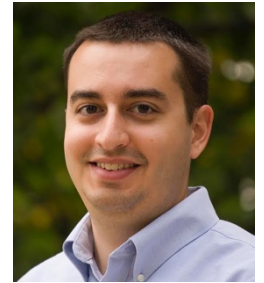


Software Tools for Mixed-Precision Program Analysis

Dr. Mike Lam

James Madison University
Lawrence Livermore National Lab

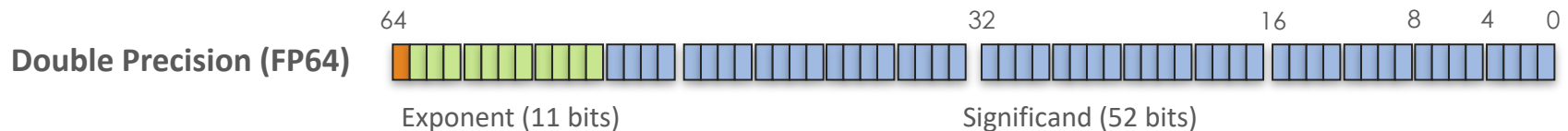
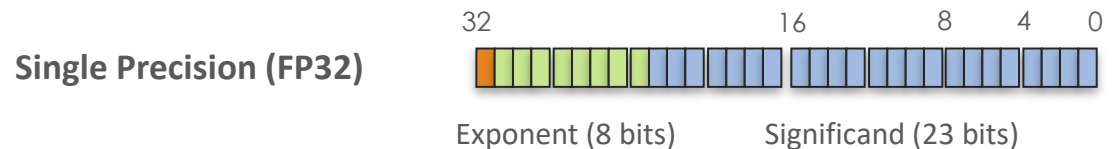
About Me



- 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

Operation	FP32	Packed FP32	FP64
Add	6	6	6
Subtract	6	6	6
Multiply	6	6	6
Divide	27	32	42
Square root	28	38	43

Instruction latencies for Intel Knights Landing



Tesla V100 PCIe

Tesla V100 SXM2

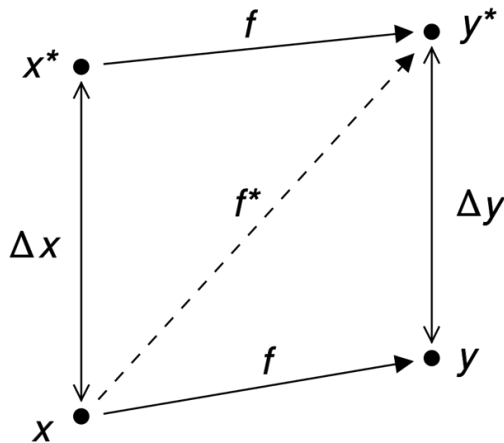
GPU Architecture		NVIDIA Volta	
NVIDIA Tensor Cores	640		
NVIDIA CUDA® Cores	5,120		
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS	FP64
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS	FP32
Tensor Performance	112 TFLOPS	125 TFLOPS	Mixed FP16 / FP32

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



```
//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)

