Tools and Techniques for Floating-Point Analysis

Ignacio Laguna

Jan 7, 2020 @ LLNL

Modified version of:

IDEAS Webinar
Best Practices for HPC Software Developers Webinar Series
October 16, 2019

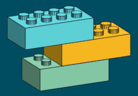
Lawrence Livermore National Laboratory

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-PRES-788144).

http://fpanalysistools.org/

What I will Present

1. Some interesting areas of floating-point analysis in HPC

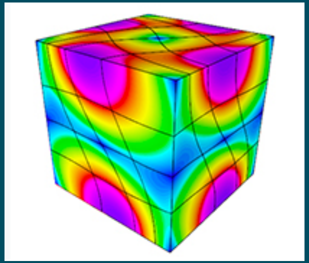
2. Potential issues when writing floating-point code
    - Will present ***principles*** 

3. Some tools (and techniques) to help programmers
    - Distinction between **research** and **tools**

Focus on
high-performance computing
applications

A Hard-To-Debug Case

Hydrodynamics mini application



Early development and porting to new system (IBM Power8, NVIDIA GPUs)

clang	–O1:	|e| = 129941.1064990107
clang	–O2:	|e| = 129941.1064990107
clang	–O3:	|e| = 129941.1064990107
gcc	–O1:	|e| = 129941.1064990107
gcc	–O2:	|e| = 129941.1064990107
gcc	–O3:	|e| = 129941.1064990107
xlc	–O1:	|e| = 129941.1064990107
xlc	–O2:	|e| = 129941.1064990107
xlc	–O3:	**|e| = 144174.9336610391**

It took several weeks of effort to debug it

IEEE Standard for Floating-Point Arithmetic (IEEE 754-2019)

● **Formats**:	how to represent floating-point data
● **Special numbers**:	Infinite, NaN, subnormal
● **Rounding rules:**	rules to be satisfied during rounding
● **Arithmetic operations:**	e.g., trigonometric functions
● **Exception handling**:	division by zero, overflow, ...

Do Programmers Understand IEEE Floating Point?

P. Dinda and C. Hetland, "Do Developers Understand IEEE Floating Point?," 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Vancouver, BC, 2018, pp. 589-598.

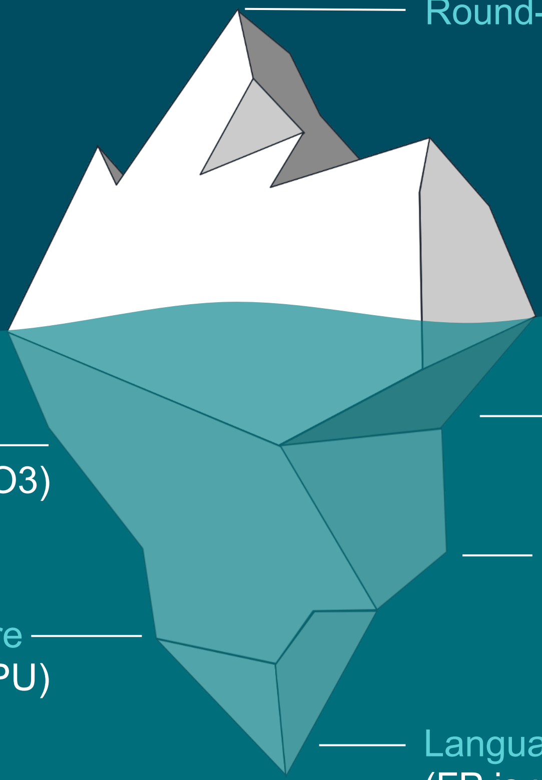
- Survey taken by 199 software developers
- Developers do little better than chance when quizzed about core properties of floating-point, yet are confident

Some misunderstood aspects:

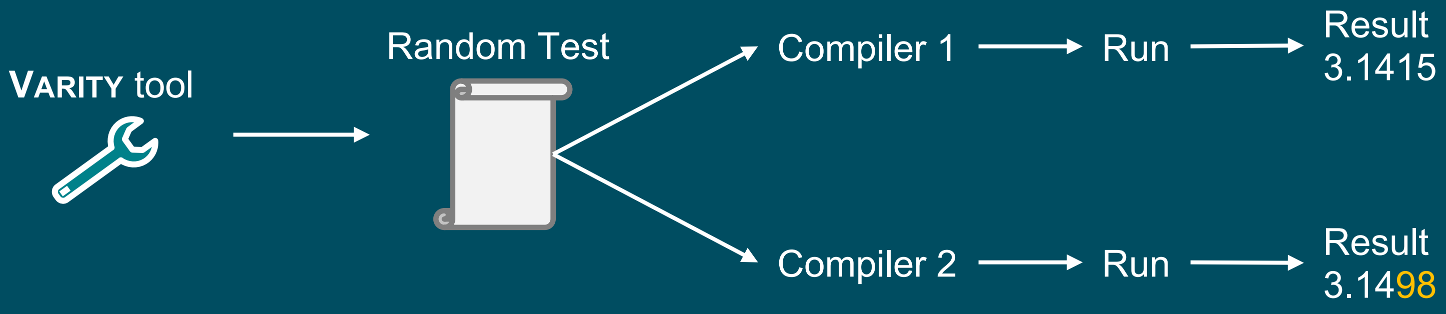
- Standard-compliant optimizations (-O2 versus –O3)
- Use of fused multiply-add (FMA) and flush-to-zero
- Can fast-math result in non-standard-compliant behavior?

