Ph.D. defense
Perpignan, December 10th, 2014

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

Mohamed Amine Najahi

Advisors: Matthieu Martel and Guillaume Revy

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

Context

- The embedded systems market is growing

Around 30% of an airplane's cost, around 40% of a car's cost



Context

- The embedded systems market is growing

Around 30% of an airplane's cost, around 40% of a car's cost

radar processing

velocity regulators

audio signal processing

signal and image processing

- Embedded systems are often dedicated to computational tasks

Context

- The embedded systems market is growing

Around 30% of an airplane's cost, around 40% of a car's cost

radar processing

velocity regulators

audio signal processing

signal and image processing

- Embedded systems are often dedicated to computational tasks

- Embedded systems face multiple constraints
  - efficiency
  - cost
  - limits on hardware resources
  - limits on energy consumption

Context

- The embedded systems market is growing

Around 30% of an airplane's cost, around 40% of a car's cost

radar processing

velocity regulators

audio signal processing

signal and image processing

- Embedded systems are often dedicated to computational tasks

- Embedded systems face multiple constraints
  - efficiency
  - cost
  - limits on hardware resources
  - limits on energy consumption

How to implement these computational tasks in embedded systems?

How to implement computational tasks?

Floating-point computations

Fixed-point computations

How to implement computational tasks?

Floating-point computations

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

Fixed-point computations

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

How to implement computational tasks?

Floating-point computations

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

Fixed-point computations

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

How to implement computational tasks?

Floating-point computations

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

Fixed-point computations

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

How to implement computational tasks?

Floating-point computations

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

Fixed-point computations

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

State of the art on how to achieve this objective?

- To make fixed-point programming easy and fast
  - develop automated code synthesis tools
- To make fixed-point programming safe numerically
  - the tools must generate bounds on rounding errors

State of the art on how to achieve this objective?

- To make fixed-point programming easy and fast
  - develop automated code synthesis tools
- To make fixed-point programming safe numerically
  - the tools must generate bounds on rounding errors

Float-to-fix conversion

- Works for a generic input
- IDFix [?], GUSTO [?], ...
- Based on floating-point simulations
  - no numeric certification
  - do not scale

State of the art on how to achieve this objective?

- To make fixed-point programming easy and fast
  - develop automated code synthesis tools
- To make fixed-point programming safe numerically
  - the tools must generate bounds on rounding errors

Float-to-fix conversion

- Works for a generic input
- IDFix [?], GUSTO [?], ...
- Based on floating-point simulations
  - no numeric certification
  - do not scale

Fixed-point code synthesis for IP blocks

- IP blocks: polynomials [?], filters [?], ...
- Based on analytic techniques
  - interval and affine arithmetics [?], [?], differentiation [?], ...
  - fast and scalable

State of the art on how to achieve this objective?

- To make fixed-point programming easy and fast
  - develop automated code synthesis tools
- To make fixed-point programming safe numerically
  - the tools must generate bounds on rounding errors

Float-to-fix conversion

- Works for a generic input
- IDFix [?], GUSTO [?], ...
- Based on floating-point simulations
  - no numeric certification
  - do not scale

Fixed-point code synthesis for IP blocks

- IP blocks: polynomials [?], filters [?], ...
- Based on analytic techniques
  - interval and affine arithmetics [?], [?], differentiation [?], ...
  - fast and scalable

The DEFIS approach

- DEFIS (ANR, 2011-)

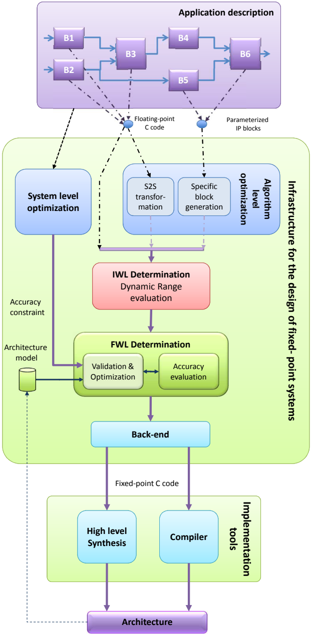
  Goal: develop techniques and tools to automate
  fixed-point programming

The DEFIS approach

- DEFIS (ANR, 2011-)

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

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

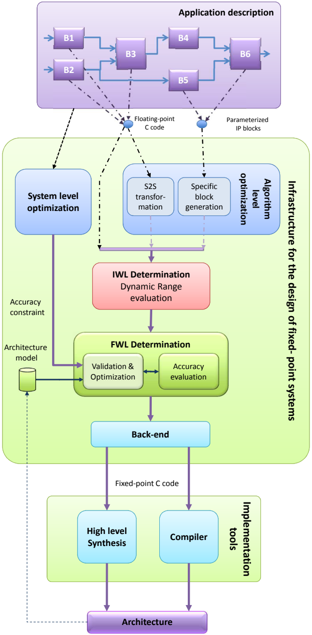


The DEFIS approach

- DEFIS (ANR, 2011-)

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

- Combines conversion and IP block synthesis
  - Ménard *et al.* (CAIRN, Univ. Rennes) [?]:
    - automatic float-to-fix conversion
  - Didier *et al.* (PEQUAN, Univ. Paris) [?]:
    - 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



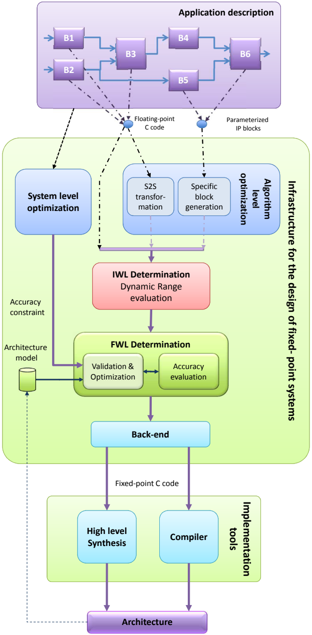
The DEFIS approach

- DEFIS (ANR, 2011-)

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

- Combines conversion and IP block synthesis
  - Ménard *et al.* (CAIRN, Univ. Rennes) [?]:
    - automatic float-to-fix conversion
  - Didier *et al.* (PEQUAN, Univ. Paris) [?]:
    - 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



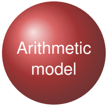
Our road-map

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

Our road-map

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

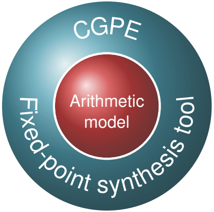
1. Specify an arithmetic model
   - Contributions:
     - formalization of $\sqrt{}$ and $/$



Our road-map

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

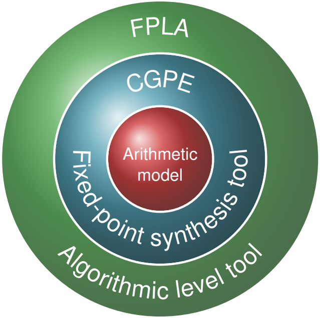
1. Specify an arithmetic model
   - Contributions:
     - formalization of $\sqrt{}$ and $/$
2. Build a synthesis tool, CGPE, for fine grained IP blocks:
   - it adheres to the arithmetic model
   - Contributions:
     - implementation of the arithmetic model
     - instruction selection



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
     - instruction selection
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



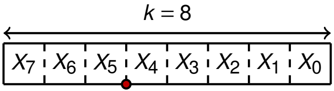
Outline of the talk

Outline of the talk

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

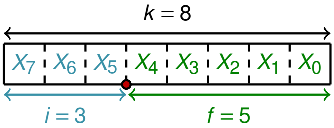


⇝ The value of x is given by $x = \dfrac{X}{2^f} = \displaystyle\sum_{\ell=-f}^{k-1-f} X_{\ell+f} \cdot 2^{\ell}$

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 $x = \frac{X}{2^f} = \sum_{\ell=-f}^{k-1-f} X_{\ell+f} \cdot 2^{\ell}$

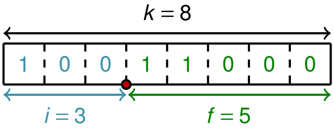
Notation

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

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

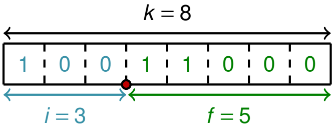
A fixed-point number with i bits of integer part and f bits of fraction part is in the $\mathbf{Q}_{i.f}$ format

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

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 $x = \dfrac{X}{2^f} = \displaystyle\sum_{\ell=-f}^{k-1-f} X_{\ell+f} \cdot 2^{\ell}$

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

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

How to compute with fixed-point numbers?

How to compute with fixed-point numbers?

Toy example: degree-1 polynomial evaluation with 8 bit numbers

- $P(x) = a_0 + (x \cdot a_1)$

	Format	Int. repr.	Decimal value
a_0	$\mathbf{Q}_{2.6}$	224	
a_1	$\mathbf{Q}_{1.7}$	192	
x	$\mathbf{Q}_{3.5}$	$[16, 208]$	

How to compute with fixed-point numbers?

Toy example: degree-1 polynomial evaluation with 8 bit numbers

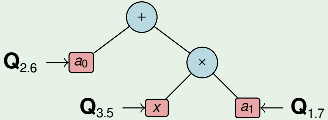
- $P(x) = a_0 + (x \cdot a_1)$

	Format	Int. repr.	Decimal value
a_0	$\mathbf{Q}_{2.6}$	224	3.5
a_1	$\mathbf{Q}_{1.7}$	192	1.5
x	$\mathbf{Q}_{3.5}$	$[16, 208]$	$[0.5, 6.5]$

How to compute with fixed-point numbers?

Toy example: degree-1 polynomial evaluation with 8 bit numbers

- $P(x) = a_0 + (x \cdot a_1)$



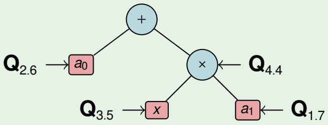
	Format	Int. repr.	Decimal value
a_0	$\mathbf{Q}_{2.6}$	224	3.5
a_1	$\mathbf{Q}_{1.7}$	192	1.5
x	$\mathbf{Q}_{3.5}$	$[16, 208]$	$[0.5, 6.5]$

1. Pick an evaluation scheme

How to compute with fixed-point numbers?

Toy example: degree-1 polynomial evaluation with 8 bit numbers

- $P(x) = a_0 + (x \cdot a_1)$



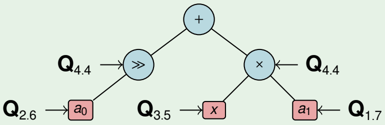
	Format	Int. repr.	Decimal value
a_0	$\mathbf{Q}_{2.6}$	224	3.5
a_1	$\mathbf{Q}_{1.7}$	192	1.5
x	$\mathbf{Q}_{3.5}$	$[16, 208]$	$[0.5, 6.5]$
$u = (x \cdot a_1)$	$\mathbf{Q}_{4.4}$	$[12, 156]$	$[0.75, 9.75]$

1. Pick an evaluation scheme
2. Determine the range and fixed-point formats of intermediate variables

How to compute with fixed-point numbers?

Toy example: degree-1 polynomial evaluation with 8 bit numbers

- $P(x) = a_0 + (x \cdot a_1)$



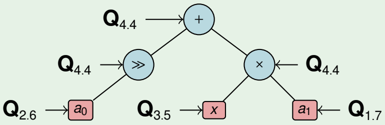
	Format	Int. repr.	Decimal value
a_0	$\mathbf{Q}_{2.6}$	224	3.5
a_1	$\mathbf{Q}_{1.7}$	192	1.5
x	$\mathbf{Q}_{3.5}$	$[16, 208]$	$[0.5, 6.5]$
$u = (x \cdot a_1)$	$\mathbf{Q}_{4.4}$	$[12, 156]$	$[0.75, 9.75]$
$v = (a_0 \gg 2)$	$\mathbf{Q}_{4.4}$	56	3.5

1. Pick an evaluation scheme
2. Determine the range and fixed-point formats of intermediate variables

How to compute with fixed-point numbers?

