Synthesis of fixed-point programs: the case of matrix multiplication

Mohamed Amine Najahi

13th École Jeunes Chercheurs en Informatique Mathématique (EJCIM 2013)
Perpignan, April 12th, 2013

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

How easy it is to program a product of matrices?

How easy it is to program a product of matrices?

Well, in floating-point, it is very easy !!

```
#define N 80
int
main()
{
  int i,j,k;
  float A[N][N]={...};
  float B[N][N]={...};
  float C[N][N]={0,...,0};
  for (i = 0; i < N ; i++)
    for (j = 0; j < N ; j++)
      for (k = 0; k < N ; k++)      /* dot product of row i and column j */
        C[i][j]+=A[i][k]*B[k][j];
  return 0;
}
```

How easy it is to program a product of matrices?

Well, in floating-point, it is very easy !!

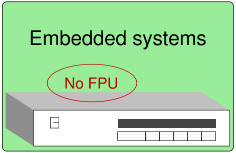
```
#define N 80
int
main()
{
  int i,j,k;
  float A[N][N]={...};
  float B[N][N]={...};
  float C[N][N]={0,...,0};
  for (i = 0; i < N ; i++)
    for (j = 0; j < N ; j++)
      for (k = 0; k < N ; k++)       /* dot product of row i and column j */
        C[i][j]+=A[i][k]*B[k][j];
  return 0;
}
```

But, what if the target does not have a floating-point unit?

Motivation

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

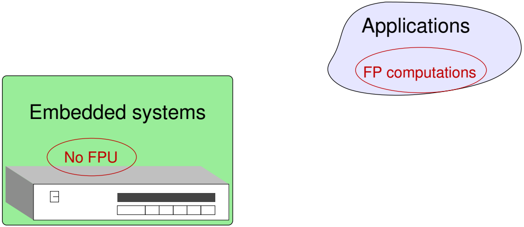
- Some embedded systems do not have any FPU (floating-point unit)



- 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)

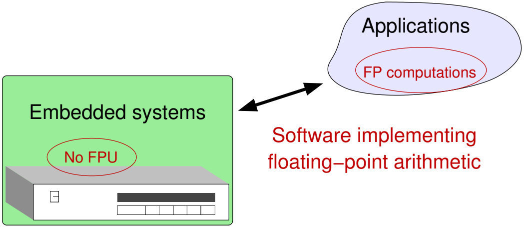


- Highly used in audio and video applications
  - demanding on floating-point computations

Motivation

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

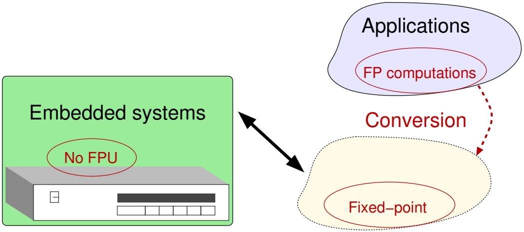
- Some embedded systems do not have any FPU (floating-point unit)



- 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)



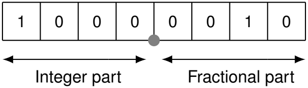
- Float to Fix conversion is tackled by the ANR project DEFIS
  - LIP6, IRISA, CEA, LIRMM, THALES and INPIXAL

Outline of the talk

Outline of the talk

Principles of fixed-point arithmetic

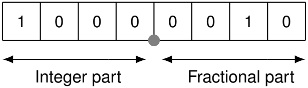
- Main idea of fixed-point arithmetic:
  - interpret bit words as integers coupled with a scale factor: $\frac{z}{2^n}$



z	$2^7 + 2^1 = 130$
Value in fixed-point	$\frac{130}{2^4} = \frac{2^7+2^1}{2^4} = 2^3 + 2^{-3} = 8.125$

Principles of fixed-point arithmetic

- Main idea of fixed-point arithmetic:
  - interpret bit words as integers coupled with a scale factor: $\frac{z}{2^n}$

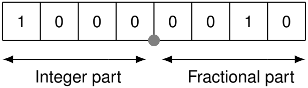


z	$2^7 + 2^1 = 130$
Value in fixed-point	$\frac{130}{2^4} = \frac{2^7+2^1}{2^4} = 2^3 + 2^{-3} = 8.125$

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

Principles of fixed-point arithmetic

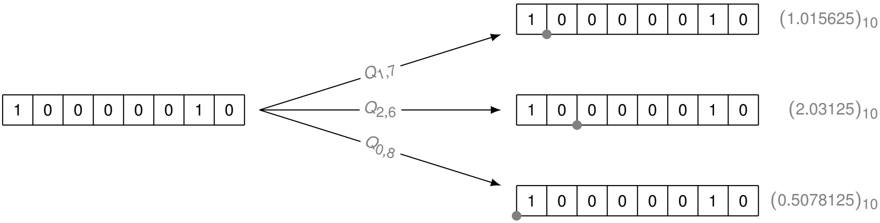
- Main idea of fixed-point arithmetic:
  - interpret bit words as integers coupled with a scale factor: $\frac{z}{2^n}$



z	$2^7 + 2^1 = 130$
Value in fixed-point	$\frac{130}{2^4} = \frac{2^7+2^1}{2^4} = 2^3 + 2^{-3} = 8.125$

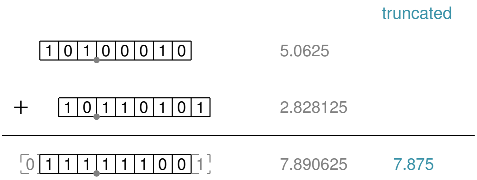
⚠ The scale factor (or fixed-point format) is implicit, only the programmer is aware of it

