

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

Amine Najahi

Advisers: M. Martel and G. Revy

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



UPVD
Université Perpignan Via Domitia

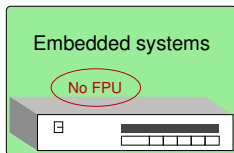


Laboratoire
d'Informatique
de Robotique
et de Microélectronique
de Montpellier



Motivation

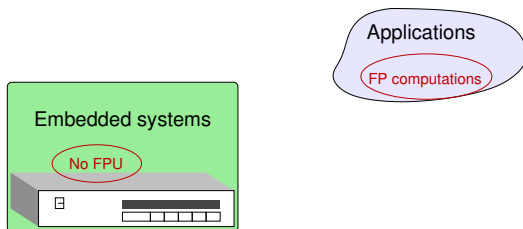
- Embedded systems are ubiquitous
 - ▶ microprocessors and/or DSPs dedicated to one or a few specific tasks
 - ▶ satisfy constraints: area, energy consumption, conception cost
- Some embedded systems **do not have any FPU** (floating-point unit)



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

Motivation

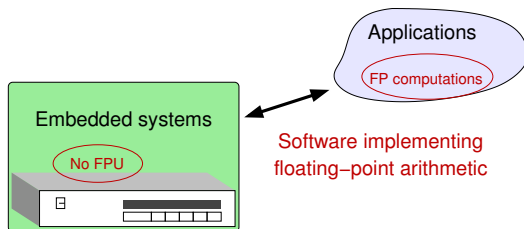
- Embedded systems are ubiquitous
 - ▶ microprocessors and/or DSPs dedicated to one or a few specific tasks
 - ▶ satisfy constraints: area, energy consumption, conception cost
- Some embedded systems **do not have any FPU** (floating-point unit)



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

Motivation

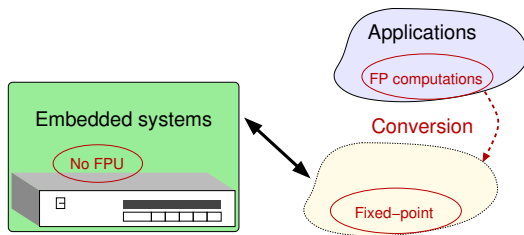
- Embedded systems are ubiquitous
 - ▶ microprocessors and/or DSPs dedicated to one or a few specific tasks
 - ▶ satisfy constraints: area, energy consumption, conception cost
- Some embedded systems **do not have any FPU** (floating-point unit)



- 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. Benchmarks and results

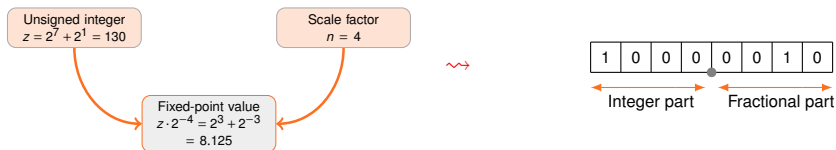
Outline of the talk

1. Synthesizing fixed-point formulas: combinatorial and numerical issues
2. Efficient matrix multiplication in fixed-point arithmetic
3. Benchmarks and results

Background on fixed-point arithmetic

■ Main idea of fixed-point arithmetic:

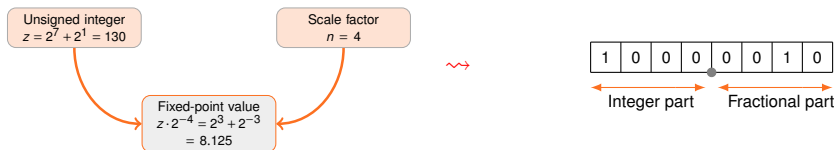
- ▶ Interpret bit packets as integers coupled with a scale factor: $z \cdot 2^{-n}$
- ▶ Example with $z = (1000010)_2$ and $n = 4$