Toy example: degree-1 polynomial evaluation with 8 bit numbers

- $P(x) = a_0 + (x \cdot a_1)$



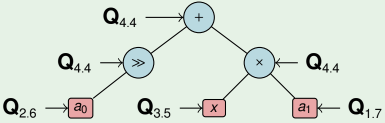
	Format	Int. repr.	Decimal value
a_0	$\mathbf{Q}_{2.6}$	224	3.5
a_1	$\mathbf{Q}_{1.7}$	192	1.5
x	$\mathbf{Q}_{3.5}$	$[16, 208]$	$[0.5, 6.5]$
$u = (x \cdot a_1)$	$\mathbf{Q}_{4.4}$	$[12, 156]$	$[0.75, 9.75]$
$v = (a_0 \gg 2)$	$\mathbf{Q}_{4.4}$	56	3.5
$w = (v + u)$	$\mathbf{Q}_{4.4}$	$[68, 212]$	$[4.25, 13.25]$

1. Pick an evaluation scheme
2. Determine the range and fixed-point formats of intermediate variables

How to compute with fixed-point numbers?

Toy example: degree-1 polynomial evaluation with 8 bit numbers

- $P(x) = a_0 + (x \cdot a_1)$



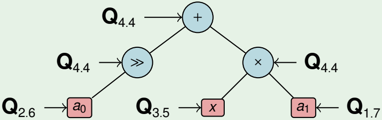
	Format	Int. repr.	Decimal value
a_0	$\mathbf{Q}_{2.6}$	224	3.5
a_1	$\mathbf{Q}_{1.7}$	192	1.5
x	$\mathbf{Q}_{3.5}$	$[16, 208]$	$[0.5, 6.5]$
$u = (x \cdot a_1)$	$\mathbf{Q}_{4.4}$	$[12, 156]$	$[0.75, 9.75]$
$v = (a_0 \gg 2)$	$\mathbf{Q}_{4.4}$	56	3.5
$w = (v + u)$	$\mathbf{Q}_{4.4}$	$[68, 212]$	$[4.25, 13.25]$

```
uint8_t pol(uint8_t x /*Q3.5*/)
{
  uint8_t u = mul ( x , 224 );
  uint8_t v = 224 >> 2;
  uint8_t w = v + u;
  return w;              /*Q4.4*/
}
```

How to compute with fixed-point numbers?

Toy example: degree-1 polynomial evaluation with 8 bit numbers

- $P(x) = a_0 + (x \cdot a_1)$



	Format	Int. repr.	Decimal value
a_0	$\mathbf{Q}_{2.6}$	224	3.5
a_1	$\mathbf{Q}_{1.7}$	192	1.5
x	$\mathbf{Q}_{3.5}$	$[16, 208]$	$[0.5, 6.5]$
$u = (x \cdot a_1)$	$\mathbf{Q}_{4.4}$	$[12, 156]$	$[0.75, 9.75]$
$v = (a_0 \gg 2)$	$\mathbf{Q}_{4.4}$	56	3.5
$w = (v + u)$	$\mathbf{Q}_{4.4}$	$[68, 212]$	$[4.25, 13.25]$

```
uint8_t pol(uint8_t x /*Q3.5*/)
{
  uint8_t u = mul ( x , 224 );
  uint8_t v = 224 >> 2;
  uint8_t w = v + u;
  return w;            /*Q4.4*/
}
```

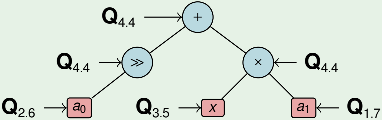
Floating-point version

```
float pol(float x)
{
  return 3.5+1.5*x;
}
```

How to compute with fixed-point numbers?

Toy example: degree-1 polynomial evaluation with 8 bit numbers

- $P(x) = a_0 + (x \cdot a_1)$



	Format	Int. repr.	Decimal value
a_0	$\mathbf{Q}_{2.6}$	224	3.5
a_1	$\mathbf{Q}_{1.7}$	192	1.5
x	$\mathbf{Q}_{3.5}$	$[16,208]$	$[0.5,6.5]$
$u = (x \cdot a_1)$	$\mathbf{Q}_{4.4}$	$[12,156]$	$[0.75,9.75]$
$v = (a_0 \gg 2)$	$\mathbf{Q}_{4.4}$	56	3.5
$w = (v + u)$	$\mathbf{Q}_{4.4}$	$[68,212]$	$[4.25,13.25]$

```
uint8_t pol(uint8_t x /*Q3.5*/)
{
  uint8_t u = mul ( x , 224 );
  uint8_t v = 224 >> 2;
  uint8_t w = v + u;
  return w;            /*Q4.4*/
}
```

Floating-point version

```
float pol(float x)
{
  return 3.5+1.5*x;
}
```

Even for small problems, this process is tedious: How to automate it?

An interval arithmetic based model

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

Val(v) is the range of v

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 $\mathbf{Val}(v) = [\underline{\mathbf{v}}, \overline{\mathbf{v}}]$
  - $i = \left\lceil \log_2 \left(\max\left(|\underline{\mathbf{v}}|, |\overline{\mathbf{v}}|\right)\right) \right\rceil + \alpha$
  - $f = k - i$

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

$\mathbf{Err}(v)$ 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

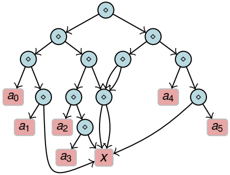
$\mathbf{Val}(v)$ is the range of v

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

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

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

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



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

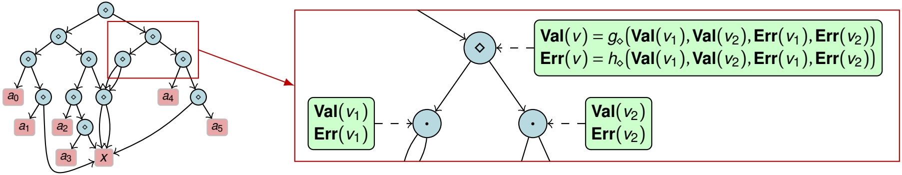
$\mathbf{Val}(v)$ is the range of v

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

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

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

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

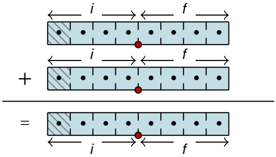


How to propagate $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ for $\diamond \in \{+, -, \times, \ll, \gg, \sqrt{\ }, /\}$?

Fixed-point addition

- The two variables v_1 and v_2 must be aligned: in the same fixed-point format $\mathbf{Q}_{i.f}$

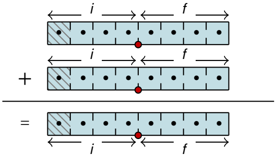
1. If $\mathbf{Val}(v_1) + \mathbf{Val}(v_2) \subseteq \mathscr{R}\mathrm{ange}(\mathbf{Q}_{i.f})$, overflow cannot occur



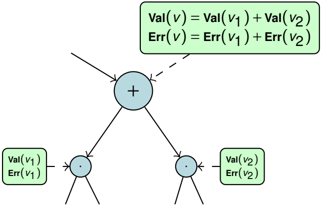
Fixed-point addition

- The two variables v_1 and v_2 must be aligned: in the same fixed-point format $\mathbf{Q}_{i.f}$

1. If $\mathbf{Val}(v_1) + \mathbf{Val}(v_2) \subseteq \mathscr{R}\text{ange}(\mathbf{Q}_{i.f})$, overflow cannot occur



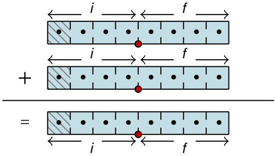
$\rightsquigarrow$



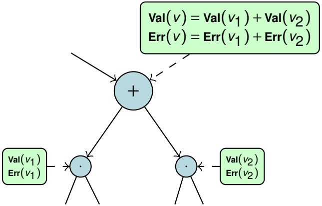
Fixed-point addition

- The two variables v_1 and v_2 must be aligned: in the same fixed-point format $\mathbf{Q}_{i.f}$

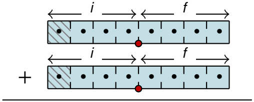
1. If $\mathbf{Val}(v_1) + \mathbf{Val}(v_2) \subseteq \mathscr{R}\text{ange}(\mathbf{Q}_{i.f})$, overflow cannot occur



$\rightsquigarrow$



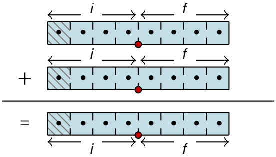
2. If $\mathscr{R}\text{ange}(\mathbf{Q}_{i.f}) \subset \mathbf{Val}(v_1) + \mathbf{Val}(v_2) \subset \mathscr{R}\text{ange}(\mathbf{Q}_{i+1.f-1})$, overflow may occur



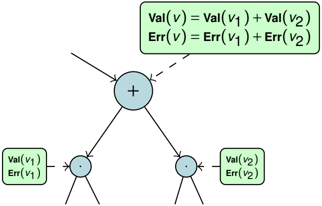
Fixed-point addition

- The two variables v_1 and v_2 must be aligned: in the same fixed-point format $\mathbf{Q}_{i.f}$

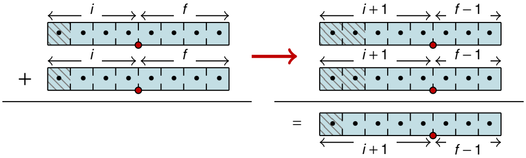
1. If $\mathbf{Val}(v_1) + \mathbf{Val}(v_2) \subseteq \mathscr{R}\text{ange}(\mathbf{Q}_{i.f})$, overflow cannot occur



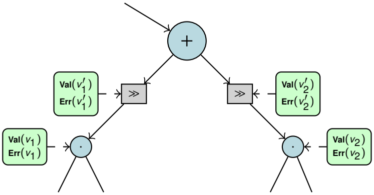
$\rightsquigarrow$



2. If $\mathscr{R}\text{ange}(\mathbf{Q}_{i.f}) \subset \mathbf{Val}(v_1) + \mathbf{Val}(v_2) \subset \mathscr{R}\text{ange}(\mathbf{Q}_{i+1.f-1})$, overflow may occur



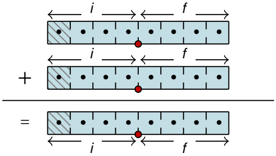
$\rightsquigarrow$



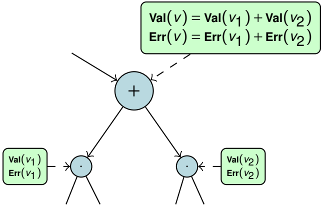
Fixed-point addition

- The two variables v_1 and v_2 must be aligned: in the same fixed-point format $\mathbf{Q}_{i.f}$

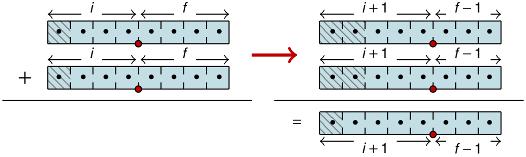
1. If $\mathbf{Val}(v_1) + \mathbf{Val}(v_2) \subseteq \mathscr{R}\text{ange}(\mathbf{Q}_{i.f})$, overflow cannot occur