```
double sum = 0.0;
```

```
void sum2pi_x()  
{
```

```
    double tmp;
```

```
    double acc;
```

```
    int i, j;
```

```
    [...]
```



```
double sum = 0.0;
```

```
void sum2pi_x()  
{
```

```
    float tmp;
```

```
    float acc;
```

```
    int i;
```

```
    int j;
```

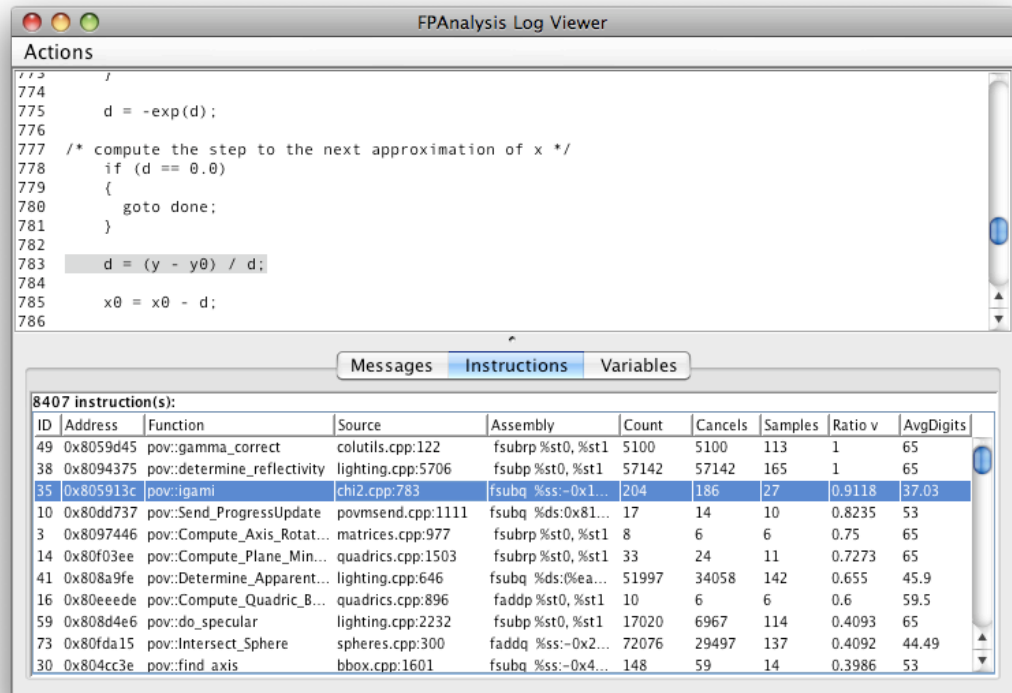
```
    [...]
```

FPAAnalysis / CRAFT (2011)

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

$$\begin{array}{r} 3.682236 \\ - 3.682234 \\ \hline 0.000002 \end{array}$$

(6 digits cancelled)



The screenshot shows the FPAAnalysis Log Viewer interface. The top pane displays assembly code with comments. The bottom pane shows a table of instructions with columns for ID, Address, Function, Source, Assembly, Count, Cancels, Samples, Ratio v, and AvgDigits.

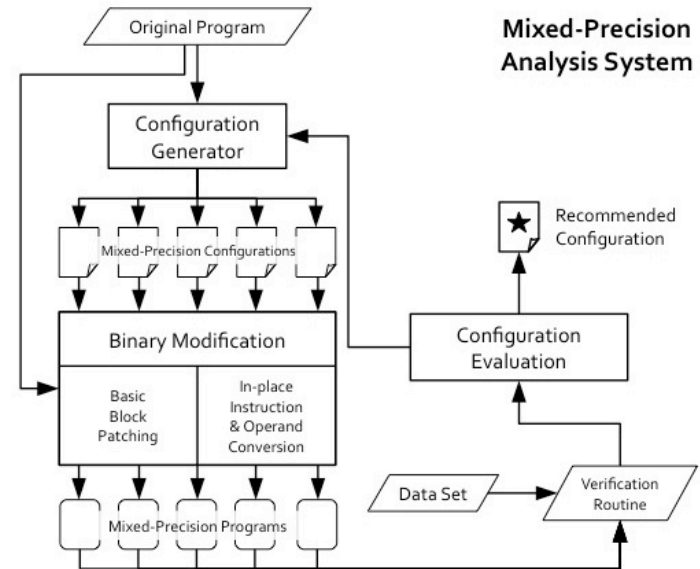
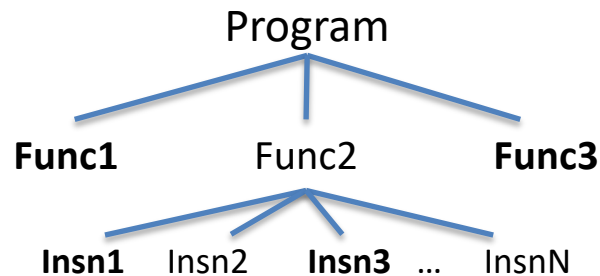
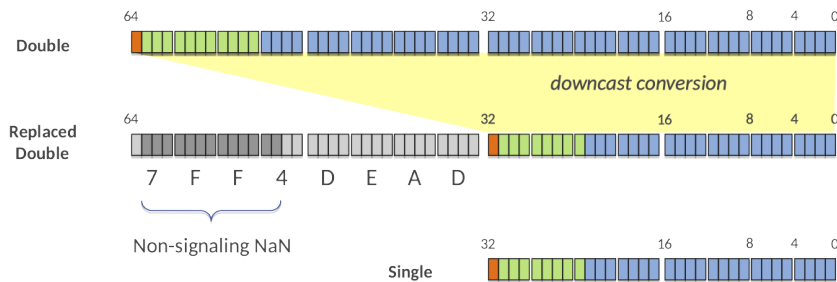
```
774 /
775     d = -exp(d);
776
777 /* compute the step to the next approximation of x */
778     if (d == 0.0)
779     {
780         goto done;
781     }
782
783     d = (y - y0) / d;
784
785     x0 = x0 - d;
786
```

ID	Address	Function	Source	Assembly	Count	Cancels	Samples	Ratio v	AvgDigits
49	0x8059d45	pov::gamma_correct	colutils.cpp:122	fsubrp %st0, %st1	5100	5100	113	1	65
38	0x8094375	pov::determine_reflectivity	lighting.cpp:5706	fsubp %st0, %st1	57142	57142	165	1	65
