Software Tools for Mixed-Precision Program Analysis

Dr. Mike Lam

James Madison University
Lawrence Livermore National Lab
About Me

• Ph.D in CS from University of Maryland ('07-'14)
  – Topic: Automated floating-point program analysis
  – Intern @ Lawrence Livermore National Lab (LLNL) in Summer '11

• Assistant professor at James Madison University since '14
  – Teaching: computer organization, parallel & distributed systems, compilers, and programming languages
  – Research: high-performance analysis research group (w/ Dee Weikle)

• Faculty scholar @ LLNL since Summer '16
  – Energy-efficient computing project (w/ Barry Roundtree)
  – Variable precision computing project (w/ Jeff Hittinger)
Motivation

- IEEE floating-point arithmetic
  - Ubiquitous in scientific computing
  - More bits => higher accuracy (usually)
  - Fewer bits => higher performance (usually)
Motivation

• Vector single precision 2X+ faster
  – Possibly better if memory pressure is alleviated
  – Newest GPUs use mixed precision for tensor ops

<table>
<thead>
<tr>
<th>Operation</th>
<th>FP32</th>
<th>Packed FP32</th>
<th>FP64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Subtract</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Multiply</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Divide</td>
<td>27</td>
<td>32</td>
<td>42</td>
</tr>
<tr>
<td>Square root</td>
<td>28</td>
<td>38</td>
<td>43</td>
</tr>
</tbody>
</table>

Instruction latencies for Intel Knights Landing

Credit: https://agner.org/optimize/ and NVIDIA Tesla V100 Datasheet
Question

• How many bits do you need?
Prior Approaches

• Rigorous: forwards/backwards error analysis
  – Requires numerical analysis expertise

• Pragmatic: “guess-and-check”
  – Requires manual code conversion effort

```c
//double x[N], y[N];
float x[N], y[N];
double alpha;
```
Research Question

• What can we learn about floating-point behavior with **automated** analysis?
  – Specifically: can we build *mixed-precision* versions of a program automatically?

• Caveat: few (or no) formal guarantees
  – Rely on user-provided representative run (and sometimes a verification routine)

```c
double sum = 0.0;
void sum2pi_x()
{
    double tmp;
    double acc;
    int i, j;

    […]
}
```
FPAnalysis / CRAFT (2011)

- Dynamic binary analysis via Dyninst
- Cancellation detection
- Range (exponent) tracking

\[
\begin{align*}
3.682236 & \quad - \quad 3.682234 \\
0.000002 & \\
\text{(6 digits cancelled)}
\end{align*}
\]
CRAFT (2013)

- Dynamic binary analysis via Dyninst
- Instruction-level replacement of doubles w/ floats
- Hierarchical search for valid replacements
if (timers_enabled) call timer_start(2)

    do 140 i = 1, nk
        x1 = 2.d0 * x(2*i-1) - 1.d0
        x2 = 2.d0 * x(2*i) - 1.d0
        t1 = x1 ** 2 + x2 ** 2
        if (t1 .le. 1.d0) then
            t2 = sqrt(-2.d0 * log(t1) / t1)
            t3 = (x1 * t2)
            t4 = (x2 * t2)
            l = max(abs(t3), abs(t4))
            q(l) = q(l) + 1.d0
            sx = sx + t3
            sy = sy + t4
        endif
        140 continue
    if (timers_enabled) call timer_stop(2)
    150 continue
<table>
<thead>
<tr>
<th>NAS Benchmark (name.CLASS)</th>
<th>Candidate Instructions</th>
<th>Configurations Tested</th>
<th>% Dynamic Replaced</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt.A</td>
<td>6,262</td>
<td>4,000</td>
<td>78.6</td>
</tr>
<tr>
<td>cg.A</td>
<td>956</td>
<td>255</td>
<td>5.6</td>
</tr>
<tr>
<td>ep.A</td>
<td>423</td>
<td>114</td>
<td>45.5</td>
</tr>
<tr>
<td>ft.A</td>
<td>426</td>
<td>74</td>
<td>0.2</td>
</tr>
<tr>
<td>lu.A</td>
<td>6,014</td>
<td>3,057</td>
<td>57.4</td>
</tr>
<tr>
<td>mg.A</td>
<td>1,393</td>
<td>437</td>
<td>36.6</td>
</tr>
<tr>
<td>sp.A</td>
<td>4,507</td>
<td>4,920</td>
<td>30.5</td>
</tr>
</tbody>
</table>
Issues

• High overhead
  – Must check and (possibly) convert operands before each instruction

• Lengthy search process
  – Search space is exponential wrt. instruction count

• Coarse-grained analysis
  – Binary decision: single or double
CRAFT (2016)

