Automated synthesis of fixed-point programs:

the case of matrix multiplication and, some elements on matrix inversion

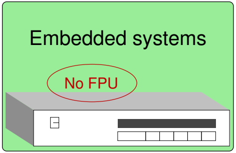
Amine Najahi

Advisers: M. Martel and G. Revy

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

Motivation

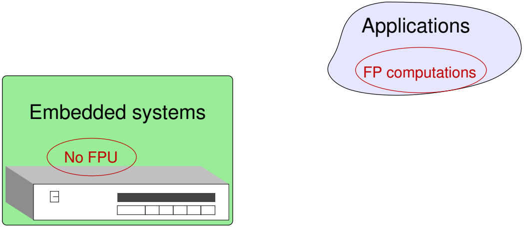
- Embedded systems are ubiquitous
  - microprocessors and/or DSPs dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost
- Some embedded systems do not have any FPU (floating-point unit)



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

Motivation

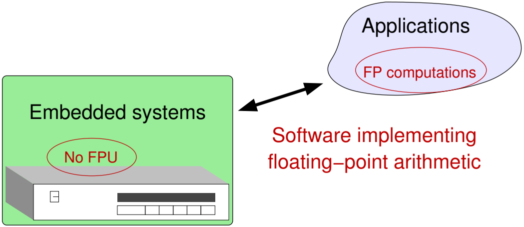
- Embedded systems are ubiquitous
  - microprocessors and/or DSPs dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost
- Some embedded systems do not have any FPU (floating-point unit)



- 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)

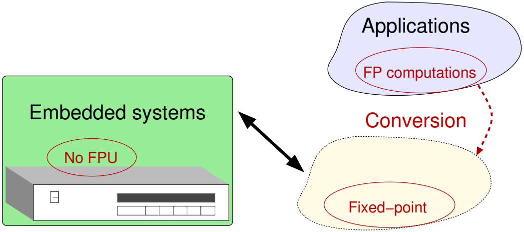


- Highly used in audio and video applications
  - demanding on floating-point computations

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

Matrix multiplication

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

```
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];
}
```

Matrix multiplication

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

```
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

Matrix multiplication

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

```
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
  - 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

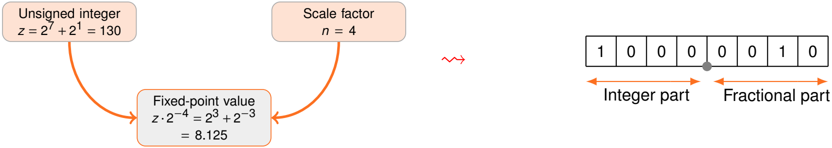
1. Synthesizing fixed-point formulas: combinatorial and numerical issues

2. Efficient matrix multiplication in fixed-point arithmetic

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

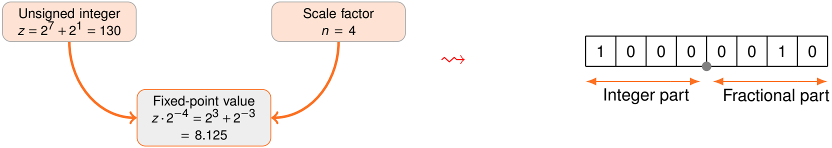
Background on fixed-point arithmetic

- Main idea of fixed-point arithmetic:
  - Interpret bit packets as integers coupled with a scale factor: $z \cdot 2^{-n}$
  - Example with $z = (1000\,0010)_2$ and $n = 4$



Background on fixed-point arithmetic

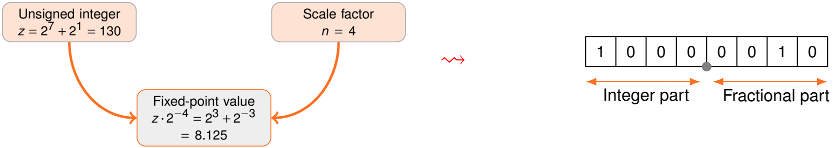
- Main idea of fixed-point arithmetic:
  - Interpret bit packets as integers coupled with a scale factor: $z \cdot 2^{-n}$
  - Example with $z = (1000\,0010)_2$ and $n = 4$



⚠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 $z = (1000\,0010)_2$ and $n = 4$



