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
Motivation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

What makes the problem harder in fixed-point?

Intermediate computations depend on the input variables range and computation scheme

Some works on linear algebra primitives in fixed-point

Lee et al. (2006): $8 \times 8$ matrix-vector products for the computation of DCT's
- Relies on some DCT properties

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

A. Najahi (DALI UPVD/LIRMM, CNRS, UM2)
- Automated synthesis of fixed-point programs: the case of matrix multiplication
Matrix multiplication

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

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

What makes the problem harder in fixed-point?
Intermediate computations depend on the input variables range and computation scheme

Some works on linear algebra primitives in fixed-point
Lee et al. (2006): 8 × 8 matrix-vector products for the computation of DCT’s
Relies on some DCT properties
Frantz et al. (2007): linear algebra routines (mostly matrix inversion) based on simulation
No strict guarantee on the error bounds
Based on lengthy simulations

A. Najahi (DALI UPVD/LIRMM, CNRS, UM2)
Automated synthesis of fixed-point programs: the case of matrix multiplication
Matrix multiplication

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

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

What makes the problem harder in fixed-point?

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

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

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

What makes the problem harder in fixed-point?

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

Some works on linear algebra primitives in fixed-point

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

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

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

2. Efficient matrix multiplication in fixed-point arithmetic

3. 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 \( z = (10000010)_2 \) 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 $z = (10000010)_2$ and $n = 4$

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

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

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

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

Arithmetic model to track errors in fixed-point computations

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

Arithmetic model to track errors in fixed-point computations

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

Addition:

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

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

\[
\begin{array}{c}
+ \\
\hline
10100010 + 01010101 \\
\hline
11110111
\end{array}
\]

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

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

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

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

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

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

  \[
  \begin{array}{c}
  10100010 \\
  \times \\
  01010101
  \end{array}
  \quad
  \begin{array}{c}
  5.0625
  \end{array}
  \]

  \[
  \begin{array}{c}
  00110101 \overset{T0}{\overline{1010}} \\
  \times \\
  01010101
  \end{array}
  \quad
  \begin{array}{c}
  6.723632812
  \end{array}
  \]

- **Physical and virtual shifts:**

  \[
  \begin{align*}
  \text{val}(r) &= \text{val}(l) \gg 2 \\
  \text{err}(r) &= \text{err}_{\text{shift}} + \text{err}(l) \gg 2
  \end{align*}
  \]

  \[
  \begin{array}{c}
  01110010 \\
  \gg \quad 2
  \end{array}
  \quad
  \begin{array}{c}
  0.875
  \end{array}
  \]

  \[
  \begin{array}{c}
  00011100 \\
  \gg_{v} \quad 2
  \end{array}
  \quad
  \begin{array}{c}
  3.5
  \end{array}
  \]

\[\text{val}(\overline{r}) = \text{val}(l) \ll 2 \quad \text{err}(\overline{r}) = \text{err}(l) \ll 2\]
Numerical issues in dot product generation

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

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

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

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

- Let us focus on 2 different schemes to compute the sum of products:
  - in full precision
  - \((c_0 + (c_1 + c_2))\)
Numerical issues in dot product generation

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

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

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

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

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

  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: \((2n - 1)!!\)

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

Remarks

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

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

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

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

\textsuperscript{1} Code Generation for Polynomial Evaluation
The CGPE library

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

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

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

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

\(^1\)Code Generation for Polynomial Evaluation

A. Najahi (DALI UPVD/LIRMM, CNRS, UM2)
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
Efficient matrix multiplication in fixed-point arithmetic

Straightforward algorithms

Accurate product

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

<table>
<thead>
<tr>
<th>AccurateProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs:</strong></td>
</tr>
<tr>
<td>Two fixed-point square matrices $A$ and $B$</td>
</tr>
<tr>
<td><strong>Outputs:</strong></td>
</tr>
<tr>
<td>C code to compute the product $AB$</td>
</tr>
<tr>
<td><strong>Steps:</strong></td>
</tr>
<tr>
<td>1: for $1 &lt; i \leq n$ do</td>
</tr>
<tr>
<td>2: for $1 &lt; j \leq n$ do</td>
</tr>
<tr>
<td>3: $\text{cgpeGenDotProduct}(A_i, B_j)$;</td>
</tr>
<tr>
<td>4: end for</td>
</tr>
<tr>
<td>5: end for</td>
</tr>
</tbody>
</table>