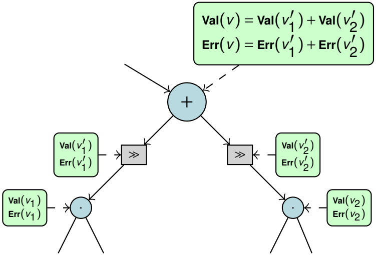
$\rightsquigarrow$



2. If $\mathscr{R}\text{ange}(\mathbf{Q}_{i.f}) \subset \mathbf{Val}(v_1) + \mathbf{Val}(v_2) \subset \mathscr{R}\text{ange}(\mathbf{Q}_{i+1.f-1})$, overflow may occur

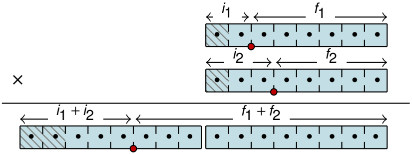


$\rightsquigarrow$



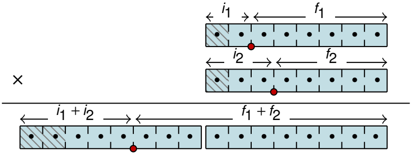
Fixed-point multiplication

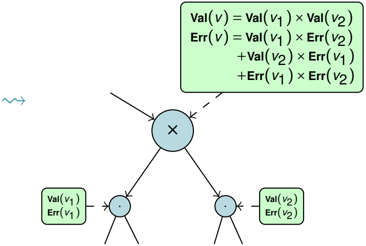
- The output format of a $\mathbf{Q}_{i_1.f_1} \times \mathbf{Q}_{i_2.f_2}$ is $\mathbf{Q}_{i_1+i_2.f_1+f_2}$



Fixed-point multiplication

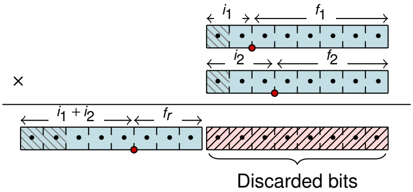
- The output format of a $\mathbf{Q}_{i_1.f_1} \times \mathbf{Q}_{i_2.f_2}$ is $\mathbf{Q}_{i_1+i_2.f_1+f_2}$



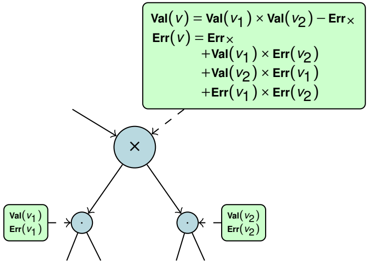


Fixed-point multiplication

- 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



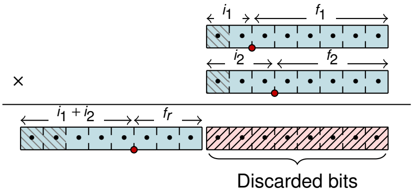
$\rightsquigarrow$

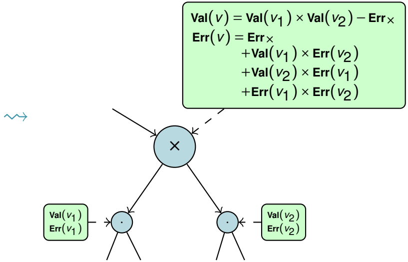


- $\mathbf{Err}_\times = \left[0, 2^{-f_r} - 2^{-(f_1+f_2)}\right]$

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



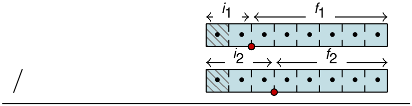


- $\mathbf{Err}_\times = \left[0, 2^{-f_r} - 2^{-(f_1+f_2)}\right]$
- This multiplication is available on integer processors and DSPs

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

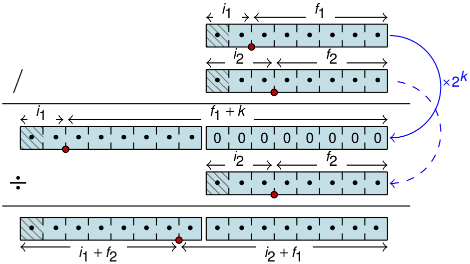
Our new fixed-point division

- 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

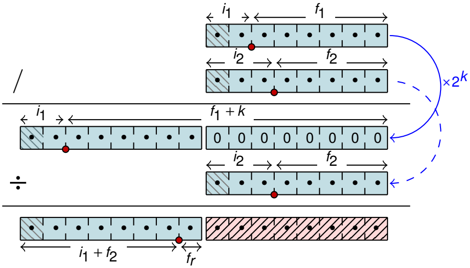
- The output integer part of $\mathbf{Q}_{i_1.f_1}/\mathbf{Q}_{i_2.f_2}$ may be as large as $i_1 + f_2$



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

Our new fixed-point division

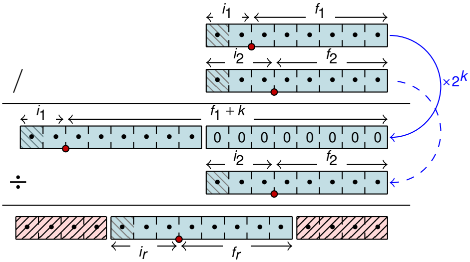
- The output integer part of $\mathbf{Q}_{i_1.f_1}/\mathbf{Q}_{i_2.f_2}$ may be as large as $i_1 + f_2$
- But, doubling the word-length is costly



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

Our new fixed-point division

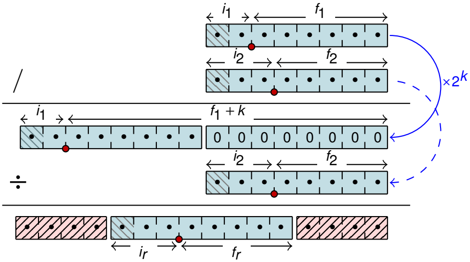
- The output integer part of $\mathbf{Q}_{i_1.f_1}/\mathbf{Q}_{i_2.f_2}$ may be as large as $i_1 + f_2$
- But, doubling the word-length is costly
- How to obtain sharper a error bounds on $\mathbf{Err}_{/}$?



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



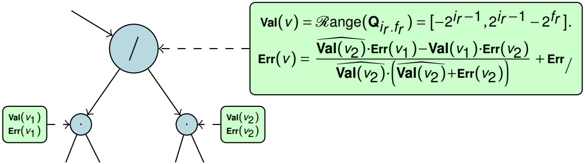
- $\mathbf{Err}_{/} = [-2^{f_r}, 2^{f_r}]$
  - ☺ sharper bound
  - ☹ risk of overflow at run-time

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

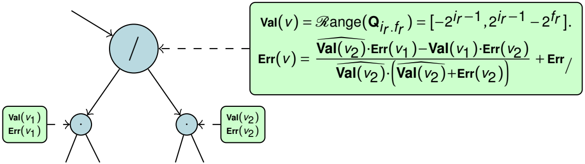
- Once the output format decided $\mathbf{Q}_{i_r.f_r}$