Myth: *It's Just Floating-Point Error…Don't Worry*

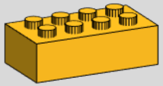
Many factors are involved in
unexpected numerical results



What Floating-Point Code Can be Produce Variability?



Principle 1



Optimization levels between compilers are not created equal

Example 1:
How Optimizations Can Bite Programmers

Random Test

```c
void compute(double comp,int var_1,double var_2,
double var_3,double var_4,double var_5,double var_6,
double var_7,double var_8,double var_9,double var_10,
double var_11,double var_12,double var_13,
double var_14) {
  double tmp_1 = +1.7948E-306;
  comp = tmp_1 + +1.2280E305 - var_2 +
    ceil((+1.0525E-307 - var_3 / var_4 / var_5));
  for (int i=0; i < var_1; ++i) {
    comp += (var_6 * (var_7 - var_8 - var_9));
  }
  if (comp > var_10 * var_11) {
    comp = (-1.7924E-320 - (+0.0 / (var_12/var_13)));
    comp += (var_14 * (+0.0 - -1.4541E-306));
  }
  printf("%.17g\n", comp);
}
```

Input

```
0.0 5 -0.0 -1.3121E-306 +1.9332E-313 +1.0351E-306
+1.1275E172 -1.7335E113 +1.2916E306 +1.9142E-319
+1.1877E-306 +1.2973E-101 +1.0607E-181 -1.9621E-306
-1.5913E118-O3
```

IBM Power9, V100 GPUs (LLNL Lassen)

clang –O3

```
$ ./test-clang
NaN
```

nvcc –O3

```
$ ./test-nvcc
-2.3139093300000002e-188
```

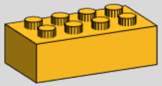
Example 2:
Can –O0 hurt you?

Random test

```
void compute(double tmp_1, double tmp_2, double tmp_3,
  double tmp_4, double tmp_5, double tmp_6) {
  if (tmp_1 > (-1.9275E54 * tmp_2 + (tmp_3 - tmp_4 * tmp_5)))
  {
      tmp_1 = (0 * tmp_6);
  }
  printf("%.17g\n", tmp_1);

  return 0;
}
```

Principle 2



Be aware of the **default behavior** of compiler optimizations

Input

```
+1.3438E306 -1.8226E305 +1.4310E306 -1.8556E305
-1.2631E305 -1.0353E3
```

IBM Power9 (LLNL Lassen)

clang –O0

```
$ ./test-clang
1.3437999999999999e+306
```

gcc –O0

```
$ ./test-gcc
1.3437999999999999e+306
```

xlc –O0

```
$ ./test-xlc
-0
```

Fused multiply-add (FMA) is used by default in XLC

Math Functions: C++ vs C

C
Using <math.h>

```
float a = 1.0f;
double b = sin(a);
```

0.8414709848078965

- <math.h> provides "`float sinf(float)`"
- Variable **a** is extended to double -> double-precision `sin()` is called

C++
Using <cmath>

```
float a = 1.0f;
double b = sin(a);
```

0.8414709568023681 6

- <cmath> provides "`float sin(float)`" in the std namespace
- Single-precision `sin()` is called -> result is extended to double precision

What is the most accurate?

FORTRAN: Compiler is Free to Apply Several Transformations

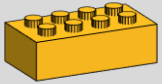
- FORTRAN compiler is free to apply mathematical identities
    - As long are they are valid in the Reals
    - a/b * c/d ➔ (a/b) * (c/d) or (a*c) / (b*d)
    - Mathematically equivalent ≠ same round-off error

- Due to compiler freedom, performance of FORTRAN is likely to be higher than C

Expression	*Allowable* alternative
X+Y	Y+X
X*Y	Y*X
-X + Y	Y-X
X+Y+Z	X + (Y + Z)
X-Y+Z	X - (Y - Z)
X*A/Z	X * (A / Z)
X*Y - X*Z	X * (Y - Z)
A/B/C	A / (B * C)
A / 5.0	0.2 * A

Source: Muller, Jean-Michel, et al. "Handbook of floating-point arithmetic.", 2010.

Principle 3



Be aware of the **language semantics**

How is Floating-Point Specified in Languages?

1. C/C++: moderately specified
2. FORTRAN: lower than C/C++
3. Python: underspecified

Python documentation warns about floating-point arithmetic:
https://python-reference.readthedocs.io/en/latest/docs/float/

float
These represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating point numbers; the savings in processor and memory usage that are usually the reason for using these is dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers.