- Reduced-precision analysis
  - Simulate conservatively via bit-mask truncation
  - Report min output precision for each instruction
  - Finer-grained analysis and lower overhead
CRAFT (2016)

- Scalability via heuristic search
  - Focus on most-executed instructions
  - Analysis time vs. benefit tradeoff

\[
\begin{align*}
> 5.0\% &\quad 4:66 \\
> 1.0\% &\quad 5:93 \\
> 0.5\% &\quad 9:45 \\
> 0.1\% &\quad 15:45 \\
> 0.05\% &\quad 23:60 \\
\text{Full} &\quad 28:71
\end{align*}
\]
Issue

- Only considers precision reduction
  - No higher precision or arbitrary-precision
  - No alternative representations
  - No dynamic tracking of error
SHVAL (2016)

- Shadow value analysis
  - Maintain “shadow” value for every memory location
  - Execute shadow operations for all computation
  - Pintool: less overhead than similar tools like Valgrind

```c
double sum = 0.0;
for (int i = 0; i < 10; i++) {
    sum += 0.1;
}
printf("%25.20f\n", sum);
```

Fig. 3. Sample C program

```
pxor xmm0, xmm0
mov  eax, 10
movsd xmm1, 0x400628
loop:
    sub  eax, 1
    addsd xmm0, xmm1
    jne  loop
movsd 0x8(rsp), xmm0
```

Original machine code:  Inserted shadow code:

```
(set to 0.0)
(load 0.1)
(increment)
(store sum)
```

```
xmm[0] = convert(0.0)
xmm[1] = convert(*0x400628)
xmm[0] += xmm[1]
mem[rsp+0x8] = xmm[0]
```

Fig. 4. Compiled assembly of program from Figure 3

<table>
<thead>
<tr>
<th>Shadow Value Type</th>
<th>Exp Size</th>
<th>Frac Size</th>
<th>Final Shadow Value</th>
<th>Relative Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit (native single)</td>
<td>8</td>
<td>23</td>
<td>1.000000</td>
<td>1.19e-07</td>
</tr>
<tr>
<td>64-bit (native double)</td>
<td>11</td>
<td>52</td>
<td>1.00000000000000000000000</td>
<td>0</td>
</tr>
<tr>
<td>128-bit GNU MPFR</td>
<td>15</td>
<td>112</td>
<td>1.0000000000000000000000005551e+00</td>
<td>1.11e-16</td>
</tr>
<tr>
<td>Unum (3.2)</td>
<td>8</td>
<td>4</td>
<td>(0.9375, 1.1875)</td>
<td>0.059</td>
</tr>
<tr>
<td>Unum (3.4)</td>
<td>8</td>
<td>16</td>
<td>(0.9999847412109375, 1.0000457763671875)</td>
<td>1.53e-05</td>
</tr>
<tr>
<td>Unum (4.6)</td>
<td>16</td>
<td>64</td>
<td>1.0000000000000000000000005551...182</td>
<td>1.11e-16</td>
</tr>
</tbody>
</table>

TABLE I
Analysis results on sample program
SHVAL (ongoing)

• Single precision shadow values
  – Trace execution and build data flow graph
  – Color nodes by error w.r.t. original double precision values
  – Highlights high-error regions
  – Inherent scaling issues

Gaussian elimination example

- Single precision shadow values
- Trace execution and build data flow graph
- Color nodes by error w.r.t. original double precision values
- Highlights high-error regions
- Inherent scaling issues
Issue

• No source-level mixed precision
  – Difficult to translate instruction-level analysis results to source-level transformations
  – Some users might be satisfied with opaque compiler-based optimization, but most HPC users want to know what changed!
CRAFT (2013)

- Memory-based replacement analysis
  - Leave computation intact but round outputs
  - Aggregate instructions that modify same variable
  - Found several valid variable-level replacements

<table>
<thead>
<tr>
<th>NAS Benchmark (name.CLASS)</th>
<th>Candidate Operands</th>
<th>Configurations Tested</th>
<th>% Executions Replaced</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt.A</td>
<td>2,342</td>
<td>300</td>
<td>97.0</td>
</tr>
<tr>
<td>cg.A</td>
<td>287</td>
<td>68</td>
<td>71.3</td>
</tr>
<tr>
<td>ep.A</td>
<td>236</td>
<td>59</td>
<td>37.9</td>
</tr>
<tr>
<td>ft.A</td>
<td>466</td>
<td>108</td>
<td>46.2</td>
</tr>
<tr>
<td>lu.A</td>
<td>1,742</td>
<td>104</td>
<td>99.9</td>
</tr>
<tr>
<td>mg.A</td>
<td>597</td>
<td>153</td>
<td>83.4</td>
</tr>
<tr>
<td>sp.A</td>
<td>1,525</td>
<td>1,094</td>
<td>88.9</td>
</tr>
</tbody>
</table>
SHVAL (2017)