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

The propagation rule and implementation of division

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



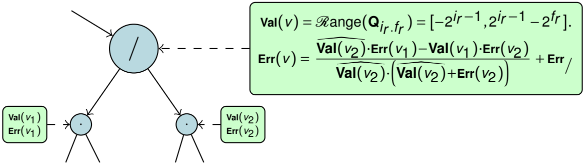
- $\widehat{\mathbf{Val}(v_2)} = \dfrac{\mathbf{Val}(v_1)}{\widehat{\mathbf{Val}(v)} + \mathbf{Err}_{/}} \cap \mathbf{Val}(v_2)$ and $\widehat{\mathbf{Val}(v)} = [-2^{i_r-1}, -2^{-f_r}] \cup [2^{-f_r}, 2^{i_r-1} - 2^{f_r}]$

```
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 $\mathbf{Q}_{i_r.f_r}$



- $\widehat{\mathbf{Val}(v_2)} = \dfrac{\mathbf{Val}(v_1)}{\widehat{\mathbf{Val}(v)} + \mathbf{Err}_/} \cap \mathbf{Val}(v_2)$ 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;
  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

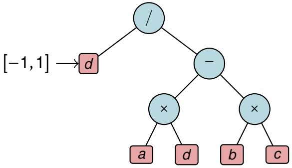
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$

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 $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

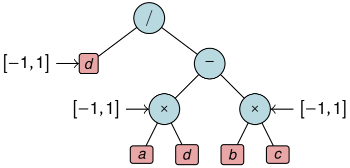
The division format trade-off: case of inverting 2×2 matrices

- 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 $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$



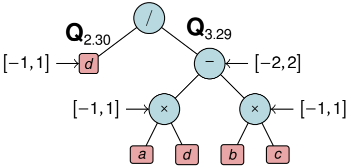
The division format trade-off: case of inverting 2×2 matrices

- 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 $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$



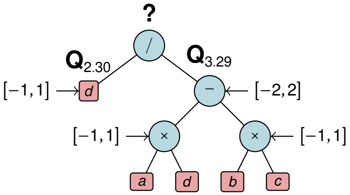
The division format trade-off: case of inverting 2×2 matrices

- 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 $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$



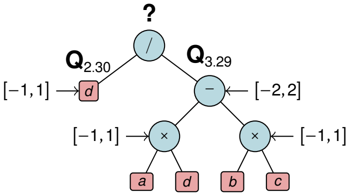
The division format trade-off: case of inverting 2×2 matrices

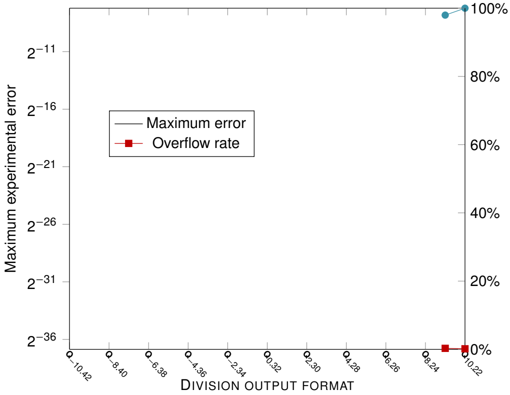
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$



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

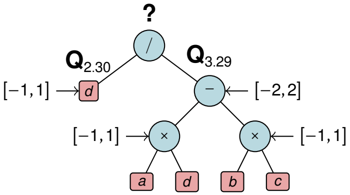
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

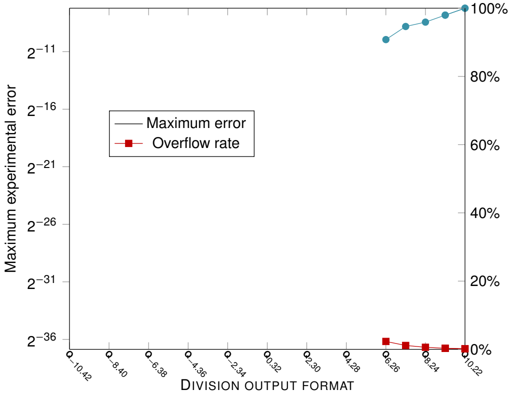




The division format trade-off: case of inverting 2×2 matrices

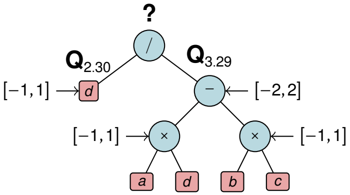
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

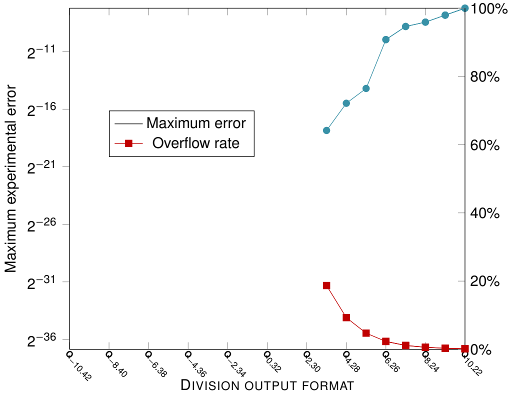




The division format trade-off: case of inverting 2×2 matrices

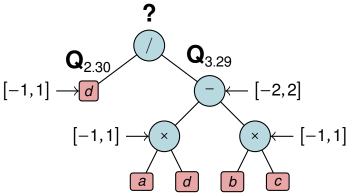
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

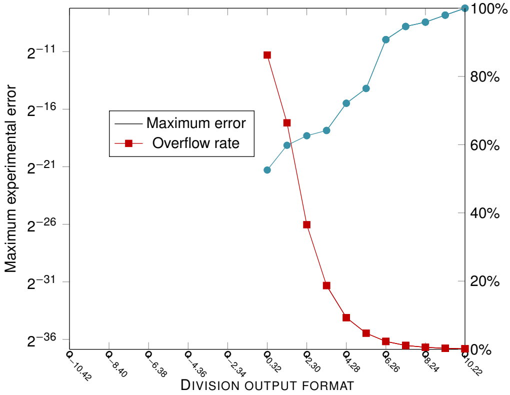




The division format trade-off: case of inverting 2×2 matrices

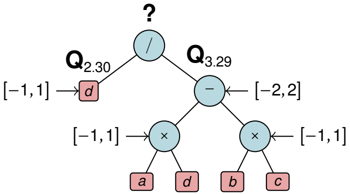
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

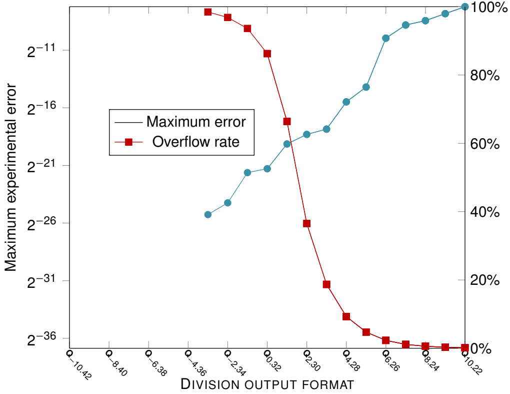




The division format trade-off: case of inverting 2×2 matrices

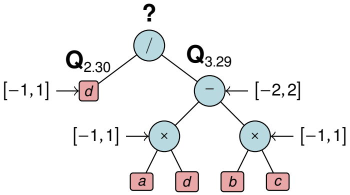
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

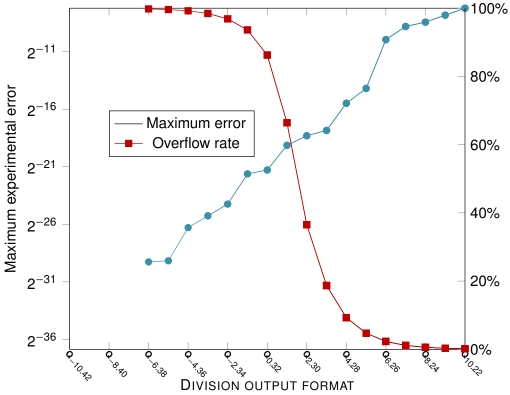




The division format trade-off: case of inverting 2×2 matrices

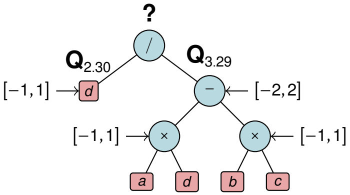
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

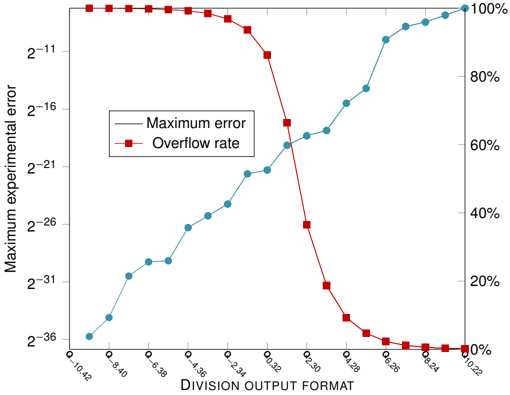




The division format trade-off: case of inverting 2×2 matrices

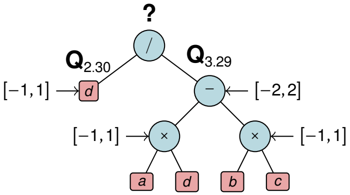
- Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbf{Q}_{2.30}$
- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} \frac{d}{\Delta} & \frac{-b}{\Delta} \\ \frac{-c}{\Delta} & \frac{a}{\Delta} \end{pmatrix}$

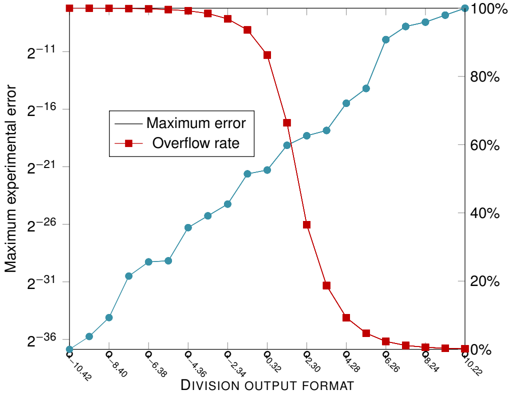




The division format trade-off: case of inverting 2×2 matrices

- 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

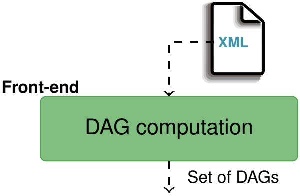
The CGPE tool

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

The CGPE tool

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

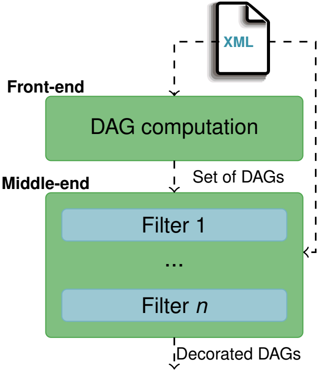
1. Computation step $\rightsquigarrow$ front-end
   - computes evaluation schemes $\rightsquigarrow$ DAGs



The CGPE tool

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

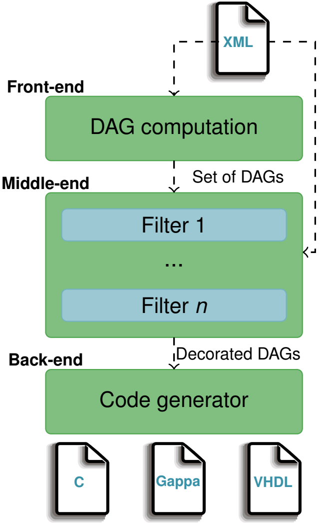
1. Computation step $\rightsquigarrow$ front-end
   - computes evaluation schemes $\rightsquigarrow$ DAGs
2. Filtering step $\rightsquigarrow$ middle-end
   - applies the arithmetic model
   - prunes the DAGs that do not satisfy different criteria:
     - latency $\rightsquigarrow$ scheduling filter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



The CGPE tool

- CGPE (*Code Generation for Polynomial Evaluation*): initiated by Revy [?]
  - 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 $0.3 \cdot \pi$:

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

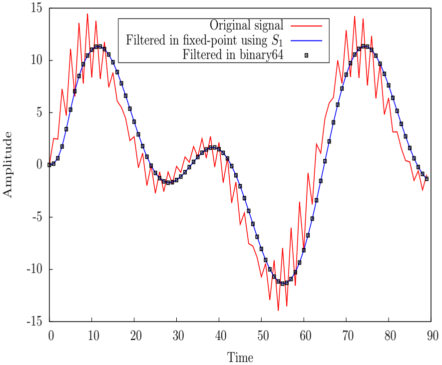
```xml
<dotproduct inf="0xb1e91685" sup="0x4e16e97b" integer_width="6" fraction_width="26" width="32">
  <coefficient name="b0" value="0x65718e3b" integer_width="-3" fraction_width="35" width="32"/>
    ...
  <variable name="y3" inf="0xb1e91685" sup="0x4e16e97b" integer_width="6" fraction_width="26"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

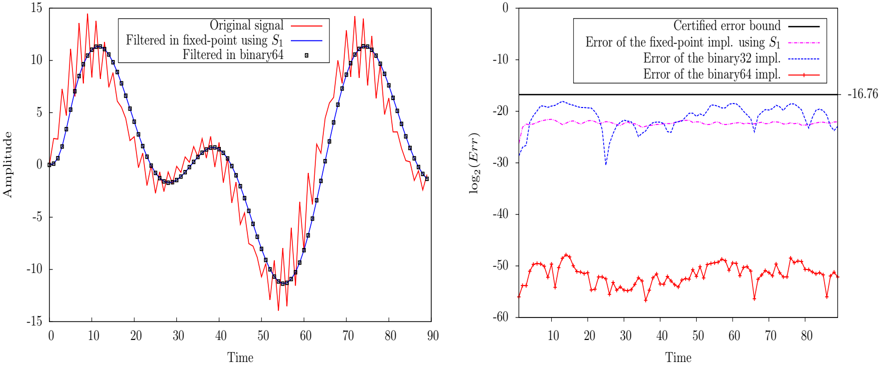


Code synthesis for an IIR filter using CGPE

- 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;
}
```

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;
}
```

Instruction selection to handle advanced instructions

- It is a well known problem in compilation $\rightsquigarrow$ proven to be NP-complete on DAGs
- Usually solved using a tiling algorithm:
  - input:
    - a DAG representing an arithmetic expression,
    - a set of tiles, with a cost for each,
    - a function that associates a cost to a DAG.
  - output: a set of covering tiles that minimize the cost function.

Instruction selection to handle advanced instructions

- It is a well known problem in compilation $\rightsquigarrow$ proven to be NP-complete on DAGs
- Usually solved using a tiling algorithm:
  - input:
    - a DAG representing an arithmetic expression,
    - a set of tiles, with a cost for each,
    - a function that associates a cost to a DAG.
  - output: a set of covering tiles that minimize the cost function.

Implementation in CGPE

- XML architecture description file that contains for each instruction:
  - its name, its type (signed or unsigned), its latency (# cycles),
  - a description of the pattern it matches,
  - a C macro to emulate it in software,
  - and a piece of Gappa script to compute the error entailed by its evaluation in fixed-point arithmetic.

The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

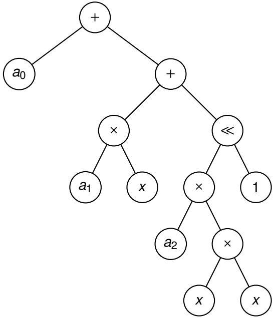
1: BottomUpDP() + TopDownSelect()
2: ImproveCSEDecision()
3: BottomUpDP() + TopDownSelect()

- Example: how to evaluate $a_0 + \Big((a_1 \cdot x) + \big((a_2 \cdot (x \cdot x)) \ll 1\big)\Big)$?

addition / shift $\rightsquigarrow$ 1 cycle

shift-and-add $\rightsquigarrow$ 1 cycle

multiplication $\rightsquigarrow$ 3 cycles

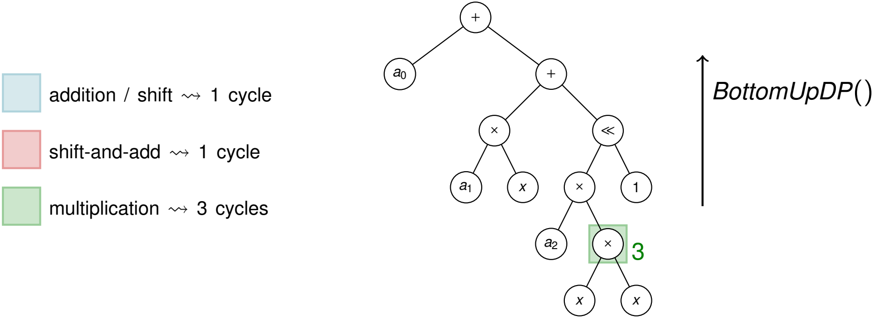


The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

1: BottomUpDP() + TopDownSelect()
2: ImproveCSEDecision()
3: BottomUpDP() + TopDownSelect()

- Example: how to evaluate $a_0 + \Big((a_1 \cdot x) + \big((a_2 \cdot (x \cdot x)) \ll 1\big)\Big)$?



