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
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

- 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**

- 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?

<table>
<thead>
<tr>
<th>Floating-point computations</th>
<th>Fixed-point computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy and fast to implement</td>
<td>Requires dedicated hardware</td>
</tr>
<tr>
<td>Easily portable</td>
<td>Slow if emulated in software</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point computations</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-point computations</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-point computations</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point computations</td>
<td>Fixed-point computations</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fixed-point computations are more tedious and time-consuming to implement. Over 50% of design time is required. They rely only on integer instructions and are efficient. They are well-suited for embedded systems such as µ-controllers, DSPs, and FPGAs, which have efficient integer instructions. However, how to make it easy, fast, and numerically safe to use by non-expert programmers is a challenge.
### How to implement computational tasks?

<table>
<thead>
<tr>
<th>Floating-point computations</th>
<th>Fixed-point computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>🧡 Easy and fast to implement</td>
<td>😞 Tedious and time consuming to implement</td>
</tr>
<tr>
<td>😊 Easily portable [?]</td>
<td>• &gt; 50% of design time [?]</td>
</tr>
</tbody>
</table>
How to implement computational tasks?

<table>
<thead>
<tr>
<th>Floating-point computations</th>
<th>Fixed-point computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>☺ Easy and fast to implement</td>
<td>☼ Tedious and time consuming to implement</td>
</tr>
<tr>
<td>☻ Easily portable [?]</td>
<td>☺ &gt; 50% of design time [?]</td>
</tr>
<tr>
<td>☹ Requires dedicated hardware</td>
<td>☻ Relies only on integer instructions</td>
</tr>
<tr>
<td>☻ Slow if emulated in software</td>
<td>☻ Efficient</td>
</tr>
</tbody>
</table>

© 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?

<table>
<thead>
<tr>
<th>Floating-point computations</th>
<th>Fixed-point computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>😊 Easy and fast to implement</td>
<td>😊 Tedious and time consuming to implement</td>
</tr>
<tr>
<td>😊 Easily portable [?]</td>
<td>• &gt; 50% of design time [?]</td>
</tr>
<tr>
<td>😞 Requires dedicated hardware</td>
<td>😊 Relies only on integer instructions</td>
</tr>
<tr>
<td>😞 Slow if emulated in software</td>
<td>😊 Efficient</td>
</tr>
</tbody>
</table>

#### Embedded systems targets

- µ-controllers
- DSPs
- FPGAs

→ 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

→ 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

---

M. A. Najahi (DALI UPVD/LIRMM, UM2, CNRS) Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks 5/1
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 $/$

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
Our road-map

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

1. Specify an arithmetic model
   ▶ Contributions:
     • formalization of √ 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 √ 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

M. A. Najahi (DALI UPVD/LIRMM, UM2, CNRS)

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

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

Notation

A fixed-point number with $i$ bits of integer part and $f$ bits of fraction part is in the $Q_{i,f}$ format.

Example:

$x$ in $Q_{3,5}$ and $X = (1001 1000)_2 = (152)_{10} \rightarrow x = (100.11000)_2 = (4.75)_{10}$
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 \( x \)

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

Notation

A fixed-point number with \( i \) bits of integer part and \( f \) bits of fraction part is in the \( Q_{i.f} \) 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 $x$

$\Leftrightarrow$ 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 $Q_{i,f}$ format.

#### Example:

- $x$ in $Q_{3,5}$ and $X = (10011000)_2 = (152)_{10} \rightarrow x = (100.11000)_2 = (4.75)_{10}$
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 $x$

$\Rightarrow$ 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 $Q_{i,f}$ format

Example:

- $x$ in $Q_{3,5}$ and $X = (10011000)_2 = (152)_{10} \quad \Rightarrow \quad x = (100.11000)_2 = (4.75)_{10}$

How to compute with fixed-point numbers?
How to compute with fixed-point numbers?

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

\[ P(x) = a_0 + (x \cdot a_1) \]

<table>
<thead>
<tr>
<th></th>
<th>Format</th>
<th>Int. repr.</th>
<th>Decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_0)</td>
<td>(Q_{2.6})</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>(a_1)</td>
<td>(Q_{1.7})</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td>(x)</td>
<td>(Q_{3.5})</td>
<td>[16, 208]</td>
<td></td>
</tr>
</tbody>
</table>