- Let us denote by $Q_{a,b}$ a fixed-point format with a integer bits and b fractional bits



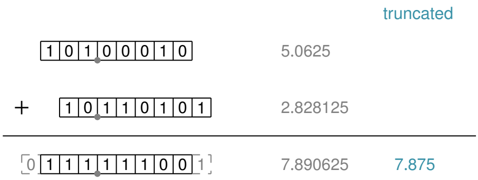
Basic fixed-point operators

- Addition
  - The two variables have to be in the same fixed-point format
  - The sum of two $Q_{a,b}$ variables yields a $Q_{a+1,b}$ variable

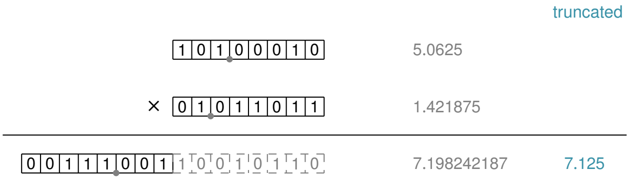


Basic fixed-point operators

- Addition
  - The two variables have to be in the same fixed-point format
  - The sum of two $Q_{a,b}$ variables yields a $Q_{a+1,b}$ variable



- Multiplication
  - No need for the two variables to have the same fixed-point format
  - The product of a $Q_{a,b}$ variable by a $Q_{c,d}$ variable yields a $Q_{a+c,b+d}$ variable



First example: a size 3 dot product

- Let us consider the arithmetic expression: $(a_0 \times b_0) + (a_1 \times b_1) + (a_2 \times b_2)$ and the following input fixed-point formats:

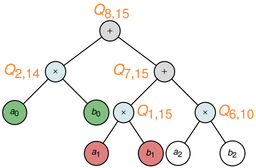
	a_0	b_0	a_1	b_1	a_2	b_2
Value	$[0.1, 1.57]$	$[0, 1.98]$	$[0.01, 0.87]$	$[1.1, 1.86]$	$[0, 15.4]$	$[2, 3.3]$
Fixed-point format	$Q_{1,7}$	$Q_{1,7}$	$Q_{0,8}$	$Q_{1,7}$	$Q_{4,4}$	$Q_{2,6}$

First example: a size 3 dot product

- Let us consider the arithmetic expression: $(a_0 \times b_0) + (a_1 \times b_1) + (a_2 \times b_2)$ and the following input fixed-point formats:

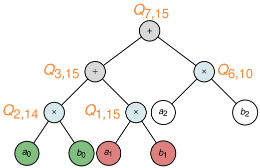
	a_0	b_0	a_1	b_1	a_2	b_2
Value	$[0.1, 1.57]$	$[0, 1.98]$	$[0.01, 0.87]$	$[1.1, 1.86]$	$[0, 15.4]$	$[2, 3.3]$
Fixed-point format	$Q_{1,7}$	$Q_{1,7}$	$Q_{0,8}$	$Q_{1,7}$	$Q_{4,4}$	$Q_{2,6}$

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



$(c_0 + (c_1 + c_2))$

in full precision



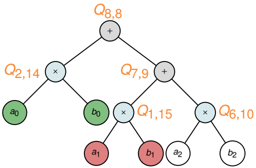
$((c_0 + c_1) + c_2)$

First example: a size 3 dot product

- Let us consider the arithmetic expression: $(a_0 \times b_0) + (a_1 \times b_1) + (a_2 \times b_2)$ and the following input fixed-point formats:

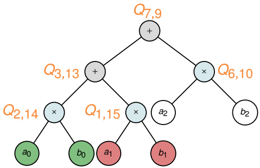
	a_0	b_0	a_1	b_1	a_2	b_2
Value	$[0.1, 1.57]$	$[0, 1.98]$	$[0.01, 0.87]$	$[1.1, 1.86]$	$[0, 15.4]$	$[2, 3.3]$
Fixed-point format	$Q_{1,7}$	$Q_{1,7}$	$Q_{0,8}$	$Q_{1,7}$	$Q_{4,4}$	$Q_{2,6}$

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



$(c_0 + (c_1 + c_2))$

with 16 bits precision



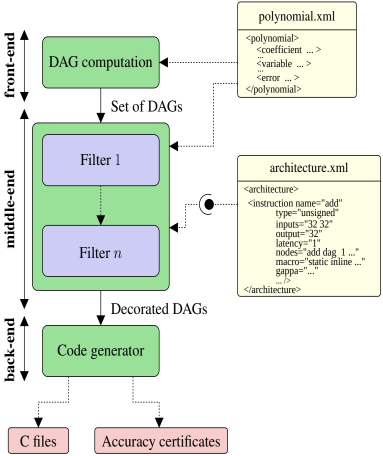
$((c_0 + c_1) + c_2)$

The CGPE [1] software tool

- Written by Revy and Mouilleron to aid in emulating floating-point in software
- A tool that generates fast and certified code

- fast ⇝ that reduce the evaluation latency on a given target, by using the target architecture features (as much as possible)

- certified ⇝ for which we can bound the error entailed by the evaluation within the given target's arithmetic

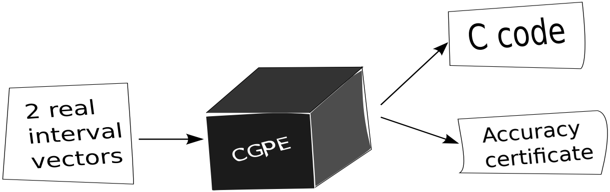


[1] Code Generation for Polynomial Evaluation

Outline of the talk

Defining the problem

- We are provided with
  - a black box (CGPE) that synthesises code for dot-products in fixed-point arithmetic