The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

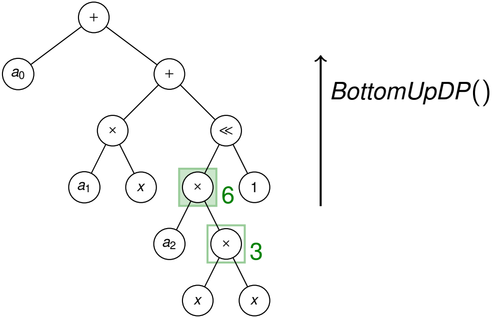
1: BottomUpDP() + TopDownSelect()
2: ImproveCSEDecision()
3: BottomUpDP() + TopDownSelect()

- Example: how to evaluate $a_0 + \Big((a_1 \cdot x) + \big((a_2 \cdot (x \cdot x)) \ll 1\big)\Big)$?

addition / shift $\rightsquigarrow$ 1 cycle

shift-and-add $\rightsquigarrow$ 1 cycle

multiplication $\rightsquigarrow$ 3 cycles



The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

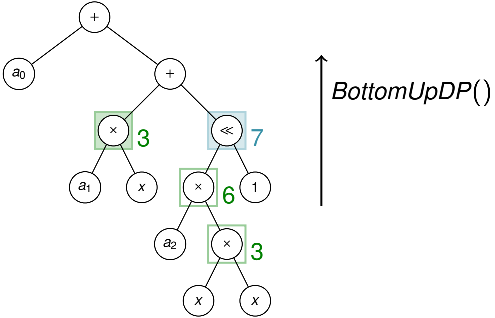
1: BottomUpDP() + TopDownSelect()
2: ImproveCSEDecision()
3: BottomUpDP() + TopDownSelect()

- Example: how to evaluate $a_0 + \Big((a_1 \cdot x) + \big((a_2 \cdot (x \cdot x)) \ll 1\big)\Big)$?

addition / shift $\rightsquigarrow$ 1 cycle

shift-and-add $\rightsquigarrow$ 1 cycle

multiplication $\rightsquigarrow$ 3 cycles



The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

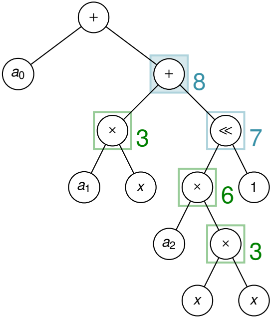
1: BottomUpDP() + TopDownSelect()
2: ImproveCSEDecision()
3: BottomUpDP() + TopDownSelect()

- Example: how to evaluate $a_0 + \Big((a_1 \cdot x) + \big((a_2 \cdot (x \cdot x)) \ll 1\big)\Big)$?

addition / shift $\rightsquigarrow$ 1 cycle

shift-and-add $\rightsquigarrow$ 1 cycle

multiplication $\rightsquigarrow$ 3 cycles



BottomUpDP()

The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

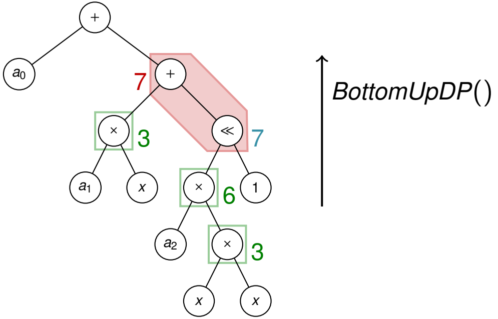
1: BottomUpDP() + TopDownSelect()
2: ImproveCSEDecision()
3: BottomUpDP() + TopDownSelect()

- Example: how to evaluate $a_0 + \Big((a_1 \cdot x) + \big((a_2 \cdot (x \cdot x)) \ll 1\big)\Big)$?

addition / shift $\rightsquigarrow$ 1 cycle

shift-and-add $\rightsquigarrow$ 1 cycle

multiplication $\rightsquigarrow$ 3 cycles

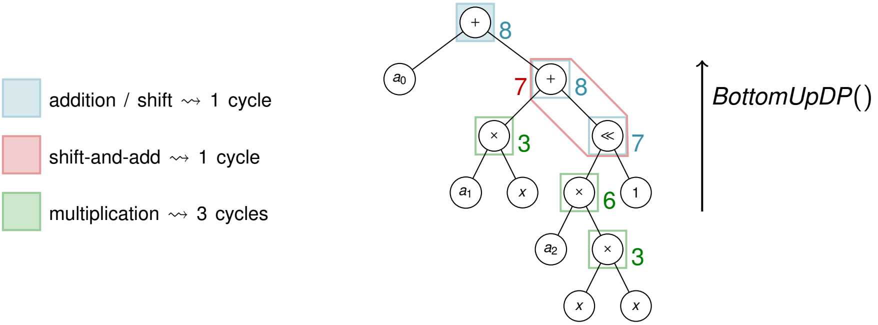


The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

1: BottomUpDP() + TopDownSelect()
2: ImproveCSEDecision()
3: BottomUpDP() + TopDownSelect()

- Example: how to evaluate $a_0 + \Big((a_1 \cdot x) + \big((a_2 \cdot (x \cdot x)) \ll 1\big)\Big)$?



The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

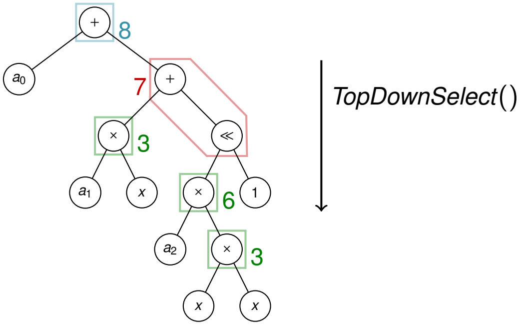
1: BottomUpDP() + TopDownSelect()
2: ImproveCSEDecision()
3: BottomUpDP() + TopDownSelect()

- Example: how to evaluate $a_0 + \Big((a_1 \cdot x) + \big((a_2 \cdot (x \cdot x)) \ll 1\big)\Big)$?

addition / shift $\rightsquigarrow$ 1 cycle

shift-and-add $\rightsquigarrow$ 1 cycle

multiplication $\rightsquigarrow$ 3 cycles



The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

1: BottomUpDP() + TopDownSelect()
2: ~~ImproveCSEDecision()~~
3: ~~BottomUpDP() + TopDownSelect()~~

- Example: how to evaluate $a_0 + \Big((a_1 \cdot x) + \big((a_2 \cdot (x \cdot x)) \ll 1\big)\Big)$?

- In our case, only the first step of NOLTIS is valuable.

- NOLTIS algorithm mainly relies on the evaluation of a cost function. We have implemented three different cost functions:
  - number of operator (regardless common subexpressions)
  - evaluation latency on unbounded parallelism
  - evaluation accuracy, computed using the Gappa script of each instruction

Impact on the number of instructions

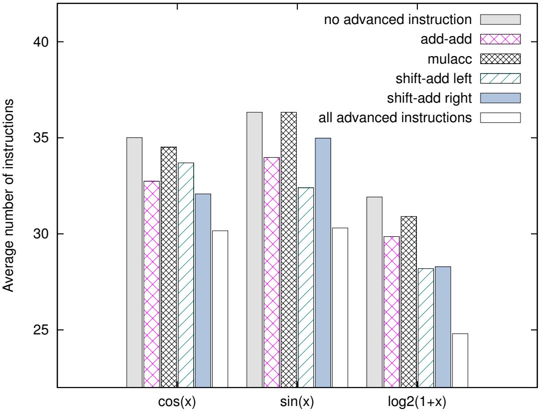


Figure: Average number of instructions in 50 synthesized codes, for the evaluation of polynomials of degree 5 up to 12 for various elementary functions.

- Remark 1: average reduction of 8.7 % up to 13.75 %
- Remark 2: interest of ST231 shift-and-add with left shift for $\sin(x)$ implementation
  $\rightsquigarrow$ reduction of 8.7 %
- Remark 3: interest of shift-and-add with right shift for $\cos(x)$ and $\log_2(1+x)$ implementation
  $\rightsquigarrow$ reduction of 12.8 % and 13.75 %, respectively

Impact on the accuracy of some functions

$f(x)$	$\mathscr{I}$	d	$\log_2(\epsilon)$ not optimized	$\log_2(\epsilon)$ optimized
$\exp(x)-1$	$[-0.25, 0.25]$	7	-26.98	-27.34
$\exp(x)$	$[0, 1]$	7	-13.94	-14.90
$\sin(x)$	$[-0.5, 0.5]$	9	-18.95	-19.91
$\cos(x)$	$[-0.5, 0.25]$	5	-27.01	-27.26
$\tan(x)$	$[0.25, 0.5]$	9	-18.81	-19.64
$\log_2(1+x)/x$	$[2^{-23}, 1]$	7	-13.94	-14.89
$\sqrt{1+x}$	$[2^{-23}, 1]$	7	-13.94	-14.90

Table: Impact of the accuracy based selection step on the certified accuracy of the generated code for various functions.

