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?

```c
# 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?

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

Synthesis of fixed-point programs: the case of matrix multiplication
How easy it is to program a product of matrices?

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

```c
#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++) /* dot product of row i and column j */
            for (k = 0; k < N ; k++)
                C[i][j]+=A[i][k]*B[k][j];
    return 0;
}
```

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

A. Najahi (DALI UPVD/LIRMM, CNRS, UM2) 
Synthesis of fixed-point programs: the case of matrix multiplication
How easy it is to program a product of matrices?

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

```c
#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

1. Background of fixed-point arithmetic
   1.1 Basics of fixed-point arithmetic
   1.2 Numerical and combinatorial issues in fixed-point programs
   1.3 CGPE

2. Matrix multiplication in fixed-point
   2.1 An accurate algorithm
   2.2 A compact algorithm
   2.3 Closest pair algorithm

3. Conclusion
Outline of the talk

1. Background of fixed-point arithmetic
   1.1 Basics of fixed-point arithmetic
   1.2 Numerical and combinatorial issues in fixed-point programs
   1.3 CGPE

2. Matrix multiplication in fixed-point
   2.1 An accurate algorithm
   2.2 A compact algorithm
   2.3 Closest pair algorithm

3. Conclusion
Principles of fixed-point arithmetic

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

<table>
<thead>
<tr>
<th>Value in fixed-point</th>
<th>$\frac{130}{2^4} = \frac{2^7 + 2^1}{2^4} = 2^3 + 2^{-3} = 8.125$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 0 1 0</td>
<td>$2^7 + 2^1 = 130$</td>
</tr>
</tbody>
</table>

Integer part       | Fractional part
Principles of fixed-point arithmetic

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

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

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: \( \frac{z}{2^n} \)

<table>
<thead>
<tr>
<th>Integer part</th>
<th>Fractional part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 0 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \quad (1.015625)_{10} \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \quad (2.03125)_{10} \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \quad (0.5078125)_{10}
\end{align*}
\]
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

\[
\begin{array}{c}
\text{1 0 1 0 0 0 1 0} \\
+ \quad \text{1 0 1 1 0 1 0 1} \\
\hline
\text{0 1 1 1 1 1 0 1} \\
\end{array}
\]

\[
\begin{array}{c}
5.0625 \\
2.828125 \\
7.890625 \\
\end{array}
\]

truncated

\[
\begin{array}{c}
7.875 \\
\end{array}
\]
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

\[
\begin{array}{c}
10100010 \\
\end{array}
\]
\[
5.0625 \\
\]
\[
+ \begin{array}{c}
10110101 \\
\end{array}
\]
\[
2.828125 \\
\]
\[
\begin{array}{c}
0111110011 \\
\end{array}
\]
\[
7.890625 \\
7.875 \\
\]

- **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

\[
\begin{array}{c}
10100010 \\
\end{array}
\]
\[
5.0625 \\
\]
\[
\times \begin{array}{c}
01011011 \\
\end{array}
\]
\[
1.421875 \\
\]
\[
\begin{array}{c}
00110011 \quad \begin{array}{c}
\text{[1]} \quad \begin{array}{c}
\text{[0]} \quad \begin{array}{c}
\text{[1]} \quad \begin{array}{c}
\text{[0]} \\
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\]
\[
7.198242187 \\
7.125 \\
\]
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:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$a_2$</td>
</tr>
<tr>
<td>Value</td>
<td>[0.1, 1.57]</td>
<td>[0.1, 1.98]</td>
<td>[0.01, 0.87]</td>
<td>[1.1, 1.86]</td>
</tr>
<tr>
<td>Fixed-point format</td>
<td>$Q_{1,7}$</td>
<td>$Q_{1,7}$</td>
<td>$Q_{0,8}$</td>
<td>$Q_{1,7}$</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th></th>
<th>(a_0)</th>
<th>(b_0)</th>
<th>(a_1)</th>
<th>(b_1)</th>
<th>(a_2)</th>
<th>(b_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>[0.1, 1.57]</td>
<td>[0.1, 1.98]</td>
<td>[0.01, 0.87]</td>
<td>[1.1, 1.86]</td>
<td>[0.15, 4]</td>
<td>[2, 3.3]</td>
</tr>
<tr>
<td>Fixed-point format</td>
<td>(Q_{1,7})</td>
<td>(Q_{1,7})</td>
<td>(Q_{0,8})</td>
<td>(Q_{1,7})</td>
<td>(Q_{4,4})</td>
<td>(Q_{2,6})</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>(a_0)</th>
<th>(b_0)</th>
<th>(a_1)</th>
<th>(b_1)</th>
<th>(a_2)</th>
<th>(b_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1, 1.57</td>
<td>[0, 1]</td>
<td>[0, 1.98]</td>
<td>[0.01, 0.87]</td>
<td>[1.1, 1.86]</td>
<td>[0, 15.4]</td>
<td>[2, 3.3]</td>
</tr>
<tr>
<td>Fixed-point format</td>
<td>(Q_{1.7})</td>
<td>(Q_{1.7})</td>
<td>(Q_{0.8})</td>
<td>(Q_{1.7})</td>
<td>(Q_{4.4})</td>
<td>(Q_{2.6})</td>
</tr>
</tbody>
</table>