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 $n = 4$

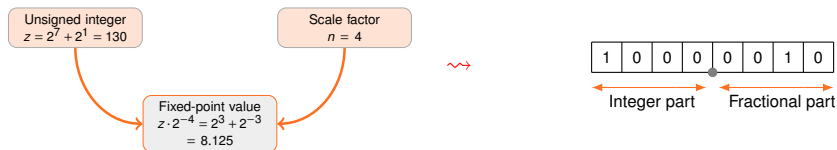


⚠ The scale factor (or fixed-point format) is implicit, only the programmer is aware of it

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 $n = 4$



⚠ 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 **val**(r_i) and **err**(r_i)
- For each basic operator, we have rules to compute **val**(r_i) and **err**(r_i)

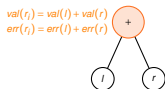
Fixed-point arithmetic model (1/2)

Arithmetic model to track errors in fixed-point computations

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

■ Addition:

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

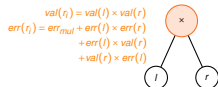


	1 0 1 0 0 0 1 0	5.0625
+	0 1 0 1 0 1 0 1	2.65625
<hr/>		
	1 1 1 1 0 1 1 1	7.7187

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



1 0 1 1 0 0 0 1 0

5.0625

truncated

× 0 1 1 0 1 0 1 0 1

1.328125

0 0 1 1 0 1 0 1 1 1 1 0 0 1 0 1 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

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



1 0 1 1 0 0 0 1 1 0

5.0625

× 0 1 1 0 1 0 1 0 1

1.328125

0 0 1 1 1 0 1 1 0 1 1 1 1 0 0 0 1 1 0 1 1 0

6.723632812

6.625

truncated

Physical and virtual shifts:

$$\begin{aligned} \text{val}(r_i) &= \text{val}(l) \gg 2 \\ \text{err}(r_i) &= \text{err}_{shift} + \text{err}(l) \gg 2 \end{aligned}$$



0 1 1 1 1 0 0 1 1 0

3.5625

»

2

0 0 0 1 1 1 1 0 0 1 1 0

0.890625 0.875

truncated

0 0 0 1 1 1 1 0 0

0.875

»_v

2

0 0 0 1 1 1 1 0 0

3.5

$$\begin{aligned} \text{val}(r_i) &= \text{val}(l) \ll 2 \\ \text{err}(r_i) &= \text{err}(l) \ll 2 \end{aligned}$$



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:

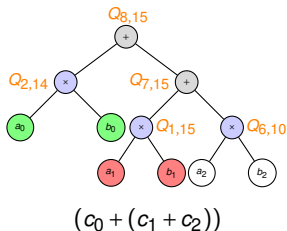
	a_0	b_0	a_1	b_1	a_2	b_2
Value	[0.1, 1.57]	[0, 1.98]	[0.01, 0.87]	[1.1, 1.86]	[0, 15.4]	[2, 3.3]
Fixed-point format	$Q_{1,7}$	$Q_{1,7}$	$Q_{0,8}$	$Q_{1,7}$	$Q_{4,4}$	$Q_{2,6}$

Numerical issues in dot product generation

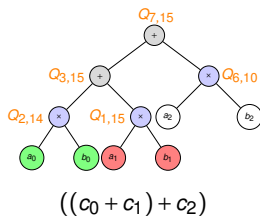
- The building block of matrix multiplication is the dot product operation
 - ▶ Let us consider a size 3 dot product: $(a_0 \times b_0) + (a_1 \times b_1) + (a_2 \times b_2)$ and the following input fixed-point formats:

	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:



in full
precision

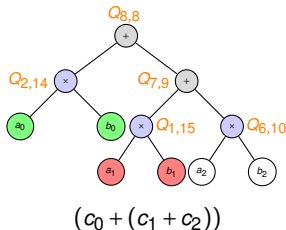


Numerical issues in dot product generation

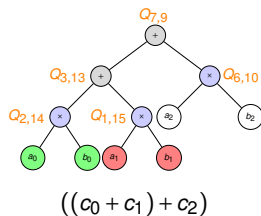
- The building block of matrix multiplication is the dot product operation
 - ▶ Let us consider a size 3 dot product: $(a_0 \times b_0) + (a_1 \times b_1) + (a_2 \times b_2)$ and the following input fixed-point formats:

	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:



with 16 bits
precision



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	$34459425 \approx 2^{25}$	$6190283353629375 \approx 2^{52}$	$8200794532637891559375 \approx 2^{73}$...