35	0x805913c	pov::igami	chi2.cpp:783	fsubq %ss:-0x1...	204	186	27	0.9118	37.03
10	0x80dd737	pov::Send_ProgressUpdate	povmsend.cpp:1111	fsubq %ds:0x81...	17	14	10	0.8235	53
3	0x8097446	pov::Compute_Axis_Rotat...	matrices.cpp:977	fsubrp %st0, %st1	8	6	6	0.75	65
14	0x80f03ee	pov::Compute_Plane_Min...	quadrics.cpp:1503	fsubrp %st0, %st1	33	24	11	0.7273	65
41	0x808a9fe	pov::Determine_Apparent...	lighting.cpp:646	fsubq %ds:%ea...	51997	34058	142	0.655	45.9
16	0x80eeede	pov::Compute_Quadric_B...	quadrics.cpp:896	faddp %st0, %st1	10	6	6	0.6	59.5
59	0x808d4e6	pov::do_specular	lighting.cpp:2232	fsubp %st0, %st1	17020	6967	114	0.4093	65
73	0x80fda15	pov::Intersect_Sphere	spheres.cpp:300	faddq %ss:-0x2...	72076	29497	137	0.4092	44.49
30	0x804cc3e	pov::find_axis	bbox.cpp:1601	fsubq %ss:-0x4...	148	59	14	0.3986	53

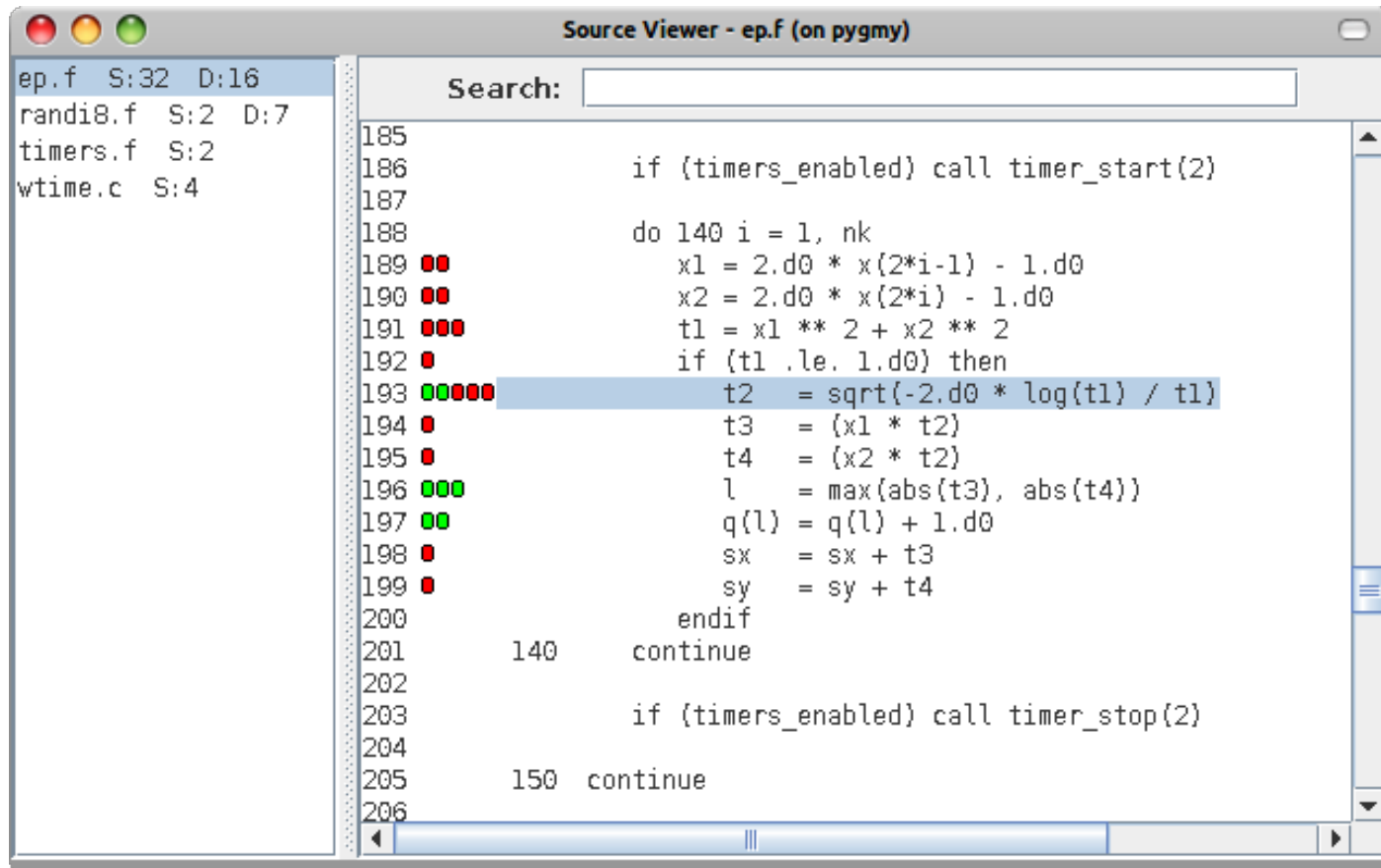
CRAFT (2013)



- Dynamic binary analysis via Dyninst
- Instruction-level replacement of doubles w/ floats
- Hierarchical search for valid replacements



CRAFT (2013)



The screenshot shows a window titled "Source Viewer - ep.f (on pygmy)". On the left, a file list shows "ep.f S:32 D:16" selected, with "randi8.f S:2 D:7", "timers.f S:2", and "wtime.c S:4" below it. The main area contains Fortran code with a search bar at the top. The code is as follows:

```
185
186     if (timers_enabled) call timer_start(2)
187
188     do 140 i = 1, nk
189         x1 = 2.d0 * x(2*i-1) - 1.d0
190         x2 = 2.d0 * x(2*i) - 1.d0
191         t1 = x1 ** 2 + x2 ** 2
192         if (t1 .le. 1.d0) then
193             t2 = sqrt(-2.d0 * log(t1) / t1)
194             t3 = (x1 * t2)
195             t4 = (x2 * t2)
196             l = max(abs(t3), abs(t4))
197             q(l) = q(l) + 1.d0
198             sx = sx + t3
199             sy = sy + t4
200         endif
201     140     continue
202
203     if (timers_enabled) call timer_stop(2)
204
205     150 continue
206
```

Execution progress is indicated by colored dots to the left of each line: red for lines 189-199, green for lines 193-197, and blue for line 193. Line 193 is highlighted in blue. A scrollbar on the right indicates the current position in the file.

CRAFT (2013)

NAS Benchmark (name.CLASS)	Candidate Instructions	Configurations Tested	% Dynamic Replaced
bt.A	6,262	4,000	78.6
cg.A	956	255	5.6
ep.A	423	114	45.5
ft.A	426	74	0.2
lu.A	6,014	3,057	57.4
mg.A	1,393	437	36.6
sp.A	4,507	4,920	30.5

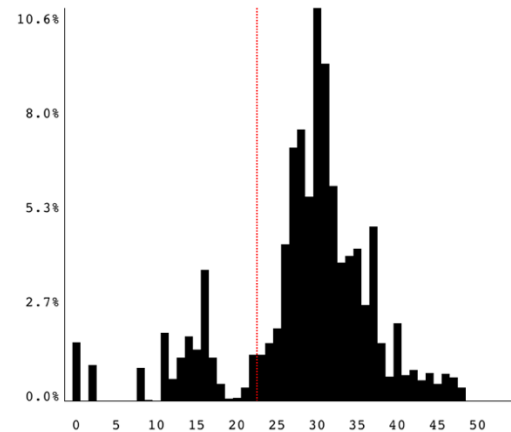
Issues

- 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)

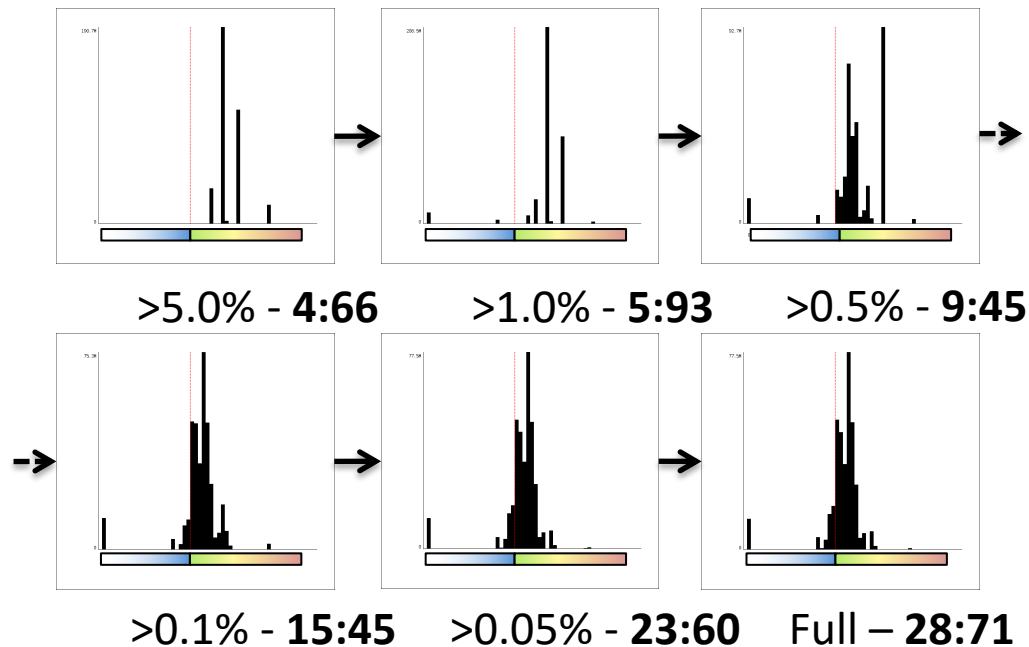
- Reduced-precision analysis
 - Simulate conservatively via bit-mask truncation
 - Report min output precision for each instruction
 - Finer-grained analysis and lower overhead