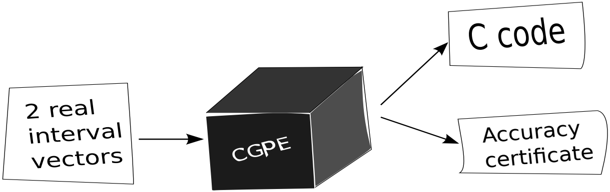


  - 2 matrices A and B in $I(\mathbb{R}^{n\times n})$

$$A = \begin{pmatrix} [-4.54, 7.78] & \cdots & [-0.789, 0.967] \\ \vdots & \ddots & \vdots \\ [12.51, 24.14] & \cdots & [-0.921, 0.791] \end{pmatrix} \quad \text{and,} \quad B = \begin{pmatrix} [-64, 45.78] & \cdots & [-0.287, 0.7] \\ \vdots & \ddots & \vdots \\ [125.1, 245.14] & \cdots & [-5.74, 7.32] \end{pmatrix}$$

Defining the problem

- We are provided with
  - a black box (CGPE) that synthesises code for dot-products in fixed-point arithmetic



  - 2 matrices A and B in $I(\mathbb{R}^{n\times n})$

$$A = \begin{pmatrix} [-4.54, 7.78] & \cdots & [-0.789, 0.967] \\ \vdots & \ddots & \vdots \\ [12.51, 24.14] & \cdots & [-0.921, 0.791] \end{pmatrix} \quad \text{and,} \quad B = \begin{pmatrix} [-64, 45.78] & \cdots & [-0.287, 0.7] \\ \vdots & \ddots & \vdots \\ [125.1, 245.14] & \cdots & [-5.74, 7.32] \end{pmatrix}$$

- We are asked to
  - Generate code that evaluates all the products $C = MN$ in fixed-point arithmetic
    - where $M \in A$ and $N \in B$

Tradeoffs to consider

- Remark: The suggested strategy should be efficient in terms of the tradeoffs below for all matrices of size smaller than 80×80

Tradeoffs to consider

- Remark: The suggested strategy should be efficient in terms of the tradeoffs below for all matrices of size smaller than 80×80

1. Size of the generated code
   - We are targeting embedded systems $\rightsquigarrow$ code size should be as tight as possible

Tradeoffs to consider

- Remark: The suggested strategy should be efficient in terms of the tradeoffs below for all matrices of size smaller than 80×80

1. Size of the generated code
   - We are targeting embedded systems $\rightsquigarrow$ code size should be as tight as possible
2. Accuracy of the generated code
   - Accuracy certificates should be produced that bound the absolute error
   - The guaranteed absolute error should be as tight as possible

Tradeoffs to consider

- Remark: The suggested strategy should be efficient in terms of the tradeoffs below for all matrices of size smaller than 80×80

1. Size of the generated code
   - We are targeting embedded systems $\rightsquigarrow$ code size should be as tight as possible
2. Accuracy of the generated code
   - Accuracy certificates should be produced that bound the absolute error
   - The guaranteed absolute error should be as tight as possible
3. Speed of generation

An accurate algorithm

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

---

AccurateProduct

Inputs:

Two square matrices $A \in I(\mathbb{R}^{n \times n})$ and $B \in I(\mathbb{R}^{n \times n})$

Outputs:

C code to compute the product MN for all $M \in A$ and $N \in B$

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
```

An accurate algorithm

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

AccurateProduct

Inputs:

Two square matrices $A \in I(\mathbb{R}^{n\times n})$ and $B \in I(\mathbb{R}^{n\times n})$

Outputs:

C code to compute the product MN for all $M \in A$ and $N \in B$

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**

Illustration on the product of two 2×2 matrices

$$C = \begin{pmatrix} C_{1,1} = cgpeGenDotProduct(A_1, B_1) & C_{1,2} = cgpeGenDotProduct(A_1, B_2) \\ C_{2,1} = cgpeGenDotProduct(A_2, B_1) & C_{2,2} = cgpeGenDotProduct(A_2, B_2) \end{pmatrix}$$

Analysis of AccurateProduct

- For square matrices of size n, n^2 calls to the cgpeGenDotProduct are issued
  - Each dot product uses more than $2n$ instructions (n multiplications + n additions)
    - ↪ The generated code for the product is proportional in size to $2n^3$

↪ More than 1 024 000 instructions for 80×80 matrices

Analysis of AccurateProduct

- For square matrices of size n, n^2 calls to the cgpeGenDotProduct are issued
  - Each dot product uses more than $2n$ instructions (n multiplications + n additions)
    - ↪ The generated code for the product is proportional in size to $2n^3$

↪ More than 1 024 000 instructions for 80×80 matrices

Advantages

- ✓ Easy to generate code
  - ✓ Two nested loops and n^2 calls to the routine cgpeGenDotProduct
- ✓ The reference in terms of numerical quality

Drawbacks

- ✗ Code size is proportional to $2n^3$
  - ✗ Similar code sizes are prohibitive in embedded systems

A compact algorithm

- Main idea: Generate a unique dot product code for all the computations

---

CompactProduct

Inputs:

Two square matrices $A \in I(\mathbb{R}^{n\times n})$ and $B \in I(\mathbb{R}^{n\times n})$

Outputs:

C code to compute the product MN for all $M \in A$ and $N \in B$

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);

A compact algorithm

- Main idea: Generate a unique dot product code for all the computations

---

CompactProduct

Inputs:

Two square matrices $A \in I(\mathbb{R}^{n\times n})$ and $B \in I(\mathbb{R}^{n\times n})$

Outputs:

C code to compute the product MN for all $M \in A$ and $N \in B$

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 on the product of two 2×2 matrices

$$C = \begin{pmatrix} C_{1,1} = cgpeGenDotProduct(A_1 \cup A_2, B_1 \cup B_2) & C_{1,2} = cgpeGenDotProduct(A_1 \cup A_2, B_1 \cup B_2) \\ C_{2,1} = cgpeGenDotProduct(A_1 \cup A_2, B_1 \cup B_2) & C_{2,2} = cgpeGenDotProduct(A_1 \cup A_2, B_1 \cup B_2) \end{pmatrix}$$

Analysis of CompactProduct

- For square matrices of size n, only one call to the cgpeGenDotProduct is issued
  - The dot product uses around $2n$ instructions (n multiplications + n additions)
    - ↪ The generated code for the product is proportional in size to $2n$

↪ Around 160 instructions for 80×80 matrices

Analysis of CompactProduct

- For square matrices of size n, only one call to the cgpeGenDotProduct is issued
  - The dot product uses around $2n$ instructions (n multiplications + n additions)
    - ↪ The generated code for the product is proportional in size to $2n$

↪ Around 160 instructions for 80×80 matrices

Advantages

✓ Easy to generate code

  ✓ Compute the union of all vectors of A and B and call the routine cgpeGenDotProduct

✓ The reference in terms of code size

Drawbacks

✗ Numerical quality deteriorates dramatically

A closest pair algorithm