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

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

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

with 16 bits precision
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** \(\Rightarrow\) that reduce the evaluation latency on a given target, by using the target architecture features (as much as possible)

- **certified** \(\Rightarrow\) 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

1. Background of fixed-point arithmetic
   1.1 Basics of fixed-point arithmetic
   1.2 Numerical and combinatorial issues in fixed-point programs
   1.3 CGPE

2. Matrix multiplication in fixed-point
   2.1 An accurate algorithm
   2.2 A compact algorithm
   2.3 Closest pair algorithm

3. Conclusion
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 $I(\mathbb{R}^{n \times n})$

$$A = \begin{bmatrix} [-4.54, 7.78] & \cdots & [-0.789, 0.967] \\ \vdots & \ddots & \vdots \\ [12.51, 24.14] & \cdots & [-0.921, 0.791] \end{bmatrix} \quad \text{and,} \quad B = \begin{bmatrix} [-64, 45.78] & \cdots & [0.287, 0.7] \\ \vdots & \ddots & \vdots \\ [125.1, 245.14] & \cdots & [-5.74, 7.32] \end{bmatrix}$$
**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 $I(R^{n \times n})$

  $A = \begin{pmatrix}
  [-4.54, 7.78] & \cdots & [-0.789, 0.967] \\
  \vdots & \ddots & \vdots \\
  [12.51, 24.14] & \cdots & [-0.921, 0.791]
  \end{pmatrix}$

  and, $B = \begin{pmatrix}
  [-64, 45.78] & \cdots & [-0.287, 0.7] \\
  \vdots & \ddots & \vdots \\
  [125.1, 245.14] & \cdots & [-5.74, 7.32]
  \end{pmatrix}$

- 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 $80 \times 80$
Tradeoffs to consider

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

1. Size of the generated code
   - We are targeting embedded systems $\rightarrow$ 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 $80 \times 80$

1. **Size of the generated code**
   - We are targeting embedded systems $\implies$ 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 $80 \times 80$

1. Size of the generated code
   - We are targeting embedded systems $\Rightarrow$ 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

---

**AcuurateProduct**

**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: \textbf{for} $1 < i \leq n$ \textbf{do}
2: \hspace{1em} \textbf{for} $1 < j \leq n$ \textbf{do}
3: \hspace{2em} \textit{cgpeGenDotProduct}(A_i, B_j);
4: \hspace{1em} \textbf{end for}
5: \textbf{end for}
An accurate algorithm

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

---

AccurateProduct

**Inputs:**
- Two square matrices \( A \in \mathbb{I}(\mathbb{R}^{n \times n}) \) and \( B \in \mathbb{I}(\mathbb{R}^{n \times n}) \)

**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. \( \text{cgpeGenDotProduct}(A_i, B_j) \);
4. end for
5. end for

---

Illustration on the product of two \( 2 \times 2 \) matrices

\[
C = \begin{pmatrix}
C_{1,1} = \text{cgpeGenDotProduct}(A_1, B_1) & C_{1,2} = \text{cgpeGenDotProduct}(A_1, B_2) \\
C_{2,1} = \text{cgpeGenDotProduct}(A_2, B_1) & C_{2,2} = \text{cgpeGenDotProduct}(A_2, B_2)
\end{pmatrix}
\]
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 $80 \times 80$ 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 $80 \times 80$ 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 \( A \in \mathbb{R}^{n \times n} \) and \( B \in \mathbb{R}^{n \times n} \)

