Automated synthesis of fixed-point programs: 
the case of matrix multiplication and, 
some elements on matrix inversion

Amine Najahi 

Advisers: M. Martel and G. Revy 

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

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

- Some embedded systems do not have any FPU (floating-point unit)

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

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

- Some embedded systems do not have any FPU (floating-point unit)

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

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

- Some embedded systems do not have any FPU (floating-point unit)

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

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

- Some embedded systems do not have any FPU (floating-point unit)

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

With the floating-point arithmetic, it is very easy to program!

```c
int main ()
{
    int i,j,k;
    float A[N][N] = {...} , B[N][N] = {...} , C[N][N] = {0,...,0};
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++) /* This inner loop computes the dot product of row i and column j */
            C[i][j] += A[i][k] * B[k][j];
}
```

What makes the problem harder in fixed-point?

Intermediate computations depend on the input variables range and computation scheme

Some works on linear algebra primitives in fixed-point

Lee et al. (2006): 8×8 matrix-vector products for the computation of DCT's

- Relies on some DCT properties

Frantz et al. (2007): linear algebra routines (mostly matrix inversion) based on simulation

- No strict guarantee on the error bounds
- Based on lengthy simulations

A. Najahi Automated synthesis of fixed-point programs: the case of matrix multiplication
Matrix multiplication

With the floating-point arithmetic, it is very easy to program!!

```c
int main()
{
    int i, j, k;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++) /* This inner loop computes the dot product of row i and column j */
                C[i][j]+=A[i][k]*B[k][j];
}
```

What makes the problem harder in fixed-point:

- Intermediate computations depend on the input variables range and computation scheme
- Some works on linear algebra primitives in fixed-point
- Lee et al. (2006): $8 \times 8$ matrix-vector products for the computation of DCT’s
  - The first matrix is constant: DCT coefficients
  - Relies on some DCT properties
- Frantz et al. (2007): linear algebra routines (mostly matrix inversion) based on simulation
  - No strict guarantee on the error bounds
  - Based on lengthy simulations
Matrix multiplication

With the floating-point arithmetic, it is very easy to program!!

```c
int main()
{
    int i,j,k;
    for (i = 0; i < N ; i++)
        for (j = 0; j < N ; j++)
            for (k = 0; k < N ; k++) /* This inner loop computes the dot product of row i and column j */
                C[i][j]+=A[i][k]*B[k][j];
}
```

What makes the problem harder in fixed-point?

- Intermediate computations depend on the input variables range and computation scheme
Matrix multiplication

With the floating-point arithmetic, it is very easy to program!!

```c
int main()
{
    int i, j, k;
    for (i = 0; i < N ; i++)
        for (j = 0; j < N ; j++)
            for (k = 0; k < N ; k++) /* This inner loop computes the dot product of row i and column j */
                C[i][j]+=A[i][k]*B[k][j];
}
```

What makes the problem harder in fixed-point?

- Intermediate computations depend on the input variables range and computation scheme

Some works on linear algebra primitives in fixed-point

- Lee et al. (2006): 8 × 8 matrix-vector products for the computation of DCT's
  - The first matrix is constant: DCT coefficients
  - Relies on some DCT properties

- Frantz et al. (2007): linear algebra routines (mostly matrix inversion) based on simulation
  - No strict guarantee on the error bounds
  - Based on lengthy simulations
Outline of the talk

1. Synthesizing fixed-point formulas: combinatorial and numerical issues

2. Efficient matrix multiplication in fixed-point arithmetic

3. State of the art on matrix inversion in fixed-point arithmetic
Outline of the talk

1. Synthesizing fixed-point formulas: combinatorial and numerical issues

2. Efficient matrix multiplication in fixed-point arithmetic

3. State of the art on matrix inversion in fixed-point arithmetic
Background on fixed-point arithmetic

- Main idea of fixed-point arithmetic:
  - Interpret bit packets as integers coupled with a scale factor: $z \cdot 2^{-n}$
  - Example with $z = (1000010)_2$ and $n = 4$

Unsigned integer $z = 2^7 + 2^1 = 130$

Scale factor $n = 4$

Fixed-point value $z \cdot 2^{-4} = 2^3 + 2^{-3} = 8.125$

The scale factor (or fixed-point format) is implicit, only the programmer is aware of it.