- Main idea: Fuse together only rows or columns that are close to each other

The Hausdorff distance d_H

$$d_H : I(\mathbb{R}^n) \times I(\mathbb{R}^n) \rightarrow \mathbb{R}$$
$$d_H(A,B) = \max_{1 \leq i \leq n} max \left\{ \left| \underline{a_i} - \underline{b_i} \right|, \left| \overline{a_i} - \overline{b_i} \right| \right\}$$

A closest pair algorithm

- Main idea: Fuse together only rows or columns that are close to each other

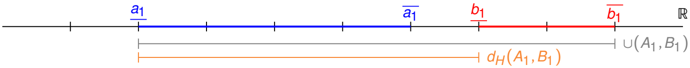
The Hausdorff distance d_H

$$d_H : I(\mathbb{R}^n) \times I(\mathbb{R}^n) \to \mathbb{R}$$
$$d_H(A,B) = \max_{1 \le i \le n} max \left\{ \left| \underline{a_i} - \underline{b_i} \right|, \left| \overline{a_i} - \overline{b_i} \right| \right\}$$

Example

Let $A = \begin{pmatrix} [-4,7] & [-11,102] \end{pmatrix}$ and $B = \begin{pmatrix} [-2,88] & [-23,1] \end{pmatrix}$ be two vectors in $I(\mathbb{R}^2)$, we have:

- $d_H(A,B) = 101$
- $\cup(A,B) = \begin{pmatrix} [-4,88] & [-23,102] \end{pmatrix}$



ClosestPairFusion

ClosestPairFusion

Inputs:

n vectors, $v_1, \ldots, v_n$ in $I(\mathbb{R}^m)$
a routine findClosestPair based on d_H
a routine Union that applies the union operator
the number k of output vectors

Outputs:

k vectors in $I(\mathbb{R}^m)$

Steps:

1: $\mathscr{B} = \{v_1, \ldots, v_n\}$
2: **while** $size(\mathscr{B}) > k$ **do**
3: $\quad (u_1, u_2) = findClosestPair(\mathscr{B})$
4: $\quad remove(u_1, \mathscr{B})$
5: $\quad remove(u_2, \mathscr{B})$
6: $\quad add(Union(u_1, u_2), \mathscr{B})$
7: **end while**

Illustration of the ClosestPairFusion

$$\begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix} \begin{pmatrix} [-4,4] & [-5,5] & [-5,5] & [-6,6] \\ [-2,2] & [-1,1] & [-3,3] & [-9,9] \\ [-7,7] & [-4,4] & [-12,12] & [-11,11] \\ [-8,8] & [-1,1] & [-10,10] & [-9,9] \end{pmatrix}$$

$$\begin{matrix} w_1 & w_2 & w_3 & w_4 \end{matrix}$$
$$\begin{pmatrix} [-3,3] & [-14,14] & [-5,5] & [-6,6] \\ [-1,1] & [-11,11] & [-3,3] & [-9,9] \\ [-4,4] & [-8,8] & [-11,11] & [-1,1] \\ [-9,9] & [-7,7] & [-10,10] & [-2,2] \end{pmatrix}$$

$d_H(v_1,v_2)$	$d_H(v_1,v_3)$	$d_H(v_1,v_4)$	$d_H(v_2,v_3)$	$d_H(v_2,v_4)$	$d_H(v_3,v_4)$
4	7	5	9	7	3

$d_H(w_1,w_2)$	$d_H(w_1,w_3)$	$d_H(w_1,w_4)$	$d_H(w_2,w_3)$	$d_H(w_2,w_4)$	$d_H(w_3,w_4)$
11	7	8	9	8	10

Illustration of the ClosestPairFusion

$$\begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix} \begin{pmatrix} [-4,4] & [-5,5] & [-5,5] & [-6,6] \\ [-2,2] & [-1,1] & [-3,3] & [-9,9] \\ [-7,7] & [-4,4] & [-12,12] & [-11,11] \\ [-8,8] & [-1,1] & [-10,10] & [-9,9] \end{pmatrix}$$

$$\begin{matrix} w_1 & w_2 & w_3 & w_4 \end{matrix}$$
$$\begin{pmatrix} [-3,3] & [-14,14] & [-5,5] & [-6,6] \\ [-1,1] & [-11,11] & [-3,3] & [-9,9] \\ [-4,4] & [-8,8] & [-11,11] & [-1,1] \\ [-9,9] & [-7,7] & [-10,10] & [-2,2] \end{pmatrix}$$

$d_H(v_1,v_2)$	$d_H(v_1,v_3)$	$d_H(v_1,v_4)$	$d_H(v_2,v_3)$	$d_H(v_2,v_4)$	$d_H(v_3,v_4)$
4	7	5	9	7	3

$d_H(w_1,w_2)$	$d_H(w_1,w_3)$	$d_H(w_1,w_4)$	$d_H(w_2,w_3)$	$d_H(w_2,w_4)$	$d_H(w_3,w_4)$
11	7	8	9	8	10

$$\begin{matrix} v_1 \\ v_2 \\ v_3 \cup v_4 \end{matrix} \begin{pmatrix} [-4,4] & [-5,5] & [-5,5] & [-6,6] \\ [-2,2] & [-1,1] & [-3,3] & [-9,9] \\ [-8,8] & [-4,4] & [-12,12] & [-11,11] \end{pmatrix}$$

$$\begin{matrix} w_1 \cup w_3 & w_2 & w_4 \end{matrix}$$
$$\begin{pmatrix} [-5,5] & [-14,14] & [-6,6] \\ [-3,3] & [-11,11] & [-9,9] \\ [-11,11] & [-8,8] & [-1,1] \\ [-10,10] & [-7,7] & [-2,2] \end{pmatrix}$$

$d_H(v_1,v_2)$	$d_H(v_1,v_3 \cup v_4)$	$d_H(v_2,v_3 \cup v_4)$
4	7	9

$d_H(w_1 \cup w_3,w_2)$	$d_H(w_1 \cup w_3,w_4)$	$d_H(w_2,w_4)$
