Efficient floating point precision tuning for approximate computing

Nhut-Minh Ho\textsuperscript{1}, Elavarasi Manogaran\textsuperscript{1}, Weng-Fai Wong\textsuperscript{1} and Asha Anoosheh\textsuperscript{2}

\textsuperscript{1} National Univ. of Singapore, Singapore
\textsuperscript{2} Univ. of California, Berkeley, U.S.A.
Energy concern in computing

Top 500 supercomputers cost $400 million/year for energy consumption:

Green computing: "FLOPS-per-Watt"

Other devices

Estimated U.S. data center electricity consumption by market segment

Reduce application energy consumption
Error tolerant applications

• Big data analytics
• Media data processing/classification
• Simulations
• Non-critical functions in each program
• ...

Approximate Computing
Approximate computing

• Sacrifice **accuracy** for **performance** => also increase energy efficiency

• Various approaches:
  • Hardware:
    • Low-power circuit with uncertainty
    • Fine-grain floating-point bitwidth hardware
  • Software:
    • Task skipping: loop perforation
    • **Floating point precision tuning**
Floating point numbers

• Appear almost in every computer program

```c
1. float a = 999.0;
2. float b = 0.0001;
3. for (int i = 0; i < 10000; i++){
   4.     a += b;
5. }
```

Expected : a = 1000.0            Actual : a \approx 1000.220703

\[
\text{Error} \approx \frac{1000.220703 - 1000}{1000} \approx 2.2 \times 10^{-4}
\]
Precision tuning, previous work

• Arbitrary-precision fixed point tuning for DSP programs
  • Many techniques: search-based, error analysis based.
  • None of them can scale to real-world floating-point programs nowadays.

• 2-type floating point precision tuning
  • Search for variables can be converted: \textit{double} \rightarrow \textit{float}.
  • Can analyze real-world applications.
  • Cannot work on finer grained precision in different architecture without modification.
2-type floating point is enough?

- Modern CPU architecture: sufficient.
- FPGA (Field-programmable gate array) prototype showed advantages of using finer grain floating point unit.
- Nvidia’s GPU (graphics processing unit) newest architecture supports half-precision.
Motivation

Current techniques for x86 applications:
• Moderately fast
• Limited precision support (float & double)

Current techniques for FPGA community:
• Slow when processing complex applications
• Fine-grained precision (any number of bits)
• HLS paves the way for big and complex applications on FPGA

Our goal:
• Fast
• Scale well
• Fine-grained precision support
• The result can be used on HLS process, as well as migrated to GPU
Overview

Input

Multiple-precision program
User specified accuracy constraints

Searching

Arbitrary-precision searching
Result refinement (optional)

Output

Final precision requirement
Tuned program(s)
High-level Synthesis

Input Searching Output
Use Multiple Precision Floating-Point Reliable (MPFR) library to create the Multiple-precision program for searching.
Arbitrary-precision tuning

Multiple-precision program

User specified accuracy constraints

Input

Arbitrary-precision searching

Searching
**Arbitrary-precision tuning**

- Lower Bound - Upper Bound - Solution

**Region 1:** $\epsilon \leq \text{Err}$

**Region 2:** $\text{Err} > \epsilon$

---

**function()**{
    mpfr_t x1;
    mpfr_t x2 = .... ;
    ....
    mpfr_t x5 = ....;
}

- User defined accuracy constraint $\epsilon$
- **Err**: error of the output

---

**Arbitrary-precision tuning**

- Lower Bound - Upper Bound - Solution

**Region 1:** $\text{Err} \leq \epsilon$

**Region 2:** $\text{Err} > \epsilon$

---

**Arbitrary-precision tuning**

- Lower Bound - Upper Bound - Solution

**Region 1:** $\text{Err} \leq \epsilon$

**Region 2:** $\text{Err} > \epsilon$

---

**Arbitrary-precision tuning**

- Lower Bound - Upper Bound - Solution

**Region 1:** $\text{Err} \leq \epsilon$

**Region 2:** $\text{Err} > \epsilon$
Step 1: Isolated downward

Find the minimum possible precision for each variable while keeping others at highest precision.

Binary search + parallelism at variable level.
Run-time $\leq \log_2(53 - 4)$

Step 1 result usually causes $Err > \epsilon$
Step 2: Grouped upward

From Step 1 result, try to get back to the solution

Strategy: Get back to the point where $Err \leq \epsilon$ as fast as possible.
Step 2: Grouped upward

Strategy: Get back to the point where $Err \leq \epsilon$ as fast as possible.

Greedily shift the blue line upward:
• 1 variable at a time: good but stuck when no variables can reduce $Err$

The effect may not propagate to the output => no change in $Err$
Step 2: Grouped upward (cont)

• Our approach: “grey-box” distributed search

When increasing precision of a variable, should increase all other variables in the path from it to the output.