We will denote by $Q_a, b$ a fixed-point format with $a$ integer bits and $b$ fractional bits.
Background on fixed-point arithmetic

- Main idea of fixed-point arithmetic:
  - Interpret bit packets as integers coupled with a scale factor: $z \cdot 2^{-n}$
  - Example with $z = (10000010)_2$ and $n = 4$

⚠️ The scale factor (or fixed-point format) is implicit, only the programmer is aware of it
Background on fixed-point arithmetic

- Main idea of fixed-point arithmetic:
  - Interpret bit packets as integers coupled with a scale factor: \( z \cdot 2^{-n} \)
  - Example with \( z = (10000010)_2 \) and \( n = 4 \)

\[
\begin{align*}
\text{Unsigned integer} & : z = 2^7 + 2^1 = 130 \\
\text{Scale factor} & : n = 4 \\
\text{Fixed-point value} & : z \cdot 2^{-4} = 2^3 + 2^{-3} = 8.125
\end{align*}
\]

⚠️ The scale factor (or fixed-point format) is implicit, only the programmer is aware of it

- We will denote by \( Q_{a,b} \) a fixed-point format with \( a \) integer bits and \( b \) fractional bits
Fixed-point arithmetic model (1/2)

Arithmetic model to track errors in fixed-point computations

- For each intermediate variable $r_i$, we store 2 intervals $\text{val}(r_i)$ and $\text{err}(r_i)$
- For each basic operator, we have rules to compute $\text{val}(r_i)$ and $\text{err}(r_i)$
Fixed-point arithmetic model (1/2)

Arithmetic model to track errors in fixed-point computations

- For each intermediate variable $r_i$, we store 2 intervals $\text{val}(r_i)$ and $\text{err}(r_i)$
- For each basic operator, we have rules to compute $\text{val}(r_i)$ and $\text{err}(r_i)$

Addition:

- The two variables have to be in the same fixed-point format

\[
\begin{align*}
\text{val}(r_i) &= \text{val}(l) + \text{val}(r) \\
\text{err}(r_i) &= \text{err}(l) + \text{err}(r)
\end{align*}
\]

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
+ & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
\hline
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1
\end{array}
\]

5.0625 + 2.65625 = 7.7187
Fixed-point arithmetic model (2/2)

- **Multiplication:**
  - The product of a $Q_{a,b}$ variable by a $Q_{c,d}$ variable yields a $Q_{a+c,b+d}$ variable

\[
\begin{align*}
\text{val}(ri) &= \text{val}(l) \times \text{val}(r) \\
\text{err}(ri) &= \text{err}_{\text{mul}} + \text{err}(l) \times \text{err}(r) \\
&\quad + \text{err}(l) \times \text{val}(r) \\
&\quad + \text{val}(r) \times \text{err}(l)
\end{align*}
\]

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
\times & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
5.0625 \\
1.328125 \\
6.723632812 \\
6.625
\end{array}
\]
Synthesizing fixed-point formulas: combinatorial and numerical issues

Fixed-point arithmetic model (2/2)

- **Multiplication:**
  - The product of a $Q_{a,b}$ variable by a $Q_{c,d}$ variable yields a $Q_{a+c,b+d}$ variable

  \[
  \text{val}(r) = \text{val}(l) \times \text{val}(r) \\
  \text{err}(r) = \text{err}_{mul} + \text{err}(l) \times \text{err}(r) \\
  + \text{err}(l) \times \text{val}(r) \\
  + \text{val}(r) \times \text{err}(l)
  \]

  \[
  \begin{array}{c}
  \times \\
  \downarrow \\
  \circ \\
  \downarrow \\
  l \\
  \uparrow \\
  r
  \end{array}
  \]

  \[
  \begin{array}{c}
  10100010 \\
  \times \\
  01010101
  \end{array}
  \]

  \[\text{val}(l) = 5.0625, \text{val}(r) = 1.328125, \text{val}(l) \times \text{val}(r) = 6.723632812, \text{val}(l) \times \text{val}(r) = 6.625\]