9	10	8

Illustration of the ClosestPairFusion

$$\begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix} \begin{pmatrix} [-4,4] & [-5,5] & [-5,5] & [-6,6] \\ [-2,2] & [-1,1] & [-3,3] & [-9,9] \\ [-7,7] & [-4,4] & [-12,12] & [-11,11] \\ [-8,8] & [-1,1] & [-10,10] & [-9,9] \end{pmatrix}$$

$d_H(v_1,v_2)$	$d_H(v_1,v_3)$	$d_H(v_1,v_4)$	$d_H(v_2,v_3)$	$d_H(v_2,v_4)$	$d_H(v_3,v_4)$
4	7	5	9	7	3

$$\begin{matrix} v_1 \\ v_2 \\ v_3 \cup v_4 \end{matrix} \begin{pmatrix} [-4,4] & [-5,5] & [-5,5] & [-6,6] \\ [-2,2] & [-1,1] & [-3,3] & [-9,9] \\ [-8,8] & [-4,4] & [-12,12] & [-11,11] \end{pmatrix}$$

$d_H(v_1,v_2)$	$d_H(v_1,v_3 \cup v_4)$	$d_H(v_2,v_3 \cup v_4)$
4	7	9

$$\begin{matrix} v_1 \cup v_2 \\ v_3 \cup v_4 \end{matrix} \begin{pmatrix} [-4,4] & [-5,5] & [-5,5] & [-9,9] \\ [-8,8] & [-4,4] & [-12,12] & [-11,11] \end{pmatrix}$$

$$\begin{matrix} w_1 & w_2 & w_3 & w_4 \end{matrix}$$
$$\begin{pmatrix} [-3,3] & [-14,14] & [-5,5] & [-6,6] \\ [-1,1] & [-11,11] & [-3,3] & [-9,9] \\ [-4,4] & [-8,8] & [-11,11] & [-1,1] \\ [-9,9] & [-7,7] & [-10,10] & [-2,2] \end{pmatrix}$$

$d_H(w_1,w_2)$	$d_H(w_1,w_3)$	$d_H(w_1,w_4)$	$d_H(w_2,w_3)$	$d_H(w_2,w_4)$	$d_H(w_3,w_4)$
11	7	8	9	8	10

$$\begin{matrix} w_1 \cup w_3 & w_2 & w_4 \end{matrix}$$
$$\begin{pmatrix} [-5,5] & [-14,14] & [-6,6] \\ [-3,3] & [-11,11] & [-9,9] \\ [-11,11] & [-8,8] & [-1,1] \\ [-10,10] & [-7,7] & [-2,2] \end{pmatrix}$$

$d_H(w_1 \cup w_3,w_2)$	$d_H(w_1 \cup w_3,w_4)$	$d_H(w_2,w_4)$
9	10	8

$$\begin{matrix} w_1 \cup w_3 & w_2 \cup w_4 \end{matrix}$$
$$\begin{pmatrix} [-5,5] & [-14,14] \\ [-3,3] & [-11,11] \\ [-11,11] & [-8,8] \\ [-10,10] & [-7,7] \end{pmatrix}$$

Analysis of the closest pair algorithm

- For square matrices of size n, $k \times l$ calls to the cgpeGenDotProduct are issued
  - Each dot product uses more than $2n$ instructions (n multiplications + n additions)
    - ↪ The generated code for the product is proportional in size to $2nkl$

↪ For 80×80 matrices, the table below gives the number of instructions

k \ l	1	2	4	5	8	10	16	20	40	80
1	160	320	640	800	1280	1600	2560	3200	6400	12800
2	320	640	1280	1600	2560	3200	5120	6400	12800	25600
4	640	1280	2560	3200	5120	6400	10240	12800	25600	51200
5	800	1600	3200	4000	6400	8000	12800	16000	32000	64000
8	1280	2560	5120	6400	10240	12800	20480	25600	51200	102400
10	1600	3200	6400	8000	12800	16000	25600	32000	64000	128000
16	2560	5120	10240	12800	20480	25600	40960	51200	102400	204800
20	3200	6400	12800	16000	25600	32000	51200	64000	128000	256000
40	6400	12800	25600	32000	51200	64000	102400	128000	256000	512000
80	12800	25600	51200	64000	102400	128000	204800	256000	512000	1024000

Advantages

✓ Code size can be controlled through the parameters k and l

Drawbacks

✗ Numerical quality deteriorates with small values of k and l

Analysis of the closest pair algorithm

- For square matrices of size n, $k \times l$ calls to the cgpeGenDotProduct are issued
  - Each dot product uses more than $2n$ instructions (n multiplications + n additions)
    - ↪ The generated code for the product is proportional in size to $2nkl$

↪ For 80×80 matrices, the table below gives the number of instructions

k \ l	1	2	4	5	8	10	16	20	40	80
1	160	320	640	800	1280	1600	2560	3200	6400	12800
2	320	640	1280	1600	2560	3200	5120	6400	12800	25600
4	640	1280	2560	3200	5120	6400	10240	12800	25600	51200
5	800	1600	3200	4000	6400	8000	12800	16000	32000	64000
8	1280	2560	5120	6400	10240	12800	20480	25600	51200	102400
10	1600	3200	6400	8000	12800	16000	25600	32000	64000	128000