Numpy package provides support for all IEEE formats

NVIDIA GPUs Deviate from IEEE Standard

- CUDA Programing Guide v10:
  - No mechanism to detect exceptions
  - Exceptions are always masked

H.2. Floating-Point Standard

All compute devices follow the IEEE 754-2008 standard for binary floating-point arithmetic with the following deviations:

- There is no dynamically configurable rounding mode; however, most of the operations support multiple IEEE rounding modes, exposed via device intrinsics;
- There is no mechanism for detecting that a floating-point exception has occurred and all operations behave as if the IEEE-754 exceptions are always masked, and deliver the masked response as defined by IEEE-754 if there is an exceptional event; for the same reason, while SNaN encodings are supported, they are not signaling and are handled as quiet;
- The result of a single-precision floating-point operation involving one or more input NaNs is the quiet NaN of bit pattern 0x7fffffff;
- Double-precision floating-point absolute value and negation are not compliant with IEEE-754 with respect to NaNs; these are passed through unchanged;

Tools & Techniques for Floating-Point Analysis

GPU Exceptions

- Floating-point exceptions
- GPUs, CUDA

Compiler Variability

- Compiler-induced variability
- Optimization flags

Mixed-Precision

- GPU mixed-precision
- Performance aspects

All tools available here

http://fpanalysistools.org/

Solved Problem: Trapping Floating-Point Exceptions in CPU Code

- When a CPU exceptions occurs, it is signaled
  - System sets a flag or takes a trap
  - Status flag FPSCR set by default
- The system (e.g., Linux) can also cause the floating-point exception signal to be raised
  - SIGFPE

Source: https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.genprogc/floating-point_except.htm

CUDA has Limited Support for Detecting Floating-Point Exceptions

- CUDA: programming language of NVIDIA GPUs
- CUDA has no mechanism to detect exceptions
  - As of CUDA version: 10
- All operations behave as if exceptions are masked

You may have **“hidden”** exceptions in your CUDA program

Detecting the Result of Exceptions in a CUDA Program

- Place **printf** statements in the code (as many a possible)

```
double x = 0;
x = x/x;
printf("res = %e\n", x);
```

- Programming checks are available in CUDA:

```
__device__   int isnan ( float  a );
__device__   int isnan ( double  a );
```

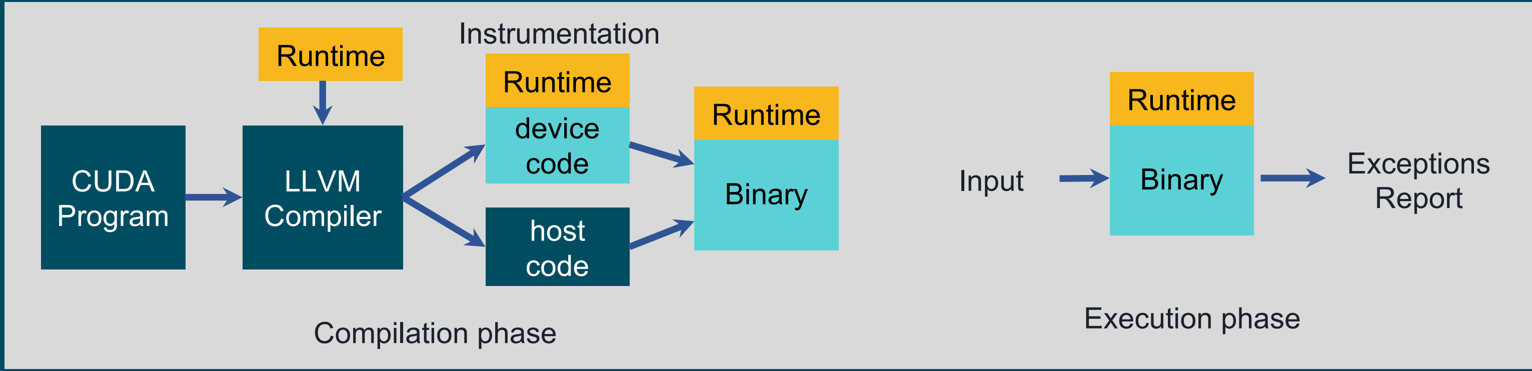
  - Also available isinf

These solutions are not ideal; they require significant programming effort

FPChecker

- Automatically detect the location of FP exceptions in NVIDIA GPUs
  - Report file & line number
  - No extra programming efforts required
- Report input operands
- Use software-based approach (compiler)
- Analyze optimized code

Workflow of FPChecker



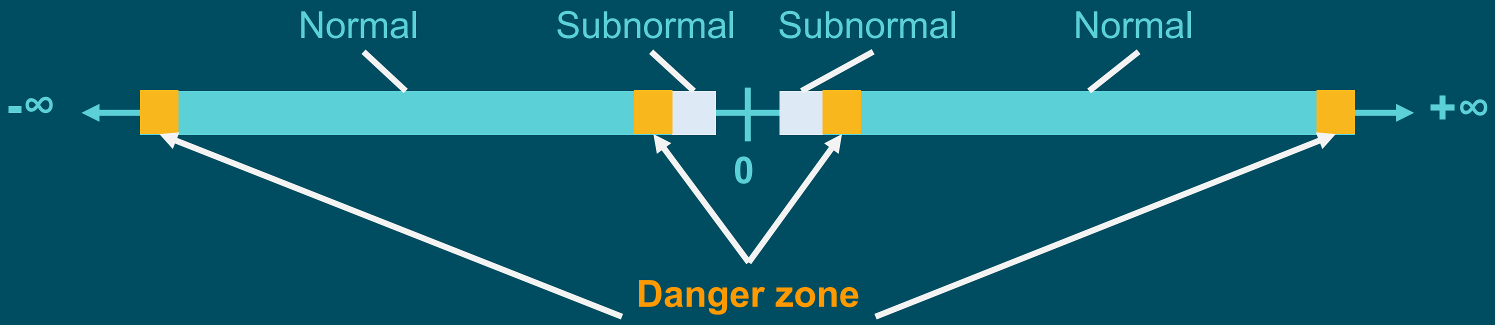
Example of Compilation Configuration for FPChecker