Compact product

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

<table>
<thead>
<tr>
<th>CompactProduct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs:</strong></td>
</tr>
<tr>
<td>Two fixed-point square matrices $A$ and $B$</td>
</tr>
<tr>
<td><strong>Outputs:</strong></td>
</tr>
<tr>
<td>C code to compute the product $AB$</td>
</tr>
<tr>
<td><strong>Steps:</strong></td>
</tr>
<tr>
<td>1: compute $v$ such that $v = A_1 \cup A_2 \cup \cdots \cup A_n$</td>
</tr>
<tr>
<td>2: compute $w$ such that $w = B_1 \cup B_2 \cup \cdots \cup B_n$</td>
</tr>
<tr>
<td>3: $\text{cgpeGenDotProduct}(v, w)$;</td>
</tr>
</tbody>
</table>
Illustration through a toy example

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

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

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

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

### 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

\[
u = \begin{pmatrix}
-1000 & 1000 \\
-2000 & 2000 \\
-4000 & 4000
\end{pmatrix}
\]

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

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

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

Number of ways to merge \(n\) vectors given by the \(n\)th Bell number

\[
\begin{align*}
\text{Number of vectors} & \quad 3 & 5 & 10 & 16 & 20 & \cdots \\
\text{Number of schemes} & \quad 5 & 52 & 115975 & \approx 2^{17} & 10480142147 & \approx 2^{33} & 51724158235372 & \approx 2^{46} & \cdots
\end{align*}
\]
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 = \[
\begin{bmatrix}
  a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\
  a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{40} & a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\]

Compact product

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

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

**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?

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

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

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

Number of ways to merge \( n \) vectors

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

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

The Hausdorff distance $d_H$

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

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

Example

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

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

The Hausdorff distance $d_H$

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

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

Example

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

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

Another possible criterion

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

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

Example

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

**ClosestPairUnion**

**Inputs:**
- $n$ fixed-point vectors $v_0, \ldots, v_{n-1}$
- a routine `findClosestPair`
- a distance $d$

**Outputs:**
- $n-1$ fixed-point vectors

**Steps:**
1. $\mathcal{B} = \{v_0, \ldots, v_{n-1}\}$
2. $(u_0, u_1) = \text{findClosestPair}(\mathcal{B}, d)$
3. `remove`(u_0, $\mathcal{B}$)
4. `remove`(u_1, $\mathcal{B}$)
5. `add`(Union(u_0, u_1), $\mathcal{B}$)
Closest pair strategy (1/2)

**ClosestPairUnion**

**Inputs:**
- $n$ fixed-point vectors $v_0, \ldots, v_{n-1}$
- a routine `findClosestPair`
- a distance $d$

**Outputs:**
- $n - 1$ fixed-point vectors

**Steps:**
1. $B = \{v_0, \ldots, v_{n-1}\}$
2. $(u_0, u_1) = \text{findClosestPair}(B, d)$
3. `remove(u_0, B)`
4. `remove(u_1, B)`
5. `add(Union(u_0, u_1), B)`

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

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

**ClosestPairUnion**

**Inputs:**
- \( n \) fixed-point vectors \( v_0, \ldots, v_{n-1} \)
- a routine \( \text{findClosestPair} \)
- a distance \( d \)

**Outputs:**
- \( n - 1 \) fixed-point vectors

**Steps:**
1. \( \mathcal{B} = \{ v_0, \ldots, v_{n-1} \} \)
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 $v_0, \cdots v_{n-1}$
- a routine $findClosestPair$
- a distance $d$

**Outputs:**
- $n-1$ fixed-point vectors

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

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

Closest pair: $A_0$ and $A_3$

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

Closest pair: $A_1$ and $A_4$
Closest pair strategy (2/2)

---

Closest Pair algorithm

**Inputs:**
- 2 fixed-point matrices $A$ and $B$
- a criterion $C$

**Outputs:**
- C code that evaluates $A \cdot B$
- s.t. $C$ is satisfied

**Steps:**

1. **while** $C$ 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 $C$: average error

**Outputs:**
- C code that evaluates $A \cdot B$
  s.t. $C$ is satisfied

**Steps:**

1. **while** average error $< \epsilon$ **do**
2. $A = \text{ClosestPairUnion}(A)$
3. $B = \text{ClosestPairUnion}(B)$
4. **for** $i < \text{numberOfRows}(A)$ **do**
5.   **for** $j < \text{numberOfCols}(B)$ **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 $\text{midrad}(H, 1)$
Results
Benchmarks and 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 FPLA$^2$ tool.
- This tool relies heavily on previous work implemented in the CGPE library.
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 FPLA\(^2\) 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

\(^2\)Fixed-Point Linear Algebra
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