16	2560	5120	10240	12800	20480	25600	40960	51200	102400	204800
20	3200	6400	12800	16000	25600	32000	51200	64000	128000	256000
40	6400	12800	25600	32000	51200	64000	102400	128000	256000	512000
80	12800	25600	51200	64000	102400	128000	204800	256000	512000	1024000

Advantages

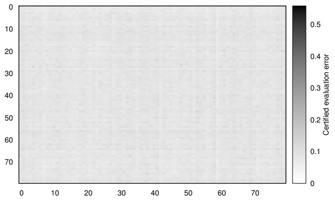
✓ Code size can be controlled through the parameters k and l

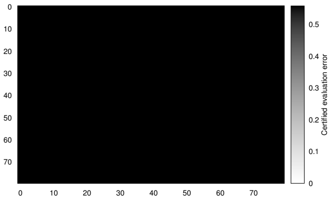
Drawbacks

✗ Numerical quality deteriorates with small values of k and l

Let us compare these algorithms

- These results were produced for interval matrices of size 80×80
  - The center of each interval is randomly selected in $[-1000, 1000]$
  - The diameter of the intervals is fixed to 100





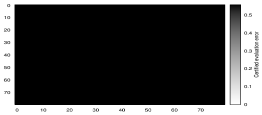
AccurateProduct

- Largest certified error: ≈ 0.1254
- Mean certified error: ≈ 0.0865
- Number of instructions: ≈ 1024000

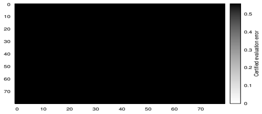
CompactProduct

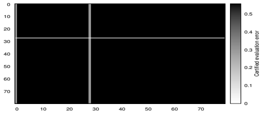
- Largest certified error: ≈ 0.5585
- Mean certified error: ≈ 0.5585
- Number of instructions: ≈ 160

Let us compare these algorithms, cont'd

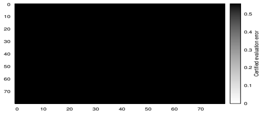


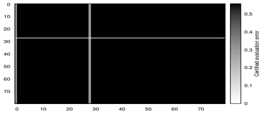
Let us compare these algorithms, cont'd

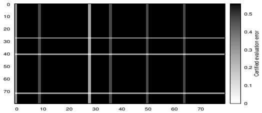




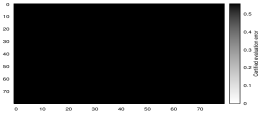
Let us compare these algorithms, cont'd

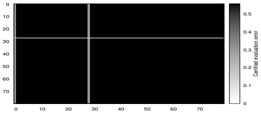


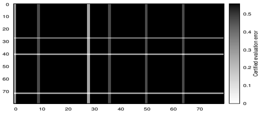


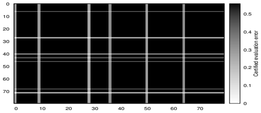


Let us compare these algorithms, cont'd

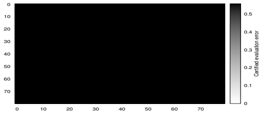


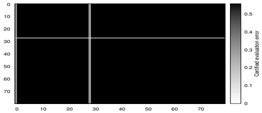


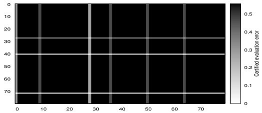


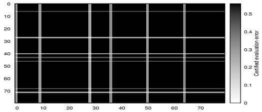


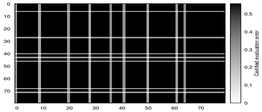
Let us compare these algorithms, cont'd



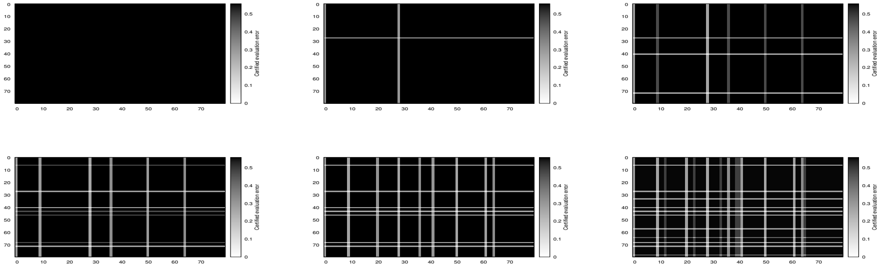




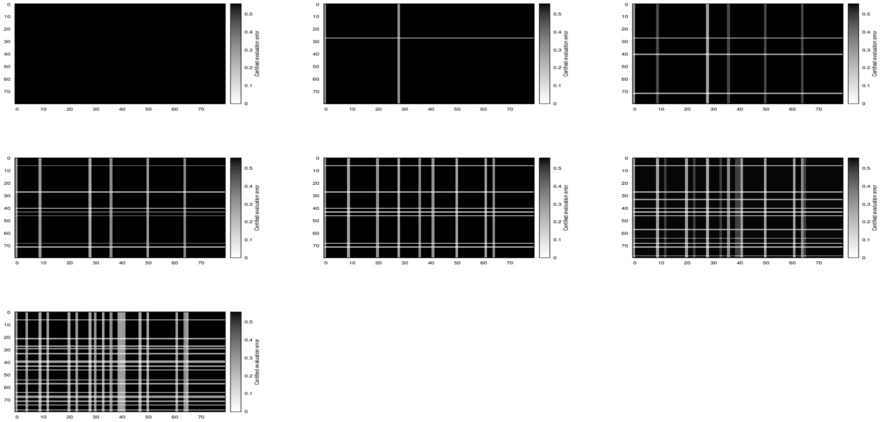




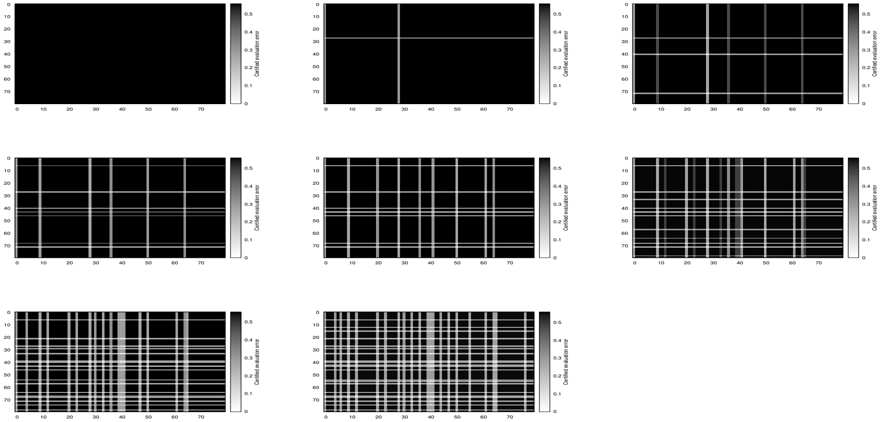
Let us compare these algorithms, cont'd



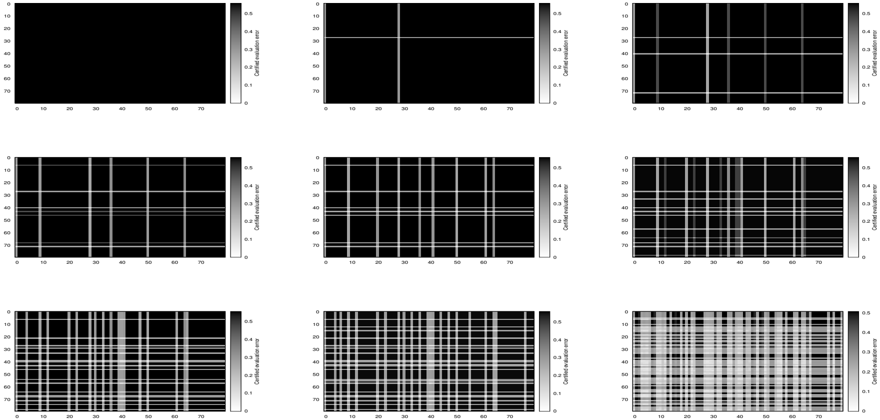
Let us compare these algorithms, cont'd



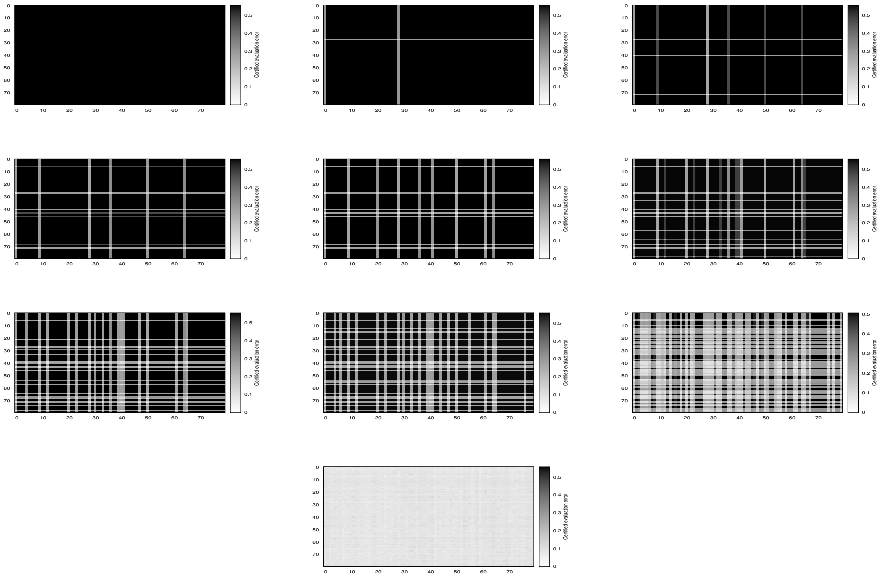
Let us compare these algorithms, cont'd



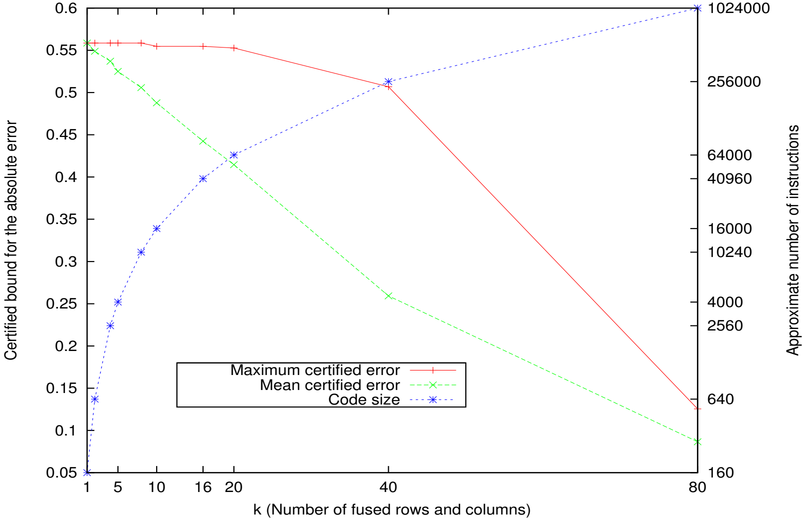
Let us compare these algorithms, cont'd



Let us compare these algorithms, cont'd



Let us compare these algorithms, cont'd



Outline of the talk

- In this talk:
  - We suggested 3 strategies to generate code for matrix product in fixed-point arithmetic
  - The accurate algorithm performs well in terms of numerical quality but is prohibitive
  - The compact algorithm generates concise codes but deteriorates the numerical quality
  - The Closest Pair algorithm enables the tradeoffs between code size and numerical quality

- In this talk:
  - We suggested 3 strategies to generate code for matrix product in fixed-point arithmetic
  - The accurate algorithm performs well in terms of numerical quality but is prohibitive
  - The compact algorithm generates concise codes but deteriorates the numerical quality
  - The Closest Pair algorithm enables the tradeoffs between code size and numerical quality

- For the future, we will be working on:
  - Suggesting similar algorithms for the discrete convolution in fixed-point arithmetic
  - Investigating the synthesis of VHDL code for building blocks like matrix multiplication

13th École Jeunes Chercheurs en Informatique Mathématique (EJCIM 2013)
Perpignan, April 12th, 2013

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

Example of code generated by CGPE