- **Physical and virtual shifts:**

  \[
  \begin{array}{c}
  \gg \\
  \downarrow \\
  \uparrow \\
  \downarrow \\
  l \\
  \uparrow \\
  l
  \end{array}
  \]

  \[
  \gg 2
  \]

  \[
  \begin{array}{c}
  01110010 \\
  \gg 2
  \end{array}
  \]

  \[\text{truncated} \]

  \[
  \begin{array}{c}
  00011100 \quad \end{array}
  \]

  \[\text{val}(l) = 3.5625, \text{val}(l) = 0.875\]

  \[
  \begin{array}{c}
  00011100 \\
  \gg v 2
  \end{array}
  \]

  \[\text{val}(l) = 3.5\]

  \[
  00011100
  \]

  \[\text{val}(l) = 0.875\]
Numerical issues in dot product generation

- The building block of matrix multiplication is the dot product operation
  - Let us consider a size 3 dot product: \((a_0 \times b_0) + (a_1 \times b_1) + (a_2 \times b_2)\) and the following input fixed-point formats:

<table>
<thead>
<tr>
<th>Value</th>
<th>Fixed-point format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1, 1.57</td>
<td>(Q_{1,7})</td>
</tr>
<tr>
<td>0.1, 1.98</td>
<td>(Q_{1,7})</td>
</tr>
<tr>
<td>0.01, 0.87</td>
<td>(Q_{0,8})</td>
</tr>
<tr>
<td>1.1, 1.86</td>
<td>(Q_{1,7})</td>
</tr>
<tr>
<td>0.15</td>
<td>(Q_{4,4})</td>
</tr>
<tr>
<td>2.3, 3.3</td>
<td>(Q_{2,6})</td>
</tr>
</tbody>
</table>
Numerical issues in dot product generation

- The building block of matrix multiplication is the dot product operation
  - Let us consider a size 3 dot product: \((a_0 \times b_0) + (a_1 \times b_1) + (a_2 \times b_2)\)
  - and the following input fixed-point formats:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_0)</td>
<td>([0.1, 1.57])</td>
</tr>
<tr>
<td>(b_0)</td>
<td>([0, 1.98])</td>
</tr>
<tr>
<td>(a_1)</td>
<td>([0.01, 0.87])</td>
</tr>
<tr>
<td>(b_1)</td>
<td>([1.1, 1.86])</td>
</tr>
<tr>
<td>(a_2)</td>
<td>([0, 15.4])</td>
</tr>
<tr>
<td>(b_2)</td>
<td>([2, 3.3])</td>
</tr>
</tbody>
</table>

- Let us focus on 2 different schemes to compute the sum of products:

\[
(c_0 + (c_1 + c_2))
\]

\[
((c_0 + c_1) + c_2)
\]
Numerical issues in dot product generation

- The building block of matrix multiplication is the dot product operation
  - Let us consider a size 3 dot product: \((a_0 \times b_0) + (a_1 \times b_1) + (a_2 \times b_2)\)
    and the following input fixed-point formats:

<table>
<thead>
<tr>
<th></th>
<th>(a_0)</th>
<th>(b_0)</th>
<th>(a_1)</th>
<th>(b_1)</th>
<th>(a_2)</th>
<th>(b_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>([0.1, 1.57])</td>
<td>([0, 1.98])</td>
<td>([0.01, 0.87])</td>
<td>([1.1, 1.86])</td>
<td>([0, 15.4])</td>
<td>([2, 3.3])</td>
</tr>
<tr>
<td>Fixed-point format</td>
<td>(Q_{1,7})</td>
<td>(Q_{1,7})</td>
<td>(Q_{0,8})</td>
<td>(Q_{1,7})</td>
<td>(Q_{4,4})</td>
<td>(Q_{2,6})</td>
</tr>
</tbody>
</table>

- Let us focus on 2 different schemes to compute the sum of products:

\[
(c_0 + c_1 + c_2) \quad \text{with 16 bits precision}
\]

\[
((c_0 + c_1) + c_2)
\]
Combinatorial issues in dot product generation

Number of dot product evaluation schemes

- Given by the sequence A001147($n$) in the OEIS and the formula: $(2n – 1)!!$

<table>
<thead>
<tr>
<th>Dot product size</th>
<th>3</th>
<th>5</th>
<th>10</th>
<th>16</th>
<th>20</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of schemes</td>
<td>3</td>
<td>105</td>
<td>34459425 ≈ $2^{25}$</td>
<td>6190283353629375 ≈ $2^{52}$</td>
<td>8200794532637891559375 ≈ $2^{73}$</td>
<td>...</td>
</tr>
</tbody>
</table>