```
▼ MODULE: 0x400000 "wtime.c" Prec=51 [51 instruction(s)]
  ▼ FUNC: 0x400b60 "MAIN_" Prec=51 [49 instruction(s)]
    ▼ BBLK: 0x401088 Prec=40
      INSN: 0x401096 "mulsd xmm6, xmm10 [ep.f:189]" Prec=39
      INSN: 0x40109b "mulsd xmm8, xmm10 [ep.f:190]" Prec=37
      INSN: 0x4010a0 "subsd xmm6, xmm9 [ep.f:189]" Prec=29
      INSN: 0x4010a5 "subsd xmm8, xmm9 [ep.f:190]" Prec=27
      INSN: 0x4010b1 "mulsd xmm1, xmm6 [ep.f:191]" Prec=25
      INSN: 0x4010b5 "mulsd xmm2, xmm8 [ep.f:191]" Prec=26
      INSN: 0x4010ba "addsd xmm1, xmm2 [ep.f:191]" Prec=27
    ▼ BBLK: 0x4010f2 Prec=51
      INSN: 0x401106 "addsd xmm0, xmm0 [ep.f:193]" Prec=25
      INSN: 0x40110a "subsd xmm2, xmm1 [ep.f:193]" Prec=25
      INSN: 0x40110e "divsd xmm0, xmm2 [ep.f:193]" Prec=25
      INSN: 0x401112 "sqrtsd xmm0, xmm0 [ep.f:193]" Prec=26
      INSN: 0x401136 "mulsd xmm6, xmm0 [ep.f:194]" Prec=27
      INSN: 0x40113a "mulsd xmm8, xmm0 [ep.f:195]" Prec=26
      INSN: 0x40113f "addsd xmm7, xmm6 [ep.f:198]" Prec=51
      INSN: 0x40115d "addsd xmm6, xmm8 [ep.f:199]" Prec=51
      INSN: 0x401178 "addsd xmm5, xmm9 [ep.f:197]" Prec=0
```



CRAFT (2016)

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



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

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

Fig. 3. Sample C program

Original machine code:

```
pxor    xmm0, xmm0          (set to 0.0)
mov     eax, 10
movsd  xmm1, 0x400628      (load 0.1)
loop:
sub     eax, 1
addsd  xmm0, xmm1         (increment)
jne    loop
movsd  0x8(rsp), xmm0     (store sum)
```

Inserted shadow code:

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

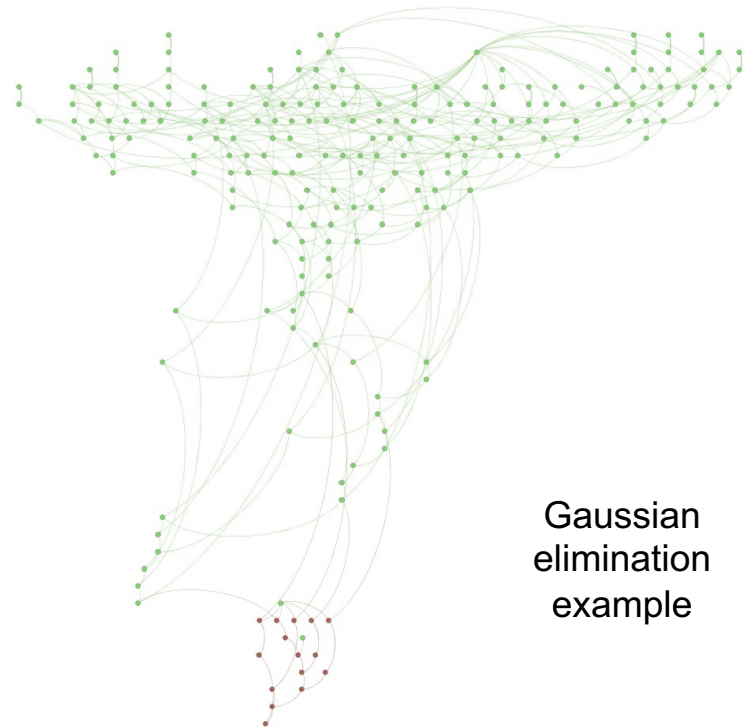
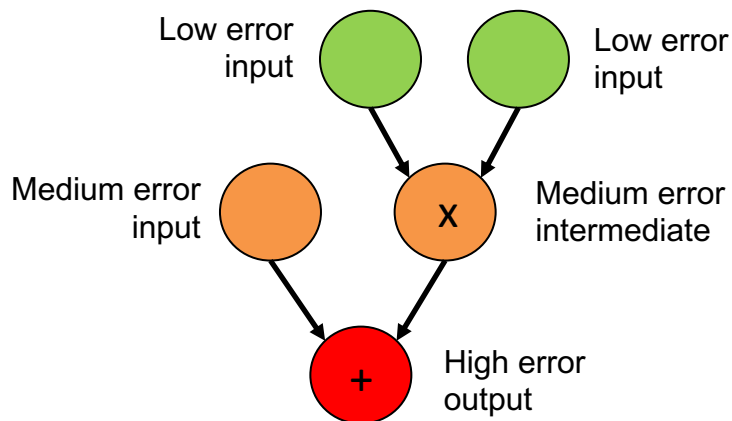
Fig. 4. Compiled assembly of program from Figure 3

Shadow Value Type	Exp Size	Frac Size	Final Shadow Value	Relative Error
32-bit (native single)	8	23	1.000000	1.19e-07
64-bit (native double)	11	52	1.0000000000000000	0
128-bit GNU MPFR	15	112	1.00000000000000005551e+00	1.11e-16
Unum (3,2)	8	4	(0.9375, 1.1875)	0.059
Unum (3,4)	8	16	(0.9999847412109375, 1.0000457763671875)	1.53e-05
Unum (4,6)	16	64	1.00000000000000005551...182	1.11e-16

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



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

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

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

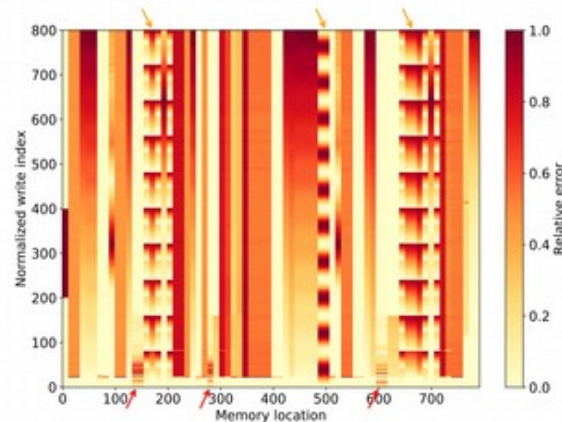


Fig. 1. Error trace per memory location. A darker pixel indicates higher error.

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