Increase precision of the whole dependence group, not single variable. 
{x1,x3,x5}, {x2,x3,x5}, {x3,x5}
Step 2: Grouped upward

- Shift Step1 result upward by competition between groups of variables.
- Group reduces most error will win 1 bit for all members.
- Parallelize at group level (5 groups)

Step 2 gives an acceptable result higher than the solution.
The iterative process

- Reuse step 1 to find another result closer to the solution.
- Then reuse step 2 to move upward to the solution.
- The algorithm converges after a few epochs.
Result

• Quality: \( \approx 6\% \) fewer in number of bits compared to an established algorithm \textit{Max-1} (for small programs).

• Complexity:
  • \( T_{mpfr} \) = time to run the input program (multiple-precision version):
  • Average: \( 25.9 \times T_{mpfr} \), for programs have 10-45 variables
  • Large program (417 variables): \( 110.5 \times T_{mpfr} \)
Compare to state-of-the-art

• *Precimonious* searches for the mixed use of 2 types: *float* and *double*.

• The fine-grain results are mapped to 2 types for comparison.
Number of double variables required for $\epsilon = 10^{-6}$

PrecB: tuned by Precimonious 2016 [7].

Ours: tuned by our tool chain
### Speedup (%) compared to the original version

<table>
<thead>
<tr>
<th>Function</th>
<th>PrecB $10^{-4}$</th>
<th>Ours $10^{-4}$</th>
<th>PrecB $10^{-4}$</th>
<th>Ours $10^{-4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>cg</td>
<td>9.2</td>
<td>15.4</td>
<td>0</td>
<td>4.8</td>
</tr>
<tr>
<td>polyroots</td>
<td>0</td>
<td>4.8</td>
<td>41.5</td>
<td>49.5</td>
</tr>
<tr>
<td>sum</td>
<td>3.4</td>
<td>3.4</td>
<td>4.8</td>
<td>4.8</td>
</tr>
</tbody>
</table>

$$\epsilon = 10^{-4}$$
Aggregated result across 11 programs

2,666 floating-point variables across 4 error thresholds ($\epsilon$)

% variable can be in half-precision (11 bits):
- $\approx 66\%$ for $10^{-4}$
- $\approx 52\%$ for $10^{-6}$
- $\approx 38\%$ for $10^{-8}$
- $\approx 31\%$ for $10^{-10}$
Result refinement

Multiple-precision program

User specified accuracy constraints

Input

Arbitrary-precision searching

Searching

Result refinement (optional)
Input variation problem

\[
\text{function (float}_32 \text{ input)}\{
\text{float}_32 \text{ output} = \text{input} \times \text{input};
\}
\]

Input = 1.2, \(\epsilon = 10^{-5}\)

\[
\text{function (float}_16 \text{ input)}\{
\text{float}_25 \text{ output} = \text{input} \times \text{input};
\}
\]

Input = 1.2, Err \leq 10^{-5}

Input = 1000.0, Err = ?

Input = 0.01, Err = ?

Statistically guided refinement for input \(\in [0.01; 1000]\)

Training set of M seeds for random number generator in range [0.01; 1000]
Statistically guided refinement

function(float_32 input){
    float_32 output = input * input;
}

worst_seed = 1, $\epsilon = 10^{-5}$

function(float_20 input){
    float_18 output = input * input;
}

Average Err over 100 seeds =?
worst_seed = ?

Average Err = $5.0 \times 10^{-3}$
worst_seed = 43 causes Err = $10^{-2}$

worst_seed = 43, $\epsilon = 10^{-5}$

Average Err over 100 seeds = ?
worst_seed = ?

function(float_16 input){
    float_22 output = input * input;
}

1 seed number = 1 representative input
Training set of 100 seeds
Result on DSP programs, target $\epsilon = 10^{-5}$

Before, average $= 2.3 \times 10^{-4}$, max $= 3.4 \times 10^{-2}$

After, average $= 4.9 \times 10^{-6}$, max $= 2.5 \times 10^{-4}$
Arbitrary precision version on Vivado HLS

Accuracy constraint: 50-60dB

Average resource consumption & execution time (normalized) of 6 programs with different precision assigned on Vivado HLS
Conclusion

• Our algorithm can scale to large and long running programs:
  • E.g. $T_{mpfr} = 20$ mins, number of variables = number of MPI threads $\leq 45$
    $\Rightarrow$ Expected searching $= 26 \times 20 \approx 520$ mins.

• We use program’s high-level dependence information to guide the distributed search process.

• Input variation problem can be mitigated with our statistics guided refinement process.

• This tool paves the way for using HLS with arbitrary precision on large programs.
Thanks for listening

Q&A
Link to github repository

https://github.com/minhnhn2910/fpPrecisionTuning