Remarks

- Picking a scheme that minimizes the evaluation error is one of the difficulties of writing fixed-point code
  - Makes it hard to write fixed-point code by hand
  - Appeals for tools with strong heuristics to automate the process
The CGPE\textsuperscript{1} library

- Initially developed by Revy and Mouilleron
  - With the aim of generating fast and certified C code for polynomial evaluation

1. fast \(\rightsquigarrow\) selects schemes that reduce the evaluation latency on a given target, by using (as much as possible) the architectural features

2. certified \(\rightsquigarrow\) produces a bound on the error entailed by the evaluation within the given target’s arithmetic
The CGPE \(^1\) library

- Initially developed by Revy and Mouilleron
  - With the aim of generating fast and certified C code for polynomial evaluation

1. fast \(\rightarrow\) selects schemes that reduce the evaluation latency on a given target, by using (as much as possible) the architectural features

2. certified \(\rightarrow\) produces a bound on the error entailed by the evaluation within the given target’s arithmetic

- Front-ends available so far: sum, dot product, univariate and bivariate polynomials
- Back-ends available so far: C code, VHDL code, GAPPAPA certificates

\(^1\) Code Generation for Polynomial Evaluation
Outline of the talk

1. Synthesizing fixed-point formulas: combinatorial and numerical issues

2. Efficient matrix multiplication in fixed-point arithmetic

3. State of the art on matrix inversion in fixed-point arithmetic
Defining the problem

- **Inputs:**
  - A black box (CGPE) that synthesises code for dot products in fixed-point arithmetic
  - Vector \( u \) of fixed-point variables
  - Vector \( v \) of fixed-point variables
  - 2 fixed-point matrices \( A \) and \( B \)
  - C code that evaluates \( u \cdot v \)
  - Accuracy certificate
Defining the problem

- **Inputs:**
  - a black box (CGPE) that synthesises code for dot products in fixed-point arithmetic
  - 2 fixed-point matrices $A$ and $B$

- **Output**
  - C code that evaluates the product $M = A \cdot B$ in fixed-point arithmetic
Straightforward algorithms

**Accurate product**

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

AccurateProduct

**Inputs:**
- Two fixed-point square matrices A and B

**Outputs:**
- C code to compute the product AB

**Steps:**
1. `for 1 < i ≤ n do`
2. `for 1 < j ≤ n do`
3. `cgpeGenDotProduct(A_i, B_j);`
4. `end for`
5. `end for`

---

**Compact product**

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

CompactProduct

**Inputs:**
- Two fixed-point square matrices A and B

**Outputs:**
- C code to compute the product AB

**Steps:**
1. compute v such that \( v = A_1 \cup A_2 \cup \cdots \cup A_n \)
2. compute w such that \( w = B_1 \cup B_2 \cup \cdots \cup B_n \)
3. `cgpeGenDotProduct(v, w);`
Illustration through a toy example

We consider the multiplication of the following two fixed-point matrices:

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

We consider the multiplication of the following two fixed-point matrices:

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

**Accurate product**

1. Output format of $\text{DotProduct}_{0,0}$: $Q_{26,6}$
2. Output format of $\text{DotProduct}_{0,1}$: $Q_{18,14}$
3. Output format of $\text{DotProduct}_{1,0}$: $Q_{15,17}$
4. Output format of $\text{DotProduct}_{1,1}$: $Q_{7,25}$

- Certified errors bounds:
  \[
  \begin{pmatrix}
  0.03125 & 0.00012207 \\
  1.52588e-05 & 5.96046e-08
  \end{pmatrix}
  \]
- Average error bound: $0.00784 \approx 2^{-7}$

**Compact product**

\[
u = \begin{pmatrix}
[-1000, 1000] & [-3000, 3000]
\end{pmatrix}
\]

\[
v = \begin{pmatrix}
[-2000, 2000] \\
[-4000, 4000]
\end{pmatrix}
\]

