Exploiting Community Structure for Floating-Point Precision Tuning

Hui Guo       Cindy Rubio-González

UC Davis
University of California

ISSTA’18 – Amsterdam, Netherlands, July 2018
Background

• Floating-point (FP) arithmetic used in many domains

• Reasoning about FP programs is difficult
  – Large variety of numerical problems
  – Most programmers are not experts in FP

• Common practice: use highest available precision
  – Disadvantage: more expensive!

• Tools have been developed for precision tuning

  Given: Accuracy constraints
  Action: Reduce precision
  Goal: Performance
Precision Tuning Example

1. long double fun(long double p) {
2.   long double pi = acos(-1.0);
3.   long double q = sin(pi * p);
4.       return q;
5. }
6
7. void simpsons() {
8.   long double a, b;
9.   long double h, s, x;
10.  const long double fuzz = 1e-26;
11.  const int n = 2000000;
12.  ...
13.  L100:
14.      x = x + h;
15.      s = s + 4.0 * fun(x);
16.      x = x + h;
17.      if (x + fuzz >= b) goto L110;
18.      s = s + 2.0 * fun(x);
19.      goto L100;
20.  L110:
21.      s = s + fun(x);
22.      //final answer:(long double)h *s/3.0
23. }

Original Program

Tuned Program

Error threshold $10^{-8}$
Precision Tuning Example

Original Program

```c
long double fun(long double p) {  
    long double pi = acos(-1.0);  
    long double q = sin(pi * p);  
    return q;  
}
```

Tuned Program

```c
long double fun(double p) {  
    double pi = acos(-1.0);  
    long double q = sin(pi * p);  
    return q;  
}
```

Tuned program runs 78.7% faster!
State-of-the-art: Black-box Precision Tuning

double precision

single precision

✔

✘
State-of-the-art: Black-box Precision Tuning

double precision

single precision
State-of-the-art: Black-box Precision Tuning
State-of-the-art: Black-box Precision Tuning

double precision

single precision
State-of-the-art: Black-box Precision Tuning

- Double precision
  - [Diagram: Multiple options, one marked with a checkmark]
- Single precision
  - [Diagram: Multiple options, all marked with an 'X']

- State-of-the-art: Black-box Precision Tuning

- Double precision: One option is selected as the best.
- Single precision: No option is selected as the best.
State-of-the-art: Black-box Precision Tuning

**double precision**

- ![Proposed configuration](image)
- ![Failed configurations](image)

**single precision**

- ![Failed configuration](image)
State-of-the-art: Black-box Precision Tuning

- State of the art groups variables arbitrarily

- Black box nature
  - Related variables assigned types independently
  - Large number of variables $\rightarrow$ Slow search
  - More type casts $\rightarrow$ Less speedup

---

Original

Local minimum
Uses lower precision
Speedup: 78.7%

Global minimum
Shifts precision less often
Speedup: 90%
Exploiting Community Structure

• Can we leverage the program to perform a more informed precision tuning?

• White box nature
  - Related variables pre-grouped into hierarchy → Same type
  - Fewer groups in search space → Faster search
  - Fewer type casts → Larger speedups

Search top to bottom

Level 0

Level 1

Level 2
Approach

1. Type Dependence Analysis + Edge Profiling
   - Weighted Dependence Graph

2. Iterative Community Detection + Ordering
   - Ordered Community Structure of Variables

3. Hierarchical Precision Tuning
   - Type Configuration
   - Speeds up program by reducing precision with respect to accuracy constraint

Source Code
Test Inputs
Accuracy Constraint
Type Dependence Analysis + Edge Profiling

1 long double fun(long double p) {
2   long double pi = acos(-1.0);
3   long double q = sin(pi * p);
4   return q;
5 }
6
7 void simpsons(){
8   long double a, b,
9      // subinterval length, integral approximation, x
10  long double h,s,x;
11  const long double fuzz = 1e-26;
12  const int n = 2000000;
13  a = 0.0;
14  b = 1.0;
15  h = (b - a) / n;
16  x = a;
17  s = fun(x);
18  L100:
19    x = x + h;
20    s = s + 4.0 * fun(x);
21    x = x + h;
22    if (x + fuzz >= b) goto L110;
23    s = s + 2.0 * fun(x);
24    goto L100;
25  L110:
26    s = s + fun(x);
27    printf("%1.16Le\n", (long double)h * s / 3.0);
28  }

Identify assignments to floating-point variables
Type Dependence Analysis
+ Edge Profiling

2 long double pi = acos(-1.0);
3 long double q = sin(pi * p);

11 const long double fuzz = 1e-26;

13 a = 0.0;
14 b = 1.0;
15 h = (b - a) / n;
16 x = a;
17 s = fun(x);

19 x = x + h;
20 s = s + 4.0 * fun(x);
21 x = x + h;
23 s = s + 2.0 * fun(x);
26 s = s + fun(x);

A vertex in the graph represents a FP variable, and an edge u → v denotes that value u is used to compute value v at least once.
Iterative Community Detection + Ordering

Use modularity maximization \cite{1, 2} to iteratively detect communities on the generated dependence graph until no new communities are found.