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 $2 \times 2$ matrices**

$$C = \begin{pmatrix}
C_{1,1} = \text{cgpeGenDotProduct}(A_1 \cup A_2, B_1 \cup B_2) & C_{1,2} = \text{cgpeGenDotProduct}(A_1 \cup A_2, B_1 \cup B_2) \\
C_{2,1} = \text{cgpeGenDotProduct}(A_1 \cup A_2, B_1 \cup B_2) & C_{2,2} = \text{cgpeGenDotProduct}(A_1 \cup A_2, B_1 \cup B_2)
\end{pmatrix}$$
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 $80 \times 80$ 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 $80 \times 80$ 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 $d_H$

\[ d_H : l(\mathbb{R}^n) \times l(\mathbb{R}^n) \rightarrow \mathbb{R} \]
\[ d_H(A, B) = \max_{1 \leq i \leq n} \max \left\{ |a_i - b_i|, |\overline{a_i} - \overline{b_i}| \right\} \]
A closest pair algorithm

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

The Hausdorff distance $d_H$

$$d_H : I(\mathbb{R}^n) \times I(\mathbb{R}^n) \rightarrow \mathbb{R}$$

$$d_H(A, B) = \max_{1 \leq i \leq n} \max \left\{ |a_i - b_i|, |\overline{a_i} - \overline{b_i}| \right\}$$

Example

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

- $d_H(A, B) = 101$
- $\cup(A, B) = \left( [-4, 88] \quad [-23, 102] \right)$
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: $\mathcal{B} = \{v_1, \ldots, v_n\}$
2: while $\text{size}(\mathcal{B}) > k$ do
3: $(u_1, u_2) = \text{findClosestPair}(\mathcal{B})$
4: $\text{remove}(u_1, \mathcal{B})$
5: $\text{remove}(u_2, \mathcal{B})$
6: $\text{add}(\text{Union}(u_1, u_2), \mathcal{B})$
7: end while
Illustration of the ClosestPairFusion with $n = 4$ and $k = l = 2$

$$v_1 = \begin{bmatrix} -4,4 & -5,5 & -5,5 & -6,6 \\ -2,2 & -1,1 & -3,3 & -9,9 \\ -7,7 & -4,4 & -12,12 & -11,11 \\ -8,8 & -1,1 & -10,10 & -9,9 \end{bmatrix}$$

$$v_2 = \begin{bmatrix} -3,3 & -14,14 & -5,5 & -6,6 \\ -1,1 & -11,11 & -3,3 & -9,9 \\ -4,4 & -8,8 & -11,11 & -1,1 \\ -9,9 & -7,7 & -10,10 & -2,2 \end{bmatrix}$$

$$d_{H}(v_1, v_2) \quad d_{H}(v_1, v_3) \quad d_{H}(v_1, v_4) \quad d_{H}(v_2, v_3) \quad d_{H}(v_2, v_4) \quad d_{H}(v_3, v_4)$$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>

$$d_{H}(w_1, w_2) \quad d_{H}(w_1, w_3) \quad d_{H}(w_1, w_4) \quad d_{H}(w_2, w_3) \quad d_{H}(w_2, w_4) \quad d_{H}(w_3, w_4)$$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>
Illustration of the ClosestPairFusion with $n = 4$ and $k = l = 2$

$$
\begin{align*}
  v_1 &= \begin{pmatrix} -4,4 & -5,5 & -5,5 & -6,6 \\ -2,2 & -1,1 & -3,3 & -9,9 \\ -7,7 & -4,4 & -12,12 & -11,11 \\ -8,8 & -1,1 & -10,10 & -9,9 
\end{pmatrix} \\
  v_2 &= \begin{pmatrix} -3,3 & -14,14 & -5,5 & -6,6 \\ -1,1 & -11,11 & -3,3 & -9,9 \\ -4,4 & -8,8 & -11,11 & -1,1 \\ -9,9 & -7,7 & -10,10 & -2,2 
\end{pmatrix} \\
  v_3 &= \begin{pmatrix} -5,5 & -5,5 & -6,6 \\ -1,1 & -3,3 & -9,9 \\ -4,4 & -12,12 & -11,11 \\ -8,8 & -10,10 & -2,2 
\end{pmatrix}
\end{align*}
$$