1. Output format of $\text{DotProduct}_{u,v}$: $Q_{26,6}$

- Certified errors bounds:
  \[
  \begin{pmatrix}
  0.03125 & 0.03125 \\
  0.03125 & 0.03125
  \end{pmatrix}
  \]
- Average error bound: $0.03125 \approx 2^{-5}$
Looking for trade-offs

Remarks
- Accurate product generates large code sizes (prohibitive in embedded systems)
- Compact product generates 1 dot product, to the expense of numerical accuracy
Looking for trade-offs

Remarks

- Accurate product generates large code sizes (prohibitive in embedded systems)
- Compact product generates 1 dot product, to the expense of numerical accuracy
- But are there interesting trade-offs to look for?
Looking for trade-offs

Remarks

- Accurate product generates large code sizes (prohibitive in embedded systems)
- Compact product generates 1 dot product, to the expense of numerical accuracy
- But are there interesting trade-offs to look for?

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

Accurate product

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

Remarks

- Accurate product generates large code sizes (prohibitive in embedded systems)
- Compact product generates 1 dot product, to the expense of numerical accuracy
- But are there interesting trade-offs to look for?

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

Compact product

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

Remarks

- Accurate product generates large code sizes (prohibitive in embedded systems)
- Compact product generates 1 dot product, to the expense of numerical accuracy
- But are there interesting trade-offs to look for?

**Idea:** Merge certain rows/columns to reduce the number of the generated dot products
Looking for trade-offs

Remarks

- Accurate product generates large code sizes (prohibitive in embedded systems)
- Compact product generates 1 dot product, to the expense of numerical accuracy
- But are there interesting trade-offs to look for?

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

- Idea: Merge certain rows/columns to reduce the number of the generated dot products

Number of ways to merge \( n \) vectors

- Given by the \( n^{th} \) Bell number

<table>
<thead>
<tr>
<th>Number of vectors</th>
<th>3</th>
<th>5</th>
<th>10</th>
<th>16</th>
<th>20</th>
<th>( \ldots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of schemes</td>
<td>5</td>
<td>52</td>
<td>115975 ( \approx 2^{17} )</td>
<td>10480142147 ( \approx 2^{33} )</td>
<td>51724158235372 ( \approx 2^{46} )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>
Distances

The Hausdorff distance $d_H$

\[ d_H : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \]

\[ d_H([a, \bar{a}], [b, \bar{b}]) = \max \{|a - b|, |\bar{a} - \bar{b}|\} \]

Example

Let $A = [-3, 1]$ and $B = [2, 4]$ be two intervals in $l(\mathbb{R})$, we have:

- $\cup (A, B) = [-3, 4]$
- $d_H(A, B) = 5$
Distances

The Hausdorff distance $d_H$

$$d_H : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

$$d_H([a, \bar{a}], [b, \bar{b}]) = \max \{ |a - b|, |\bar{a} - \bar{b}| \}$$

Example

Let $A = [-3, 1]$ and $B = [2, 4]$ be two intervals in $I(\mathbb{R})$, we have:

- $\cup(A, B) = [-3, 4]$
- $d_H(A, B) = 5$

Another possible criterion

$$d_d : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

$$d_d([a, \bar{a}], [b, \bar{b}]) = \text{diam}([a, \bar{a}] \cup [b, \bar{b}])$$

Example

- $\cup(A, B) = [-3, 4]$
- $d_d(A, B) = 7$
Closest pair strategy (1/2)

- Given, a distance and a collection of vectors:
  - we can write a routine that merges the closest pair of vectors.
Closest pair strategy (1/2)

- Given, a distance and a collection of vectors:
  - we can write a routine that merges the closest pair of vectors.

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

Closest pair: \( A_0 \) and \( A_3 \)
Closest pair strategy (1/2)

- Given, a distance and a collection of vectors:
  - we can write a routine that merges the closest pair of vectors.

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

Closest pair: \(A_0\) and \(A_3\)

\[
A' = A_0 \cup A_3
\]

Closest pair: \(A_1\) and \(A_4\)
**Closest pair strategy (1/2)**

- Given, a distance and a collection of vectors:
  - we can write a routine that merges the closest pair of vectors.

