Expanding the Horizons of Finite-Precision Analysis

Debasmita Lohar

PhD Defense Talk

27th March, 2024
def controller(x: Real, y: Real, z: Real): Real = {
    val res = -x*y - 2*y*z - x - z
    return res
}
def controller(x: Real, y: Real, z: Real): Real = {
    val res = -x*y - 2*y*z - x - z
    return res
}

• Reals are implemented in Floating-point / Fixed-point data type
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    val res = -x*y - 2*y*z - x - z
    return res
}

- Reals are implemented in Floating-point / Fixed-point data type
- Introduces roundoff errors at potentially every operation
Errors in Finite-Precision

def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    val res = -x*y - 2*y*z - x - z
    return res
}

+-/ error

0.1 + 0.2 = 0.3

32-bit floating-point arithmetic
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    val res = -x*y - 2*y*z - x - z
    return res
}

+/- error

\[
\begin{array}{c}
0.1 + 0.2 = 0.3 \\
\text{real arithmetic}
\end{array}
\]

\[
\begin{array}{c}
0.1 + 0.2 \\
\text{32-bit floating-point arithmetic}
\end{array}
\]

Does it even affect real-world systems?
Finite-Precision Errors in Real World

February 1991, Dhahran, Saudi Arabia
Gulf War: Loss of accuracy led to failure in US defense system, 28 killed!

April 1992, Schleswig-Holstein, Germany
Rounding error changed Parliament makeup!

June 1996
Overflow led to explosion of Ariane 5, 39s after lift-off, $370 million lost!

...
Finite-Precision Errors in Real World

February 1991, Dhahran, Saudi Arabia
Gulf War: Loss of accuracy led to failure in US defense system, 28 killed!

April 1992, Schleswig-Holstein, Germany
Rounding error changed Parliament makeup!

June 1996
Overflow led to explosion of Ariane, 39s after lift-off, $370 million lost!

May, 2020
Rounding error in luminance computation crashed Android phones
Finite-Precision Errors in Real World

February 1991, Dhahran, Saudi Arabia

Gulf War: Loss of accuracy led to failure in US defense system, 28 killed!

April 1992, Schleswig-Holstein, Germany

Rounding error changed Parliament makeup!

June 1996

Overflow led to explosion of Ariane 5, 39s after lift-off, $370 million lost!

May 2020

Rounding error in luminance computation crashed Android phones

How do we compute the errors?
Finite-Precision Accuracy Analysis

```python
(x:Float32, y:Float32, z:Float32): Float32

def controller(x, y, z): = {
    val res = -x*y - 2*y*z - x - z
    return res
}
```
compute a bound on the error

```python
(x:Float32, y:Float32, z:Float32): Float32
def controller(x, y, z): = {
    val res = -x*y - 2*y*z - x - z
    return res
}
```
def controller(x, y, z):
    val res = -x*y - 2*y*z - x - z
    return res
}

ensuring (res +/− ?)

absolute error:

\[
\max_{x,y,z \in \mathbb{I}} | f(x,y,z) - \tilde{f}(\tilde{x}, \tilde{y}, \tilde{z}) |
\]
Finite-Precision Accuracy Analysis

\[
\max_{x,y,z \in I} | f(x, y, z) - \tilde{f}(\tilde{x}, \tilde{y}, \tilde{z}) |
\]

worst-case error analysis for small programs

Daisy  FLUCTUAT  Rosa
FPTaylor  PRECiSA  ...
def controller(x, y, z): ___ = {
    val res = -x*y - 2*y*z - x - z
    return res
} ensuring (res +/- ?)
def controller(x, y, z):
    res = -x*y - 2*y*z - x - z
    return res

Errors depend on Precision used

(x:Float16, y:Float16, z:Float16)

2.02e+00
(x: __, y: __, z: __)

```python
def controller(x, y, z): ___ = {
    val res = -x*y - 2*y*z - x - z
    return res
}
```

Errors depend on Precision used

```
2.02e+00
1.58e-4
```
def controller(x, y, z):
    res = -x*y - 2*y*z - x - z
    return res

Ensuring (res +/- ?)

Errors depend on Precision used

(x:__, y:__, z:__)

2.02e+00
1.58e-4
2.95e-13
4.84e-31
def controller(x, y, z):
    res = -x*y - 2*y*z - x - z
    return res

ensuring (res +/- ?)

So are the Resource Costs!

(x: __, y: __, z: __)
def controller(x, y, z):
    res = -x*y - 2*y*z - x - z
    return res

ensuring (res +/- ?)
def controller(x, y, z): ___ = {
    val res = -x*y - 2*y*z - x - z
    return res
} ensuring (res +/- ?)

So are the Resource Costs!

We need to find a tradeoff between accuracy and resources!
def controller(x, y, z): 
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)

find the lowest precision satisfying error bound
Finite-Precision Optimization

```scala
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- 0.00197)
```

mixed-precision optimization

- minimize resource cost still satisfying the error
- assign different precisions to different variables
def controller(x, y, z): ___ = {
    val res = -x*y - 2*y*z - x - z
    return res
} ensuring (res +/- ?)

Finite-Precision Optimization

- minimize resource cost still satisfying the error
- assign different precisions to different variables

worst-case tuning for small (floating-point) programs

Daisy FPTuner
The Horizons of Finite-Precision Analysis

Accuracy Analysis

- worst-case error analysis for small programs
- Daisy
- FLUCTUAT
- Rosa
- FPTaylor
- PRECiSA
- ...