Community structure of floating-point variables

Iterative Community Detection + Ordering

Sort the items at each level of the hierarchy using topological order to follow the dependence flow

Ordered community structure
Hierarchical Precision Tuning

Search through the hierarchy from top down to the bottom

Original precision configuration
Reduce precision to speed up program

Top-level precision configuration
Reduce precision to speed up program

Bottom-level precision configuration

Global minimum configuration with 90% speedup!
Experimental Setup

- Hierarchical search algorithm implemented in tool HiFPTuner
- Benchmarks: 4 GSL programs (inputs that maximize coverage), 2 NAS Parallel Benchmarks (inputs Class A), 3 other numerical programs including simpsons (input free)
- Error thresholds
  - Multiple error thresholds: $10^{-4}, 10^{-6}, 10^{-8},$ and $10^{-10}$
  - User can evaluate trade-off between accuracy and speedup
  - 35 experiments in total
- Evaluated search efficiency and effectiveness in comparison with state-of-the-art tool Precimonious
## Number of Communities

<table>
<thead>
<tr>
<th>Initial Type Configuration</th>
<th>Items to Tune</th>
<th>Communities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Items</td>
<td>L2</td>
</tr>
<tr>
<td>Program</td>
<td></td>
<td></td>
</tr>
<tr>
<td>simpsons</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>arclenght</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>piqpr</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>fft</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>gaussian</td>
<td>58</td>
<td>25</td>
</tr>
<tr>
<td>sum</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>bessel</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>ep</td>
<td>17</td>
<td>9</td>
</tr>
<tr>
<td>cp</td>
<td>35</td>
<td>21</td>
</tr>
</tbody>
</table>

The number of tunable items at the top level of the hierarchy is reduced by 53% from 239 to 112
RQ1: Search Efficiency

How *efficient* is hierarchical search for precision tuning in comparison with Precimonious?

Answer: HiFPTuner exhibits higher search efficiency over Precimonious for 75.9% (22 out of 29) of the experiments that require tuning.

Overall, HiFPTuner explores 45% (3,326) fewer configurations than Precimonious.
Configurations for Error Threshold $10^{-8}$

Number of Configurations

<table>
<thead>
<tr>
<th>Function</th>
<th>HiFPTuner</th>
<th>Precimious</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpsons</td>
<td>24</td>
<td>116</td>
</tr>
<tr>
<td>arclength</td>
<td>30</td>
<td>142</td>
</tr>
<tr>
<td>piqpr</td>
<td>52</td>
<td>164</td>
</tr>
<tr>
<td>fft</td>
<td>43</td>
<td>297</td>
</tr>
<tr>
<td>gaussian</td>
<td>211</td>
<td>275</td>
</tr>
<tr>
<td>sum</td>
<td>533</td>
<td>433</td>
</tr>
<tr>
<td>ep</td>
<td>45</td>
<td>77</td>
</tr>
<tr>
<td>cp</td>
<td>497</td>
<td>735</td>
</tr>
</tbody>
</table>
## Configurations for Error Threshold 10^{-8}

### Initial Type Configuration

<table>
<thead>
<tr>
<th>Program</th>
<th>L</th>
<th>D</th>
<th>F</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpsons</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>arclength</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>piqpr</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fft</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>gaussian</td>
<td>0</td>
<td>56</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>sum</td>
<td>0</td>
<td>34</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ep</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>cp</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

### HiFPTuner

#### Error threshold: 10^{-8}

<table>
<thead>
<tr>
<th></th>
<th>L</th>
<th>D</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpsons</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>arclength</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>piqpr</td>
<td>3</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fft</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>gaussian</td>
<td>0</td>
<td>10</td>
<td>46</td>
<td>2</td>
</tr>
<tr>
<td>sum</td>
<td>0</td>
<td>10</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>ep</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>cp</td>
<td>0</td>
<td>24</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

### Precimonious

#### Error threshold: 10^{-8}

<table>
<thead>
<tr>
<th></th>
<th>L</th>
<th>D</th>
<th>F</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpsons</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>arclength</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>piqpr</td>
<td>3</td>
<td>13</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>fft</td>
<td>0</td>
<td>21</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>gaussian</td>
<td>0</td>
<td>56</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>sum</td>
<td>0</td>
<td>34</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ep</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>cp</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
HiFPTuner top level’s configuration has the best performance
24 vs. 116 configurations
RQ2: Search Effectiveness

How *effective* is hierarchical search in finding higher quality configurations than Precimoniuous?

Answer: HiFPTuner finds better configurations for 51.7% (15 out of 29) of the experiments that require tuning compared to Precimoniuous
Summary

• White-box approach for dynamic precision tuning

• Analyzes code and runtime behavior to construct a weighted dependence graph used to detect communities of variables and construct a hierarchy

• Experimental evaluation on 9 programs shows
  – HiFPTUNER reduces the search space by 53% on average
  – HiFPTUNER finds better configurations for 51.7% of the programs x error thresholds

• HiFPTUNER makes a step towards more scalable and effective floating-point precision tuning