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

Closest pair: \(A_0\) and \(A_3\)

\[
A' = A_0 \cup A_3
\]

Closest pair: \(A_1\) and \(A_4\)

\[
A' = A_0 \cup A_4
\]
Closest pair strategy (2/2)

Algorithm 1 Dynamic Closest Pair algorithm

Inputs:
- Two square matrices \( A \in \mathbb{F}ix^{n \times n} \) and \( B \in \mathbb{F}ix^{n \times n} \)
  - a criterion \( C \)

Outputs:
- \( C \) code to compute the product \( AB \) s.t. \( C \) is satisfied

Steps:
1. \( \mathcal{S}_1 = \{ A_1, \ldots, A_n \} \)
2. \( \mathcal{S}_2 = \{ B_1, \ldots, B_n \} \)
3. while \( C \) is satisfied do
4. \( (u_1, u_2, d_{u_1, u_2}) = \text{findClosestPair}(\mathcal{S}_1) \)
5. \( (v_1, v_2, d_{v_1, v_2}) = \text{findClosestPair}(\mathcal{S}_2) \)
6. if \( d_{u_1, u_2} \leq d_{v_1, v_2} \) then
7. \( \text{remove}(u_1, \mathcal{S}_1); \text{remove}(u_2, \mathcal{S}_1); \text{insert}(u_1 \cup u_2, \mathcal{S}_1) \)
8. else
9. \( \text{remove}(v_1, \mathcal{S}_2); \text{remove}(v_2, \mathcal{S}_2); \text{insert}(v_1 \cup v_2, \mathcal{S}_2) \)
10. end if
11. for \( A_i \in \mathcal{S}_1 \) do
12. for \( B_j \in \mathcal{S}_2 \) do
13. \( \text{cgpeGenDotProduct}(A_i, B_j) \)
14. end for
15. end for
16. end while
Closest pair strategy (2/2)

Algorithm 2 Dynamic Closest Pair algorithm

Inputs:
- Two square matrices $A \in \mathbb{F}ix^{n \times n}$ and $B \in \mathbb{F}ix^{n \times n}$
- A criterion $\mathcal{C}$: the average error bound is $\leq \epsilon$

Outputs:
- C code to compute the product $AB$ s.t. $\mathcal{C}$ is satisfied

Steps:
1: $\mathcal{S}_1 = \{A_1, \ldots, A_n\}$
2: $\mathcal{S}_2 = \{B_1, \ldots, B_n\}$
3: while average error $\leq \epsilon$ do
4: \hspace{1em} $(u_1, u_2, d_{u_1}, d_{u_2}) = \text{findClosestPair}(\mathcal{S}_1)$
5: \hspace{1em} $(v_1, v_2, d_{v_1}, d_{v_2}) = \text{findClosestPair}(\mathcal{S}_2)$
6: \hspace{1em} if $d_{u_1}, u_2 \leq d_{v_1}, v_2$ then
7: \hspace{2em} remove$(u_1, \mathcal{S}_1)$; remove$(u_2, \mathcal{S}_1)$; insert$(u_1 \cup u_2, \mathcal{S}_1)$
8: \hspace{1em} else
9: \hspace{2em} remove$(v_1, \mathcal{S}_2)$; remove$(v_2, \mathcal{S}_2)$; insert$(v_1 \cup v_2, \mathcal{S}_2)$
10: \hspace{1em} end if
11: \hspace{1em} for $A_i \in \mathcal{S}_1$ do
12: \hspace{2em} for $B_j \in \mathcal{S}_2$ do
13: \hspace{3em} cgpeGenDotProduct$(A_i, B_j)$
14: \hspace{2em} end for
15: \hspace{1em} end for
16: end while
Benchmarks

1. Weight matrices with dynamic range $2^{\left\lfloor \frac{n}{2} \right\rfloor - 1}$
2. Normally distributed random matrices (generated by matlab)
3. We took the Hadamard product of both matrices $H$
4. The matrices fed to the algorithm are $\text{midrad}(H, 1)$
Results

![Graph showing the number of dot product codes vs. average precision bound for different algorithms.]