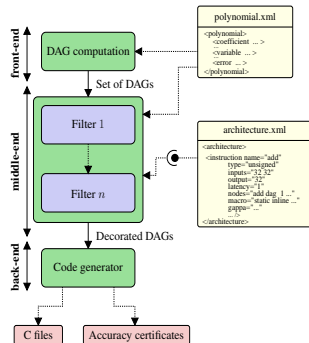
Remarks

- 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 CGPE¹ library

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

- fast \rightsquigarrow selects schemes that reduce the evaluation latency on a given target, by using (as much as possible) the architectural features
- certified \rightsquigarrow produces a bound on the error entailed by the evaluation within the given target's arithmetic

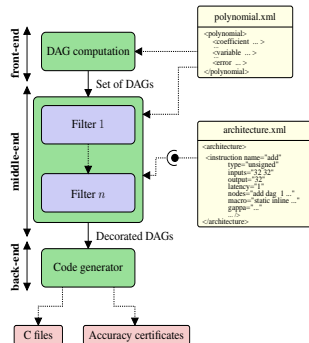


¹Code Generation for Polynomial Evaluation

The CGPE¹ library

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

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

¹Code Generation for Polynomial Evaluation

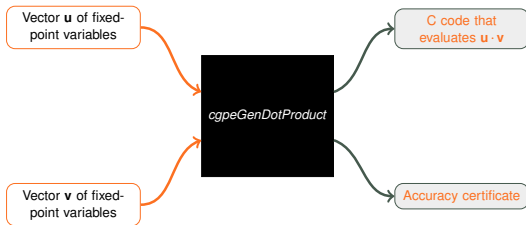
Outline of the talk

1. Synthesizing fixed-point formulas: combinatorial and numerical issues
2. Efficient matrix multiplication in fixed-point arithmetic
3. Benchmarks and results

Defining the problem

■ Inputs:

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

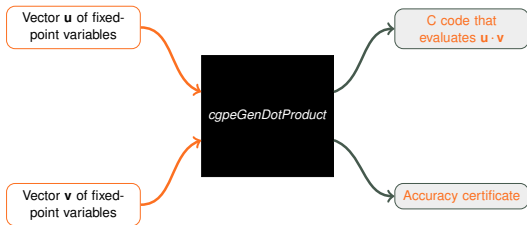


- ▶ 2 fixed-point matrices A and B

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:

- 1: **for** $1 < i \leq n$ **do**
- 2: **for** $1 < j \leq n$ **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 $v = A_1 \cup A_2 \cup \dots \cup A_n$
- 2: compute w such that $w = B_1 \cup B_2 \cup \dots \cup B_n$
- 3: $cgpeGenDotProduct(v, w)$;

Illustration through a toy example

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

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

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 $\text{DotProduct}_{0,0}$: $Q_{26,6}$
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?

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

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

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

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

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

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

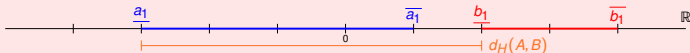
$$d_H([\underline{a}, \bar{a}], [\underline{b}, \bar{b}]) = \max\{|\underline{a} - \underline{b}|, |\bar{a} - \bar{b}|\}$$

Example

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

■ $\cup(A, B) = [-3, 4]$

■ $d_H(A, B) = 5$



Distances

The Hausdorff distance d_H

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

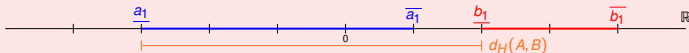
$$d_H([\underline{a}, \bar{a}], [\underline{b}, \bar{b}]) = \max\{|\underline{a} - \underline{b}|, |\bar{a} - \bar{b}|\}$$

Example

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

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

$$\blacksquare d_H(A, B) = 5$$



Another possible criterion

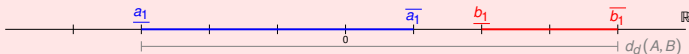
$$d_d : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