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

Floating-point version

```c
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

\[ P(x) = a_0 + (x \cdot a_1) \]

<table>
<thead>
<tr>
<th>Format</th>
<th>Int. repr.</th>
<th>Decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_0 \quad Q_{2.6})</td>
<td>224</td>
<td>3.5</td>
</tr>
<tr>
<td>(a_1 \quad Q_{1.7})</td>
<td>192</td>
<td>1.5</td>
</tr>
<tr>
<td>(x \quad Q_{3.5})</td>
<td>[16, 208]</td>
<td>[0.5, 6.5]</td>
</tr>
</tbody>
</table>
How to compute with fixed-point numbers?

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

\[ P(x) = a_0 + (x \cdot a_1) \]

1. Pick an evaluation scheme
How to compute with fixed-point numbers?

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

\[ P(x) = a_0 + (x \cdot a_1) \]

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

<table>
<thead>
<tr>
<th>Format</th>
<th>Int. repr.</th>
<th>Decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_0)</td>
<td>(Q_{2.6})</td>
<td>224</td>
</tr>
<tr>
<td>(a_1)</td>
<td>(Q_{1.7})</td>
<td>192</td>
</tr>
<tr>
<td>(x)</td>
<td>(Q_{3.5})</td>
<td>[16,208]</td>
</tr>
<tr>
<td>(u = (x \cdot a_1))</td>
<td>(Q_{4.4})</td>
<td>[12,156]</td>
</tr>
</tbody>
</table>
How to compute with fixed-point numbers?

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

\[ P(x) = a_0 + (x \cdot a_1) \]

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

**Format Int. repr. Decimal value**

<table>
<thead>
<tr>
<th></th>
<th>Format</th>
<th>Int. repr.</th>
<th>Decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_0 )</td>
<td>( Q_{2.6} )</td>
<td>224</td>
<td>3.5</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( Q_{1.7} )</td>
<td>192</td>
<td>1.5</td>
</tr>
<tr>
<td>( x )</td>
<td>( Q_{3.5} )</td>
<td>[16,208]</td>
<td>[0.5, 6.5]</td>
</tr>
<tr>
<td>( u = (x \cdot a_1) )</td>
<td>( Q_{4.4} )</td>
<td>[12,156]</td>
<td>[0.75, 9.75]</td>
</tr>
<tr>
<td>( v = (a_0 \gg 2) )</td>
<td>( Q_{4.4} )</td>
<td>56</td>
<td>3.5</td>
</tr>
</tbody>
</table>

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

\[ P(x) = a_0 + (x \cdot a_1) \]

<table>
<thead>
<tr>
<th>Format</th>
<th>Int. repr.</th>
<th>Decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_0 )</td>
<td>( Q_{2.6} )</td>
<td>224</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( Q_{1.7} )</td>
<td>192</td>
</tr>
<tr>
<td>( x )</td>
<td>( Q_{3.5} )</td>
<td>[16, 208]</td>
</tr>
<tr>
<td>( u = (x \cdot a_1) )</td>
<td>( Q_{4.4} )</td>
<td>[12, 156]</td>
</tr>
<tr>
<td>( v = (a_0 \gg 2) )</td>
<td>( Q_{4.4} )</td>
<td>56</td>
</tr>
<tr>
<td>( w = (v + u) )</td>
<td>( Q_{4.4} )</td>
<td>[68, 212]</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Format</th>
<th>Int. repr.</th>
<th>Decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>$Q_{2.6}$</td>
<td>224</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$Q_{1.7}$</td>
<td>192</td>
</tr>
<tr>
<td>$x$</td>
<td>$Q_{3.5}$</td>
<td>[16,208]</td>
</tr>
<tr>
<td>$u = (x \cdot a_1)$</td>
<td>$Q_{4.4}$</td>
<td>[12,156]</td>
</tr>
<tr>
<td>$v = (a_0 \gg 2)$</td>
<td>$Q_{4.4}$</td>
<td>56</td>
</tr>
<tr>
<td>$w = (v + u)$</td>
<td>$Q_{4.4}$</td>
<td>[68,212]</td>
</tr>
</tbody>
</table>

```c
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

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

\[
\begin{align*}
\text{Format} & \quad \text{Int. repr.} & \quad \text{Decimal value} \\
\text{a}_0 & \quad Q_{2.6} & \quad 224 & \quad 3.5 \\
\text{a}_1 & \quad Q_{1.7} & \quad 192 & \quad 1.5 \\
\text{x} & \quad Q_{3.5} & \quad [16,208] & \quad [0.5, 6.5] \\
\text{u} = (x \cdot \text{a}_1) & \quad Q_{4.4} & \quad [12,156] & \quad [0.75, 9.75] \\
\text{v} = (\text{a}_0 \gg 2) & \quad Q_{4.4} & \quad 56 & \quad 3.5 \\
\text{w} = (\text{v} + \text{u}) & \quad Q_{4.4} & \quad [68,212] & \quad [4.25, 13.25] \\
\end{align*}
\]

```
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*/
}
```

```
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

\[ P(x) = a_0 + (x \cdot a_1) \]

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

