Rigorous Floating-point Mixed-precision Tuning

Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Ganesh Gopalakrishnan, Zvonimir Rakamarić, and Alexey Solovyev

University of Utah, Salt Lake City, USA

This work was supported in part by NSF awards CCCF 1531140, CCF 1643056, and CCF 1552975.
Efficiency/Precision Trade-off

Energy Efficient (lower bit-width)

Accurate (higher bit-width)

Higher Bit-width
e.g., 64- or 128-bit

Lower Bit-width
e.g., 16- or 32-bit

Photo courtesy to cdn.wcfctech.com, dreamstime.com, technologystudent.com, ucsdnews.ucsd.edu and blog.technospares.co.uk.
Today’s Choices

• All 32-bit or all 64-bit

\[
E = x - (x + y)
\]

or

\[
E = x - (x + y)
\]
Can we have both?

Efficiency and Precision
Yes, often!

• Use mixed-precision!

\[ E = x - (x + y) \]
Mixed-precision: a Solution for the Trade-off

- **Key thrust**: Selecting high precision for some sub-expressions is often better than selecting high precision for all.

- **Objective**: find out the mappings of operators to bit-widths that satisfy the desired precision level and result in the optimal program efficiency.

- Need for high program efficiency
  - Example: The ExaScale is constrained by an energy budget [Bergman. `08]

- Need for rigor. Examples:
  - Missed collisions in the simulations in the Large Hadron Collider [Bailey. `13]
  - Intel’s update for the trigonometric library
Previous Efforts in Mixed-precision Tuning

- Manual tuning. Examples:
  - Linear system solver [Buttari. `07]
  - Arc length calculation [Bailey. `12]

- Execution-based automatic tuning.
  - Tools: Precimonious [Rubio-González. `13], CRAFT [Lam. `13], etc.
Main Issues of Execution-based Tuning

• Tools: Precimonious [Rubio-González. `13], CRAFT [Lam. `13], etc.

• Main Issue: the error analysis only guarantees for a few inputs. No guarantees for all the other billions!
FPTuner: First Rigorous Mixed-precision Tuning Method

- Rigorous error analysis
- The desired precision level is guarantees for ALL inputs!
Key Challenges of Rigorous Mixed-Precision Tuning

- Need rigorous error estimation methods for **interval inputs**
  - Tight over-approximations are essential
  - Must handle a variety of expression types (non-linear, transcendental, ..)

- Need flexible ways of controlling precision allocation
  - “Gang” operators: If one set at high, all related (vector) must be similarly set
  - Constrain the number of high-precision, casting, etc.
Main Problem of Rigorous Mixed-Precision Tuning

• The act of setting precision affects the high-to-low precision jumps (casts)
  • Where such jumps occur AND how much of a jump it is!
• Each such jump is a rounding step
• But that in turn affects error estimates!
  • Need to break this meta-circularity
Main Ideas Behind FPTuner

• Floating-point error analysis
  • Build an error function which reflects precision selections
  • **Our solution**: Symbolic Taylor forms [Solovyev. ‘15]

• Program efficiency analysis
  • Similarly, the efficiency function needs to reflect precision selections
  • **Our solution**: weighted sum of the lower bit-width operators and constraint of the # of type casts

• Optimization for considering both the efficiency and the error functions
  • **Our solution**: convert the tuning problem to an optimization problem
Main Contributions

• The first rigorous mixed-precision tuning method
  • Converting the tuning problem to an optimization problem
    • With a set of knobs to guide precision selection
• FPTuner: a prototype tool implementation
  • Release on Github: https://github.com/soarlab/FPTuner
• Report the results of precision tuning with FPTuner in terms of performance and energy measurements
  • Point out delicate compilation issues
Roadmap

- Background on floating-point and rounding error
- Our method – rigorous mixed-precision tuning
- Experimental results
- Related work and conclusions
Roadmap

• Background on floating-point and rounding error
• Our method – rigorous mixed-precision tuning
• Experimental results
• Related work and conclusions
Rounding – the Source of Floating-point Imprecision

• A floating-point domain is a subset of real numbers
• Each real number is “rounded” to a floating-point number
Rounding – the Source of Floating-point Imprecision

- Binade: a group of floating-point number having the same exponent
  - Numbers in the same binade are equally distributed
- Any two binades have the same amount of numbers
- The ranges of any two consecutive binades are in a factor of 2
Rounding – the Source of Floating-point Imprecision

- The max errors of rounding to different binades are also different
- Error \((\tilde{x} - x)\) is bounded by the binade/the magnitude of \(x\): \((\tilde{x} - x) = x \times e_x\)
  - \(\tilde{x} = x \times (1 + e_x)\)
  - \(|e_x| \leq \epsilon\) where \(\epsilon = 2^{-53}\) under 64-bit
Rounding – the Source of Floating-point Imprecision

- The lower bit-width domain is a subset of the higher bit-width
- $\tilde{x} = x \times (1 + \epsilon)$ applies with a different machine epsilon $\epsilon = 2^{-24}$
Rounding – the Source of Floating-point Imprecision