⚠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

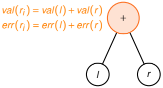
- For each intermediate variable r_i, we store 2 intervals $\textbf{val}(r_i)$ and $\textbf{err}(r_i)$
- For each basic operator, we have rules to compute $\textbf{val}(r_i)$ and $\textbf{err}(r_i)$

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 $\textbf{val}(r_i)$ and $\textbf{err}(r_i)$
- For each basic operator, we have rules to compute $\textbf{val}(r_i)$ and $\textbf{err}(r_i)$

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

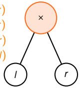


	101.00010	5.0625
+	010.10101	2.65625
	111.10111	7.7187

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

$$
\begin{aligned}
val(r_i) &= val(l) \times val(r) \\
err(r_i) &= err_{mul} + err(l) \times err(r) \\
&\quad + err(l) \times val(r) \\
&\quad + val(r) \times err(l)
\end{aligned}
$$



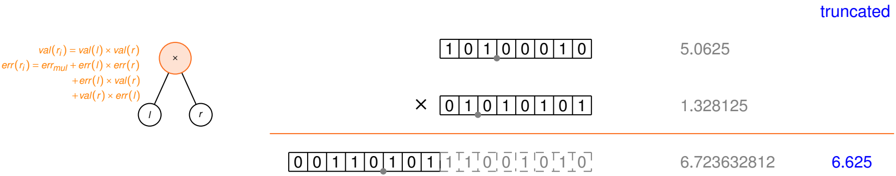
truncated

	1 0 1 0 0 0 1 0	5.0625	
×	0 1 0 1 0 1 0 1	1.328125	
0 0 1 1 0 1 0 1	1 1 0 0 1 0 1 0	6.723632812	6.625

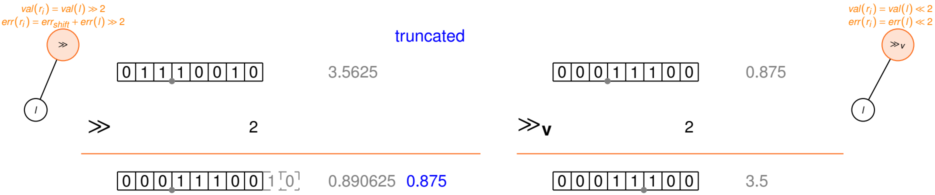
Fixed-point arithmetic model (2/2)

Multiplication:

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



Physical and virtual shifts:



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:

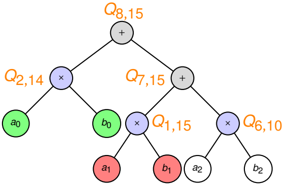
	a_0	b_0	a_1	b_1	a_2	b_2
Value	$[0.1, 1.57]$	$[0, 1.98]$	$[0.01, 0.87]$	$[1.1, 1.86]$	$[0, 15.4]$	$[2, 3.3]$
Fixed-point format	$Q_{1,7}$	$Q_{1,7}$	$Q_{0,8}$	$Q_{1,7}$	$Q_{4,4}$	$Q_{2,6}$

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:

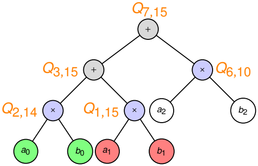
	a_0	b_0	a_1	b_1	a_2	b_2
Value	$[0.1, 1.57]$	$[0, 1.98]$	$[0.01, 0.87]$	$[1.1, 1.86]$	$[0, 15.4]$	$[2, 3.3]$
Fixed-point format	$Q_{1,7}$	$Q_{1,7}$	$Q_{0,8}$	$Q_{1,7}$	$Q_{4,4}$	$Q_{2,6}$

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



$(c_0 + (c_1 + c_2))$

in full precision



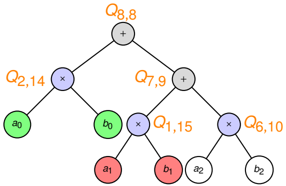
$((c_0 + c_1) + c_2)$

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:

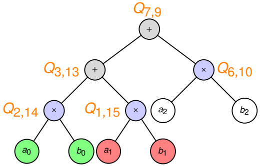
	a_0	b_0	a_1	b_1	a_2	b_2
Value	$[0.1, 1.57]$	$[0, 1.98]$	$[0.01, 0.87]$	$[1.1, 1.86]$	$[0, 15.4]$	$[2, 3.3]$
Fixed-point format	$Q_{1,7}$	$Q_{1,7}$	$Q_{0,8}$	$Q_{1,7}$	$Q_{4,4}$	$Q_{2,6}$