- Remark 1: with a `mulacc` that computes `(a * b) + (c >> n)` with $n \in \{1, \cdots, 31\}$ with one final rounding
- Remark 2: more accurate results for all the cases, almost up to 1 bit of accuracy for *exp*, *sin*, $\log_2$ and $\sqrt{1+x}$

Outline of the talk

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

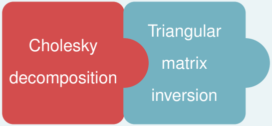
Let M be a matrix of fixed-point variables,

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

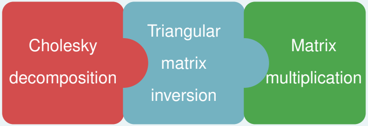
Let M be a matrix of fixed-point variables,

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

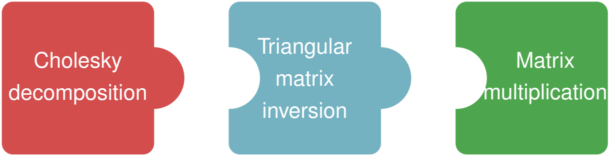
1. Generate certified code to compute B a lower triangular s.t. $M' = B \cdot B^T$
2. Generate certified code to compute $N = B^{-1}$
3. Generate certified code to compute $M'^{-1} = N^T \cdot N$

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

Matrix multiplication

Inputs

- Two matrices A and B of fixed-point variables

$$A \in \mathbb{F}\mathrm{ix}^{n \times n} \quad \text{and} \quad B \in \mathbb{F}\mathrm{ix}^{n \times n}$$

- A bound $\mathscr{C}_1$ on the roundoff error
- A bound $\mathscr{C}_2$ on the code size

Matrix multiplication

Inputs

- Two matrices A and B of fixed-point variables

$$A \in \mathbb{F}\mathrm{ix}^{n \times n} \quad \text{and} \quad B \in \mathbb{F}\mathrm{ix}^{n \times n}$$

- A bound $\mathscr{C}_1$ on the roundoff error
- A bound $\mathscr{C}_2$ on the code size

Output

- Fixed-point code (C, VHDL, ...) that evaluates the product

$$C' = A' \cdot B', \quad \text{where} \quad A' \in A \quad \text{and} \quad B' \in B$$

  that satisfy both $\mathscr{C}_1$ and $\mathscr{C}_2$

- Accuracy certificate (verifiable by a formal proof checker)

Straightforward algorithms

Accurate algorithm

- Main idea: a dot product code for each coefficient of the resulting matrix

Accurate algorithm

Inputs:

Two matrices $A \in \mathbb{F}ix^{n \times n}$ and $B \in \mathbb{F}ix^{n \times n}$

Outputs:

C code to compute the product $A \cdot B$

$n \cdot n$ accuracy certificates

Steps:

1: **for** $1 < i \leq n$ **do**
2:     **for** $1 < j \leq n$ **do**
3:         $DPSynthesis(A_{i,:}, B_{:,j})$
4:     **end for**
5: **end for**
6: *Check* $\mathscr{C}_1$ *and* $\mathscr{C}_2$

Straightforward algorithms

Accurate algorithm

- Main idea: a dot product code for each coefficient of the resulting matrix

Accurate algorithm

Inputs:

Two matrices $A \in \mathbb{F}ix^{n \times n}$ and $B \in \mathbb{F}ix^{n \times n}$

Outputs:

C code to compute the product $A \cdot B$

$n \cdot n$ accuracy certificates

Steps:

1: **for** $1 < i \leq n$ **do**
2:     **for** $1 < j \leq n$ **do**
3:         $\mathit{DPSynthesis}(A_{i,:}, B_{:,j})$
4:     **end for**
5: **end for**
6: *Check* $\mathscr{C}_1$ *and* $\mathscr{C}_2$

Compact algorithm

- Main idea: a unique dot product code for all the coefficient of the resulting matrix

Compact algorithm

Inputs:

Two matrices $A \in \mathbb{F}ix^{n \times n}$ and $B \in \mathbb{F}ix^{n \times n}$

Outputs:

C code to compute the product $A \cdot B$

1 accuracy certificate

Steps:

1: $\mathscr{U} = A_{1,:} \cup A_{2,:} \cup \cdots \cup A_{n,:}$, with $\mathscr{U} \in \mathbb{F}ix^{1 \times n}$
2: $\mathscr{V} = B_{:,1} \cup B_{:,2} \cup \cdots \cup B_{:,n}$, with $\mathscr{V} \in \mathbb{F}ix^{n \times 1}$
3: $\mathit{DPSynthesis}(\mathscr{U}, \mathscr{V})$
4: *Check* $\mathscr{C}_1$ *and* $\mathscr{C}_2$

Example of a multiplication of 2×2 matrices

We consider code synthesis for the multiplication of matrices $A' \in A$ and $B' \in B$:

$$A = \begin{pmatrix} [-1000, 1000] & [-3000, 3000] \\ [-1, 1] & [-1, 1] \end{pmatrix} \text{ and } B = \begin{pmatrix} [-2000, 2000] & [-2, 2] \\ [-4000, 4000] & [-10, 10] \end{pmatrix}$$

Coefficient	$A_{1,1}$	$A_{1,2}$	$A_{2,1}$	$A_{2,2}$	$B_{1,1}$	$B_{1,2}$	$B_{2,1}$	$B_{2,2}$
Fixed-point format	$\mathbf{Q}_{11.21}$	$\mathbf{Q}_{12.20}$	$\mathbf{Q}_{2.30}$	$\mathbf{Q}_{2.30}$	$\mathbf{Q}_{11.21}$	$\mathbf{Q}_{3.29}$	$\mathbf{Q}_{2.30}$	$\mathbf{Q}_{5.27}$

Example of a multiplication of 2×2 matrices

We consider code synthesis for the multiplication of matrices $A' \in A$ and $B' \in B$:

$$A = \begin{pmatrix} [-1000,1000] & [-3000,3000] \\ [-1,1] & [-1,1] \end{pmatrix} \text{ and } B = \begin{pmatrix} [-2000,2000] & [-2,2] \\ [-4000,4000] & [-10,10] \end{pmatrix}$$

Coefficient	$A_{1,1}$	$A_{1,2}$	$A_{2,1}$	$A_{2,2}$	$B_{1,1}$	$B_{1,2}$	$B_{2,1}$	$B_{2,2}$
Fixed-point format	$\mathbf{Q}_{11.21}$	$\mathbf{Q}_{12.20}$	$\mathbf{Q}_{2.30}$	$\mathbf{Q}_{2.30}$	$\mathbf{Q}_{11.21}$	$\mathbf{Q}_{3.29}$	$\mathbf{Q}_{2.30}$	$\mathbf{Q}_{5.27}$

Accurate algorithm

Dot-product	$A_{1,:} \cdot B_{:,1}$	$A_{1,:} \cdot B_{:,2}$	$A_{2,:} \cdot B_{:,1}$	$A_{2,:} \cdot B_{:,2}$
Evaluated using	$\text{DPCode}_{1,1}$	$\text{DPCode}_{1,2}$	$\text{DPCode}_{2,1}$	$\text{DPCode}_{2,2}$
Output format	$Q_{26,6}$	$Q_{18,14}$	$Q_{15,17}$	$Q_{7,25}$
Certified error	$\approx 2^{-5}$	$\approx 2^{-14}$	$\approx 2^{-16}$	$\approx 2^{-24}$
Maximum error	$\approx 2^{-5}$			
Average error	$\approx 2^{-7}$			

Compact algorithm

Dot-product	$A_{1,:} \cdot B_{:,1}$	$A_{1,:} \cdot B_{:,2}$	$A_{2,:} \cdot B_{:,1}$	$A_{2,:} \cdot B_{:,2}$
Evaluated using	$\text{DPCode}_{\mathscr{U},\mathscr{V}}$			
Output format	$Q_{26,6}$			
Certified error	$\approx 2^{-5}$			
Maximum error	$\approx 2^{-5}$			
Average error	$\approx 2^{-5}$			

Trade-off algorithms

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \qquad B = \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix}$$

$A_{0:}$ $A_{1:}$ $A_{4:}$ $A_{2:}$ $A_{3:}$

Accurate algorithm:

(25 dot-product codes)

$B_{:0}$ $B_{:1}$ $B_{:4}$ $B_{:2}$ $B_{:3}$

Trade-off algorithms

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix}$$

$A_{0:}$ $A_{1:}$ $A_{4:}$ $A_{2:}$ $A_{3:}$

Compact algorithm:

(1 dot-product code)

$B_{:0}$ $B_{:1}$ $B_{:4}$ $B_{:2}$ $B_{:3}$

Trade-off algorithms

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix}$$

$A_{0:}$ $A_{1:}$ $A_{4:}$ $A_{2:}$ $A_{3:}$

Trade-off algorithm:
(9 dot-product codes)

$B_{:0}$ $B_{:1}$ $B_{:4}$ $B_{:2}$ $B_{:3}$

Trade-off algorithms

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \qquad B = \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix}$$

Number of possible trade-off algorithms

- Given by $\mathscr{B}(n)^2$ with $\mathscr{B}(n)$ the n^{th} Bell number

n	5	6	10	16	25	64	···
$\mathscr{B}(\mathbf{n})^2$	2704	41 209	$\approx 2^{34}$	$\approx 2^{66}$	$\approx 2^{124}$	$\approx 2^{433}$	···

Trade-off algorithms

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix}$$

Number of possible trade-off algorithms

- Given by $\mathscr{B}(n)^2$ with $\mathscr{B}(n)$ the n^{th} Bell number

n	5	6	10	16	25	64	$\cdots$
$\mathscr{B}(\mathbf{n})^{\mathbf{2}}$	2704	41 209	$\approx 2^{34}$	$\approx 2^{66}$	$\approx 2^{124}$	$\approx 2^{433}$	$\cdots$

How to find in reasonable time a partition that reduces cost size without harming the accuracy?

Distances between fixed-point variables

The Hausdorff distance

$$d_H : \mathbb{F}ix \times \mathbb{F}ix \rightarrow \mathbb{R}^+$$
$$d_H(I_1, I_2) = max\left\{\left|\underline{I_1} - \underline{I_2}\right|, \left|\overline{I_1} - \overline{I_2}\right|\right\}$$

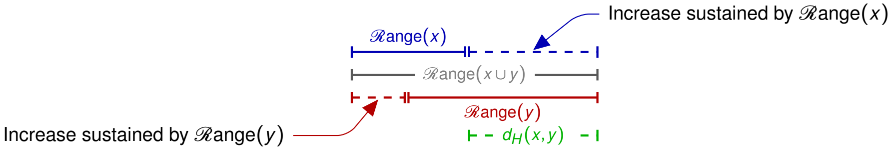
Fixed-point distance

$$d_F : \mathbb{F}ix \times \mathbb{F}ix \rightarrow \mathbb{N}$$
$$d_F(I_1, I_2) = \left|IntegerPart(I_1) - IntegerPart(I_2)\right|$$

Distances between fixed-point variables

The Hausdorff distance

$$d_H : \mathbb{F}ix \times \mathbb{F}ix \to \mathbb{R}^+$$
$$d_H\left(I_1, I_2\right) = max\left\{\left|\underline{I_1} - \underline{I_2}\right|, \left|\overline{I_1} - \overline{I_2}\right|\right\}$$



Fixed-point distance

$$d_F : \mathbb{F}ix \times \mathbb{F}ix \to \mathbb{N}$$
$$d_F\left(I_1, I_2\right) = \left|IntegerPart\left(I_1\right) - IntegerPart\left(I_2\right)\right|$$

Closest pair algorithm

Input:

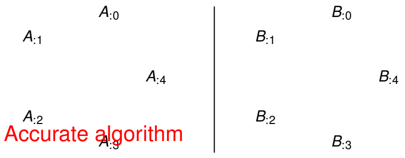
Two matrices $A \in \mathbb{F}\mathrm{ix}^{n \times n}$ and $B \in \mathbb{F}\mathrm{ix}^{n \times n}$
An accuracy bound $\mathscr{C}_1$ (ex. the average error bound is $< \epsilon$)
A code size bound $\mathscr{C}_2$
A metric d

Output:

Code to compute $A \cdot B$ s.t. $\mathscr{C}_1$ and $\mathscr{C}_2$ are satisfied,
or no code otherwise

Algorithm:

$\mathscr{S}_A \leftarrow \{A_{0,:}, \ldots, A_{n-1,:}\}$
$\mathscr{S}_B \leftarrow \{B_{:,0}, \ldots, B_{:,n-1}\}$
while $\mathscr{C}_1$ is satisfied **do**
    $(u_A, v_A), d_A \leftarrow \mathit{findClosestPair}(\mathscr{S}_A, d)$
    $(u_B, v_B), d_B \leftarrow \mathit{findClosestPair}(\mathscr{S}_B, d)$
    if $d_A \leq d_B$ **then**
        $\mathit{remove}(u_A, v_A, \mathscr{S}_A)$
        $\mathit{insert}(u_A \cup v_A, \mathscr{S}_A)$
    else
        $\mathit{remove}(u_B, v_B, \mathscr{S}_B)$
        $\mathit{insert}(u_B \cup v_B, \mathscr{S}_B)$
    end if
    for $(A_i, B_j) \in \mathscr{S}_A \times \mathscr{S}_B$ **do**
        $\mathit{DPSynthesis}(A_i, B_j)$
    end for
end while
/* Revert the last merging step, and check the bound $\mathscr{C}_2$. */