```c
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

```c
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 \( \text{Val}(v) \) and \( \text{Err}(v) \)
- 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 arithmetic model for fixed-point code synthesis

An interval arithmetic based model

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

\( \text{Val}(v) \) is the range of \( v \)

- the format \( Q_{i,f} \) of \( v \) is deduced from \( \text{Val}(v) = [v, \bar{v}] \)

\[ i = \left\lceil \log_2 \left( \max(|v|, |\bar{v}|) \right) \right\rceil + \alpha \]

\[ f = k - i \]

\[ \alpha = \begin{cases} 
1, & \text{if } \text{mod} \left( \log_2(\bar{v}), 1 \right) \neq 0, \\
2, & \text{otherwise} 
\end{cases} \]

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

- a bound \( \epsilon \) on rounding errors is deduced from \( \text{Err}(v) = [e, \bar{e}] \)

\[ \epsilon = \max \left( |e|, |\bar{e}| \right) \]
An arithmetic model for fixed-point code synthesis

Range and error analysis by propagating intervals

An interval arithmetic based model

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

**Val\( (v) \)** is the range of \( v \)

- the format \( Q_{i,f} \) of \( v \) is deduced from
  \[
  \text{Val}(v) = [v, \overline{v}]
  \]
- \[
  i = \left\lceil \log_2 \left( \max (|v|, |\overline{v}|) \right) \right\rceil + \alpha
  \]
- \[
  f = k - i
  \]
- \[
  \alpha = \begin{cases} 
  1, & \text{if } \text{mod} \left( \log_2(\overline{v}), 1 \right) \neq 0, \\
  2, & \text{otherwise}
  \end{cases}
  \]

**Err\( (v) \)** encloses the rounding error of computing \( v \)

- a bound \( \varepsilon \) on rounding errors is deduced from
  \[
  \text{Err}(v) = [e, \overline{e}]
  \]
- \[
  \varepsilon = \max (|e|, |\overline{e}|)
  \]
An arithmetic model for fixed-point code synthesis

An interval arithmetic based model

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

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

- the format $Q_{i,f}$ of $v$ is deduced from
  \[
  \text{Val}(v) = [v, \bar{v}]
  \]

- $i = \left\lceil \log_2 (\max(|v|, |\bar{v}|)) \right\rceil + \alpha$
- $f = k - i$

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

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

- a bound $\epsilon$ on rounding errors is deduced from
  \[
  \text{Err}(v) = [e, \bar{e}]
  \]

- $\epsilon = \max(|e|, |\bar{e}|)$

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

M. A. Najahi (DALI UPVD/LIRMM, UM2, CNRS)
Fixed-point addition

- The two variables \( v_1 \) and \( v_2 \) must be aligned: in the same fixed-point format \( Q_{i,f} \).

1. If \( \text{Val}(v_1) + \text{Val}(v_2) \subseteq \text{Range}(Q_{i,f}) \), overflow cannot occur.

\[
\begin{array}{c}
\text{Val}(v_1) \\
+ \\
\text{Val}(v_2) \\
\hline
\text{Val}(v_1 + v_2)
\end{array}
\]
Fixed-point addition

1. The two variables \( v_1 \) and \( v_2 \) must be aligned: in the same fixed-point format \( Q_{i,f} \).

2. If \( \text{Val}(v_1) + \text{Val}(v_2) \subseteq \text{Range}(Q_{i,f}) \), overflow cannot occur.

\[
\begin{align*}
\text{Val}(v) &= \text{Val}(v_1) + \text{Val}(v_2) \\
\text{Err}(v) &= \text{Err}(v_1) + \text{Err}(v_2)
\end{align*}
\]
Fixed-point addition

- The two variables $v_1$ and $v_2$ must be aligned: in the same fixed-point format $Q_{i,f}$

1. If $\text{Val}(v_1) + \text{Val}(v_2) \subseteq \text{Range}(Q_{i,f})$, overflow cannot occur

2. If $\text{Range}(Q_{i,f}) \subset \text{Val}(v_1) + \text{Val}(v_2) \subset \text{Range}(Q_{i+1.f-1})$, overflow may occur
Fixed-point addition

- The two variables \( v_1 \) and \( v_2 \) must be aligned: in the same fixed-point format \( Q_{i,f} \).

1. If \( \text{Val}(v_1) + \text{Val}(v_2) \subseteq \text{Range}(Q_{i,f}) \), overflow cannot occur.

\[
\begin{array}{c}
\text{Val}(v_1) \\
\text{Val}(v_2)
\end{array}
\]

\[
\begin{array}{c}
\text{Err}(v_1) \\
\text{Err}(v_2)
\end{array}
\]

\[
\begin{array}{c}
\text{Val}(v_1) \\
\text{Val}(v_2)
\end{array}
\]

\[
\begin{array}{c}
\text{Err}(v_1) \\
\text{Err}(v_2)
\end{array}
\]

\[
\begin{array}{c}
\text{Val}(v_1) \\
\text{Val}(v_2)
\end{array}
\]

\[
\begin{array}{c}
\text{Err}(v_1) \\
\text{Err}(v_2)
\end{array}
\]

\[
\begin{array}{c}
\text{Val}(v_1) \\
\text{Val}(v_2)
\end{array}
\]

\[
\begin{array}{c}
\text{Err}(v_1) \\
\text{Err}(v_2)
\end{array}
\]

2. If \( \text{Range}(Q_{i,f}) \subset \text{Val}(v_1) + \text{Val}(v_2) \subset \text{Range}(Q_{i+1,f-1}) \), overflow may occur.
Fixed-point addition

- The two variables $v_1$ and $v_2$ must be aligned: in the same fixed-point format $Q_{i,f}$

1. If $\text{Val}(v_1) + \text{Val}(v_2) \subseteq \text{Range}(Q_{i,f})$, overflow cannot occur

\[
\begin{array}{c}
\text{Val}(v) = \text{Val}(v_1) + \text{Val}(v_2) \\
\text{Err}(v) = \text{Err}(v_1) + \text{Err}(v_2)
\end{array}
\]

2. If $\text{Range}(Q_{i,f}) \subset \text{Val}(v_1) + \text{Val}(v_2) \subset \text{Range}(Q_{i+1,f-1})$, overflow may occur

\[
\begin{array}{c}
\text{Val}(v) = \text{Val}(v'_1) + \text{Val}(v'_2) \\
\text{Err}(v) = \text{Err}(v'_1) + \text{Err}(v'_2)
\end{array}
\]
Fixed-point multiplication

- The output format of a $Q_{i_1.f_1} \times Q_{i_2.f_2}$ is $Q_{i_1 + i_2.f_1 + f_2}$.
Fixed-point multiplication

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

```
int32_t mul ( int32_t v1 , int32_t v2)
{
  int64_t prod = (( int64_t ) v1) * (( int64_t ) v2);
  return ( int32_t ) (prod >> 32);
}
```
### Fixed-point multiplication

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

\[
\text{Err}_x = \left[ 0, 2^{-f_r} - 2^{-(f_1+f_2)} \right]
\]
Fixed-point multiplication

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

![Diagram of fixed-point multiplication]

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

```c
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 $Q_{i_1.f_1} / Q_{i_2.f_2}$ may be as large as $i_1 + f_2$
Our new fixed-point division

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

\[
\text{Err}/ = [-2^{i_2+f_1}, 2^{i_2+f_1}]
\]
Our new fixed-point division

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

\[
\text{Err} / = \left[ -2^{f_r}, 2^{f_r} \right]
\]
Our new fixed-point division

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

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

-Sharper bound
- Risk of overflow at run-time
Our new fixed-point division

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

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

- sharper bound
- risk of overflow at run-time

How to decide on 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 $Q_{ir.fr}$

\[
\text{Val}(v) = \text{Range}(Q_{ir.fr}) = [-2^{ir-1}, 2^{ir-1} - 2^{fr}].
\]

\[
\text{Err}(v) = \frac{\text{Val}(v_2) \cdot \text{Err}(v_1) - \text{Val}(v_1) \cdot \text{Err}(v_2)}{\text{Val}(v_2) \cdot (\text{Val}(v_2) + \text{Err}(v_2))} + \text{Err}/\text{Val}(v_2).
\]

- \[
\text{Val}(v_2) = \frac{\text{Val}(v_1)}{\text{Val}(v) + \text{Err}/\text{Val}(v)} \cap \text{Val}(v_2) \text{ and } \text{Val}(v) = [-2^{ir-1}, -2^{-fr}] \cup [2^{-fr}, 2^{ir-1} - 2^{fr}].
\]
The propagation rule and implementation of division

- Once the output format decided $Q_{ir,fr}$

\[
\text{Val}(v) = \mathbb{R}ange(Q_{ir,fr}) = [-2^{ir-1}, 2^{ir-1} - 2^{fr}].
\]

\[
\text{Err}(v) = \frac{\text{Val}(v_2) \cdot \text{Err}(v_1) - \text{Val}(v_1) \cdot \text{Err}(v_2)}{\text{Val}(v_2) \cdot (\text{Val}(v_2) + \text{Err}(v_2))} + \text{Err} / \text{Val}(v_2).
\]