Optimization

- worst-case tuning for small (floating-point) programs
- Daisy
- FPTuner
Our Work: Extending the Horizon of Finite-Precision Analysis

**Accuracy Analysis**

- Considering probability distribution of inputs
- Probabilistic Analysis
  - iFM '19
  - EMSOFT '18

**Optimization**

- Worst-case tuning for small (floating-point) programs

**Tools**

- Daisy
- FLUCTUAT
- Rosa
- FPTaylor
- PRECiSA
- ...
Our Work: Extending the Horizon of Finite-Precision Analysis

Accuracy Analysis

- worst-case error analysis for small programs

Optimization

- handling larger programs

Probabilistic Analysis

- Static + Dynamic Analysis

- worst-case tuning for small (floating-point) programs

Daisy
FLUCTUAT
Rosa
FPTaylor
PRECiSA

Daisy
FPTuner
Our Work: Extending the Horizon of Finite-Precision Analysis

Accuracy Analysis

Probabilistic Analysis

Static + Dynamic Analysis

worst-case error analysis for small programs

Daisy  FLUCTUAT  Rosa  FPTaylor  PRECiSA  ...

Optimization

specializing mixed fixed tuning for NNs

NN Quantization

worst-case tuning for small (floating-point) programs

Daisy  FPTuner
Today's Talk: Probabilistic Error Analysis and NN Quantization

Accuracy Analysis

- Probabilistic Analysis
- Static + Dynamic Analysis

- worst-case error analysis for small programs

- Daisy
- FLUCTUAT
- Rosa
- FPTaylor
- PRECiSA
- ...

Optimization

- NN Quantization
- Dynamic Analysis

- worst-case tuning for small (floating-point) programs

- Daisy
- FPTuner

EMSOFT '19
EMSOFT '18
TACAS '21
EMSOFT '23
How do we take into account uncertainties in the inputs and compute the distribution of errors at the output?
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    require (-15.0 <= x, y, z <= 15.0)
    val res = -x*y - 2*y*z - x - z
    return res
} ensuring (res +/- ?)

State-of-the-Art: Worst-Case Error Analysis

Daisy  FLUCTUAT  Rosa
FPTaylor  PRECiSA  ...

absolute error: 1.7e-4
Worst-case can be pessimistic!

def controller(x:Float32, y:Float32, z:Float32): Float32 = {
  require (-15.0 <= x, y, z <= 15.0)
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
Worst-case can be pessimistic!

```scala
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
  require (-15.0 <= x, y, z <= 15.0)
  val res = -x*y - 2*y*z - x - z
  return res
} ensuring (res +/- ?)
```

For most of inputs, errors are small!
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    require (-15.0 <= x, y, z <= 15.0)
    val res = -x*y - 2*y*z - x - z
    return res
} ensuring (error <= 1.5e-4, 0.85)

tolerates big errors occurring with <= 0.15 probability
Scenario 1: Applications may tolerate large infrequent errors

def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    require (-15.0 <= x, y, z <= 15.0)
    val res = -x*y - 2*y*z - x - z
    return res
}

ensuring (error <= 1.5e-4, 0.85)

worst-case error: 1.7e-4

tolerates big errors occurring with <= 0.15 probability
scenario 1: applications may tolerate large infrequent errors

```python
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    require (-15.0 <= x, y, z <= 15.0)
    val res = -x*y - 2*y*z - x - z
    return res
}
```

ensuring (error <= 1.5e-4, 0.85)

worst-case error: 1.7e-4

tolerates big errors occurring with <= 0.15 probability

We need to analyze roundoff errors probabilistically!
Our Contribution: Probabilistic Analysis for Roundoff Errors

```python
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    require (-15.0 <= x, y, z <= 15.0)

    x := gaussian(4.0, 0.5)
    y := gaussian(4.75, 0.2)
    z := gaussian(4.8, 0.25)

    val res = -x*y - 2*y*z - x - z
    return res
} ensuring (error <= 1.5e-4, 0.85)
```
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    require (-15.0 <= x, y, z <= 15.0)
    x := gaussian(4.0, 0.5)
    y := gaussian(4.75, 0.2)
    z := gaussian(4.8, 0.25)
    val res = -x*y - 2*y*z - x - z
    return res
} ensuring (error <= 1.5e-4, 0.85)

✓ probability distribution of errors
✓ a refined error that occurs with the threshold probability
Overview: Sound Probabilistic Roundoff Error Analysis

finite-precision program with probabilistic inputs

probabilistic error analysis
Overview: Sound Probabilistic Roundoff Error Analysis

finite-precision program with probabilistic inputs

probabilistic interval subdivision

probabilistic error analysis
Probabilistic Interval Subdivision

finite-precision program with probabilistic inputs

probabilistic interval subdivision
probabilistic error analysis

require (-15.0 <= x, y, z <= 15.0)
Probabilistic Interval Subdivision

finite-precision program with probabilistic inputs

probabilistic interval subdivision

probabilistic error analysis

require \((-15.0 \leq x, y, z \leq 15.0)\)
Probabilistic Interval Subdivision

finite-precision program with probabilistic inputs

probabilistic interval subdivision

probabilistic error analysis

require (-15.0 <= x, y, z <= 15.0)

subdomain with a probability taking Cartesian Product:

\[ \forall i \in x, \forall j \in y, \forall k \in z, p_{ijk} = x_i \times y_j \times z_k \]
Probabilistic Error Analysis

finite-precision program with probabilistic inputs

probabilistic interval subdivision

probabilistic error analysis

\[
<s_{ijk}, p_{ijk} >
\]

\[-x*y - 2*y*z - x - z\]
Probabilistic Error Analysis

finite-precision program with probabilistic inputs

probabilistic interval subdivision
probabilistic error analysis

error distribution:

\[-x*y - 2*y*z - x - z\]

cumulative prob.
lower bound
upper bound
abs. error
Probability Distribution of Errors

finite-precision program with probabilistic inputs

probabilistic interval subdivision

probabilistic error analysis

\[ x \cdot y \cdot z \]
Refined Error Bounds

finite-precision program with probabilistic inputs

probabilistic interval subdivision
probabilistic error analysis

error metric extraction

threshold = 0.85
Refined Error Bounds

finite-precision program with probabilistic inputs

probabilistic interval subdivision

probabilistic error analysis

error metric extraction

threshold = 0.85

refined error, probability

1 - 0.85

worst-case
Summary of Results: Probabilistic Error Analysis

threshold probability: 0.85, 32-bit floating-point error

<table>
<thead>
<tr>
<th>#benchmarks</th>
<th>#inputs</th>
<th>#arith-ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1 – 9</td>
<td>4 – 25</td>
</tr>
</tbody>
</table>
Summary of Results: Probabilistic Error Analysis

threshold probability: 0.85, 32-bit floating-point error

<table>
<thead>
<tr>
<th>#benchmarks</th>
<th>#inputs</th>
<th>#arith-ops</th>
<th>error reduction (%) from worst-case to the largest frequent average</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1 - 9</td>
<td>4 - 25</td>
<td>17.0</td>
<td>48.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16.2</td>
<td>45.1</td>
</tr>
</tbody>
</table>
Summary of Results: Probabilistic Error Analysis

Threshold probability: 0.85, 32-bit floating-point error

<table>
<thead>
<tr>
<th>#benchmarks</th>
<th>#inputs</th>
<th>#arith-ops</th>
<th>error reduction (%) from worst-case to the largest frequent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>average gaussian uniform max gaussian uniform</td>
</tr>
<tr>
<td>25</td>
<td>1 – 9</td>
<td>4 – 25</td>
<td>17.0 16.2 48.9 45.1</td>
</tr>
</tbody>
</table>

Reductions up to 73.1% with approximate hardware specifications!
• Not all applications need worst-case guarantees

• Providing bounds on most frequent errors can be resource-efficient

• An automated probabilistic error analyzer: PrAn
  - strikes a balance between accuracy and complexity
  - handles different distributions, dependencies, and thresholds
Today’s Talk: Probabilistic Error Analysis and NN Quantization

Accuracy Analysis

- Probabilistic Analysis
- Static + Dynamic Analysis
- worst-case error analysis for small programs
- Daisy, FLUCTUAT, Rosa, FPTaylor, PRECiSA, ...

Optimization

- NN Quantization
- worst-case tuning for small (floating-point) programs
- Daisy, FPTuner
How do we generate quantized implementations for neural networks that meet specified worst-case error bounds?
Neural networks are ubiquitous in safety-critical systems!

- Adaptive Cruise Control
- Collision Avoidance System
- Unicycle Controller
- Drone Controller
Neural Networks as Controllers

Unicycle Controller
Neural Networks as Controllers

**Unicycle Controller**

```python
def UnicycleController(in: Vector): Vector = {
    weights1 = Matrix(...)
    weights2 = Matrix(...)
    bias1 = Vector(...)
    bias2 = Vector(...)
    x1 = relu(weights1 * in + bias1)
    out = linear(weights2 * x1 + bias2)
    return out
}
```

feed-forward regression models
Models are trained in High-Precision

\[ x_1 = \text{relu}(W_1 \cdot \text{in} + b_1) \]

\[ \text{out} = \text{linear}(W_2 \cdot x_1 + b_2) \]
Models are trained in High-Precision

$x_1 = \text{relu}(W_1 * \text{in} + b_1)$

$out = \text{linear}(W_2 * x_1 + b_2)$
Models are trained in High-Precision

\[ x_1 = \text{relu}(W_1 \cdot \text{in} + b_1) \]

\[ \text{out} = \text{linear}(W_2 \cdot x_1 + b_2) \]

Input data → training → high-precision GPU → 64-bit floating-point

\[
\begin{bmatrix}
-5.23724322e-03 & \cdots & 1.30853499e-04 \\
\vdots & \ddots & \vdots \\
-7.29779880e-01 & \cdots & -2.27958648e-04
\end{bmatrix}
\]
Models are trained in High-Precision

\[
x_1 = \text{relu}(W_1 \cdot \text{in} + b_1)
\]

\[
\text{out} = \text{linear}(W_2 \cdot x_1 + b_2)
\]

in real-valued arithmetic

+/- error

64-bit floating-point

input data \rightarrow \text{training}
Model Deployment requires Quantization

- input data
- training
- high-precision
- GPU
- quantization
- fixed low precision system
- +/- error

model deployment
We need to quantize respecting the error bound!