$$
\begin{align*}
  d_H(v_1, v_2) &= 4 \\
  d_H(v_1, v_3) &= 5 \\
  d_H(v_2, v_3) &= 7 \\
  d_H(v_2, v_4) &= 9 \\
  d_H(v_3, v_4) &= 3 \\
  d_H(w_1, w_2) &= 11 \\
  d_H(w_1, w_3) &= 7 \\
  d_H(w_1, w_4) &= 8 \\
  d_H(w_2, w_3) &= 9 \\
  d_H(w_2, w_4) &= 10 \\
  d_H(w_3, w_4) &= 10 \\
  d_H(v_1 \cup v_3, v_4) &= 4 \\
  d_H(v_1 \cup v_3, v_4) &= 7 \\
  d_H(v_2, v_3 \cup v_4) &= 9 \\
  d_H(w_1 \cup w_3, w_2) &= 9 \\
  d_H(w_1 \cup w_3, w_4) &= 10 \\
  d_H(w_2, w_4) &= 8 \\
\end{align*}
$$
Illustration of the ClosestPairFusion with $n = 4$ and $k = l = 2$

$v_1 = \begin{bmatrix} -4,4 & -5,5 & -5,5 & -6,6 \\ -2,2 & -1,1 & -3,3 & -9,9 \\ -7,7 & -4,4 & -12,12 & -11,11 \\ -8,8 & -1,1 & -10,10 & -9,9 \end{bmatrix}$

$v_2, v_3, v_4$

$W_1 = \begin{bmatrix} -3,3 & -14,14 & -5,5 & -6,6 \\ -1,1 & -11,11 & -3,3 & -9,9 \\ -4,4 & -8,8 & -11,11 & -1,1 \\ -9,9 & -7,7 & -10,10 & -2,2 \end{bmatrix}$

$W_1 \cup W_3, W_2, W_4$

$W_1 \cup W_3 = \begin{bmatrix} -5,5 & -14,14 & -6,6 \\ -3,3 & -11,11 & -9,9 \\ -11,11 & -8,8 & -1,1 \\ -10,10 & -7,7 & -2,2 \end{bmatrix}$

$W_2 \cup W_4$
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 \( 80 \times 80 \) matrices, the table below gives the number of instructions

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>8</th>
<th>10</th>
<th>16</th>
<th>20</th>
<th>40</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>160</td>
<td>320</td>
<td>640</td>
<td>800</td>
<td>1280</td>
<td>1600</td>
<td>2560</td>
<td>3200</td>
<td>6400</td>
<td>12800</td>
</tr>
<tr>
<td>2</td>
<td>320</td>
<td>640</td>
<td>1280</td>
<td>1600</td>
<td>2560</td>
<td>3200</td>
<td>5120</td>
<td>6400</td>
<td>12800</td>
<td>25600</td>
</tr>
<tr>
<td>4</td>
<td>640</td>
<td>1280</td>
<td>2560</td>
<td>3200</td>
<td>5120</td>
<td>6400</td>
<td>10240</td>
<td>12800</td>
<td>25600</td>
<td>51200</td>
</tr>
<tr>
<td>5</td>
<td>800</td>
<td>1600</td>
<td>3200</td>
<td>4000</td>
<td>6400</td>
<td>8000</td>
<td>12800</td>
<td>16000</td>
<td>32000</td>
<td>64000</td>
</tr>
<tr>
<td>8</td>
<td>1280</td>
<td>2560</td>
<td>5120</td>
<td>6400</td>
<td>10240</td>
<td>12800</td>
<td>20480</td>
<td>25600</td>
<td>51200</td>
<td>102400</td>
</tr>
<tr>
<td>10</td>
<td>1600</td>
<td>3200</td>
<td>6400</td>
<td>8000</td>
<td>12800</td>
<td>16000</td>
<td>25600</td>
<td>32000</td>
<td>64000</td>
<td>128000</td>
</tr>
<tr>
<td>16</td>
<td>2560</td>
<td>5120</td>
<td>10240</td>
<td>12800</td>
<td>20480</td>
<td>25600</td>
<td>40960</td>
<td>51200</td>
<td>102400</td>
<td>204800</td>
</tr>
<tr>
<td>20</td>
<td>3200</td>
<td>6400</td>
<td>12800</td>
<td>16000</td>
<td>25600</td>
<td>32000</td>
<td>51200</td>
<td>64000</td>
<td>128000</td>
<td>256000</td>
</tr>
<tr>
<td>40</td>
<td>6400</td>
<td>12800</td>
<td>25600</td>
<td>32000</td>
<td>51200</td>
<td>64000</td>
<td>102400</td>
<td>128000</td>
<td>256000</td>
<td>512000</td>
</tr>
<tr>
<td>80</td>
<td>12800</td>
<td>25600</td>
<td>51200</td>
<td>64000</td>
<td>102400</td>
<td>128000</td>
<td>204800</td>
<td>256000</td>
<td>512000</td>
<td>1024000</td>
</tr>
</tbody>
</table>