25 DPcodes

$\mathscr{C}_1$ is satisfied

Closest pair algorithm

Input:

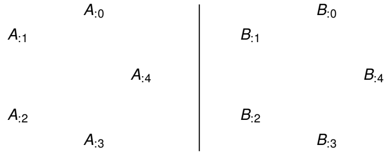
Two matrices $A \in \mathbb{F}\mathrm{ix}^{n \times n}$ and $B \in \mathbb{F}\mathrm{ix}^{n \times n}$
An accuracy bound $\mathscr{C}_1$ (ex. the average error bound is $< \epsilon$)
A code size bound $\mathscr{C}_2$
A metric d

Output:

Code to compute $A \cdot B$ s.t. $\mathscr{C}_1$ and $\mathscr{C}_2$ are satisfied,
or no code otherwise

Algorithm:

$\mathscr{S}_A \leftarrow \{A_{0,:}, \ldots, A_{n-1,:}\}$
$\mathscr{S}_B \leftarrow \{B_{:,0}, \ldots, B_{:,n-1}\}$
while $\mathscr{C}_1$ is satisfied **do**
    $(u_A, v_A), d_A \leftarrow \mathit{findClosestPair}(\mathscr{S}_A, d)$
    $(u_B, v_B), d_B \leftarrow \mathit{findClosestPair}(\mathscr{S}_B, d)$
    if $d_A \leq d_B$ **then**
        $\mathit{remove}(u_A, v_A, \mathscr{S}_A)$
        $\mathit{insert}(u_A \cup v_A, \mathscr{S}_A)$
    else
        $\mathit{remove}(u_B, v_B, \mathscr{S}_B)$
        $\mathit{insert}(u_B \cup v_B, \mathscr{S}_B)$
    end if
    for $(A_i, B_j) \in \mathscr{S}_A \times \mathscr{S}_B$ **do**
        $\mathit{DPSynthesis}(A_i, B_j)$
    end for
end while
/* Revert the last merging step, and check the bound $\mathscr{C}_2$. */



20 DPcodes

$\mathscr{C}_1$ is satisfied

Closest pair algorithm

Input:

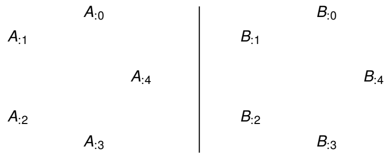
Two matrices $A \in \mathbb{F}\mathrm{ix}^{n\times n}$ and $B \in \mathbb{F}\mathrm{ix}^{n\times n}$
An accuracy bound $\mathscr{C}_1$ (ex. the average error bound is $< \epsilon$)
A code size bound $\mathscr{C}_2$
A metric d

Output:

Code to compute $A \cdot B$ s.t. $\mathscr{C}_1$ and $\mathscr{C}_2$ are satisfied,
or no code otherwise

Algorithm:

$\mathscr{S}_A \leftarrow \{A_{0,:}, \ldots, A_{n-1,:}\}$
$\mathscr{S}_B \leftarrow \{B_{:,0}, \ldots, B_{:,n-1}\}$
while $\mathscr{C}_1$ is satisfied **do**
    $(u_A, v_A), d_A \leftarrow \mathit{findClosestPair}(\mathscr{S}_A, d)$
    $(u_B, v_B), d_B \leftarrow \mathit{findClosestPair}(\mathscr{S}_B, d)$
    if $d_A \le d_B$ **then**
        $\mathit{remove}(u_A, v_A, \mathscr{S}_A)$
        $\mathit{insert}(u_A \cup v_A, \mathscr{S}_A)$
    else
        $\mathit{remove}(u_B, v_B, \mathscr{S}_B)$
        $\mathit{insert}(u_B \cup v_B, \mathscr{S}_B)$
    end if
    for $(A_i, B_j) \in \mathscr{S}_A \times \mathscr{S}_B$ **do**
        $\mathit{DPSynthesis}(A_i, B_j)$
    end for
end while
/* Revert the last merging step, and check the bound $\mathscr{C}_2$. */



16 DPcodes

$\mathscr{C}_1$ is satisfied

Closest pair algorithm

Input:

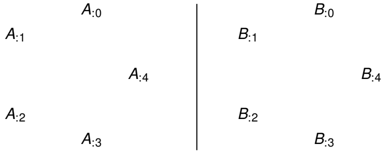
Two matrices $A \in \mathbb{F}\mathrm{ix}^{n\times n}$ and $B \in \mathbb{F}\mathrm{ix}^{n\times n}$
An accuracy bound $\mathscr{C}_1$ (ex. the average error bound is $< \epsilon$)
A code size bound $\mathscr{C}_2$
A metric d

Output:

Code to compute $A \cdot B$ s.t. $\mathscr{C}_1$ and $\mathscr{C}_2$ are satisfied,
or no code otherwise

Algorithm:

$\mathscr{S}_A \leftarrow \{A_{0,:}, \ldots, A_{n-1,:}\}$
$\mathscr{S}_B \leftarrow \{B_{:,0}, \ldots, B_{:,n-1}\}$
while $\mathscr{C}_1$ is satisfied **do**
    $(u_A, v_A), d_A \leftarrow \mathit{findClosestPair}(\mathscr{S}_A, d)$
    $(u_B, v_B), d_B \leftarrow \mathit{findClosestPair}(\mathscr{S}_B, d)$
    if $d_A \le d_B$ **then**
        $\mathit{remove}(u_A, v_A, \mathscr{S}_A)$
        $\mathit{insert}(u_A \cup v_A, \mathscr{S}_A)$
    else
        $\mathit{remove}(u_B, v_B, \mathscr{S}_B)$
        $\mathit{insert}(u_B \cup v_B, \mathscr{S}_B)$
    end if
    for $(A_i, B_j) \in \mathscr{S}_A \times \mathscr{S}_B$ **do**
        $\mathit{DPSynthesis}(A_i, B_j)$
    end for
end while
/* Revert the last merging step, and check the bound $\mathscr{C}_2$. */



12 DPcodes

$\mathscr{C}_1$ is satisfied

Closest pair algorithm

Input:

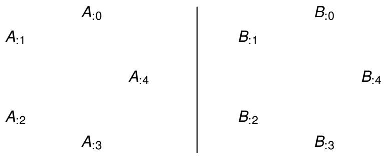
Two matrices $A \in \mathbb{F}\mathrm{ix}^{n\times n}$ and $B \in \mathbb{F}\mathrm{ix}^{n\times n}$
An accuracy bound $\mathscr{C}_1$ (ex. the average error bound is $< \epsilon$)
A code size bound $\mathscr{C}_2$
A metric d

Output:

Code to compute $A \cdot B$ s.t. $\mathscr{C}_1$ and $\mathscr{C}_2$ are satisfied,
or no code otherwise

Algorithm:

$\mathscr{S}_A \leftarrow \{A_{0,:}, \dots, A_{n-1,:}\}$
$\mathscr{S}_B \leftarrow \{B_{:,0}, \dots, B_{:,n-1}\}$
while $\mathscr{C}_1$ is satisfied **do**
  $(u_A, v_A), d_A \leftarrow \mathit{findClosestPair}(\mathscr{S}_A, d)$
  $(u_B, v_B), d_B \leftarrow \mathit{findClosestPair}(\mathscr{S}_B, d)$
  if $d_A \le d_B$ **then**
    $\mathit{remove}(u_A, v_A, \mathscr{S}_A)$
    $\mathit{insert}(u_A \cup v_A, \mathscr{S}_A)$
  else
    $\mathit{remove}(u_B, v_B, \mathscr{S}_B)$
    $\mathit{insert}(u_B \cup v_B, \mathscr{S}_B)$
  end if
  for $(A_i, B_j) \in \mathscr{S}_A \times \mathscr{S}_B$ **do**
    $\mathit{DPSynthesis}(A_i, B_j)$
  end for
end while
/* Revert the last merging step, and check the bound $\mathscr{C}_2$. */



9 DPcodes

$\mathscr{C}_1$ is no longer satisfied

Closest pair algorithm

Input:

Two matrices $A \in \mathbb{F}\mathrm{ix}^{n\times n}$ and $B \in \mathbb{F}\mathrm{ix}^{n\times n}$
An accuracy bound $\mathscr{C}_1$ (ex. the average error bound is $< \epsilon$)
A code size bound $\mathscr{C}_2$
A metric d

Output:

Code to compute $A \cdot B$ s.t. $\mathscr{C}_1$ and $\mathscr{C}_2$ are satisfied,
or no code otherwise

Algorithm:

$\mathscr{S}_A \leftarrow \{A_{0,:}, \dots, A_{n-1,:}\}$
$\mathscr{S}_B \leftarrow \{B_{:,0}, \dots, B_{:,n-1}\}$
while $\mathscr{C}_1$ is satisfied **do**
  $(u_A, v_A), d_A \leftarrow$ *findClosestPair*$(\mathscr{S}_A, d)$
  $(u_B, v_B), d_B \leftarrow$ *findClosestPair*$(\mathscr{S}_B, d)$
  if $d_A \leq d_B$ **then**
    remove$(u_A, v_A, \mathscr{S}_A)$
    insert$(u_A \cup v_A, \mathscr{S}_A)$
  else
    remove$(u_B, v_B, \mathscr{S}_B)$
    insert$(u_B \cup v_B, \mathscr{S}_B)$
  end if
  for $(A_i, B_j) \in \mathscr{S}_A \times \mathscr{S}_B$ **do**
    DPSynthesis(A_i, B_j)
  end for
end while
/* Revert the last merging step, and check the bound $\mathscr{C}_2$. */

$A_{:0}$ $A_{:1}$ $A_{:4}$ $A_{:2}$ $A_{:3}$ | $B_{:0}$ $B_{:1}$ $B_{:4}$ $B_{:2}$ $B_{:3}$

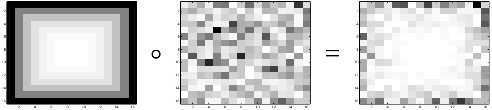
12 `DPcodes`

$\mathscr{C}_1$ is satisfied

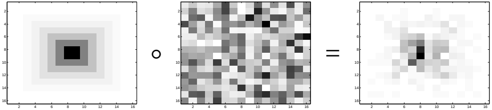
$\rightsquigarrow$ Revert the last merging step and check if $\mathscr{C}_2$ is satisfied

Benchmark matrices

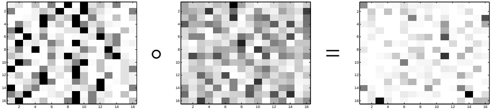
Edges



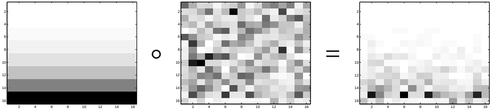
Center



Random

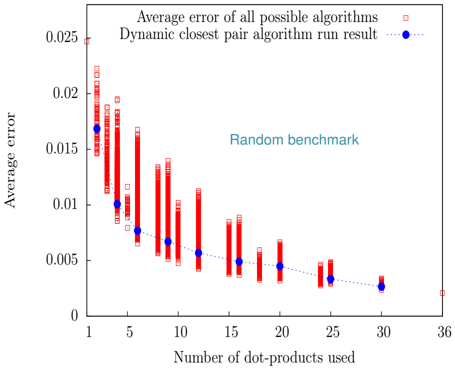


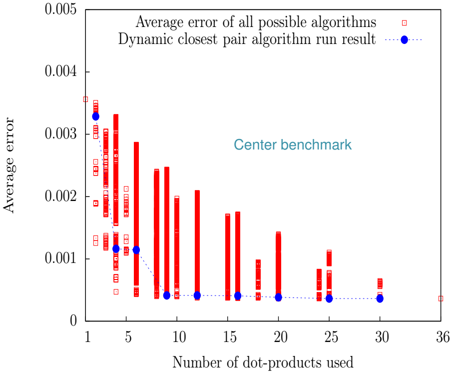
Rows



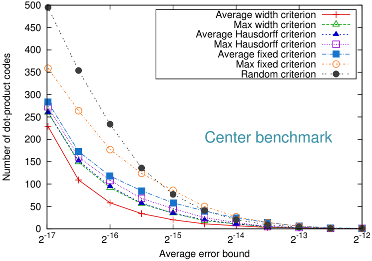
Efficiency of the distance-based heuristic: 6×6 matrix product