Model Deployment requires Quantization

input data → training → model deployment

quantization

+/- error

fixed low precision system
Sound Mixed Fixed-Point Quantization

Unicycle Controller

\[
\begin{align*}
-0.6 & \leq \text{in1} \leq 9.55 \\
-4.5 & \leq \text{in2} \leq 0.2 \\
-0.06 & \leq \text{in3} \leq 2.11 \\
-0.3 & \leq \text{in4} \leq 1.51
\end{align*}
\]

res +/- 1e-3

mixed precision fixed-point code

directly synthesized

XILINX
State-of-the-art is not enough!

-0.6 <= in1 <= 9.55
-4.5 <= in2 <= 0.2
-0.06 <= in3 <= 2.11
-0.3 <= in4 <= 1.51

no fixed-point support!

res +/- 1e-3

mixed precision
fixed-point code

directly synthesized

• not scalable
• needs unrolled structures
• over-approximates a lot
State-of-the-art is not enough!

-0.6 ≤ in1 ≤ 9.55
-4.5 ≤ in2 ≤ 0.2
-0.06 ≤ in3 ≤ 2.11
-0.3 ≤ in4 ≤ 1.51

Our Contribution: Sound Scalable Quantizer for NNs
Key Idea: Quantization for efficiency is an optimization problem!

\[ \gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha} \]

- integer-valued cost

\begin{align*}
-0.6 &\leq in1 \leq 9.55 \\
-4.5 &\leq in2 \leq 0.2 \\
-0.06 &\leq in3 \leq 2.11 \\
-0.3 &\leq in4 \leq 1.51
\end{align*}
Key Idea: Quantization for efficiency is an optimization problem!

```
res +/- 1e-3
```

-0.6 <= in1 <= 9.55
-4.5 <= in2 <= 0.2
-0.06 <= in3 <= 2.11
-0.3 <= in4 <= 1.51

\[
\begin{align*}
\text{minimize: } & \gamma = \sum_{i=1}^{n} \gamma_i^{\text{dot}} + \gamma_i^{\text{bias}} + \gamma_i^{\alpha} \\
\text{subject to: } & \epsilon_n \leq \epsilon_{\text{target}}
\end{align*}
\]

- integer-valued cost
- real-valued error constraint
**Key Idea:** Quantization for efficiency is an optimization problem!

\[
\begin{align*}
\text{minimize:} & \quad \gamma = \sum_{i=1}^{n} \gamma_i^{\text{dot}} + \gamma_i^{\text{bias}} + \gamma_i^{\alpha} \\
\text{subject to:} & \quad \epsilon_n \leq \epsilon_{\text{target}} \\
& \quad I_{i}^{\text{op}} \geq \text{intBits} \left( R_{i}^{\text{op}} + \epsilon_i \right)
\end{align*}
\]

- integer-valued cost
- real-valued error constraint
- integer-valued range constraint

Ensure: no overflow

\[-0.6 \leq \text{in1} \leq 9.55 \]
\[-4.5 \leq \text{in2} \leq 0.2 \]
\[-0.06 \leq \text{in3} \leq 2.11 \]
\[-0.3 \leq \text{in4} \leq 1.51 \]

\[\text{res} \pm/\mp 1\times10^{-3}\]
mixed-integer problem

minimize: \[ \gamma = \sum_{i=1}^{n} \gamma_{i}^{\text{dot}} + \gamma_{i}^{\text{bias}} + \gamma_{i}^{\alpha} \]

subject to:

\[ \epsilon_n \leq \epsilon_{\text{target}} \]

\[ I_{i}^{\text{op}} \geq \text{intBits} \left( R_{i}^{\text{op}} + \epsilon_i \right) \]
minimize: \( \gamma = \sum_{i=1}^{n} \gamma_i^{\text{dot}} + \gamma_i^{\text{bias}} + \gamma_i^{\alpha} \)

subject to:

\[ \epsilon_n \leq \epsilon_{\text{target}} \]

\[ I_i^{\text{op}} \geq \text{intBits} \left( R_i^{\text{op}} + \epsilon_i \right) \]
Our Solution: Reduce to Mixed Integer Linear Programming (MILP) Problem!
Aster: Sound Quantizer for NNs

```python
def UnicycleController(in: Vector): Vector = {
    weights1 = Matrix [...]
    weights2 = Matrix [...]
    bias1 = Vector(...)
    bias2 = Vector(...)
    x1 = relu(weights1 * in + bias1)
    out = linear(weights2 * x1 + bias2)
    return out
}
```

ensuring (res +/- 1e-3)

high-level model
Aster: Sound Quantizer for NNs

```python
def UnicycleController(in: Vector): Vector = {
    weights1 = Matrix[...]
    weights2 = Matrix[...]
    bias1 = Vector(...)
    bias2 = Vector(...)
    x1 = relu(weights1 * in + bias1)
    out = linear(weights2 * x1 + bias2)
    return out
}
```

ensuring (res +/- 1e-3)

mixed-precision fixed-point code

```c
#include <math.h>
#include <ap_fixed.h>
#include <hls_math.h>
#include <ap_fixed.h>

void nn1(ap_fixed<24,5> x_0, ap_fixed<24,4> x_1, ap_fixed<24,3> x_2, ap_fixed<24,2> x_3, ap_fixed<27,8> _result[2]) {
    ap_fixed<24,1> weights1_0_0 = -0.036691424;
    ...
    ap_fixed<27,8> layer2_dot_1 = (_tmp4994 + _tmp4995);
    ap_fixed<27,8> layer2_bias_0 = (layer2_dot_0 + (ap_fixed<27,1>) (bias2_0));
    ap_fixed<27,8> layer2_bias_1 = (layer2_dot_1 + (ap_fixed<27,1>) (bias2_1));
    ap_fixed<27,8> layer2_0 = (layer2_bias_0);
    ap_fixed<27,8> layer2_1 = (layer2_bias_1);
    _result[0] = layer2_0;
    _result[1] = layer2_1;
}
```