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



$(c_0 + (c_1 + c_2))$

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: $(2n-1)!!$

Dot product size	3	5	10	16	20	···
Number of schemes	3	105	$34459425 \approx 2^{25}$	$6190283353629375 \approx 2^{52}$	$8200794532637891559375 \approx 2^{73}$	···

Remarks

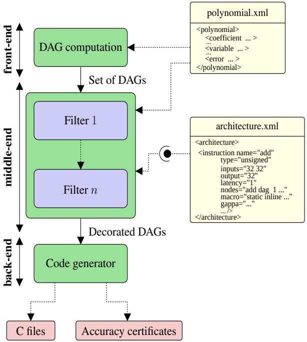
- Picking a scheme that minimizes the evaluation error is one of the difficulties of writing fixed-point code
  - Makes it hard to write fixed-point code by hand
  - Appeals for tools with strong heuristics to automate the process

The CGPE [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

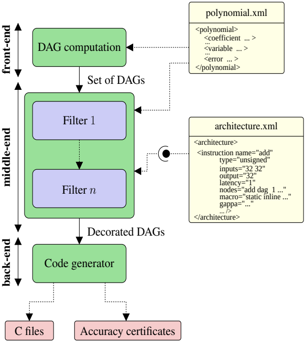


[1] Code Generation for Polynomial Evaluation

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 $\rightsquigarrow$ 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



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

[1] Code Generation for Polynomial Evaluation

Outline of the talk

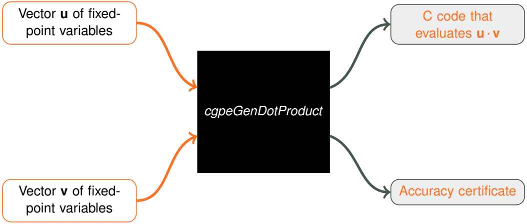
1. Synthesizing fixed-point formulas: combinatorial and numerical issues

2. Efficient matrix multiplication in fixed-point arithmetic

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

Defining the problem

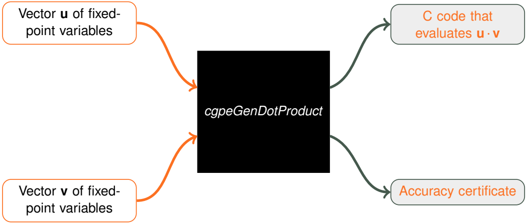
- Inputs:
  - a black box (CGPE) that synthesises code for dot products in fixed-point arithmetic



  - 2 fixed-point matrices A and B

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:

for $1 < i \leq n$ **do**
    for $1 < j \leq n$ **do**
        cgpeGenDotProduct(A_i, B_j);
    end for
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:

compute v such that $v = A_1 \cup A_2 \cup \cdots \cup A_n$
compute w such that $w = B_1 \cup B_2 \cup \cdots \cup B_n$
cgpeGenDotProduct(v, w);

Illustration through a toy example

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

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

Illustration through a toy example

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

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

Accurate product

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

- 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 $DotProduct_{u,v}$: $Q_{26,6}$

- 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?

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

Compact product

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

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?

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

- 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?

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

- 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

Number of vectors	3	5	10	16	20	⋯
Number of schemes	5	52	$115975 \approx 2^{17}$	$10480142147 \approx 2^{33}$	$51724158235372 \approx 2^{46}$	⋯

Distances

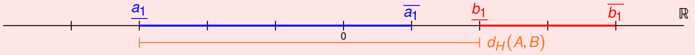
The Hausdorff distance d_H

$$d_H : \mathbb{IR} \times \mathbb{IR} \rightarrow \mathbb{R}$$
$$d_H\left([\underline{a}, \overline{a}], [\underline{b}, \overline{b}]\right) = max\left\{\left|\underline{a} - \underline{b}\right|, \left|\overline{a} - \overline{b}\right|\right\}$$