- Rounding a higher bit-with value to the lower introduces an error
- No error for rounding from the lower to the higher bit-width
Rigorous Rounding Error Estimation

- Model floating-point computation of $E = x - (x + y)$ with Reals

$$\tilde{E} = \left( \left( (x \cdot (1 + e_0)) + (y \cdot (1 + e_1)) \right) \cdot (1 + e_2) \right) - (x \cdot (1 + e_3)) \right) \cdot (1 + e_4)$$

- $|e_0| \leq \epsilon_0$, $|e_1| \leq \epsilon_1$, $|e_2| \leq \epsilon_2$, $|e_3| \leq \epsilon_3$, $|e_4| \leq \epsilon_4$

- Rounding error: $|\tilde{E} - E|$
Rigorous Rounding Error Estimation

• Interval arithmetic [Melquiond. `06]
• Affine arithmetic [Darulova. `12]
• Affine arithmetic + SMT [Darulova. `14]
• Semidefinite programming [Magron. `16]
• **Symbolic Taylor Form** [Solovyev. `15]
  • Provides the tightest error bounds of all these tools
  • Handles a large classes of expressions (e.g., non-linear and transcendental)
Roadmap

• Background on floating-point and rounding error
• **Our method – rigorous mixed-precision tuning**
• Experimental results
• Related work and conclusions
Uni-precision Case

• The relation between $E$ and its floating-point representation $\tilde{E}$ can be described in a Taylor form [Solovyev. `15]
  
  • $e_0 \sim e_4$ serve as the noise variables
  • $M_2$ summarizes the second and the higher order terms

\[
\tilde{E} = E + \frac{\partial \tilde{E}}{\partial e_0} (0) \times e_0 + \cdots + \frac{\partial \tilde{E}}{\partial e_4} (0) \times e_4 + M_2
\]
Uni-precision Case

• The noise/error variables are bounded by the machine epsilons

\[ |e_0| \leq \varepsilon_0, |e_1| \leq \varepsilon_1, |e_2| \leq \varepsilon_2, |e_3| \leq \varepsilon_3, |e_4| \leq \varepsilon_4 \]

\[ |\bar{E} - E| \leq \left| \frac{\partial \bar{E}}{\partial e_0}(0) \right| \times \varepsilon_0 + \cdots + \left| \frac{\partial \bar{E}}{\partial e_4}(0) \right| \times \varepsilon_4 + M_2 \]
Uni-precision Case

- **Gelpia**: our global optimization tool
- With global optimization, calculate constant $U_0$:

$$\forall x, y. \left| \frac{\partial \tilde{E}}{\partial e_0}(0) \right| \times \varepsilon_0 \leq U_0$$

$$|\tilde{E} - E| \leq \left| \frac{\partial \tilde{E}}{\partial e_0}(0) \right| \times \varepsilon_0 + \cdots + \left| \frac{\partial \tilde{E}}{\partial e_4}(0) \right| \times \varepsilon_4 + M_2$$
Uni-precision Case

• Omit $M_2$ in this presentation. Our paper offers the details of handling it.
• Each term bounds the error “contributed” by an operator

Machine epsilon (bit-width dependent)

$$|\tilde{E} - E| \leq U_0 \times \epsilon_0 + \cdots + U_4 \times \epsilon_4$$

• Bit-width independent constant
Uni-precision Case

• All operators are 32-bit:

\[ \epsilon_0 = \epsilon_1 = \epsilon_2 = \epsilon_3 = \epsilon_4 = 2^{-24} (\epsilon_{32\text{-bit}}) \]

\[ |\tilde{E} - E| \leq U_0 \times \epsilon_{32\text{-bit}} + \cdots + U_4 \times \epsilon_{32\text{-bit}} \]
Uni-precision Case

- All operators are 64-bit:

\[ \epsilon_0 = \epsilon_1 = \epsilon_2 = \epsilon_3 = \epsilon_4 = 2^{-53} (\epsilon_{64-bit}) \]

\[ |\tilde{E} - E| \leq U_0 \times \epsilon_{64-bit} + \ldots + U_4 \times \epsilon_{64-bit} \]
Mixed-precision Case

• All operators are 64-bit:

\[ \epsilon_0 = \epsilon_1 = \epsilon_2 = \epsilon_3 = \epsilon_{64\text{–bit}} \quad \epsilon_4 = \epsilon_{32\text{–bit}} \]

\[ |\tilde{E} - E| \leq U_0 \times \epsilon_{64\text{–bit}} + \cdots + U_4 \times \epsilon_{32\text{–bit}} \]
Mixed-precision Case

• Replace machine epsilons with symbolic variables:

\[ s_0, s_1, s_2, s_3, s_4 \in \{\epsilon_{32\text{-bit}}, \epsilon_{64\text{-bit}}\} \]

\[ |\tilde{E} - E| \leq U_0 \times s_0 + \cdots + U_4 \times s_4 \]
Mixed-precision Case

• Mixed-precision derives type casts!