Use clang instead of NVCC

```
#CXX = nvcc
CXX = /path/to/clang++
CUFLAGS = -std=c++11 --cuda-gpu-arch=sm_60 -g
FPCHECK_FLAGS = -Xclang -load -Xclang /path/libfpchecker.so \
  -include Runtime.h -I/path/fpchecker/src
CXXFLAGS += $(FPCHECK_FLAGS)
```

- Load instrumentation library
- Include runtime header file

We report **Warnings** for Latent Underflows/Overflows



- -D **FPC_DANGER_ZONE_PERCENT**=x.x:
  a. Changes the size of the danger zone.
  b. By default, x.x is 0.10, and it should be a number between 0.0 and 1.0.

Example of Error Report

```
+--------------------------- FPChecker Error Report ----------------------------+
Error           : Underflow
Operation       : MUL (9.999888672e-321)
File            : dot_product_raja.cpp
Line            : 32
+-------------------------------------------------------------------------------+
```

Slowdown: 1.2x – 1.5x

Tools & Techniques for Floating-Point Analysis

GPU Exceptions

- Floating-point exceptions
- GPUs, CUDA

Compiler Variability

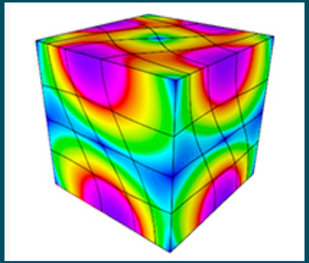
- Compiler-induced variability
- Optimization flags

Mixed-Precision

- GPU mixed-precision
- Performance aspects

A Hard-To-Debug Case

Hydrodynamics mini application



<u>Early development and porting to new system (IBM Power8, NVIDIA GPUs)</u>

clang	–O1:	|e| = 129941.1064990107
clang	–O2:	|e| = 129941.1064990107
clang	–O3:	|e| = 129941.1064990107
gcc	–O1:	|e| = 129941.1064990107
gcc	–O2:	|e| = 129941.1064990107
gcc	–O3:	|e| = 129941.1064990107
xlc	–O1:	|e| = 129941.1064990107
xlc	–O2:	|e| = 129941.1064990107
xlc	–O3:	**|e| = 144174.9336610391**

How to debug it?

Root-Cause Analysis Process

Buggy Program → File → Function (code region) → Line of Code

Delta Debugging

- Identifies **<u>input</u>** that makes problem manifest
  - **<u>Input</u>** for us: *file & function*
- Identifies minimum input
- Iterative algorithm
  - Average case: O(log N)
  - Worst case: O(N)

Delta Debugging Example

Input: $func_1$, $func_2$, $func_3$, $func_4$, $func_5$, $func_6$, $func_7$, $func_8$

Bug: Wrong results when:
1. $func_3$ and $func_7$ are compiled with high optimization
2. Remaining functions compiled low optimization

Step 1
Split input

$func_1$, $func_2$, $func_3$, $func_4$ | $func_5$, $func_6$, $func_7$, $func_8$

Step 2

chunk 1 → low optimization

$func_1$, $func_2$, $func_3$, $func_4$

chunk 2 → high optimization

$func_5$, $func_6$, $func_7$, $func_8$ ✓

chunk 1 → high optimization

$func_1$, $func_2$, $func_3$, $func_4$

chunk 2 → low optimization

$func_5$, $func_6$, $func_7$, $func_8$ ✓

Delta Debugging Example

Step 3 use chunks of finer granularity

$func_1$, $func_2$ / $func_3$, $func_4$ / $func_5$, $func_6$ / $func_7$, $func_8$

chunk 1 → low optimization

$func_1$, $func_2$

chunks 2,3,4 → high optimization

$func_3$, $func_4$, $func_5$, $func_6$, $func_7$, $func_8$

- Chunk 1 can be removed (also chunk 3 later)
- Restart from smaller input ($func_3$, $func_4$, $func_7$, $func_8$)
- Final result: $func_3$, $func_7$

Results: File & Function Isolated

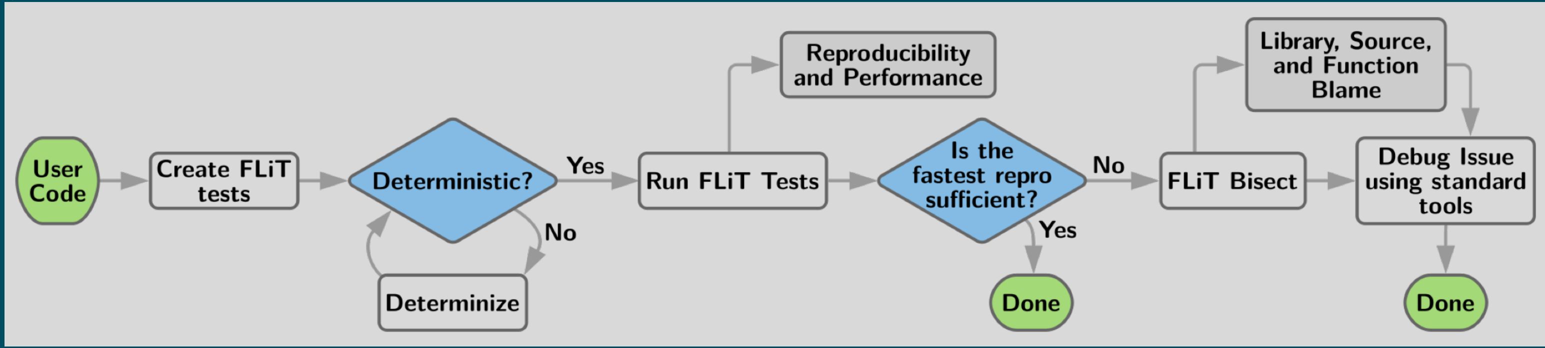
- File: raja/kernels/quad/rQDataUpdate.cpp
- Function: rUpdateQuadratureData2D