Example

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

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



Distances

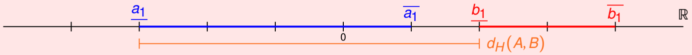
The Hausdorff distance d_H

$$d_H : \mathbb{IR} \times \mathbb{IR} \rightarrow \mathbb{R}$$
$$d_H\left([\underline{a},\overline{a}],[\underline{b},\overline{b}]\right) = max\left\{\left|\underline{a}-\underline{b}\right|,\left|\overline{a}-\overline{b}\right|\right\}$$

Example

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

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

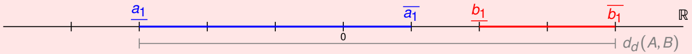


Another possible criterion

$$d_d : \mathbb{IR} \times \mathbb{IR} \rightarrow \mathbb{R}$$
$$d_d\left([\underline{a},\overline{a}],[\underline{b},\overline{b}]\right) = diam\left([\underline{a},\overline{a}] \cup [\underline{b},\overline{b}]\right)$$

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.

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{array}{c} A_0 \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array}$$

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.

$$A = \begin{array}{c} \left(\begin{array}{ccccc} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right) \end{array} \begin{array}{l} A_0 \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array}$$

Closest pair: A_0 and A_3

$$A = \begin{array}{c} \left(\begin{array}{ccccc} a'_{00} & a'_{01} & a'_{02} & a'_{03} & a'_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right) \end{array} \begin{array}{l} A'_0 = A_0 \cup A_3 \\ A_1 \\ A_2 \\ A_4 \end{array}$$

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.

$$A = \begin{array}{cc} \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} & \begin{array}{l} A_0 \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \end{array}$$

Closest pair: A_0 and A_3

$$A = \begin{array}{cc} \begin{pmatrix} a'_{00} & a'_{01} & a'_{02} & a'_{03} & a'_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} & \begin{array}{l} A'_0 = A_0 \cup A_3 \\ A_1 \\ A_2 \\ A_4 \end{array} \end{array}$$

Closest pair: A_1 and A_4

$$A = \begin{array}{cc} \begin{pmatrix} a'_{00} & a'_{01} & a'_{02} & a'_{03} & a'_{04} \\ a'_{10} & a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix} & \begin{array}{l} A'_0 = A_0 \cup A_3 \\ A'_1 = A_1 \cup A_4 \\ A_2 \end{array} \end{array}$$

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 $\mathscr{C}$

Outputs:

`C` code to compute the product AB s.t. $\mathscr{C}$ is satisfied

Steps:

$\mathscr{S}_1 = \{A_1, \ldots, A_n\}$
$\mathscr{S}_2 = \{B_1, \ldots, B_n\}$
while $\mathscr{C}$ is satisfied **do**
    $(u_1, u_2, d_{u_1,u_2}) = \mathit{findClosestPair}(\mathscr{S}_1)$
    $(v_1, v_2, d_{v_1,v_2}) = \mathit{findClosestPair}(\mathscr{S}_2)$
    if $d_{u_1,u_2} \leq d_{v_1,v_2}$ **then**
        $\mathit{remove}(u_1, \mathscr{S}_1); \mathit{remove}(u_2, \mathscr{S}_1); \mathit{insert}(u_1 \cup u_2, \mathscr{S}_1)$
    else
        $\mathit{remove}(v_1, \mathscr{S}_2); \mathit{remove}(v_2, \mathscr{S}_2); \mathit{insert}(v_1 \cup v_2, \mathscr{S}_2)$
    end if
    for $A_i \in \mathscr{S}_1$ **do**
        for $B_j \in \mathscr{S}_2$ **do**
            $\mathit{cgpeGenDotProduct}(A_i, B_j)$
        end for
    end for
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 $\mathscr{C}$: the average error bound is $\leq \epsilon$

Outputs:

C code to compute the product AB s.t. $\mathscr{C}$ is satisfied

Steps:

$\mathscr{S}_1 = \{A_1, \dots, A_n\}$
$\mathscr{S}_2 = \{B_1, \dots, B_n\}$
while average error $\leq \epsilon$ **do**
    $(u_1, u_2, d_{u_1,u_2}) = \mathit{findClosestPair}(\mathscr{S}_1)$
    $(v_1, v_2, d_{v_1,v_2}) = \mathit{findClosestPair}(\mathscr{S}_2)$
    if $d_{u_1,u_2} \leq d_{v_1,v_2}$ **then**
        $\mathit{remove}(u_1, \mathscr{S}_1); \mathit{remove}(u_2, \mathscr{S}_1); \mathit{insert}(u_1 \cup u_2, \mathscr{S}_1)$
    else
        $\mathit{remove}(v_1, \mathscr{S}_2); \mathit{remove}(v_2, \mathscr{S}_2); \mathit{insert}(v_1 \cup v_2, \mathscr{S}_2)$
    end if
    for $A_i \in \mathscr{S}_1$ **do**
        for $B_j \in \mathscr{S}_2$ **do**
            $\mathit{cgpeGenDotProduct}(A_i, B_j)$
        end for
    end for
end while

Benchmarks

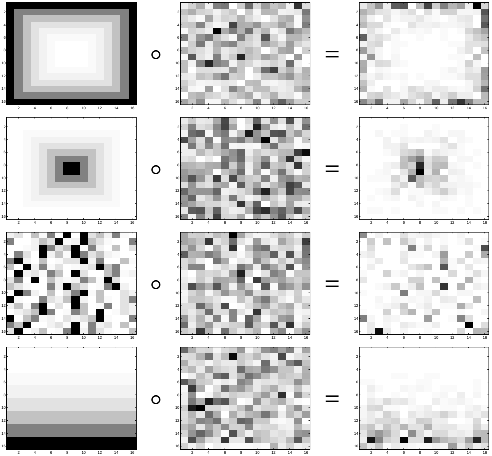
1. Weight matrices with dynamic range $2^{\lfloor \frac{n}{2} \rfloor - 1}$
2. Normally distributed random matrices (generated by matlab)
3. We took the Hadamard product of both matrices H
4. The matrices fed to the algorithm are $midrad(H, 1)$

$W_{i,j} = 2^{max(i,j,n-1-i,n-1-j)-\lfloor n/2 \rfloor}$ — Edges

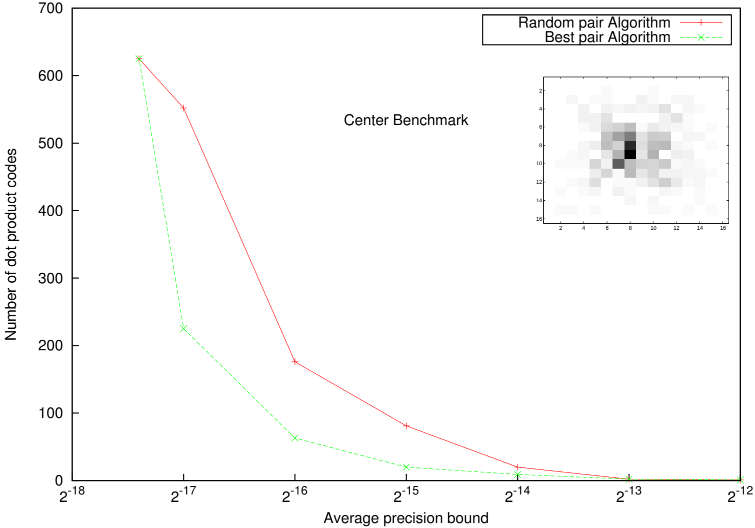
$W_{i,j} = 2^{min(i,j,n-1-i,n-1-j)}$ — Center