```
double sum = 0.0;
```

```
void sum2pi_x()
```

```
{
```

```
    double tmp;
```

```
    double acc;
```

```
    int i, j;
```

```
    [...]
```



```
double sum = 0.0;
```

```
void sum2pi_x()
```

```
{
```

```
    float tmp;
```

```
    float acc;
```

```
    int i;
```

```
    int j;
```

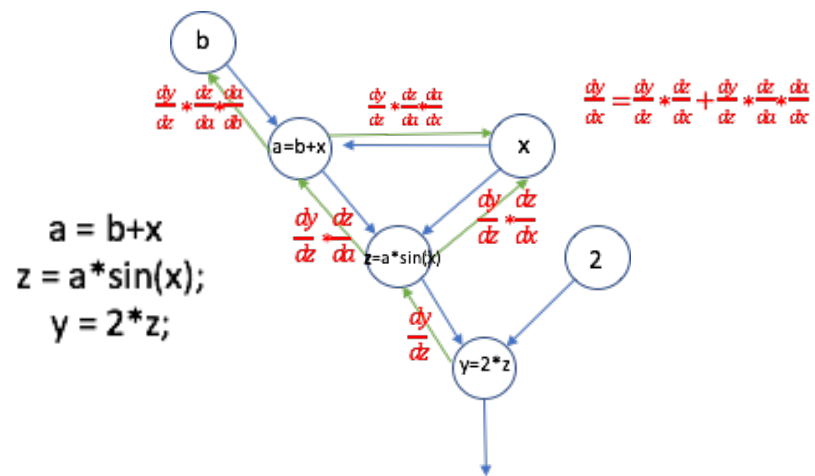
```
    [...]
```

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



ADAPT (2018)

Original C Code

```
#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

```
#include <iostream>
#include <adapt.h>
#include <adapt-impl.cpp> ] – AD Libraries

AD_real sum = 0.0;
AD_real inc = 0.1; ] – Type Changes

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

int main() {
    AD_begin();
    AD_independent(inc, "inc"); ] – Initialization
    do_sum();
    cout << AD_value(sum) << endl;

    AD_dependent(sum, "sum", 8);
    AD_report(); ] – Output
    return 0;
}
```

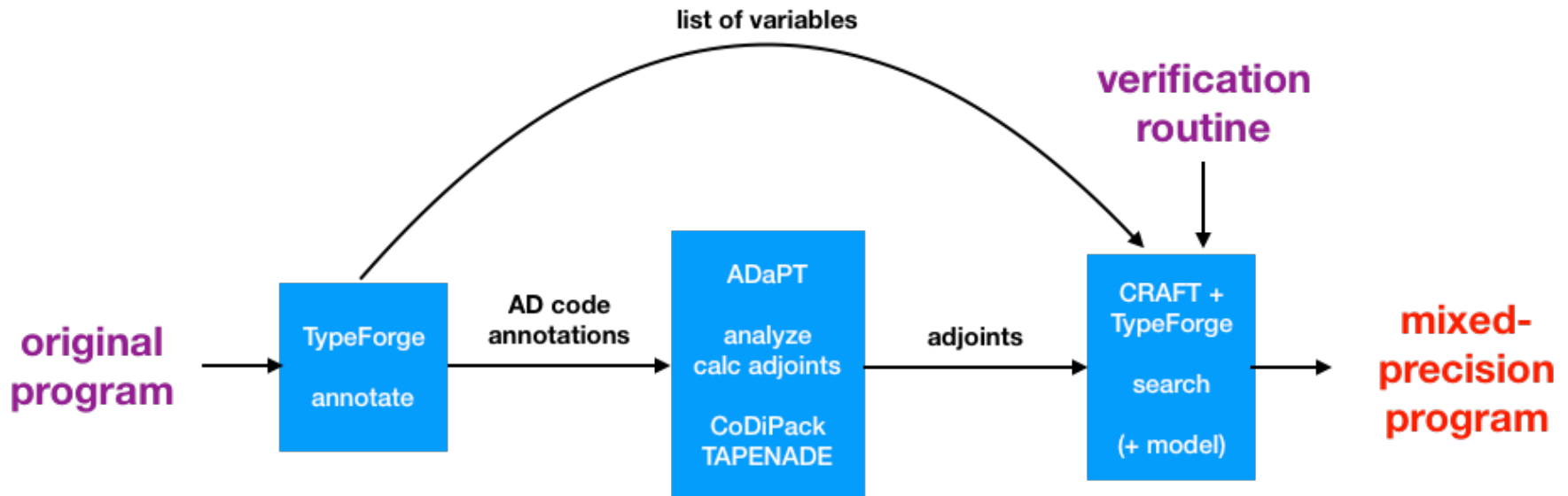
ADAPT (2018)