quantization
directly synthesized
Summary of Results: Mixed Fixed-Point Quantization of NNs

target error: $1 \times 10^{-3}$, max precision: 32-bit, TO: 5 hours

<table>
<thead>
<tr>
<th>#benchmarks</th>
<th>#params</th>
<th>analysis time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Daisy</td>
<td>Aster</td>
</tr>
<tr>
<td>mid-sized (14)</td>
<td>60 - 3920</td>
<td></td>
</tr>
<tr>
<td>large (4)</td>
<td>12K - 44.5K</td>
<td></td>
</tr>
</tbody>
</table>
### Summary of Results: Mixed Fixed-Point Quantization of NNs

target error: $1e^{-3}$, max precision: 32-bit, **TO**: 5 hours

<table>
<thead>
<tr>
<th>#benchmarks</th>
<th>#params</th>
<th>analysis time Daisy</th>
<th>analysis time Aster</th>
</tr>
</thead>
<tbody>
<tr>
<td>mid-sized</td>
<td>60 - 3920</td>
<td>4s - 2h 46m 20s</td>
<td>2s - 50s</td>
</tr>
<tr>
<td>(14)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>large</td>
<td>12K - 44.5K</td>
<td>TO</td>
<td>12m 7s - 3h 49m 31s</td>
</tr>
<tr>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary of Results: Mixed Fixed-Point Quantization of NNs

<table>
<thead>
<tr>
<th>#benchmarks</th>
<th>#params</th>
<th>analysis time</th>
<th>latency (clock-cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Daisy</td>
<td>Aster</td>
</tr>
<tr>
<td>mid-sized</td>
<td>14</td>
<td>60 - 3920</td>
<td>4s - 2h 46m 20s</td>
</tr>
<tr>
<td>(14)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>large</td>
<td>4</td>
<td>12K - 44.5K</td>
<td>TO</td>
</tr>
<tr>
<td>(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

target error: $1e^{-3}$, max precision: 32-bit, TO: 5 hours
**Summary of Results: Mixed Fixed-Point Quantization of NNs**

Target error: $1e-3$, max precision: 32-bit, **TO**: 5 hours

<table>
<thead>
<tr>
<th>#benchmarks</th>
<th>#params</th>
<th>analysis time</th>
<th>latency (clock-cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mid-sized</td>
<td>60 – 3920</td>
<td>Daisy: 4s – 2h 46m 20s</td>
<td>Aster: 2s – 50s</td>
</tr>
<tr>
<td>large</td>
<td>12K – 44.5K</td>
<td>TO</td>
<td></td>
</tr>
</tbody>
</table>

Aster is more precise than Daisy — Daisy reports 5 infeasibility, Aster reports 3!
**Takeaways**

- Specializing optimization in application contexts can be beneficial.
- Optimization with linearizations and abstractions is effective for NNs.
- An automated NN quantizer: Aster generates sound quantized code that can be directly synthesized in Xilinx.
  - is precise and scalable.
Expanding the Horizons of Finite-Precision Analysis

Accuracy Analysis

- iFM '19
- EMSOFT '18
- Probabilistic Analysis

- Static + Dynamic Analysis
- TACAS '21
- Static + Dynamic Analysis

Thesis Contributions

Optimization

- EMSOFT '23
- NN Quantization

worst-case error analysis for small programs

- Daisy
- FLUCTUAT
- Rosa
- FPTaylor
- PRECiSA
- ...

worst-case tuning for small (floating-point) programs

- Daisy
- FPTuner
Future Research Directions

• Scalable Accuracy Analysis
  - considering probabilistic inputs
  - by combining static, dynamic analysis and machine learning techniques
Future Research Directions

• Scalable Accuracy Analysis
  - considering probabilistic inputs
  - by combining static, dynamic analysis and machine learning techniques

• Scalable Optimization
  - considering probabilistic inputs
  - specialize in other application contexts
Future Research Directions

• Scalable Accuracy Analysis
  - considering probabilistic inputs
  - by combining static, dynamic analysis and machine learning techniques

• Scalable Optimization
  - considering probabilistic inputs
  - specialize in other application contexts

• Finite-precision in the context of
  - heterogeneous HPC systems

... and others!
Thank You
For Your Attention!
BACKUP SLIDES
Probabilistic Error Analysis
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    require (-15.0 <= x, y, z <= 15.0)
    val res = -x*y - 2*y*z - x - z
    return res
} ensuring (error <= 1.5e-4, 0.85)

resource efficient but has probabilistic error specification:
<4 x \epsilon_m, 0.1>, <\epsilon_m, 0.9>
Scenario 2: Approximate Hardware Specifications

```scala
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
  require (-15.0 <= x, y, z <= 15.0)
  val res = -x*y - 2*y*z - x - z
  return res
}
```

ensuring (error <= 1.5e-4, 0.85)

resource efficient but has probabilistic error specification:

\(<4 \times \epsilon_m, 0.1>, <\epsilon_m, 0.9>\)