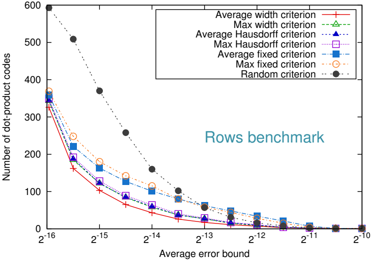
- Generating the 41 209 different algorithms: 2h15 per benchmark
- Running our algorithm: 10 seconds per benchmark

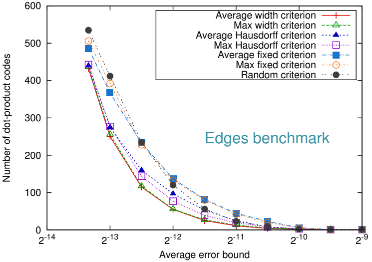


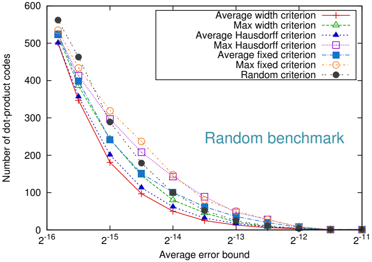


Impact of the metric on the trade-off strategy

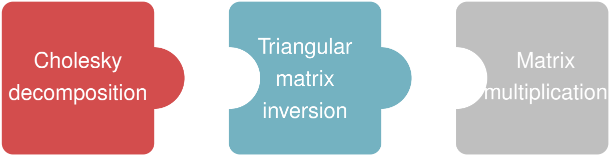








Linear algebra basic blocks



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

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

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

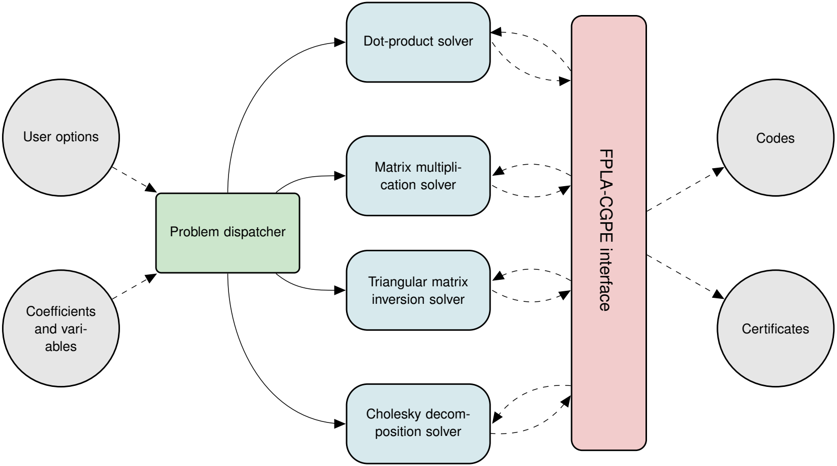
Triangular matrix inversion

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

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

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

FPLA (Fixed-Point Linear Algebra)

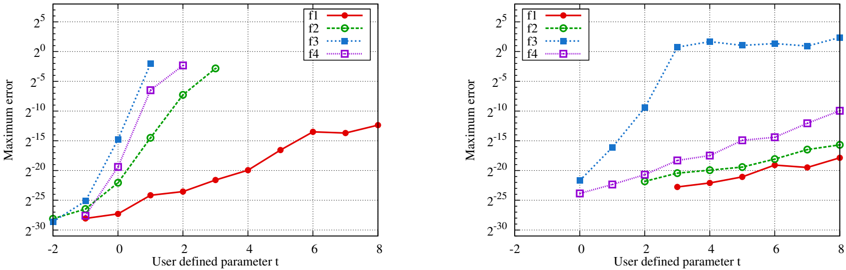


Impact of the output format of division

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 $t \in [-2, 8]$

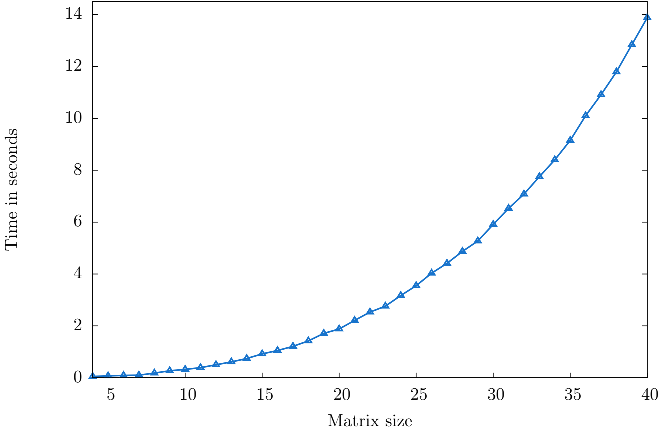


(a) Cholesky 5×5.   (b) Triangular 10×10.

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

How fast is generating triangular matrix inversion codes?

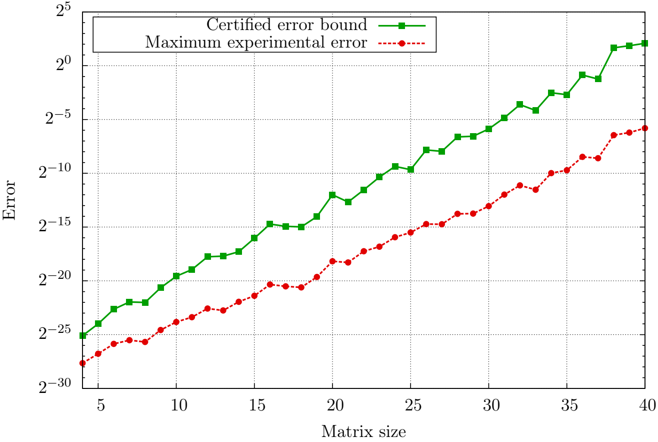
- We use $f_4(i_1, i_2) = \lfloor (i_1 + i_2)/2 \rfloor + 1$ to set the output format of division



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

How fast is generating triangular matrix inversion codes?

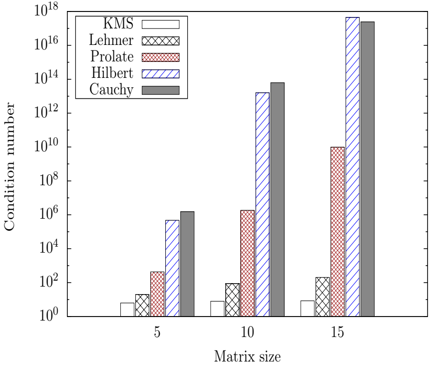
- We use $f_4(i_1, i_2) = \lfloor (i_1 + i_2)/2 \rfloor + 1$ to set the output format of division



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

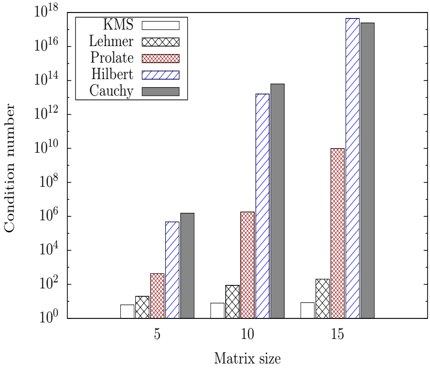
Decomposing some well known matrices

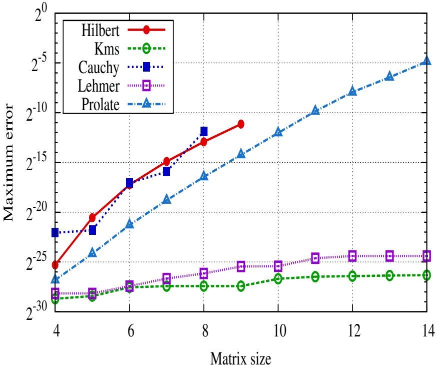
- 2 ill-conditioned matrices: Hilbert and Cauchy
- 2 well-conditioned matrices: KMS and Lehmer



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

Outline of the talk

Contributions

A framework for certified fixed-point code synthesis

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

Contributions

A framework for certified fixed-point code synthesis

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

Contributions

A framework for certified fixed-point code synthesis

- 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
    - → matrix multiplication: accuracy vs. code size trade-offs,
    - → Cholesky decomposition and triangular matrix inversion: study of divisions' impact

Contributions

A framework for certified fixed-point code synthesis

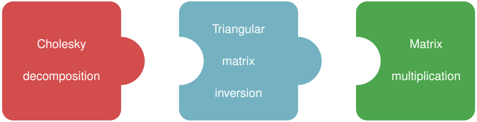
- Formalization and implementation of an arithmetic model
  - allows certification
  - handles $\sqrt{}$ and $/$
- Adaptation of the CGPE tool to the model:
  - generates code for fine grained expressions
  - instruction selection
- Development of FPLA:
  - automated and certified code synthesis for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

Publications

- DASIP14: *Toward the synthesis of fixed-point code for matrix inversion based on Cholesky decomposition* [?]
- SYNASC14: *Automated Synthesis of Target-Dependent Programs for Polynomial Evaluation in Fixed-Point Arithmetic* [?]
- PECCS14: *Code Size and Accuracy-Aware Synthesis of Fixed-Point Programs for Matrix Multiplication* [?]
- DASIP12: *Design of Fixed-point Embedded Systems (DEFIS) French ANR Project.* [?]
- SCAN12: *Approach based on instruction selection for fast and certified code generation* [?]

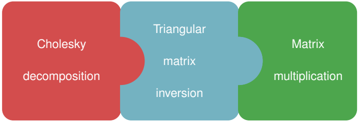
Perspectives

- Integrate the matrix inversion flow



Perspectives

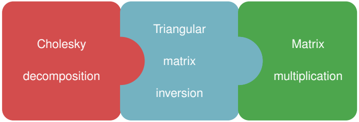
- Integrate the matrix inversion flow



  - Extend the code size vs. accuracy trade-off strategy to Cholesky decomposition and triangular matrix inversion

Perspectives

- Integrate the matrix inversion flow

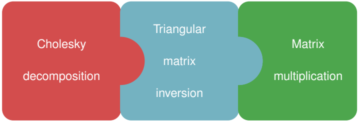


  - Extend the code size vs. accuracy trade-off strategy to Cholesky decomposition and triangular matrix inversion

- Extend the arithmetic model to support complex arithmetic
  - Objective: inverting co-variance matrices for Space Time Adaptive Processing

Perspectives

- Integrate the matrix inversion flow



  - Extend the code size vs. accuracy trade-off strategy to Cholesky decomposition and triangular matrix inversion

- Extend the arithmetic model to support complex arithmetic
  - Objective: inverting co-variance matrices for Space Time Adaptive Processing

- Target hardware implementations
  - Suggest an arithmetic model for fully custom word-length variables
  - Code size vs. accuracy → chip area vs. accuracy

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.