```c
// File automatically generated by CGPE (Code Generation for Polynomial Evaluation)
// Scheme  : ((((a0+(S*(a1+(T*a2))))+((S*(T*T))*(a3+(T*a4))))+((S*(T*T))*((T*T)*(a5+(T*a6)))))+(((S*(T*T))*((T*T)*(
       T*T)))*((a7+(T*a8))+((T*T)*(a9+(T*a10))))))
// Degree  : [9,1]


uint32_t func_0(uint32_t T, uint32_t S)
{
  uint32_t r0 = mul(T, 0x5a82685d);        // 1.31
  uint32_t r1 = 0xb504f31f - r0;           // 1.31
  uint32_t r2 = mul(S, r1);                // 2.30
  uint32_t r3 = 0x00000020 + r2;           // 2.30
  uint32_t r4 = mul(T, T);                 // 0.32
  uint32_t r5 = mul(S, r4);                // 1.31
  uint32_t r6 = mul(T, 0x386fd5f4);        // 1.31
  uint32_t r7 = 0x43df72f7 - r6;           // 1.31
  uint32_t r8 = mul(r5, r7);               // 2.30
  uint32_t r9 = r3 + r8;                   // 2.30
  uint32_t r10 = mul(T, 0x28724100);       // 1.31
  uint32_t r11 = 0x308b1798 - r10;         // 1.31
  uint32_t r12 = mul(r4, r11);             // 1.31
  uint32_t r13 = mul(r5, r12);             // 2.30
  uint32_t r14 = r9 + r13;                 // 2.30
  uint32_t r15 = mul(r4, r4);              // 0.32
  uint32_t r16 = mul(r5, r15);             // 1.31
  uint32_t r17 = mul(T, 0x106c5cd9);       // 1.31
  uint32_t r18 = 0x1d7bf968 - r17;         // 1.31
  uint32_t r19 = mul(T, 0x00fa9aa4);       // 1.31
  uint32_t r20 = 0x05dfffa4 - r19;         // 1.31
  uint32_t r21 = mul(r4, r20);             // 1.31
  uint32_t r22 = r18 + r21;                // 1.31
  uint32_t r23 = mul(r16, r22);            // 2.30
  uint32_t r24 = r14 + r23;                // 2.30
  return r24;
}

/* Error bound computed using MPFI:
 *                              [-10143016495730090498777923426474946138419878708211621348969002255633701140171 5b-283,
 *                                 2558666639368261567540009240052639411998875810271185469392236048075044418576 7b-282]
 *                               ~ [-2^{-27.191},2^{-28.1781}]
 */
```