The worst-case assumes \(4 \times \epsilon_m\) error occurs always!
Probabilistic Error Analysis

For each arithmetic operation:

step 1: compute range for intermediate value starting with initial distributions
For each arithmetic operation:

step 1: compute range for intermediate value starting with initial distributions

step 2: propagate existing errors — probabilistic affine arithmetic
Probabilistic Error Analysis

For each arithmetic operation:

- **step 1**: compute range for intermediate value starting with initial distributions
- **step 2**: propagate existing errors
- **step 3**: compute new errors — as multiple fresh noise terms
Sound Mixed Fixed-Point Quantization
Overview: Reduction to MILP

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{\text{dot}} + \gamma_i^{\text{bias}} + \gamma_i^{\alpha}$

subject to:

$\epsilon_n \leq \epsilon_{\text{target}}$

$I_{i}^{\text{op}} \geq \text{intBits} \left( R_{i}^{\text{op}} + \epsilon_i \right)$
Overview: Reduction to MILP

\[
\text{minimize: } \gamma = \sum_{i=1}^{n} \gamma_i^{\text{dot}} + \gamma_i^{\text{bias}} + \gamma_i^{\alpha}
\]

subject to:

\[
\epsilon_n \leq \epsilon_{\text{target}}
\]

\[
I_{i}^{\text{op}} \geq \text{intBits} \left( R_{i}^{\text{op}} + \epsilon_i \right)
\]

over-approximate integer bits separately using interval arithmetic

fixed-point representation
Overview: Reduction to MILP

\[
\text{minimize: } \gamma = \sum_{i=1}^{n} \gamma_i^{\text{dot}} + \gamma_i^{\text{bias}} + \gamma_i^{\alpha} \\
\text{subject to: } \\
\epsilon_n \leq \epsilon_{\text{target}} \\
I_i^{op} \geq \text{intBits} \left( R_i^{op} + \epsilon_i \right)
\]

- over-approximate integer bits separately
Linearization Step 2: Exact Linearization of Cost

minimize: \( \gamma = \sum_{i=1}^{n} \gamma_i^{\text{dot}} + \gamma_i^{\text{bias}} + \gamma_i^{\alpha} \)

subject to:

\[ \epsilon_n \leq \epsilon_{\text{target}} \]

\[ I_{i}^{op} \geq \text{intBits} \left( R_{i}^{op} + \epsilon_i \right) \]

\( \gamma_i^{\text{bias}} = \max(\pi_i^{\text{dot}}, \pi_i^{\text{bias}}) \)

non-linear function
Linearization Step 2: Exact Linearization of Cost

minimize: \( \gamma = \sum_{i=1}^{n} \gamma_i^{\text{dot}} + \gamma_i^{\text{bias}} + \gamma_i^\alpha \)

subject to:

\( \epsilon_n \leq \epsilon_{\text{target}} \)

\( I_i^{op} \geq \text{intBits} \left( R_i^{op} + \epsilon_i \right) \)

\( \gamma_i^{\text{bias}} = \max(\pi_i^{\text{dot}}, \pi_i^{\text{bias}}) \)

\( c1: \gamma_i^{\text{bias}} \geq \pi_i^{\text{dot}} \)

\( c2: \gamma_i^{\text{bias}} \geq \pi_i^{\text{bias}} \)
Overview: Reduction to MILP

minimize: $\gamma = \sum_{i=1}^{n} \gamma_i^{\text{dot}} + \gamma_i^{\text{bias}} + \gamma_i^{\alpha}$

subject to:

$\epsilon_n \leq \epsilon_{\text{target}}$

$I_i^{op} \geq \text{intBits} \left( R_i^{op} + \epsilon_i \right)$

- over-approximate integer bits separately
- linearize bias cost and error constraint exactly
Overview: Reduction to MILP

Linearized Problem

minimize: \( \gamma = \sum_{i=1}^{n} \gamma_{i}^{\text{dot}} + \gamma_{i}^{\text{bias}} + \gamma_{i}^{\alpha} \)

subject to:

\( \epsilon_n \leq \epsilon_{\text{target}} \)

\( I_{i}^{\text{op}} \geq \text{intBits} \left( R_{i}^{\text{op}} + \epsilon_{i} \right) \)

• over-approximate integer bits separately
• linearize bias cost and error constraint exactly
• abstract dot product
Optimizing Fractional Bits for Dot and Bias Products

**Linearized Problem**

minimize: \( \gamma = \sum_{i=1}^{n} \gamma_i^{dot} + \gamma_i^{bias} + \gamma_i^{\alpha} \)

subject to:

\( \epsilon_n \leq \epsilon_{\text{target}} \)

\( I_i^{op} \geq \text{intBits} \left( R_i^{op} + \epsilon_i \right) \)
Optimizing Fractional Bits for Dot and Bias Products

**Linearized Problem**

minimize: \( \gamma = \sum_{i=1}^{n} \gamma_i^{\text{dot}} + \gamma_i^{\text{bias}} + \gamma_i^{\alpha} \)

subject to:

\[ \varepsilon_n \leq \varepsilon_{\text{target}} \]

\[ I_i^{op} \geq \text{intBits} \left( R_i^{op} + \varepsilon_i \right) \]

**MILP solver**

**sign bit**

**integer bits**

**fractional bits**
Generate Full-Fledged Quantized Implementation

• reduced to MILP problem
• optimized fractional bits for dot and bias results assuming precision of weights
• assigning correctly rounded precision for all variables and constants
Generate Full-Fledged Quantized Implementation