- Center Benchmark
- Random pair Algorithm
- Best pair Algorithm

A. Najahi
Automated synthesis of fixed-point programs: the case of matrix multiplication
Results

![Graph showing efficiency of matrix multiplication algorithms]

- Efficient matrix multiplication in fixed-point arithmetic
- Random pair Algorithm
- Best pair Algorithm

Number of dot product codes vs. Average precision bound forEdges Benchmark.

- Random pair Algorithm: Red line
- Best pair Algorithm: Green dashed line

A. Najahi
Results
Results

Efficient matrix multiplication in fixed-point arithmetic

A. Najahi

Automated synthesis of fixed-point programs: the case of matrix multiplication
Demo
Outline of the talk

1. Synthesizing fixed-point formulas: combinatorial and numerical issues

2. Efficient matrix multiplication in fixed-point arithmetic

3. State of the art on matrix inversion in fixed-point arithmetic
Matrix inversion in THALES STAP benchmark

```c
void Mat_Invert (int ntt, int nsa, Cplfloat In[ntt*nsa][ntt*nsa], Cplfloat Out[ntt*nsa][ntt*nsa]) {
    double inv[nsa*ntt][2*nsa*ntt][2];
    double pivot[2], coef[2];
    float re, im;
    int i=0, j=0, k=0, l=0;
    for (i=0; i<ntt*nsa; i++) {
        for (j=0; j<ntt*nsa; j++) {
            inv[i][j][0] = (double) In[i][j].re;
            inv[i][j][1] = (double) In[i][j].im;
            if (i == j) {
                inv[i][j+nsa*ntt][0] = 1.0; inv[i][j+nsa*ntt][1] = 0.0;
            } else {
                inv[i][j+nsa*ntt][0] = 0.0; inv[i][j+nsa*ntt][1] = 0.0;
            }
        }
    }
    for (i=0; i<nsa*ntt; i++) {
        pivot[0]=inv[i][i][0]; pivot[1]=inv[i][i][1];
        if (pivot[0] == 0.) {
            printf("\n Pivot nul re = %.2f , im = %.2f\n", pivot[0], pivot[1]);
            exit(0);
        }
        for (j=i; j<2*nsa*ntt; j++) {
            re = inv[i][j][0]; im = inv[i][j][1];
            inv[i][j][0] = (re * pivot[0] + im * pivot[1]) / (pivot[0] * pivot[0] + pivot[1] * pivot[1]);
            inv[i][j][1] = (im * pivot[0] - re * pivot[1]) / (pivot[0] * pivot[0] + pivot[1] * pivot[1]);
        }
        for (k=0; k<nsa*ntt; k++) {
            if (i!=k) {
                coef[0] = inv[k][i][0]; coef[1] = inv[k][i][1];
                for (l=i; l<2*nsa*ntt; l++) {
                    inv[k][l][0] -= (coef[0] * inv[i][l][0] - coef[1] * inv[i][l][1]);
                    inv[k][l][1] -= (coef[0] * inv[i][l][1] + coef[1] * inv[i][l][0]);
                }
            }
        }
    }
    for (i=0; i<nsa*ntt; i++) {
        for (j=0; j<nsa*ntt; j++) {
            Out[i][j].re = (float) inv[i][j+nsa*ntt][0];
            Out[i][j].im = (float) inv[i][j+nsa*ntt][1];
        }
    }
}
```
The algorithm step by step

Initialization:

\[
\begin{pmatrix}
  I_{n,0} & I_{n,1} & \cdots & I_{n,n-1} \\
  I_{n-1,0} & I_{n-1,1} & \cdots & I_{n-1,n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  I_{0,0} & I_{0,1} & \cdots & I_{0,n-1}
\end{pmatrix}
\]

\[
\begin{pmatrix}
  1 & \cdots & \cdots & 0 \\
  0 & 1 & \cdots & 0 \\
  0 & \vdots & \ddots & 0 \\
  0 & \cdots & \cdots & 1
\end{pmatrix}
\]
The algorithm step by step

Step 1: Picking the pivot

\[
\begin{pmatrix}
In_{0,0} & In_{0,1} & \cdots & In_{0,n-1} & 1 & \cdots & \cdots & 0 \\
In_{1,0} & In_{1,1} & \cdots & In_{1,n-1} & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
In_{n-1,0} & In_{n-1,1} & \cdots & In_{n-1,n-1} & 0 & \cdots & \cdots & 1
\end{pmatrix}
\]
The algorithm step by step