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 $80 \times 80$ matrices, the table below gives the number of instructions

<table>
<thead>
<tr>
<th>$k$</th>
<th>$l$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>8</th>
<th>10</th>
<th>16</th>
<th>20</th>
<th>40</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>160</td>
<td>320</td>
<td>640</td>
<td>800</td>
<td>1280</td>
<td>1600</td>
<td>2560</td>
<td>3200</td>
<td>6400</td>
<td>12800</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>320</td>
<td>640</td>
<td>1280</td>
<td>1600</td>
<td>2560</td>
<td>3200</td>
<td>5120</td>
<td>6400</td>
<td>12800</td>
<td>25600</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>640</td>
<td>1280</td>
<td>2560</td>
<td>3200</td>
<td>5120</td>
<td>6400</td>
<td>10240</td>
<td>12800</td>
<td>25600</td>
<td>51200</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>800</td>
<td>1600</td>
<td>3200</td>
<td>4000</td>
<td>6400</td>
<td>8000</td>
<td>12800</td>
<td>16000</td>
<td>32000</td>
<td>64000</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1280</td>
<td>2560</td>
<td>5120</td>
<td>6400</td>
<td>10240</td>
<td>12800</td>
<td>20480</td>
<td>25600</td>
<td>51200</td>
<td>102400</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1600</td>
<td>3200</td>
<td>6400</td>
<td>8000</td>
<td>12800</td>
<td>16000</td>
<td>25600</td>
<td>32000</td>
<td>64000</td>
<td>128000</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>2560</td>
<td>5120</td>
<td>10240</td>
<td>12800</td>
<td>20480</td>
<td>25600</td>
<td>40960</td>
<td>51200</td>
<td>102400</td>
<td>204800</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>3200</td>
<td>6400</td>
<td>12800</td>
<td>16000</td>
<td>25600</td>
<td>32000</td>
<td>51200</td>
<td>64000</td>
<td>128000</td>
<td>256000</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>6400</td>
<td>12800</td>
<td>25600</td>
<td>32000</td>
<td>51200</td>
<td>64000</td>
<td>102400</td>
<td>128000</td>
<td>256000</td>
<td>512000</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>12800</td>
<td>25600</td>
<td>51200</td>
<td>64000</td>
<td>102400</td>
<td>128000</td>
<td>204800</td>
<td>256000</td>
<td>512000</td>
<td>1024000</td>
</tr>
</tbody>
</table>

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 $80 \times 80$
  - The center of each interval is randomly selected in $[-1000, 1000]$
  - The diameter of the intervals is fixed to 100

<table>
<thead>
<tr>
<th>AccurateProduct</th>
<th>CompactProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Largest certified error: $\approx 0.1254$</td>
<td>Largest certified error: $\approx 0.5585$</td>
</tr>
<tr>
<td>Mean certified error: $\approx 0.0865$</td>
<td>Mean certified error: $\approx 0.5585$</td>
</tr>
<tr>
<td>Number of instructions: $\approx 1024000$</td>
<td>Number of instructions: $\approx 160$</td>
</tr>
</tbody>
</table>
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

![Graph showing certified bound for the absolute error, approximate number of instructions, maximum certified error, and mean certified error as functions of the number of fused rows and columns (k).]
Outline of the talk

1. Background of fixed-point arithmetic
   1.1 Basics of fixed-point arithmetic
   1.2 Numerical and combinatorial issues in fixed-point programs
   1.3 CGPE

2. Matrix multiplication in fixed-point
   2.1 An accurate algorithm
   2.2 A compact algorithm
   2.3 Closest pair algorithm

3. Conclusion
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
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
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 = 0x05dffa4 - 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:
   [-1014301649573009049877792342647494613841987877082116213489690022556337011401715b-283,
   25586666393682615675400092400526394119988758102711854693922360480750444185767b-282]
   ~ [-2^(-27.191),2^(-28.1781)] */
```