$W_{i,j} = 2^{Rand(0,\lfloor n/2 \rfloor - 1)}$ — Random

$W_{i,j} = 2^{\lfloor i/2 \rfloor}$ — Rows

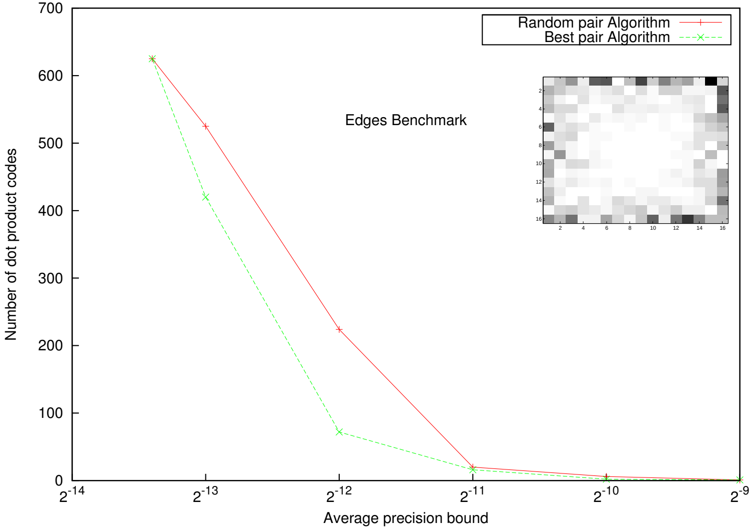


Results



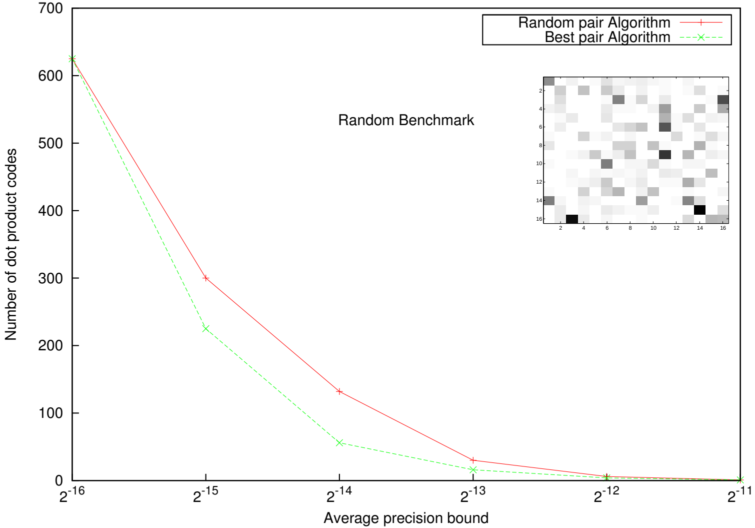
Random pair Algorithm
Best pair Algorithm
Center Benchmark
Number of dot product codes
Average precision bound
700
600
500
400
300
200
100
0
2^{-18}
2^{-17}
2^{-16}
2^{-15}
2^{-14}
2^{-13}
2^{-12}

Results



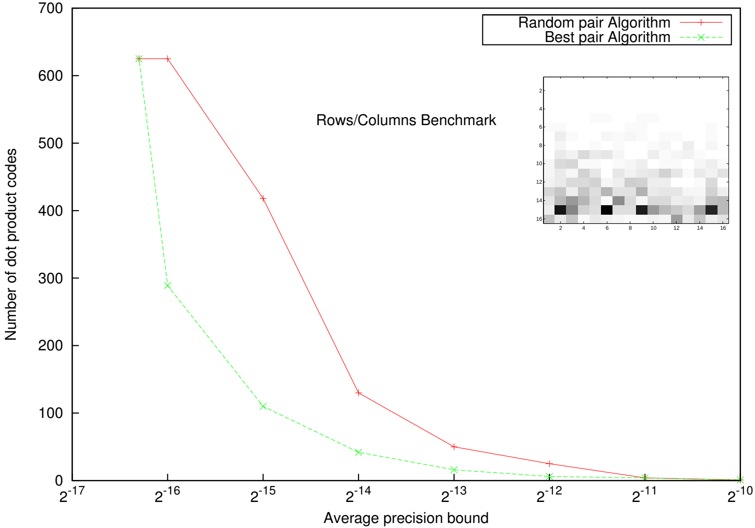
Random pair Algorithm
Best pair Algorithm
Edges Benchmark
Number of dot product codes
Average precision bound
0
100
200
300
400
500
600
700
2^{-14}
2^{-13}
2^{-12}
2^{-11}
2^{-10}
2^{-9}

Results



Random pair Algorithm
Best pair Algorithm
Random Benchmark
Number of dot product codes
Average precision bound
700
600
500
400
300
200
100
0
2^{-16}
2^{-15}
2^{-14}
2^{-13}
2^{-12}
2^{-11}

Results


Random pair Algorithm
Best pair Algorithm
Rows/Columns Benchmark
Number of dot product codes
Average precision bound
700
600
500
400
300
200
100
0
2^-17
2^-16
2^-15
2^-14
2^-13
2^-12
2^-11
2^-10

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

```
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 = %f , im = %f\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];
    }
  }
}
```