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

Example

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

$$\blacksquare d_d(A, B) = 7$$



Closest pair strategy (1/2)

ClosestPairUnion

Inputs:

- n fixed-point vectors v_0, \dots, v_{n-1}
- a routine *findClosestPair*
- a distance d

Outputs:

- $n - 1$ fixed-point vectors

Steps:

- 1: $\mathcal{B} = \{v_0, \dots, v_{n-1}\}$
 - 2: $(u_0, u_1) = \text{findClosestPair}(\mathcal{B}, d)$
 - 3: $\text{remove}(u_0, \mathcal{B})$
 - 4: $\text{remove}(u_1, \mathcal{B})$
 - 5: $\text{add}(\text{Union}(u_0, u_1), \mathcal{B})$
-

Closest pair strategy (1/2)

ClosestPairUnion

Inputs:

- n fixed-point vectors v_0, \dots, v_{n-1}
- a routine *findClosestPair*
- a distance d

Outputs:

- $n - 1$ fixed-point vectors

Steps:

- 1: $\mathcal{B} = \{v_0, \dots, v_{n-1}\}$
 - 2: $(u_0, u_1) = \text{findClosestPair}(\mathcal{B}, d)$
 - 3: $\text{remove}(u_0, \mathcal{B})$
 - 4: $\text{remove}(u_1, \mathcal{B})$
 - 5: $\text{add}(\text{Union}(u_0, u_1), \mathcal{B})$
-

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

Closest pair: A_0 and A_3

Closest pair strategy (1/2)

ClosestPairUnion

Inputs:

- n fixed-point vectors v_0, \dots, v_{n-1}
- a routine *findClosestPair*
- a distance d

Outputs:

- $n - 1$ fixed-point vectors

Steps:

- 1: $\mathcal{B} = \{v_0, \dots, v_{n-1}\}$
 - 2: $(u_0, u_1) = \text{findClosestPair}(\mathcal{B}, d)$
 - 3: $\text{remove}(u_0, \mathcal{B})$
 - 4: $\text{remove}(u_1, \mathcal{B})$
 - 5: $\text{add}(\text{Union}(u_0, u_1), \mathcal{B})$
-

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

Closest pair: A_0 and A_3

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

Closest pair: A_1 and A_4

Closest pair strategy (1/2)

ClosestPairUnion

Inputs:

- n fixed-point vectors v_0, \dots, v_{n-1}
- a routine *findClosestPair*
- a distance d

Outputs:

- $n - 1$ fixed-point vectors

Steps:

- 1: $\mathcal{B} = \{v_0, \dots, v_{n-1}\}$
 - 2: $(u_0, u_1) = \text{findClosestPair}(\mathcal{B}, d)$
 - 3: $\text{remove}(u_0, \mathcal{B})$
 - 4: $\text{remove}(u_1, \mathcal{B})$
 - 5: $\text{add}(\text{Union}(u_0, u_1), \mathcal{B})$
-

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

Closest pair: A_0 and A_3

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

Closest pair: A_1 and A_4

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

Closest pair strategy (2/2)

Closest Pair algorithm

Inputs:

2 fixed-point matrices A and B
 a criterion \mathcal{C}

Outputs:

C code that evaluates $A \cdot B$
 s.t. \mathcal{C} is satisfied

Steps:

```

1: while  $\mathcal{C}$  is satisfied do
2:    $A = \text{ClosestPairUnion}(A)$ 
3:    $B = \text{ClosestPairUnion}(B)$ 
4:   for  $i < \text{numberRows}(A)$  do
5:     for  $j < \text{numberCols}(B)$  do
6:        $\text{cgpeGenDotProduct}(A_i, B_j)$ 
7:     end for
8:   end for
9: end while
  
```

Closest pair strategy (2/2)

Closest Pair algorithm

Inputs:

- 2 fixed-point matrices A and B
- a criterion \mathcal{C} : **average error**

Outputs:

- C code that evaluates $A \cdot B$
- s.t. \mathcal{C} is satisfied

Steps:

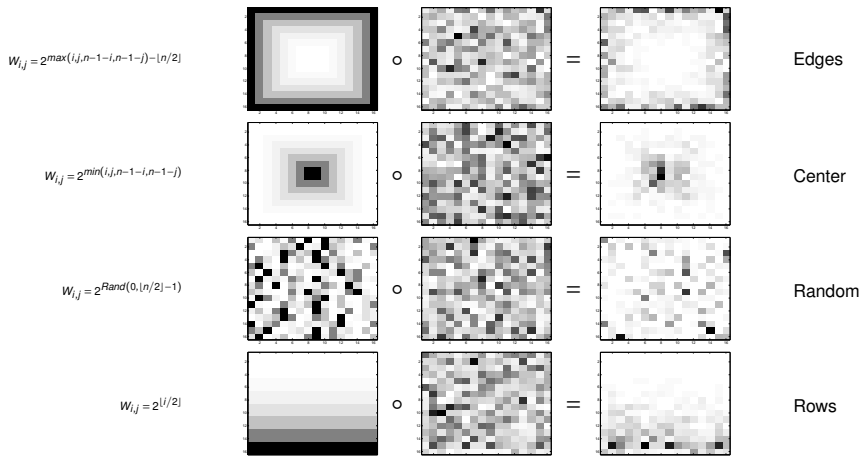
- 1: **while** **average error** $< \epsilon$ **do**
 - 2: $A = \text{ClosestPairUnion}(A)$
 - 3: $B = \text{ClosestPairUnion}(B)$
 - 4: **for** $i < \text{numberRows}(A)$ **do**
 - 5: **for** $j < \text{numberCols}(B)$ **do**
 - 6: $\text{cgpeGenDotProduct}(A_i, B_j)$
 - 7: **end for**
 - 8: **end for**
 - 9: **end while**
-

Outline of the talk

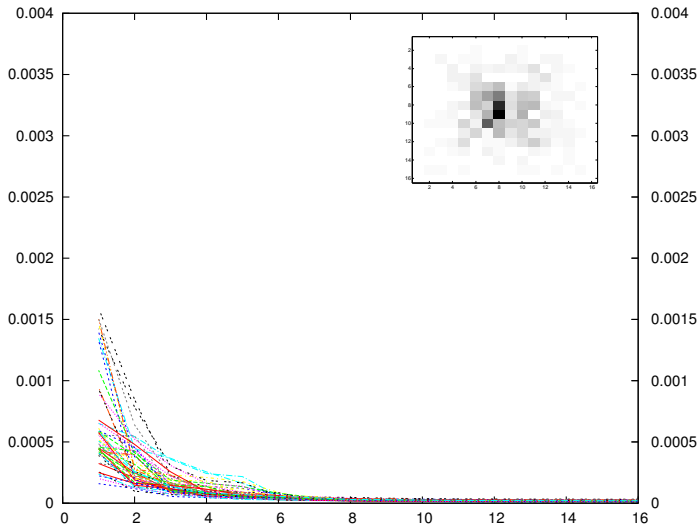
1. Synthesizing fixed-point formulas: combinatorial and numerical issues
2. Efficient matrix multiplication in fixed-point arithmetic
3. Benchmarks and results

Benchmarks

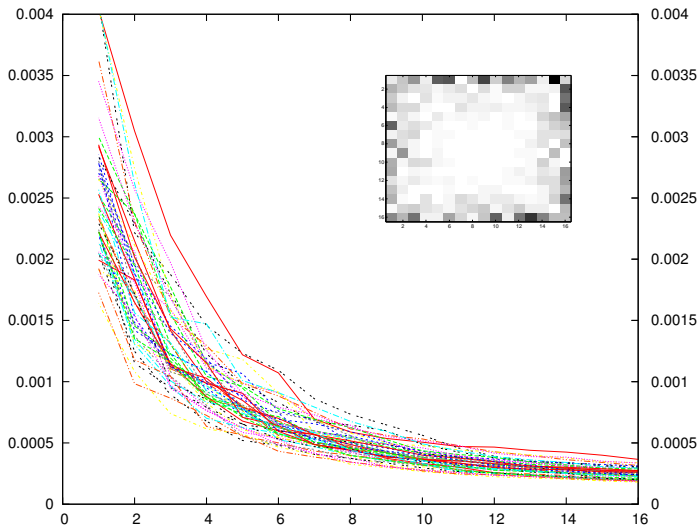
1. Weight matrices with dynamic range $2^{\lfloor \frac{n}{2} \rfloor - 1}$
2. Normally distributed random matrices (generated by matlab)
3. We took the Hadamard product of both matrices H
4. The matrices fed to the algorithm are $midrad(H, 1)$