- Problem goes away when:
  - `rUpdateQuadratureData2D` compiled with –O2
  - Other functions with –O3

Optimization level	Energy
-O2	|e| = 129941.1064990107
-O3	|e| = 144174.9336610391
-O3 (except rUpdateQuadratureData2D)	|e| = 129664.9230608184

FLiT

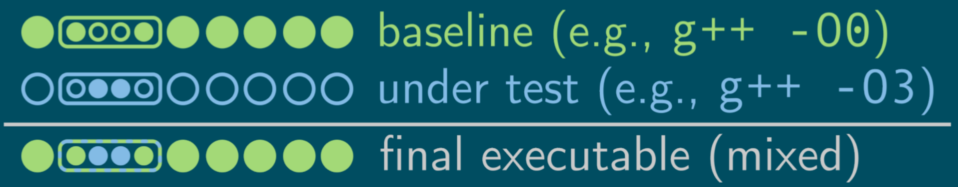
Michael Bentley
University of Utah



Multiple Levels:

- Determine variability-inducing compilations
- Analyze the tradeoff of reproducibility and performance
- Locate variability by identifying files and functions causing variability

Bisection Method

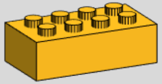


http://fpanalysistools.org/

Other Problems: Subnormal Numbers

- Subnormal numbers + -O3 = bad results

Principle 4



Avoid subnormal numbers if possible

- **Reason 1:** may impact performance
- **Reason 2:** you lose too much precision

Subnormal Numbers May be Inaccurate

```
double x = 1/3.0;
printf("Original    : %e\n", x);
x = x * 7e-323;
printf("Denormalized: %e\n", x);
x = x / 7e-323;
printf("Restored    : %e\n", x);
```

```
long double x = 1/3.0;
printf("Original    : %Le\n", x);
x = x * 7e-323;
printf("Denormalized: %Le\n", x);
x = x / 7e-323;
printf("Restored    : %Le\n", x);
```

```
Original    : 3.333333e-01
Denormalized: 2.470328e-323
Restored    : 3.571429e-01
```

```
Original    : 3.333333e-01
Denormalized: 2.305640e-323
Restored    : 3.333333e-01
```

Exact Computations for Subnormal Numbers

It can be proved that:

- Assuming that $RN(\,)$ is the rounding function operation
- If x, y are floating-point numbers, and
- $RN(x+y)$ is a subnormal number
- Then $RN(x+y) = x+y$, i.e., it is computed exactly

Hauser, John R. "Handling floating-point exceptions in numeric programs." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18, no. 2 (1996): 139-174.

Subnormal numbers resulting from addition or subtraction are exact

Not necessarily the case for division, multiplication, or other functions

How to Avoid Subnormal Numbers?

- Use higher precision
  - Research problem: could we selectively expand precision on some code?
- Scale up, scale down
  - Could work for simple problems only
  - You lose precision
- Flush underflows to zero
  - Doesn’t fix the underlying problem
  - Eliminates performance issues
- Algorithmic change

Tools & Techniques for Floating-Point Analysis

GPU Exceptions

- Floating-point exceptions
- GPUs, CUDA

Compiler Variability

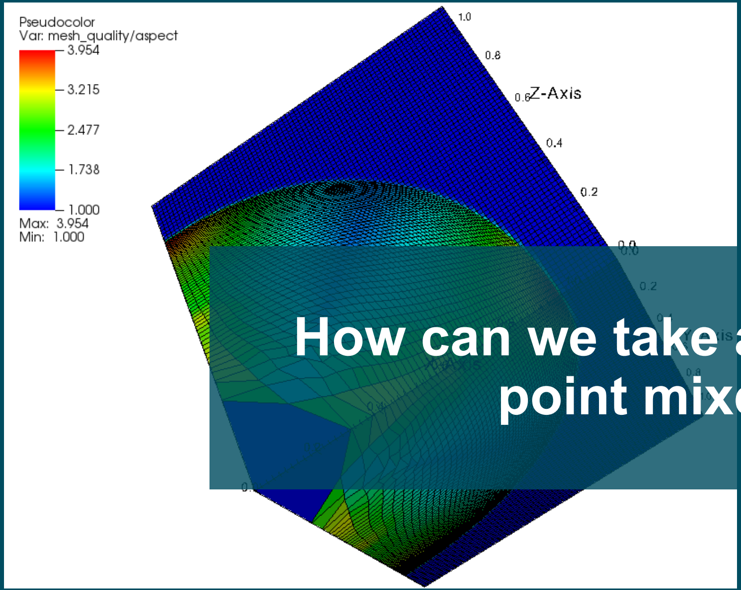
- Compiler-induced variability
- Optimization flags