1 Initialization

Iterate on number of rows

Assign. the pivot

Norm. pivot's row

Applying to the rest

Copying in Out

The algorithm step by step

Initialization:

$$\left(\begin{array}{cccc|cccc} In_{0,0} & In_{0,1} & \cdots & In_{0,n-1} & 1 & \cdots & \cdots & 0 \\ In_{1,0} & In_{1,1} & \cdots & In_{1,n-1} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & 0 & \vdots & \ddots & 0 \\ In_{n-1,0} & In_{n-1,1} & \cdots & In_{n-1,n-1} & 0 & \cdots & \cdots & 1 \end{array}\right)$$

The algorithm step by step

Step 1: Picking the pivot

$$\left(\begin{array}{cccc|cccc} In_{0,0} & In_{0,1} & \cdots & In_{0,n-1} & 1 & \cdots & \cdots & 0 \\ In_{1,0} & In_{1,1} & \cdots & In_{1,n-1} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & 0 & \vdots & \ddots & 0 \\ In_{n-1,0} & In_{n-1,1} & \cdots & In_{n-1,n-1} & 0 & \cdots & \cdots & 1 \end{array}\right)$$

The algorithm step by step

Step 1: Normalizing the pivot's row

$$\left(\begin{array}{cccc|cccc} \boxed{1} & \frac{In_{0,1}}{In_{0,0}} & \cdots & \frac{In_{0,n-1}}{In_{0,0}} & \frac{1}{In_{0,0}} & \cdots & \cdots & 0 \\ In_{1,0} & In_{1,1} & \cdots & In_{1,n-1} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & 0 & \vdots & \ddots & 0 \\ In_{n-1,0} & In_{n-1,1} & \cdots & In_{n-1,n-1} & 0 & \cdots & \cdots & 1 \end{array}\right)$$

The algorithm step by step

Step 1: Putting zeros in the pivot's column

$$
\left(\begin{array}{ccccc|cccc}
\boxed{1} & \frac{In_{0,1}}{In_{0,0}} & \cdots & \frac{In_{0,n-1}}{In_{0,0}} & & \frac{1}{In_{0,0}} & \cdots & \cdots & 0 \\
0 & In_{1,1} - In_{1,0}\,\frac{In_{0,1}}{In_{0,0}} & \cdots & In_{1,n-1} - In_{1,0}\,\frac{In_{0,n-1}}{In_{0,0}} & & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & & 0 & \vdots & \ddots & 0 \\
0 & In_{n-1,1} - In_{n-1,0}\,\frac{In_{0,1}}{In_{0,0}} & \cdots & In_{n-1,n-1} - In_{n-1,0}\,\frac{In_{0,n-1}}{In_{0,0}} & & 0 & \cdots & \cdots & 1
\end{array}\right)
$$

The algorithm step by step

Step 2: Picking the pivot

$$\left(\begin{array}{ccccc|cccc}
1 & \frac{In_{0,1}}{In_{0,0}} & \cdots & \frac{In_{0,n-1}}{In_{0,0}} & & \frac{1}{In_{0,0}} & \cdots & \cdots & 0 \\
0 & \boxed{In_{1,1} - In_{1,0}\frac{In_{0,1}}{In_{0,0}}} & \cdots & In_{1,n-1} - In_{1,0}\frac{In_{0,n-1}}{In_{0,0}} & & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & & 0 & \vdots & \ddots & 0 \\
0 & In_{n-1,1} - In_{n-1,0}\frac{In_{0,1}}{In_{0,0}} & \cdots & In_{n-1,n-1} - In_{n-1,0}\frac{In_{0,n-1}}{In_{0,0}} & & 0 & \cdots & \cdots & 1
\end{array}\right)$$

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: $A^{-1} = \dfrac{com(A)^T}{det(A)}$
- For a 2×2 matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, $A^{-1} = \dfrac{1}{det(A)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$

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, $det(A) = ad - bc \in [-3, 3] \rightsquigarrow Q_{3,29}$
- Now consider the matrix $M = \begin{pmatrix} 1+2^{-28} & 1 \\ 1 & 1 \end{pmatrix} \rightsquigarrow det(M) = 2^{-28}$
- Then what should be the format of $\dfrac{a}{det(A)}$?
  - 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

- ?

DEFIS WP3 meeting
Perpignan, July 9 th, 2013

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