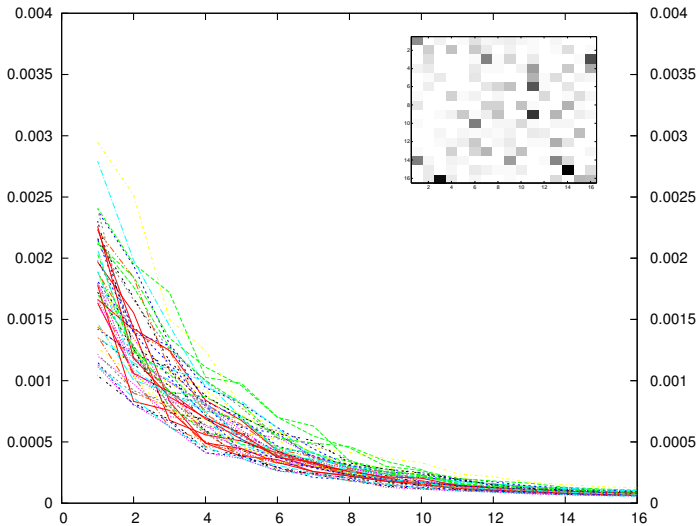
Results



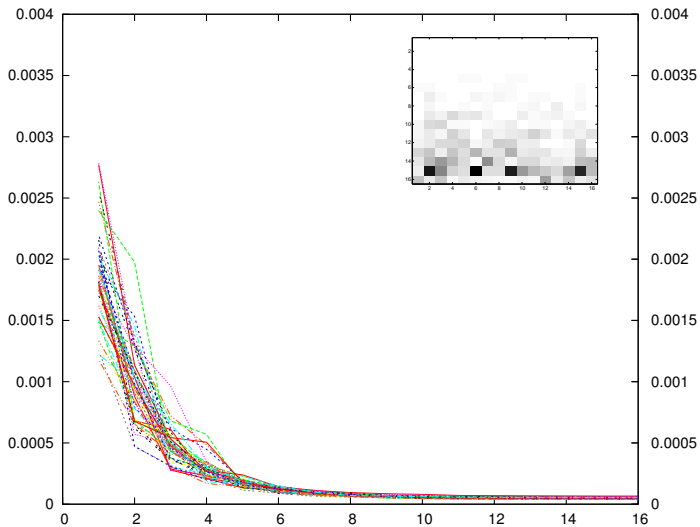
Results



Results



Results



Demo

Conclusion

■ Contributions:

- ▶ We suggested heuristics based on distances between rows/columns to tackle the code size/accuracy trade-off
- ▶ We have implemented these algorithms in the FPLA² tool
- ▶ This tool relies heavily on previous work implemented in the CGPE library

²Fixed-Point Linear Algebra

Conclusion

■ Contributions:

- ▶ We suggested heuristics based on distances between rows/columns to tackle the code size/accuracy trade-off
- ▶ We have implemented these algorithms in the FPLA² tool
- ▶ This tool relies heavily on previous work implemented in the CGPE library

■ Future work:

- ▶ Work toward handling code generation for 2-D convolution and matrix inversion
- ▶ Investigate the generation of VHDL instead of C code
- ▶ Better handling of structured matrices

²Fixed-Point Linear Algebra

Paris, May 30th, 2013

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

Amine Najahi

Advisers: M. Martel and G. Revy

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



UPVD
Université Perpignan Via Domitia



Laboratoire
d'Informatique
de Robotique
et de Microélectronique
de Montpellier