- $\overline{\text{Val}(v_2)} = \frac{\text{Val}(v_1)}{\text{Val}(v) + \text{Err} / \text{Val}(v_2)} \cap \text{Val}(v_2)$ and $\overline{\text{Val}(v)} = [-2^{ir-1}, -2^{-fr}] \cup [2^{-fr}, 2^{ir-1} - 2^{fr}]$

```c
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 $Q_{ir,fr}$

\[ \text{Val}(v) = \text{Range}(Q_{ir,fr}) = [-2^{ir-1}, 2^{ir-1} - 2^{fr}] \]

\[ \text{Err}(v) = \frac{\text{Val}(v_2) \cdot \text{Err}(v_1) - \text{Val}(v_1) \cdot \text{Err}(v_2)}{\text{Val}(v_2) \cdot (\text{Val}(v_2) + \text{Err}(v_2))} + \text{Err} / \]

\[ \overline{\text{Val}(v_2)} = \frac{\text{Val}(v_1)}{\text{Val}(v) + \text{Err}} \cap \text{Val}(v_2) \quad \text{and} \quad \overline{\text{Val}(v)} = [-2^{ir-1}, -2^{-fr}] \cup [2^{-fr}, 2^{ir-1} - 2^{fr}] \]

```c
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) == 0x0000000080000000ll)
        || ((V & 0x0000000080000000ll) == 0));
    return (int32_t) V;
}
```

- Additional code to check for run-time overflows
The division format trade-off: case of inverting $2 \times 2$ matrices

Consider $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $a, b, c, d \in [-1, 1]$ in the format $\mathbb{Q}_{2.30}$.
The division format trade-off: case of inverting $2 \times 2$ matrices

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

- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} d/\Delta & -b/\Delta \\ -c/\Delta & a/\Delta \end{pmatrix}$.
The division format trade-off: case of inverting $2 \times 2$ matrices

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

- 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 $2 \times 2$ matrices

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

- 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 $2 \times 2$ matrices

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

- Cramer’s rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} d/\Delta & -b/\Delta \\ -c/\Delta & a/\Delta \end{pmatrix}$.
The division format trade-off: case of inverting $2 \times 2$ matrices

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

- 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 $2 \times 2$ matrices

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

Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} d/\Delta & -b/\Delta \\ -c/\Delta & a/\Delta \end{pmatrix}$
The division format trade-off: case of inverting $2 \times 2$ matrices

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

- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} / \Delta$.
The division format trade-off: case of inverting $2 \times 2$ matrices

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

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

![Diagram of division format trade-off]

- Division output format
  - Maximum experimental error
  - Overflow rate

M. A. Najahi (DALI UPVD/LIRMM, UM2, CNRS)
The division format trade-off: case of inverting $2 \times 2$ matrices

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

- Cramer’s rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} d/\Delta & -b/\Delta \\ -c/\Delta & a/\Delta \end{pmatrix}$
The division format trade-off: case of inverting $2 \times 2$ matrices

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

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

![Diagram of division output format with maximum experimental error and overflow rate graphs.](image)
The division format trade-off: case of inverting $2 \times 2$ matrices

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

- Cramer’s rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} d/\Delta & -b/\Delta \\ -c/\Delta & a/\Delta \end{pmatrix}$
The division format trade-off: case of inverting $2 \times 2$ matrices

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

- Cramer's rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} d/\Delta & -b/\Delta \\ -c/\Delta & a/\Delta \end{pmatrix}$
The division format trade-off: case of inverting $2 \times 2$ matrices

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

- Cramer’s rule: if $\Delta = ad - bc \neq 0$ then $A^{-1} = \begin{pmatrix} d/\Delta & -b/\Delta \\ -c/\Delta & a/\Delta \end{pmatrix}$
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 $\rightarrow$ front-end
   - computes evaluation schemes $\rightarrow$ DAGs
The CGPE tool

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

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

2. **Filtering step** $\leadsto$ middle-end
   - applies the arithmetic model
   - prunes the DAGs that do not satisfy different criteria:
     - latency $\leadsto$ scheduling filter
     - accuracy $\leadsto$ numerical filter
     - ...

3. **Generation step** $\leadsto$ back-end
   - generates C codes and Gappa accuracy certificates
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 $0.3 \cdot \pi$:

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

$\begin{dotproduct}
  \text{coefficient name=}&b0\text{ value=}&0x65718e3b\text{ integer_width=-3 fraction_width=}&35\text{ width=}&32/
  \ldots
  \text{variable name=}&y3\text{ inf=}&0xb1e91685\text{ sup=}&0x4e16e97b\text{ integer_width=}&6 fraction_width=}&26\text{ width=}&32/
\end{dotproduct}$
Code synthesis for an IIR filter using CGPE

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

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

Original signal

Filtered in fixed-point using $S_1$

Filtered in binary64

Amplitude

Time
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 $0.3 \cdot \pi$:

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

```c
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*/ )
{
    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 = r9 + 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 = r13 + 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 $0.3 \cdot \pi$:

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

```c
int32_t filter ( int32_t u0, int32_t u1, int32_t u2, int32_t u3,
                 int32_t y1, int32_t y2, int32_t y3 )
{
    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; //Q2.30 [-2^{-27},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^{-22.5324},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 → 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 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 $a_0 + \left( (a_1 \cdot x) + ((a_2 \cdot (x \cdot x)) \ll 1) \right)$?

- addition / shift $\leadsto$ 1 cycle
- shift-and-add $\leadsto$ 1 cycle
- multiplication $\leadsto$ 3 cycles
The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]  
1: BottomUpDP() + TopDownSelect()  
2: ImproveCSEDecision()  
3: BottomUpDP() + TopDownSelect()  

Example: how to evaluate $a_0 + \left( (a_1 \cdot x) + ((a_2 \cdot (x \cdot x)) \ll 1) \right)$?

- Addition / shift $\sim 1$ cycle
- Shift-and-add $\sim 1$ cycle
- Multiplication $\sim 3$ cycles

Diagram:

```
  +   +   ≤   1
  |   |   |   |
  |   |   |   |
  +   x   x   3
  |   |   |
  |   |   |
  x   x   1
  a_0
  a_1
  a_2
```

BottomUpDP()
The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

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

Example: how to evaluate $a_0 + \left( (a_1 \cdot x) + \left( (a_2 \cdot (x \cdot x)) \ll 1 \right) \right)$?

- Addition / shift $\sim 1$ cycle
- Shift-and-add $\sim 1$ cycle
- Multiplication $\sim 3$ cycles
The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

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

Example: how to evaluate \( a_0 + \left( (a_1 \cdot x) + ((a_2 \cdot (x \cdot x)) \ll 1) \right) \)?

- Addition / shift \( \leadsto \) 1 cycle
- Shift-and-add \( \leadsto \) 1 cycle
- Multiplication \( \leadsto \) 3 cycles

\[ \begin{align*}
+ & \quad + \\
\times & \quad \ll
\end{align*} \]
The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [1]

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

Example: how to evaluate $a_0 + \left( (a_1 \cdot x) + ((a_2 \cdot (x \cdot x)) \ll 1) \right)$?

- Addition / shift $\leadsto$ 1 cycle
- Shift-and-add $\leadsto$ 1 cycle
- Multiplication $\leadsto$ 3 cycles

Diagram:

```
BottomUpDP()
```

```
addition / shift $\leadsto$ 1 cycle
shift-and-add $\leadsto$ 1 cycle
multiplication $\leadsto$ 3 cycles
```

M. A. Najahi (DALI UPVD/LIRMM, UM2, CNRS)
The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

1: \text{BottomUpDP}() + \text{TopDownSelect}()
2: \text{ImproveCSEDDecision}()
3: \text{BottomUpDP}() + \text{TopDownSelect}()

Example: how to evaluate $a_0 + \left( (a_1 \cdot x) + ((a_2 \cdot (x \cdot x)) \ll 1) \right)$?

- Addition / shift $\leadsto 1$ cycle
- Shift-and-add $\leadsto 1$ cycle
- Multiplication $\leadsto 3$ cycles
The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?] 

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

Example: how to evaluate $a_0 + \left( (a_1 \cdot x) + \left( (a_2 \cdot (x \cdot x)) \ll 1 \right) \right)$?

## BottomUpDP() 

- addition / shift $\sim 1$ cycle
- shift-and-add $\sim 1$ cycle
- multiplication $\sim 3$ cycles

M. A. Najahi (DALI UPVD/LIRMM, UM2, CNRS)
The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm

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

Example: how to evaluate \( a_0 + \left( a_1 \cdot x + \left( a_2 \cdot (x \cdot x) \ll 1 \right) \right) \)?

- addition / shift \( \leadsto \) 1 cycle
- shift-and-add \( \leadsto \) 1 cycle
- multiplication \( \leadsto \) 3 cycles
The NOLTIS-like tiling algorithm

Near-Optimal Instruction Selection algorithm [?]

1. BottomUpDP() + TopDownSelect()
2. ImproveGSEDecision()
3. BottomUpDP() + TopDownSelect()

Example: how to evaluate \( a_0 + \left( (a_1 \cdot x) + ((a_2 \cdot (x \cdot x)) \ll 1) \right) \)?

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

<table>
<thead>
<tr>
<th>$f(x)$</th>
<th>$\mathcal{J}$</th>
<th>$d$</th>
<th>$\log_2(\epsilon)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>not optimized</td>
</tr>
<tr>
<td>$\exp(x) - 1$</td>
<td>$[-0.25, 0.25]$</td>
<td>7</td>
<td>$-26.98$ $-27.34$</td>
</tr>
<tr>
<td>$\exp(x)$</td>
<td>$[0, 1]$</td>
<td>7</td>
<td>$-13.94$ $-14.90$</td>
</tr>
<tr>
<td>$\sin(x)$</td>
<td>$[-0.5, 0.5]$</td>
<td>9</td>
<td>$-18.95$ $-19.91$</td>
</tr>
<tr>
<td>$\cos(x)$</td>
<td>$[-0.5, 0.25]$</td>
<td>5</td>
<td>$-27.01$ $-27.26$</td>
</tr>
<tr>
<td>$\tan(x)$</td>
<td>$[0.25, 0.5]$</td>
<td>9</td>
<td>$-18.81$ $-19.64$</td>
</tr>
<tr>
<td>$\log_2(1 + x)/x$</td>
<td>$[2^{-23}, 1]$</td>
<td>7</td>
<td>$-13.94$ $-14.89$</td>
</tr>
<tr>
<td>$\sqrt{1 + x}$</td>
<td>$[2^{-23}, 1]$</td>
<td>7</td>
<td>$-13.94$ $-14.90$</td>
</tr>
</tbody>
</table>

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 $(a \ast b) + (c \gg n)$ with $n \in \{1, \ldots, 31\}$ 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
  - 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

- Cholesky decomposition
- Triangular matrix inversion
- Matrix multiplication
Linear algebra basic blocks

- Cholesky decomposition
- Triangular matrix inversion
- Matrix multiplication
Matrix multiplication

Inputs

- Two matrices $A$ and $B$ of fixed-point variables
  \[ A \in \text{Fix}^{n \times n} \quad \text{and} \quad B \in \text{Fix}^{n \times n} \]

- A bound $C_1$ on the roundoff error

- A bound $C_2$ on the code size
Matrix multiplication

**Inputs**

- Two matrices $A$ and $B$ of fixed-point variables
  \[ A \in \text{Fix}^{n \times n} \quad \text{and} \quad B \in \text{Fix}^{n \times n} \]
- A bound $C_1$ on the roundoff error
- A bound $C_2$ 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 $C_1$ and $C_2$
- 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

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 $\mathcal{C}_1$ and $\mathcal{C}_2$
Straightforward algorithms

Accurate algorithm

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

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 $\mathcal{C}_1$ and $\mathcal{C}_2$

Compact algorithm

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

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: $\mathcal{U} = A_{1,:), \cup A_{2,:), \cdots \cup A_{n,:), \text{ with } \mathcal{U} \in \mathbb{F}ix^{1 \times n}$
2: $\mathcal{V} = B_{:,1} \cup B_{:,2} \cup \cdots \cup B_{:,n}, \text{ with } \mathcal{V} \in \mathbb{F}ix^{n \times 1}$
3: $DPSynthesis(\mathcal{U}, \mathcal{V})$
4: Check $\mathcal{C}_1$ and $\mathcal{C}_2$
Example of a multiplication of 2 × 2 matrices

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

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

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>$A_{1,1}$</th>
<th>$A_{1,2}$</th>
<th>$A_{2,1}$</th>
<th>$A_{2,2}$</th>
<th>$B_{1,1}$</th>
<th>$B_{1,2}$</th>
<th>$B_{2,1}$</th>
<th>$B_{2,2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point format</td>
<td>$Q_{11,21}$</td>
<td>$Q_{12,20}$</td>
<td>$Q_{2,30}$</td>
<td>$Q_{2,30}$</td>
<td>$Q_{11,21}$</td>
<td>$Q_{3,29}$</td>
<td>$Q_{2,30}$</td>
<td>$Q_{5,27}$</td>
</tr>
</tbody>
</table>
Example of a multiplication of $2 \times 2$ matrices