- reduced to MILP problem
- optimized fractional bits for dot and bias results assuming precision of weights
- assigning correctly rounded precision for all variables and constants

using fixed-point sum of products by constants*

* A Correctly-Rounded Fixed-Point-Arithmetic DotProduct Algorithm, Sylvie Boldo, Diane Gallois-Wong, and Thibault Hilaire, ARITH 2020
Discrete Choice in the Presence of Numerical Uncertainties
Scenario 1: Wrong Discrete Decisions

```python
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
    x:= gaussian(4.0, 0.5)
    y:= gaussian(4.75, 2.0)
    z:= gaussian(4.8, 2.5)
    val res = -x*y - 2*y*z - x - z
    if (res <= 0.0)
        raise_alarm()
    else
        do_nothing()
}
```

real-valued execution
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
    require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)

    x:= gaussian(4.0, 0.5)
    y:= gaussian(4.75, 2.0)
    z:= gaussian(4.8, 2.5)

    val res = -x*y - 2*y*z - x - z

    if (res <= 0.0)
        raise_alarm()
    else
        do_nothing()
}
Scenario 1: Wrong Discrete Decisions

def controller(x:Float32, y:Float32, z:Float32): Float32 = {
    require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
    
    x:= gaussian(4.0, 0.5)
    y:= gaussian(4.75, 2.0)
    z:= gaussian(4.8, 2.5)

    val res = -x*y - 2*y*z - x - z

    if (res <= 0.0)
        raise_alarm()
    else
        do_nothing()
}

Program always takes the wrong decision in the worst-case!
def controller(x: Float32, y: Float32, z: Float32): Float32 = {
  require (0.0 <= x <= 4.6 && 0.0 <= y, z <= 10.0)
  x := gaussian(4.0, 0.5)
  y := gaussian(4.75, 2.0)
  z := gaussian(4.8, 2.5)
  val res = -x*y - 2*y*z - x - z
  if (res <= 0.0)
    raise_alarm()
  else
    do_nothing()
}
scalability of probabilistic analysis for numerical programs

<table>
<thead>
<tr>
<th>#benchmark</th>
<th>#ops</th>
<th>#vars</th>
<th>uniform</th>
<th>gaussian</th>
<th>over-approx.</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>4–25</td>
<td>1–9</td>
<td>48s–7m 28s</td>
<td>42s–11m 1s</td>
<td>~e–4(7)*</td>
</tr>
</tbody>
</table>

* compared our analysis with symbolic inference

https://doi.org/10.5281/zenodo.8042198
scalability of probabilistic analysis for numerical programs

<table>
<thead>
<tr>
<th>#benchmark</th>
<th>#ops</th>
<th>#vars</th>
<th>uniform</th>
<th>gaussian</th>
<th>over-approx.</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>4–25</td>
<td>1–9</td>
<td>48s–7m 28s</td>
<td>42s–11m 1s</td>
<td>~e–4(7)*</td>
</tr>
</tbody>
</table>

* compared our analysis with symbolic inference

[link](https://doi.org/10.5281/zenodo.8042198)

Sound and precise WPPs for small programs with different distributions
Two-Phase Approach for Conditional Floating-Point Verification
void linpack(double cray, double init[4])

# define N 5
# define LDA (N + 1)
double *a, a_max, *b, b_max, eps, ops, *resid, resid_max, residn, *rhs;
int i, info,*ipvt, j, job;
double t1, t2, time[6], total, *x;
...
dgesl(a, LDA, N, ipvt, b, job);
...
a = r8mat_gen(LDA, N, init);
...
# undef LDA
# undef N
}

void dgesl(double a[], int lda, int n, int ipvt[], double b[], int job ) {
int k, l;
double t;
if ( job == 0 ) {
for ( k = 1; k <= n-1; k++ ) {
...  
daxpy(n-k, t, a+k+(k-1)*lda, 1, b+k, 1);
}
for ( k = n; 1 <= k; k-- ) {
...  
daxpy(k-1, t, a+0+(k-1)*lda, 1, b, 1);
}
} else {
for ( k = 1; k <= n; k++ ) {
t = ddot(k-1, a+0+(k-1)*lda, 1, b);
b[k-1] = ( b[k-1] - t ) / a[k-1+(k-1)*lda];
}
for ( k = n-1; 1 <= k; k-- ) {
b[k-1] = b[k-1] + ddot(n-k, a+k+(k-1)*lda, 1, b+k);
    l = ipvt[k-1];
    if ( l != k ) {
...  
}
}
}
return;
}
double ddot(int n, double dx[], int incx, double dy[], int incy ) {
double dtemp = 0.0;
int i, ix, it, m;
if ( n <= 0 ) {
return dtemp;
}
if ( incx != 1 || incy != 1 ) {
    if ( 0 <= incx ) {
ix = 0;
} else {
ix = ( -n + 1) * incx;
}
    if ( 0 <= incy ) {
iy = 0;
} else {
iy = ( -n + 1) * incy;
}
    dtemp = incr(dtemp, dx, dy, n, ix, iy, incx, incy);
} else {
    dtemp = dot(dx, dy);
}
return dtemp;
}

double r8mat_gen(int lda, int n, int init[4]) {
double *a;
int I, j;
a = ( double * ) malloc( lda * n * sizeof(double) );
for ( j = 1; j <= n; j++ ) {
    for ( i = 1; i <= n; i++ ) {
a[i-1+(j-1)*lda] = r8_random(init) - 0.5;
    }
}
return a;
...
void linpack(double cray, double init[4])