Mixed-Precision

- GPU mixed-precision
- Performance aspects

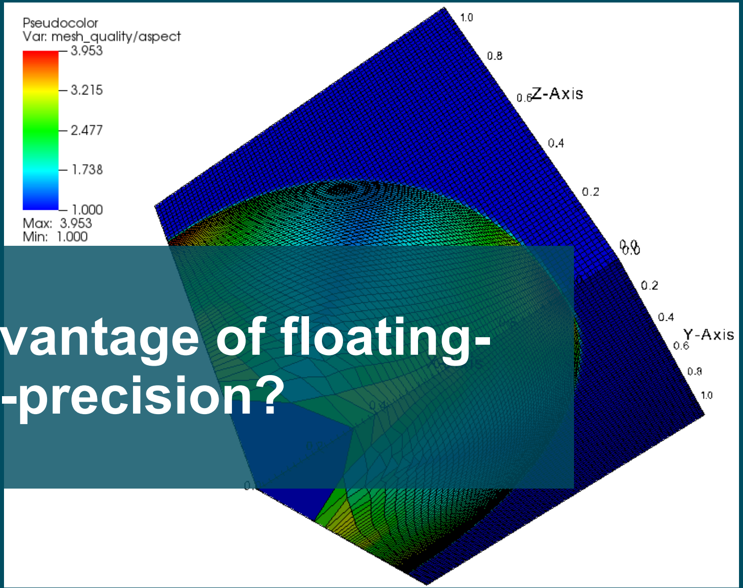
LULESH
NVIDIA P100 GPU

Run 1



FP64 (double precision)

Run 2



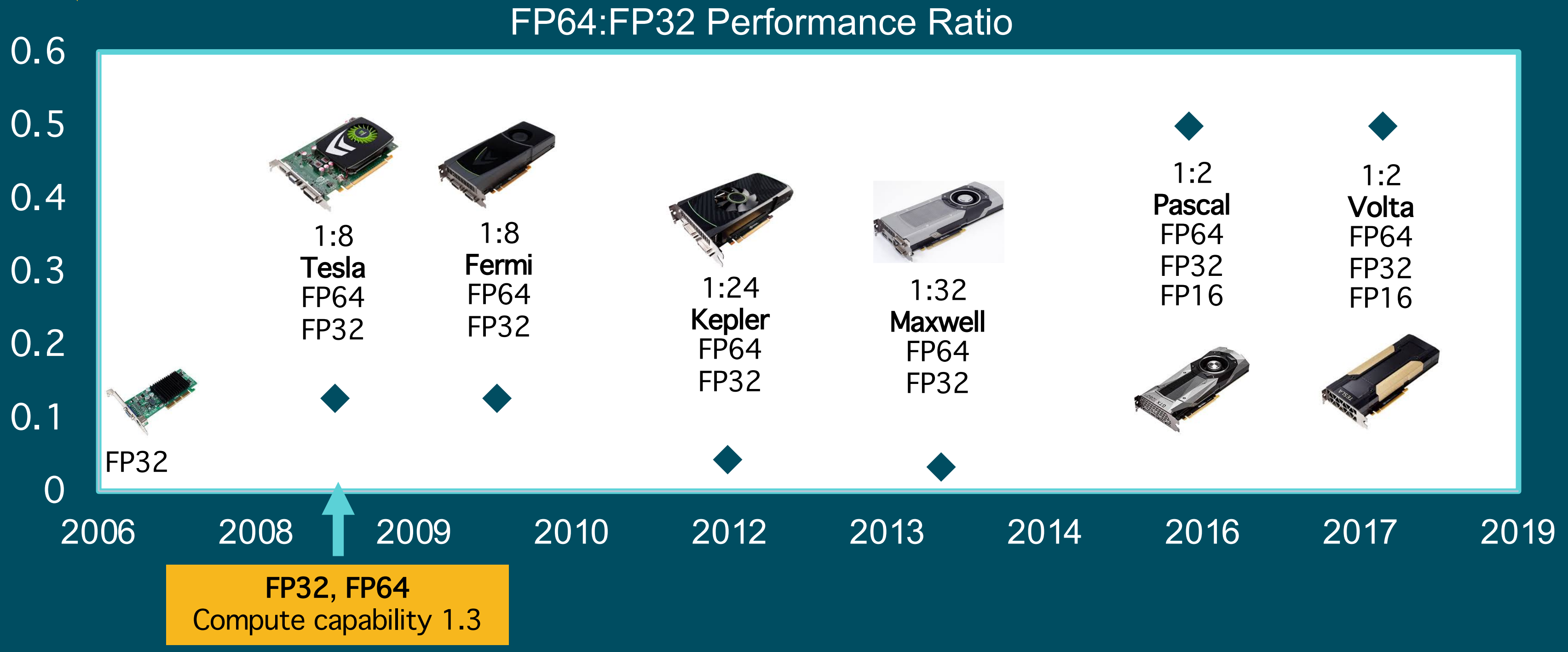
Mixed-Precision (FP64 & FP32)

How can we take advantage of floating-point mixed-precision?