• Single-vs-double shadow value analysis
  – Aggregate error by instruction or memory location over time

• Computer vision case study (Apriltags)
  – 1.7x speedup on average with only 4% error
  – 40% energy savings in embedded experiments

Credit: Ramy Medhat (ramy.medhat@uwaterloo.ca)
Issues

• Each instruction or variable is tested in isolation
  – Union of valid replacements is often invalid

• Cannot ensure speedup
  – Instrumentation overhead
  – Added casts to convert data between regions
  – Lack of vectorization
CRAFT (ongoing)

- Variable-centric mixed precision analysis
  - Use TypeForge (an AST-level type conversion tool) for source-to-source mixed precision

- Search for best speedup
  - Run full compiler backend w/ optimizations
  - Report fastest configuration that passes verification

```c
double sum = 0.0;

void sum2pi_x()
{
    double tmp;
    double acc;
    int i, j;

    [...]
}
```

```c
double sum = 0.0;

void sum2pi_x()
{
    float tmp;
    float acc;
    int i;
    int j;

    [...]
}
Related Work

• CRAFT, SHVAL, and Precimonious [Rubio’13]
  – Very practical
  – Widely-used tool frameworks (Dyninst, Pin, LLVM)
  – Few (or no) formal guarantees
  – Tested on HPC benchmarks on Linux/x86

• Daisy [Darulova’18] and FPTuner [Chiang’17]
  – Very rigorous
  – Custom input formats
  – Provable error bounds for given input range
  – Impractical for HPC benchmarks
**ADAPT (2018)**

- Automatic backwards error analysis
  - Obtain gradients via reverse-mode algorithmic differentiation (CoDiPack or TAPENADE)
  - Calculate error contribution of intermediate results
  - Aggregate by program variable
  - Greedy algorithm builds mixed-precision allocation

Credit: Harshitha Menon (gopalakrishn1@llnl.gov)
Original C Code

```c
#include <iostream>

double sum = 0.0;
double inc = 0.1;

double do_sum() {
    int i;
    for (i = 0; i < 1000; i++) {
        sum += inc;
    }
    return sum;
}

int main() {
    do_sum();
    cout << sum << endl;
    return 0;
}
```

AD Instrumented Code

```c
#include <iostream>
#include <adapt.h>
#include <adapt-impl.cpp>

AD_real sum = 0.0;
AD_real inc = 0.1;

AD_real do_sum() {
    int i;
    for (i = 0; i < 1000; i++) {
        sum += inc;
    }
    return sum;
}

int main() {
    AD_begin();
    AD_independent(inc, "inc");
    do_sum();
    cout << AD_value(sum) << endl;
    AD_dependent(sum, "sum", 8);
    AD_report();
    return 0;
}
```

- AD Libraries
- Type Changes
- Initialization
- Output
• Used ADAPT on LULESH benchmark to help develop a mixed-precision CUDA version

• Achieved speedup of 20% within original error threshold on NVIDIA GK110 GPU
FloatSmith (ongoing)

- Mixed-precision search via CRAFT
- Source-to-source translation via TypeForge
- Optionally, use ADAPT analysis to narrow search and provide more rigorous guarantees
FPHP (ongoing)

• Benchmark suite aimed at facilitating scale-up for mixed-precision analysis tools
  – “Middle ground” between real-valued expressions and full applications
  – Currently looking for good case studies
Future Work

• (Better) OpenMP/MPI support
• (Better) GPU and FPGA support
• Model-based performance prediction
• Dynamic runtime precision tuning
• Ensemble floating-point analysis
Summary

• Automated mixed precision is possible
  – Practicality vs. rigor tradeoff
• Multiple active projects
  – Various goals and approaches
  – All target HPC applications
• Many avenues for future research
Papers

• **CRAFT**

• **SHVAL**

• **ADAPT**
Acknowledgements

Jeff Hollingsworth
Bronis de Supinski
Barry Rountree
Jeff Hittinger

Scott Lloyd
Matthew Legendre
Harshitha Menon
Markus Schordan

Lindsay Lam
Shelby Funk
Ramy Medhat
Nathan Pinnow

Dee Weikle
Garrett Folks
Logan Moody
Nkeng Atabong

U.S. Department of Energy
DE-CFC02-01ER25489, DE-FG02-01ER25510, DE-FC02-06ER25763, and DE-AC52-07NA27344

Lawrence Livermore National Laboratory
LDRD project 17-SI-004

James Madison University
various provost awards, college grants, and department student funding
Thank you!

github.com/crafthpc
github.com/llnl/adapt-fp
tinyurl.com/fpanalysis

Contact me:
lam2mo@jmu.edu