\[ s_0, s_1, s_2, s_3, s_4 \in \{ \epsilon_{32\text{-}bit}, \epsilon_{64\text{-}bit} \} \quad s_{t0}, s_{t1} = \epsilon_{32\text{-}bit} \]

\[ |\tilde{E} - E| \leq U_0 \times s_0 + \cdots + U_4 \times s_4 + U_{t0} \times s_{t0} + U_{t1} \times s_{t1} \]
Mixed-precision Case

• Different mixed-precision schemes derive different type casts!

\[ s_0, s_1, s_2, s_3, s_4 \in \{ \epsilon_{32\text{-}bit}, \epsilon_{64\text{-}bit} \} \quad s_{t2} = \epsilon_{32\text{-}bit} \]

\[ |\tilde{E} - E| \leq U_0 \times s_0 + \cdots + U_4 \times s_4 + U_{t2} \times s_{t2} + 0 \]
Mixed-precision Case

• Assume type casts are always there
• But some of them introduce zero errors:

\[ s_0, s_1, s_2, s_3, s_4 \in \{ \epsilon_{32\text{-bit}}, \epsilon_{64\text{-bit}} \} \quad s_{t0}, s_{t1}, s_{t2}, s_{t3} \in \{ 0, \epsilon_{32\text{-bit}} \} \]

\[
|\tilde{E} - E| \leq U_0 \times s_0 + \cdots + U_4 \times s_4 \\
\quad + U_{t0} \times s_{t0} + U_{t1} \times s_{t1} \\
\quad + U_{t2} \times s_{t2} + U_{t3} \times s_{t3}
\]
Mixed-precision Case

• Assume type casts are always there
• But some of them introduce zero errors:

\[ s_0, s_1, s_2, s_3, s_4 \in \{ \epsilon_{32-bit}, \epsilon_{64-bit} \} \]
\[ s_{t0}, s_{t1}, s_{t2}, s_{t3} \in \{ 0, \epsilon_{32-bit} \} \]

\[ s_{t0} = \begin{cases} 
\epsilon_{32-bit}, & \text{‘x’ is 64-bit and ‘-’ is 32-bit} \\
0, & \text{otherwise}
\end{cases} \]
Additional Constraints

• Limit the maximum number of type casts
  • Achieve by encoding the symbolic variables with pseudo Boolean variables
• “Gang” operators
  • Assign the same bit-width to a group of operators
  • Can tune SIMD programs
  • E.g., ganging the two ‘x’s:
Efficiency Model

- Count the weighted sum of the lower bit-width operators
  - E.g., assigning 32-bit to the subtraction gets score $w_4$
  - Prioritize assigning 32-bit to the expensive operators such as square root
- Control the # of type casts with an additional constraint
Solution Overview: Tune with Mathematical Optimization

Maximize  program efficiency model

Subject to  error model (the worst-case error) ≤ Err (user-specified threshold)

constraints for limiting the # of type casts

constraints for ganging operations
FP Tuner Toolflow

- Program: Real-valued Expression
  - Gelpia: Global Optimizer
    - Generic Error Model
    - Efficiency Model
  - Optimization Problem
  - Optimal Mixed-precision

User Specifications
- Error Threshold
- Operators’ Weights
- Additional Constrains

Additional Constrains

39
Roadmap

• Background on floating-point and rounding error
• Our method – rigorous mixed-precision tuning
• Experimental results
• Related work and conclusions
Experimental Results

- Benchmarks: important math primitives
  - Previous work [Darulova. `14]
  - Synthetic primitives
- Two types of efficiency
  - Performance
  - Energy – measured with an actual hardware platform + wattmeter
- Generate mixed-precision versions with two bit-widths
  - The mixed-precision versions have the moderate efficiency/precision
- Current limitations: not handling branches and loops
Performance Benefits of Using Mixed-precision

- Mixed 64- and 128-bit
- Speedup comparing to the all-128s
  - 1.1x ~ 9.4x
  - Average: 2.4x
Energy Consumption Benefits of Using Mixed-precision

- Mixed 32- and 64-bit
- Energy saving: 31.5%
Energy Consumption Benefits of Using Mixed-precision

- Mixed 32- and 64-bit
- Energy saving: 31.5%
Roadmap

• Background on floating-point and rounding error
• Our method – rigorous mixed-precision tuning
• Experimental results
• Related work and conclusions
Related Work

• Floating-point error analysis methods and tools
  [Melquiond. `06] [Goubault. `06] [Rummer. `10] [Boldo. `11] [Darulova. `12 `14] [Chiang. `14] 
  [Zou. `15] [Solovyev. `15] [Magron. `16], etc.

• Rewriting-based precision tuning [Panchekha et al. PLDI 2015]

• Rigorous program-wise precision selection [Darulova et al. POPL 2014]
  • All-32 or all-64
Conclusions

• We offer the first solution of rigorous floating-point mixed-precision tuning
  • Convert a tuning problem to an optimization problem

• Tool releases:
  • FPTuner: https://github.com/soarlab/FPTuner
  • Gelpia, the global optimizer: https://github.com/soarlab/gelpia