6 digits of accuracy, **10%** speedup
3 digits of accuracy, **46%** speedup

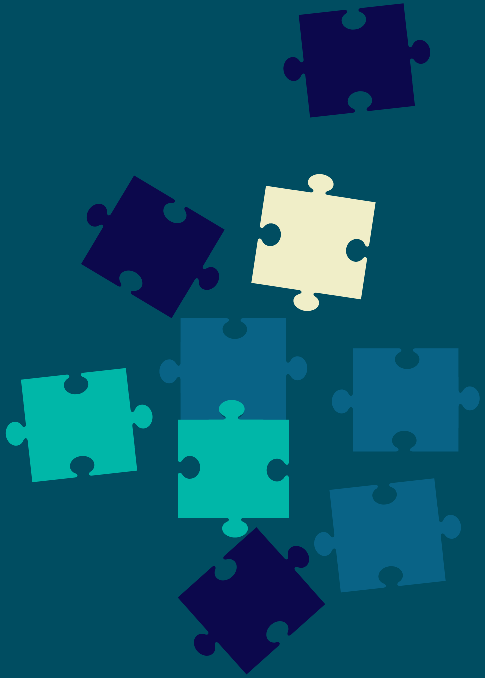
http://fpanalysistools.org/

Floating-Point Precision Levels in NVIDIA GPUs Have Increased



http://fpanalysistools.org/

Mixed-Precision Programing is Challenging



- Scientific programs have many variables
- {FP32, FP64} precision: **2^N combinations**
- {FP16, FP32, FP64} precision: **3^N combinations**

Example of Mixed-Precision Tuning

Force computation kernel in **n-body simulation** (CUDA)

```
__global__ void bodyForce(double *x, double *y,
 double *z, double *vx, double *vy, double *vz,
 double dt, int n)
{
 int i = blockDim.x * blockIdx.x + threadIdx.x;
 if (i < n) {
  double Fx=0.0; double Fy=0.0; double Fz=0.0;
  for (int j = 0; j < n; j++) {
   double dx = x[j] - x[i];
   double dy = y[j] - y[i];
   double dz = z[j] - z[i];
   double distSqr = dx*dx + dy*dy + dz*dz + 1e-9;
   double invDist = rsqrt(distSqr);
   double invDist3 = invDist * invDist * invDist;
   Fx += dx*invDist3; Fy += dy*invDist3; Fz += dz*invDist3;
  }
  vx[i] += dt*Fx; vy[i] += dt*Fy; vz[i] += dt*Fz;
 }
}
```

double -> float

Error of particle position (x,y,z)

$$\left|\frac{x-x_0}{x}\right| + \left|\frac{y-y_0}{y}\right| + \left|\frac{z-z_0}{z}\right|$$

(x,y,z): baseline position

(x_0,y_0,z_0): new configuration

http://fpanalysistools.org/

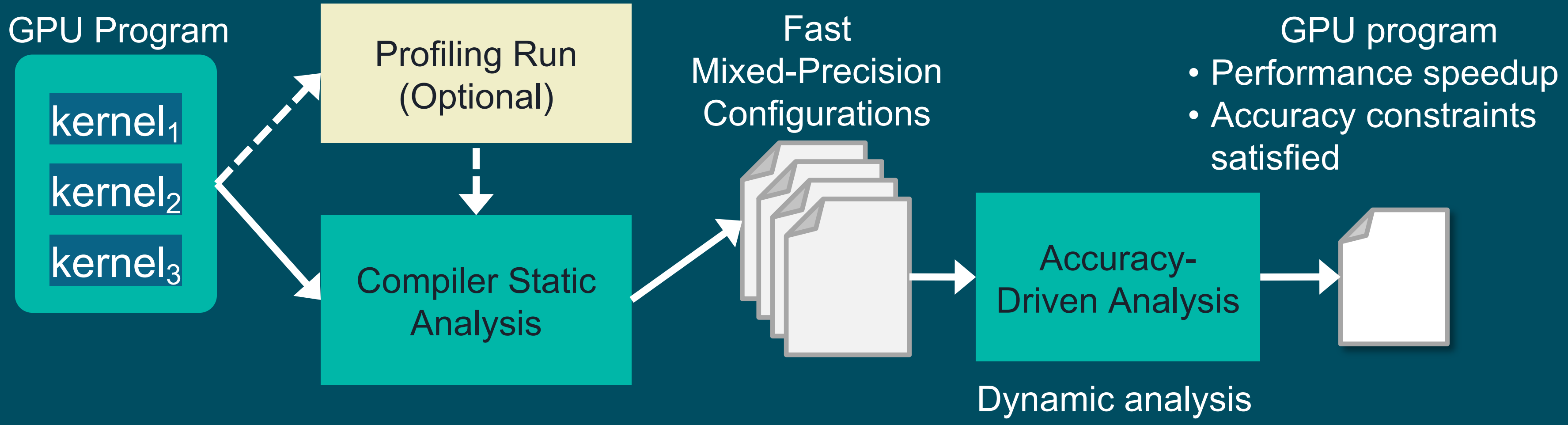
Example of Mixed-Precision Tuning (2)