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

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

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>$A_{1,1}$</th>
<th>$A_{1,2}$</th>
<th>$A_{2,1}$</th>
<th>$A_{2,2}$</th>
<th>$B_{1,1}$</th>
<th>$B_{1,2}$</th>
<th>$B_{2,1}$</th>
<th>$B_{2,2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point format</td>
<td>$Q_{11,21}$</td>
<td>$Q_{12,20}$</td>
<td>$Q_{2,30}$</td>
<td>$Q_{2,30}$</td>
<td>$Q_{11,21}$</td>
<td>$Q_{3,29}$</td>
<td>$Q_{2,30}$</td>
<td>$Q_{5,27}$</td>
</tr>
</tbody>
</table>

**Accurate algorithm**

<table>
<thead>
<tr>
<th>Dot-product</th>
<th>$A_{1,1} \cdot B_{1,1}$</th>
<th>$A_{1,2} \cdot B_{1,2}$</th>
<th>$A_{2,1} \cdot B_{1,1}$</th>
<th>$A_{2,2} \cdot B_{1,2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluated using</td>
<td>DPCode$_{1,1}$</td>
<td>DPCode$_{1,2}$</td>
<td>DPCode$_{2,1}$</td>
<td>DPCode$_{2,2}$</td>
</tr>
<tr>
<td>Output format</td>
<td>$Q_{26,6}$</td>
<td>$Q_{18,14}$</td>
<td>$Q_{15,17}$</td>
<td>$Q_{7,25}$</td>
</tr>
<tr>
<td>Certified error</td>
<td>$\approx 2^{-5}$</td>
<td>$\approx 2^{-14}$</td>
<td>$\approx 2^{-16}$</td>
<td>$\approx 2^{-24}$</td>
</tr>
<tr>
<td>Maximum error</td>
<td>$\approx 2^{-5}$</td>
<td>$\approx 2^{-7}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average error</td>
<td>$\approx 2^{-7}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Compact algorithm**

<table>
<thead>
<tr>
<th>Dot-product</th>
<th>$A_{1,1} \cdot B_{1,1}$</th>
<th>$A_{1,2} \cdot B_{1,2}$</th>
<th>$A_{2,1} \cdot B_{1,2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluated using</td>
<td>DPCode$_{U,V}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output format</td>
<td>$Q_{26,6}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Certified error</td>
<td>$\approx 2^{-5}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum error</td>
<td>$\approx 2^{-5}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average error</td>
<td>$\approx 2^{-5}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trade-off algorithms

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

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

Accurate algorithm:
(25 dot-product codes)
### 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 = \[
\begin{pmatrix}
  b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\
  b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\
  b_{20} & b_{21} & b_{22} & b_{23} & b_{24} \\
  b_{30} & b_{31} & b_{32} & b_{33} & b_{34} \\
  b_{40} & b_{41} & b_{42} & b_{43} & b_{44}
\end{pmatrix}
\]

#### Compact algorithm:
(1 dot-product code)

- \(A_0\):
- \(A_1\):
- \(A_2\):
- \(A_3\):
- \(A_4\):

Number of possible trade-off algorithms given by \(B(n)^2\) with \(B(n)\) the \(n\)th Bell number:

\[
\begin{align*}
B_0 & = 16 \\
B_1 & = 45 \\
B_2 & = 124 \\
B_3 & = 433 \\
B_4 & = \ldots
\end{align*}
\]

How to find in reasonable time a partition that reduces cost size without harming the accuracy?
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 \( B(n) \) with \( B(n) \) the \( n \)th Bell number

\[
\begin{align*}
B(5) & \approx 2704 \\
B(6) & \approx 41209 \\
B(10) & \approx 2^{34} \\
B(25) & \approx 2^{66} \\
B(43) & \approx 2^{124} \\
\end{align*}
\]

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

Trade-off algorithm:
(9 dot-product codes)
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 \( \mathcal{B}(n)^2 \) with \( \mathcal{B}(n) \) the \( n^{th} \) Bell number

<table>
<thead>
<tr>
<th>( n )</th>
<th>5</th>
<th>6</th>
<th>10</th>
<th>16</th>
<th>25</th>
<th>64</th>
<th>( \mathcal{B}(n)^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2704</td>
<td>41209</td>
<td>( \approx 2^{34} )</td>
<td>( \approx 2^{66} )</td>
<td>( \approx 2^{124} )</td>
<td>( \approx 2^{433} )</td>
<td></td>
</tr>
</tbody>
</table>
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 \( \mathcal{B}(n)^2 \) with \( \mathcal{B}(n) \) the \( n^{th} \) Bell number

<table>
<thead>
<tr>
<th>( n )</th>
<th>5</th>
<th>6</th>
<th>10</th>
<th>16</th>
<th>25</th>
<th>64</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{B}(n)^2 )</td>
<td>2704</td>
<td>41209</td>
<td>( \approx 2^{34} )</td>
<td>( \approx 2^{66} )</td>
<td>( \approx 2^{124} )</td>
<td>( \approx 2^{433} )</td>
<td>...</td>
</tr>
</tbody>
</table>

How to find in reasonable time a partition that reduces cost size without harming the accuracy?
Distances between fixed-point variables

The Hausdorff distance

\[ d_H : \text{Fix} \times \text{Fix} \rightarrow \mathbb{R}^+ \]
\[ d_H(l_1, l_2) = \max \left\{ \left| l_1 - l_2 \right|, \left| \overline{l_1} - \overline{l_2} \right| \right\} \]

Fixed-point distance

\[ d_F : \text{Fix} \times \text{Fix} \rightarrow \mathbb{N} \]
\[ d_F(l_1, l_2) = \left| \text{IntegerPart}(l_1) - \text{IntegerPart}(l_2) \right| \]
Distances between fixed-point variables

The Hausdorff distance