- 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

```
main
|__ TimeIncrement
|__ LagrangeLeapFrog
|  |__ LagrangeNodal
|  |  |__ CalcForceForNodes
|  |  |  |__ CalcVolumeForceForElems
|  |  |  |__ InitStressTermsForElems
|  |  |  |__ IntegrateStressForElems
|  |  |  |__ CollectDomainNodesToElemNodes
|  |  |  |__ CalcElemShapeFunctionDerivatives
|  |  |  |__ CalcElemNodeNormals
|  |  |  |__ SumElemFaceNormal
|  |  |  |__ SumElemStressesToNodeForces
|  |  |  |__ CalcHourglassControlForElems
|  |  |  |__ CollectDomainNodesToElemNodes
|  |  |  |__ CalcElemVolumeDerivative
|  |  |  |__ VoluDer
|  |  |  |__ CalcFBHourglassForceForElems
|  |  |  |__ CalcAccelerationForNodes
|  |  |  |__ ApplyAccelerationBoundaryConditionsForNodes
|  |  |  |__ CalcVelocityForNodes
|  |  |  |__ CalcPositionForNodes
|  |  |  |__ CalcQForElems
|  |  |  |__ CalcMonotonicQGradientsForElems
|  |  |  |__ CalcMonotonicQForElems
|  |  |  |__ CalcMonotonicQRegionForElems
|  |  |  |__ ApplyMaterialPropertiesForElems
|  |  |  |__ EvalEOSForElems
|  |  |  |__ CalcEnergyForElems
|  |  |  |__ CalcPressureForElems
|  |  |  |__ CalcSoundSpeedForElems
|  |  |  |__ UpdateVolumesForElems
|  |  |  |__ CalcTimeConstraintsForElems
|  |  |  |__ CalcCourantConstraintForElems
|  |  |  |__ CalcHydroConstraintForElems
```

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



FPHPC (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**

- **2016:** Michael O. Lam and Jeffrey K. Hollingsworth. “Fine-Grained Floating-Point Precision Analysis.” *Int. J. High Perform. Comput. Appl.* 32, 2 (March 2018), 231-245.
- **2013:** Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. “Automatically Adapting Programs for Mixed-Precision Floating-Point Computation.” In *Proceedings of the International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 369-378.
- **2011:** Michael O. Lam, Jeffrey K. Hollingsworth, and G. W. Stewart. “Dynamic Floating-Point Cancellation Detection.” *Parallel Comput.* 39, 3 (March 2013), 146-155.

- **SHVAL**

- **2017:** Ramy Medhat, Michael O. Lam, Barry L. Rountree, Borzoo Bonakdarpour, and Sebastian Fischmeister. “Managing the Performance/Error Tradeoff of Floating-point Intensive Applications.” *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 184 (October 2017), 19 pages.
- **2016:** Michael O. Lam and Barry L. Rountree. “Floating-Point Shadow Value Analysis.” In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools (ESPT '16)*. IEEE Press, Piscataway, NJ, USA, 18-25.

- **ADAPT**

- **2018:** Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. “ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning.” In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 48.

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