#define N 5
#define LDA (N + 1)
double *a, a_max, *b, b_max, eps, ops, *resid, resid_max, residn, *rhs;

int i, info, *ipvt, j, job;
double t1, t2, time[6], total, *x;

... dgesl(a, LDA, N, ipvt, b, job);
...
a = r8mat_gen(LDA, N, init);
...
#undef LDA
#undef N
}

void dgesl(double a[], int lda, int n, int ipvt[], double b[], int job) {
int k, l;
double t;
if (job == 0) {
    for (k = 1; k <= n - 1; k++) {
        ...
daxpy(n-k, t, a+k+(k-1)*lda, 1, b+k, 1);
    }
    for (k = n; 1 <= k; k--) {
        ...
daxpy(k-1, t, a+0+(k-1)*lda, 1, b, 1);
    }
} else {
    for (k = 1; k <= n; k++) {
        t = ddot(k-1, a+0+(k-1)*lda, 1, b);
        b[k-1] = (b[k-1] - t) / a[k-1+(k-1)*lda];
    }
    for (k = n-1; 1 <= k; k--) {
        b[k-1] = b[k-1] + ddot(n-k, a+k+(k-1)*lda, 1, b+k, 1);
        l = ipvt[k-1];
        if (l != k) {
            ...
        }
    }
}
return;

double ddot(int n, double dx[], int incx, double dy[], int incy) {
    double dtemp = 0.0;
    int i, ix, it, m;
    if (n <= 0) {
        return dtemp;
    }
    if (incx != 1 || incy != 1) {
        if (0 <= incx) {
            ix = 0;
        } else {
            ix = (-n + 1) * incx;
        }
        if (0 <= incy) {
            iy = 0;
        } else {
            iy = (-n + 1) * incy;
        }
        dtemp = incr(dtemp, dx, dy, n, ix, iy, incx, incy);
    } else {
        dtemp = dot(dx, dy);
    }
    return dtemp;
}

double *r8mat_gen(int lda, int n, int init[4]) {
double *a;
int I, j;
a = (double *) malloc(lda * n * sizeof(double));
    for (j = 1; j <= n; j++) {
        for (i = 1; i <= n; i++) {
            a[i-1+(j-1)*lda] = r8_random(init) - 0.5;
        }
    }
return a;
...

Numerical analyzers do not scale!
void linpack(double cray, double init[4])

double dot(double dx[4], double dy[4])

double ddot(int n, double dx[], int incx, double dy[], int incy)
void linpack(double cray, double init[4])

double dot(double dx[4], double dy[4])

Does not require complex numerical analysis

phase I: whole program analysis

Static / Dynamic Analysis
+ scales well
- imprecise numerical analysis

kernel input ranges

numerically interesting: numerical kernel
void linpack(double cray, double init[4])

... 

dgesl(a, LDA, N, ipvt, b, job);
...

a = r8mat_gen(LDA, N, init);
...

#undef LDA
#undef N
}

void dgesl(double a[], int lda, int n, int ipvt[], double b[], int job)

int k, l;

double t;

if (job == 0)

for (k = 1; k <= n-1; k++)

...

daxpy(n-k, t, a+k+(k-1)*lda, 1, b+k, 1);
...

for (k = n; 1 <= k; k--)

...

daxpy(k-1, t, a+0+(k-1)*lda, 1, b, 1);
...

} else

for (k = 1; k <= n; k++)

t = ddot(k-1, a+0+(k-1)*lda, 1, b, 1);

b[k-1] = (b[k-1] - t) / a[k-1+(k-1)*lda];
...

for (k = n-1; 1 <= k; k--)

b[k-1] = b[k-1] + ddot(n-k, a+k+(k-1)*lda, 1, b+k, 1);

l = ipvt[k-1];

if (l != k) {
...

} 

return;

double ddot(int n, double dx[], int incx, double dy[], int incy)

double dot(double dx[4], double dy[4])

double ddot(double dx[], int incx, double dy[], int incy)

double dot(double dx[], int incx, double dy[], int incy)

double dot(double dx[], int incx, double dy[], int incy)

... 

r8_random(init) - 0.5

numercially interesting: numerical kernel 

double dot(double dx[4], double dy[4])

phase I: whole program analysis 

Static / Dynamic Analysis

+ scales well
- imprecise numerical analysis

kernel input ranges 

phase II: numerical analysis 

Numerical Analysis

+ precise numerical analysis
- not scalable

no error / NaN, \infty, cancellation warnings
### Summary: (Conditional) Floating-Point Verification

<table>
<thead>
<tr>
<th>#benchmark</th>
<th>#kernel</th>
<th>lang</th>
<th>#in</th>
<th>LOC</th>
<th>(conditional) verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>24</td>
<td>C, C++</td>
<td>1–24</td>
<td>31–2187</td>
<td>14 verified, 10 warnings</td>
</tr>
</tbody>
</table>

(2 cancellation, 8 NaN/∞)

[https://doi.org/10.5281/zenodo.8043359](https://doi.org/10.5281/zenodo.8043359)
(Conditional) verification of floating-point kernels ‘hidden’ in large applications