\[ d_H : \text{Fix} \times \text{Fix} \rightarrow \mathbb{R}^+ \]
\[ d_H (l_1, l_2) = \max \{ |l_1 - l_2|, |\bar{l}_1 - \bar{l}_2| \} \]

Fixed-point distance

\[ d_F : \text{Fix} \times \text{Fix} \rightarrow \mathbb{N} \]
\[ d_F (l_1, l_2) = |\text{IntegerPart}(l_1) - \text{IntegerPart}(l_2)| \]
Closest pair algorithm

Input:
- Two matrices $A \in \text{Fix}^{n \times n}$ and $B \in \text{Fix}^{n \times n}$
- An accuracy bound $C_1$ (ex. the average error bound is $< \varepsilon$)
- A code size bound $C_2$
- A metric $d$

Output:
- Code to compute $A \cdot B$ s.t. $C_1$ and $C_2$ are satisfied, or no code otherwise

Algorithm:
1. $\mathcal{G}_A \leftarrow \{A_0, \ldots, A_{n-1}\}$
2. $\mathcal{G}_B \leftarrow \{B_0, \ldots, B_{n-1}\}$
3. while $C_1$ is satisfied do
4.      $(u_A, v_A), d_A \leftarrow \text{findClosestPair}(\mathcal{G}_A, d)$
5.      $(u_B, v_B), d_B \leftarrow \text{findClosestPair}(\mathcal{G}_B, d)$
6.      if $d_A \leq d_B$ then
7.         remove$(u_A, v_A, \mathcal{G}_A)$
8.         insert$(u_A \cup v_A, \mathcal{G}_A)$
9.      else
10.       remove$(u_B, v_B, \mathcal{G}_B)$
11.       insert$(u_B \cup v_B, \mathcal{G}_B)$
12.     end if
13.     for $(A_i, B_j) \in \mathcal{G}_A \times \mathcal{G}_B$ do
14.        $\text{DPSynthesis}(A_i, B_j)$
15.     end for
16. end while
17. /* Revert the last merging step, and check the bound $C_2$. */
Closest pair algorithm

Input:
Two matrices $A \in \text{Fix}^{n \times n}$ and $B \in \text{Fix}^{n \times n}$
An accuracy bound $\epsilon_1$ (ex. the average error bound is $< \epsilon$)
A code size bound $\epsilon_2$
A metric $d$

Output:
Code to compute $A \cdot B$ s.t. $\epsilon_1$ and $\epsilon_2$ are satisfied, or no code otherwise

Algorithm:
1: $\mathcal{S}_A \leftarrow \{A_0, \ldots, A_{n-1}\}$
2: $\mathcal{S}_B \leftarrow \{B_0, \ldots, B_{n-1}\}$
3: while $\epsilon_1$ is satisfied do
4: $(u_A, v_A), d_A \leftarrow \text{findClosestPair}(\mathcal{S}_A, d)$
5: $(u_B, v_B), d_B \leftarrow \text{findClosestPair}(\mathcal{S}_B, d)$
6: if $d_A \leq d_B$ then
7: remove$(u_A, v_A, \mathcal{S}_A)$
8: insert$(u_A \cup v_A, \mathcal{S}_A)$
9: else
10: remove$(u_B, v_B, \mathcal{S}_B)$
11: insert$(u_B \cup v_B, \mathcal{S}_B)$
12: end if
13: for $(A_i, B_j) \in \mathcal{S}_A \times \mathcal{S}_B$ do
14: $\text{DPSynthesis}(A_i, B_j)$
15: end for
16: end while
17: /* Revert the last merging step, and check the bound $\epsilon_2$. */
Closest pair algorithm

Input:
Two matrices $A \in \text{Fix}^{n \times n}$ and $B \in \text{Fix}^{n \times n}$
An accuracy bound $C_1$ (ex. the average error bound is $\leq \epsilon$)
A code size bound $C_2$
A metric $d$

Output:
Code to compute $A \cdot B$ s.t. $C_1$ and $C_2$ are satisfied,
or no code otherwise

Algorithm:
1: $\mathcal{S}_A \leftarrow \{A_0, \ldots, A_{n-1}\}$
2: $\mathcal{S}_B \leftarrow \{B_0, \ldots, B_{n-1}\}$
3: while $C_1$ is satisfied do
4:   $(u_A, v_A), d_A \leftarrow \text{findClosestPair}(\mathcal{S}_A, d)$
5:   $(u_B, v_B), d_B \leftarrow \text{findClosestPair}(\mathcal{S}_B, d)$
6:   if $d_A \leq d_B$ then
7:     remove($u_A, v_A, \mathcal{S}_A$)
8:     insert($u_A \cup v_A, \mathcal{S}_A$)
9:   else
10:    remove($u_B, v_B, \mathcal{S}_B$)
11:    insert($u_B \cup v_B, \mathcal{S}_B$)
12:  end if
13:  for $(A_i, B_j) \in \mathcal{S}_A \times \mathcal{S}_B$ do
14:     $\text{DPSynthesis}(A_i, B_j)$
15:  end for
16: end while
17: /* Revert the last merging step, and check if $C_2$ is satisfied */
Closest pair algorithm

Input:
- Two matrices $A \in \text{Fix}^{n \times n}$ and $B \in \text{Fix}^{n \times n}$
- An accuracy bound $C_1$ (e.g., the average error bound is $< \epsilon$)
- A code size bound $C_2$
- A metric $d$

Output:
- Code to compute $A \cdot B$ s.t. $C_1$ and $C_2$ are satisfied, or no code otherwise

Algorithm:
1: $\mathcal{S}_A \leftarrow \{A_0, \ldots, A_{n-1}\}$
2: $\mathcal{S}_B \leftarrow \{B_0, \ldots, B_{n-1}\}$
3: while $C_1$ is satisfied do
4: \((u_A, v_A), d_A \leftarrow \text{findClosestPair}(\mathcal{S}_A, d)\)
5: \((u_B, v_B), d_B \leftarrow \text{findClosestPair}(\mathcal{S}_B, d)\)
6: if $d_A \leq d_B$ then
7: remove\((u_A, v_A, \mathcal{S}_A)\)
8: insert\((u_A \cup v_A, \mathcal{S}_A)\)
9: else
10: remove\((u_B, v_B, \mathcal{S}_B)\)
11: insert\((u_B \cup v_B, \mathcal{S}_B)\)
12: end if
13: for $(A_i, B_j) \in \mathcal{S}_A \times \mathcal{S}_B$ do
14: \(\text{DPSynthesis}(A_i, B_j)\)
15: end for
16: end while
17: /* Revert the last merging step, and check the bound $C_2$. */
Closest pair algorithm

Input:
Two matrices $A \in \text{Fix}^{n \times n}$ and $B \in \text{Fix}^{n \times n}$
An accuracy bound $c_1$ (ex. the average error bound is $< \epsilon$)
A code size bound $c_2$
A metric $d$

Output:
Code to compute $A \cdot B$ s.t. $c_1$ and $c_2$ are satisfied,
or no code otherwise

Algorithm:
1: $\mathcal{S}_A \leftarrow \{A_0, \ldots, A_{n-1}\}$