Step 1: Normalizing the pivot’s row

\[
\begin{pmatrix}
1 & \frac{In_{0,1}}{In_{0,0}} & \cdots & \frac{In_{0,n-1}}{In_{0,0}} \\
In_{1,0} & \frac{In_{1,1}}{In_{1,0}} & \cdots & \frac{In_{1,n-1}}{In_{1,0}} \\
\vdots & \vdots & \ddots & \vdots \\
In_{n-1,0} & \frac{In_{n-1,1}}{In_{n-1,n-1}} & \cdots & \frac{In_{n-1,n-1}}{In_{n-1,n-1}} \\
\end{pmatrix}
\]
The algorithm step by step

Step 1: Putting zeros in the pivot’s column

\[
\begin{pmatrix}
1 & \frac{\ln_{0,1}}{\ln_{0,0}} & \cdots & \frac{\ln_{0,n-1}}{\ln_{0,0}} & \frac{1}{\ln_{0,0}} & \cdots & 0 \\
0 & \frac{\ln_{1,1} - \ln_{1,0}}{\ln_{0,0}} & \cdots & \frac{\ln_{1,n-1} - \ln_{1,0}}{\ln_{0,0}} & 0 & \cdots & 0 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & \frac{\ln_{n-1,1} - \ln_{n-1,0}}{\ln_{0,0}} & \cdots & \frac{\ln_{n-1,n-1} - \ln_{n-1,0}}{\ln_{0,0}} & 0 & \cdots & 0 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{pmatrix}
\]
The algorithm step by step

Step 2: Picking the pivot

\[
\begin{pmatrix}
1 & \frac{In_{0,1}}{In_{0,0}} & \cdots & \frac{In_{0,n-1}}{In_{0,0}} & \frac{1}{In_{0,0}} \\
0 & In_{1,1} - In_{1,0} & \cdots & In_{1,n-1} - In_{1,0} & 0 \\
0 & \vdots & \ddots & \vdots & \vdots \\
0 & In_{n-1,1} - In_{n-1,0} & \cdots & In_{n-1,n-1} - In_{n-1,0} & 0 \\
0 & \vdots & \cdots & \vdots & 1
\end{pmatrix}
\]
Some remarks on this implementation

- It is based on row reduction
- The implementation is optimized
- Computes the inverse in place
  - avoids backward as well as forward substitution

Which parts are easily convertible to fixed-point?

1. Initialization ✓
2. Picking the pivot ✓
3. Normalizing the pivot’s row (Division by the pivot) ✗
4. Putting zeros in the pivot’s column (Additions and multiplication) ✓
Issues with the division operator

- Consider the analytic method to invert a matrix: \( A^{-1} = \frac{\text{com}(A)^T}{\det(A)} \)

- For a \( 2 \times 2 \) matrix \( A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \), \( A^{-1} = \frac{1}{\det(A)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix} \)

**Example**

Let \( a, b, c, d \in [1, 2] \):

- Suppose the format of \( a, b, c \) and \( d \) is \( Q_{3,29} \)
- Using interval arithmetic, \( \det(A) = ad - bc \in [-3, 3] \Rightarrow Q_{3,29} \)
- Now consider the matrix \( M = \begin{pmatrix} 1 + 2^{-28} & 1 \\ 1 & 1 \end{pmatrix} \Rightarrow \det(M) = 2^{-28} \)
- Then what should be the format of \( \frac{a}{\det(A)} \)?
  - To avoid overflow, it should be at least \( Q_{31,1} \)
Available works on linear algebra routines

Works that exploits simulation-based methods

- Frantz et al. (2007):
  - Decompositions investigated: SVD, Cholesky, LU and QR
  - Matrices size: up to 30
  - A posteriori error estimation

- Irturk et al. (2006): GUSTO tool
  - Decompositions investigated: Cholesky, LU, QR and analytic
  - Matrices size: up to 8
  - A posteriori error estimation

Works with proven and certified error bounds

- ?
Automated synthesis of fixed-point programs:
the case of matrix multiplication and,
some elements on matrix inversion

Amine Najahi

Advisers: M. Martel and G. Revy

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