault Hilaire, Benoît Lopez, Eric Goubault, Sylvie Putot, Franck Vedrine, Amine Najahi, Guillaume Revy, Laurent Fangain, Christian Samoyeau, Fabrice Lemonnier, and Christophe Clienti.
Design of Fixed-Point Embedded Systems (defis) French ANR Project.

[LHD14] Benoit Lopez, Thibault Hilaire, and Laurent-Stéphane Didier.
Formatting bits to better implement signal processing algorithms.

[Rev09] Guillaume Revy.
Implementation of binary floating-point arithmetic on embedded integer processors - Polynomial evaluation-based algorithms and certified code generation.

[MNR12] Christophe Mouilleron, Amine Najahi, and Guillaume Revy.
Approach based on instruction selection for fast and certified code generation.

[MR11] Christophe Mouilleron and Guillaume Revy.
Automatic Generation of Fast and Certified Code for Polynomial Evaluation.

[KG08] David R. Koes and Seth C. Goldstein.
Near-optimal instruction selection on DAGs.

[MNR14b] Matthieu Martel, Amine Najahi, and Guillaume Revy.
Toward the synthesis of fixed-point code for matrix inversion based on cholesky decomposition.

[MNR14c] Christophe Mouilleron, Amine Najahi, and Guillaume Revy.
Automated Synthesis of Target-Dependent Programs for Polynomial Evaluation in Fixed-Point Arithmetic.

[MNR14a] Matthieu Martel, Amine Najahi, and Guillaume Revy.
Code Size and Accuracy-Aware Synthesis of Fixed-Point Programs for Matrix Multiplication.

[CG09] Jason Cong, Karthik Gururaj, Bin Liu 0006, Chunyue Liu, Zhiru Zhang, Sheng Zhou, and Yi Zou.
Evaluation of static analysis techniques for fixed-point precision optimization.

[LV09] Dong-U Lee and John D. Villasenor.
Optimized custom precision function evaluation for embedded processors.