2: $\mathcal{S}_B \leftarrow \{B_0, \ldots, B_{n-1}\}$
3: while $c_1$ is satisfied do
4: $(u_A, v_A), d_A \leftarrow \text{findClosestPair}(\mathcal{S}_A, d)$
5: $(u_B, v_B), d_B \leftarrow \text{findClosestPair}(\mathcal{S}_B, d)$
6: if $d_A \leq d_B$ then
7: remove$(u_A, v_A, \mathcal{S}_A)$
8: insert$(u_A \cup v_A, \mathcal{S}_A)$
9: else
10: remove$(u_B, v_B, \mathcal{S}_B)$
11: insert$(u_B \cup v_B, \mathcal{S}_B)$
12: end if
13: for $(A_i, B_j) \in \mathcal{S}_A \times \mathcal{S}_B$ do
14: $\text{DPSynthesis}(A_i, B_j)$
15: end for
16: end while
17: /* Revert the last merging step, and check the bound $c_2$. */
**Closest pair algorithm**

**Input:**
- Two matrices $A \in \text{Fix}^{n \times n}$ and $B \in \text{Fix}^{n \times n}$
- An accuracy bound $\mathcal{C}_1$ (ex. the average error bound is $< \epsilon$)
- A code size bound $\mathcal{C}_2$
- A metric $d$

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

**Algorithm:**

1. $\mathcal{S}_A \leftarrow \{A_0, \ldots, A_{n-1}\}$
2. $\mathcal{S}_B \leftarrow \{B_0, \ldots, B_{n-1}\}$
3. while $\mathcal{C}_1$ is satisfied do
4. \hspace{1em} $(u_A, v_A), d_A \leftarrow \text{findClosestPair}\left(\mathcal{S}_A, d\right)$
5. \hspace{1em} $(u_B, v_B), d_B \leftarrow \text{findClosestPair}\left(\mathcal{S}_B, d\right)$
6. \hspace{1em} if $d_A \leq d_B$ then
7. \hspace{2em} remove$(u_A, v_A, \mathcal{S}_A)$
8. \hspace{2em} insert$(u_A \cup v_A, \mathcal{S}_A)$
9. \hspace{1em} else
10. \hspace{2em} remove$(u_B, v_B, \mathcal{S}_B)$
11. \hspace{2em} insert$(u_B \cup v_B, \mathcal{S}_B)$
12. \hspace{1em} end if
13. for $(A_i, B_j) \in \mathcal{S}_A \times \mathcal{S}_B$ do
14. \hspace{1em} DPSynthesis$(A_i, B_j)$
15. end for
16. end while
17. /* Revert the last merging step, and check the bound $\mathcal{C}_2$. */

$\mathcal{C}_1$ is satisfied

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

12 DP codes
Benchmark matrices

Edges

Center

Random

Rows
Efficiency of the distance-based heuristic: $6 \times 6$ matrix product

- Generating the 41209 different algorithms: 2h15 per benchmark
- Running our algorithm: 10 seconds per benchmark
Impact of the metric on the trade-off strategy

Center benchmark

Rows benchmark

Edges benchmark

Random benchmark
Linear algebra basic blocks

- Cholesky decomposition
- Triangular matrix inversion
- Matrix multiplication
## Cholesky decomposition and triangular matrix inversion

**Cholesky decomposition**

\[
 b_{i,j} = \begin{cases} 
 \sqrt{c_{i,i}} & \text{if } i = j \\
 \frac{c_{i,j}}{b_{j,j}} & \text{if } i \neq j 
\end{cases}
\]

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

**Triangular matrix inversion**

\[
 n_{i,j} = \begin{cases} 
 \frac{1}{b_{i,i}} & \text{if } i = j \\
 -\frac{c_{i,j}}{b_{i,i}} & \text{if } i \neq j 
\end{cases}
\]

where \( c_{i,j} = \sum_{k=j}^{i-1} b_{i,k} \cdot n_{k,j} \)
## Cholesky decomposition and triangular matrix inversion

### Cholesky decomposition

\[ b_{i,j} = \begin{cases} \sqrt{c_{i,i}} & \text{if } i = j \\ \frac{c_{i,j}}{b_{j,j}} & \text{if } i \neq j \end{cases} \]

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

### Triangular matrix inversion

\[ n_{i,j} = \begin{cases} 1 & \text{if } i = j \\ \frac{-c_{i,j}}{b_{i,i}} & \text{if } i \neq j \end{cases} \]

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. \( f_4(i_1, i_2) = \left\lfloor \frac{(i_1 + i_2)}{2} \right\rfloor + t, \)

\( i_1 \) 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 $f_4(i_1, i_2) = \lceil (i_1 + i_2)/2 \rceil + 1$ 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 $\times$ and $/$
Conclusions and perspectives

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
Contributions

A framework for certified fixed-point code synthesis

- Formalization and implementation of an arithmetic model
  - allows certification
  - handles √ 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
Conclusions and perspectives

Contributions

A framework for certified fixed-point code synthesis

- Formalization and implementation of an arithmetic model
  - allows certification
  - handles √ 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
- SCAN12: Approach based on instruction selection for fast and certified code generation
Perspectives

- Integrate the matrix inversion flow

  - Cholesky decomposition
  - Triangular matrix inversion
  - Matrix multiplication

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
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
Conclusions and perspectives

- [MCCS02] Daniel Menard, Daniel Chillet, François Charot, and Olivier Sentieys. Automatic floating-point to fixed-point conversion for DSP code generation.
- [FRC03] Claire F. Fang, Rob A. Rutenbar, and Tsuhan Chen. Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs.
- [LHD14] Benoît Lopez, Thibault Hilaire, and Laurent-Stéphane Didier. Formatting bits to better implement signal processing algorithms.