Force computation kernel in **n-body simulation** (CUDA)

```
__global__ void bodyForce(double *x, double *y,
 double *z, double *vx, double *vy, double *vz,
 double dt, int n)
{
 int i = blockDim.x * blockIdx.x + threadIdx.x;
 if (i < n) {
  double Fx=0.0; double Fy=0.0; double Fz=0.0;
  for (int j = 0; j < n; j++) {
   double dx = x[j] - x[i];
   double dy = y[j] - y[i];
   double dz = z[j] - z[i];
   double distSqr = dx*dx + dy*dy + dz*dz + 1e-9;
   double invDist = rsqrt(distSqr);
   double invDist3 = invDist * invDist * invDist;
   Fx += dx*invDist3; Fy += dy*invDist3; Fz += dz*invDist3;
  }
  vx[i] += dt*Fx; vy[i] += dt*Fy; vz[i] += dt*Fz;
 }
}
```

No.	Variables in FP32	Error	Speedup(%)
1	All	15.19	53.70
2	invDist3	4.08	5.78
3	distSqr	1.93	-43.35
4	invDist3, invDist, distSqr	1.80	11.69

GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications



Ignacio Laguna, Paul C. Wood, Ranvijay Singh, Saurabh Bagchi. GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications. ISC High Performance, Frankfurt, Germany, Jun 16-20, 2019 **(Best paper award)**

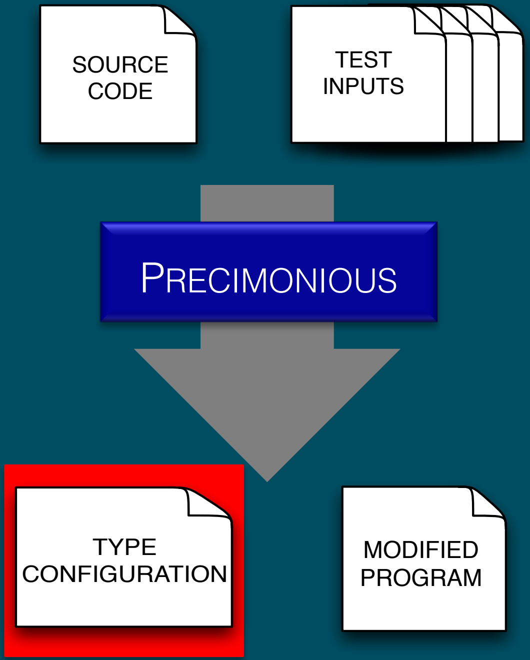
Precimonious

“Parsimonious or Frugal with Precision”

Cindy Rubio González
University of California, Davis

Dynamic Analysis for Floating-Point Precision Tuning

Annotated with error threshold



Less Precision

↓

Speedup

Modified program in executable format

http://fpanalysistools.org/

ADAPT: Algorithmic Differentiation for Error Analysis

Computer architectures support multiple levels of precision

- Higher precision – improves accuracy
- Lower precision – reduces run time, memory pressure, and energy consumption

APPROACH

For a given y = f(x)

First order Taylor series approximation at x=a

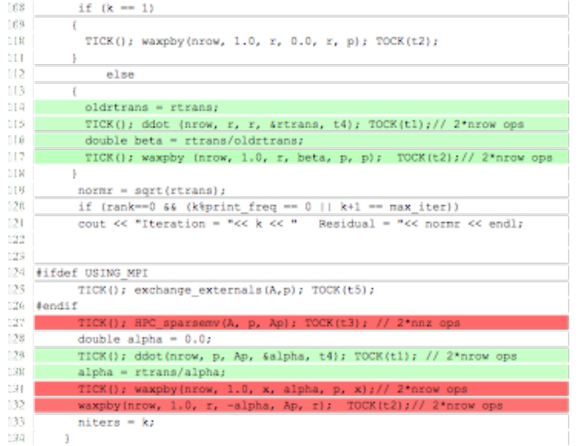
$$
\begin{aligned}
y &= f(x) \\
&= f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \dots \\
&\approx f(a) + f'(a)(x-a).
\end{aligned}
$$

$\Delta y = f'(a)\,\Delta x$

Obtain f'(a) using Algorithmic Differentiation (AD)

http://fpanalysistools.org/

Identifies critical sections that need to be in higher precision



```
    if (k == 1)
   {
     TICK(); waxpby(nrow, 1.0, r, 0.0, r, p); TOCK(t2);
   }
       else
   {
     oldrtrans = rtrans;
     TICK(); ddot (nrow, r, r, &rtrans, t4); TOCK(t1);// 2*nrow ops
     double beta = rtrans/oldrtrans;
     TICK(); waxpby (nrow, 1.0, r, beta, p, p);  TOCK(t2);// 2*nrow ops
   }
    normr = sqrt(rtrans);
    if (rank==0 && (k%print_freq == 0 || k+1 == max_iter))
    cout << "Iteration = "<< k << "   Residual = "<< normr << endl;


#ifdef USING_MPI
    TICK(); exchange_externals(A,p); TOCK(t5);
#endif
    TICK(); HPC_sparsemv(A, p, Ap); TOCK(t3); // 2*nnz ops
    double alpha = 0.0;
    TICK(); ddot(nrow, p, Ap, &alpha, t4); TOCK(t1); // 2*nrow ops
    alpha = rtrans/alpha;
    TICK(); waxpby(nrow, 1.0, x, alpha, p, x);// 2*nrow ops
    waxpby(nrow, 1.0, r, -alpha, Ap, r);  TOCK(t2);// 2*nrow ops
    niters = k;
   }
```

Mixed precision speedup:

- 1.1x HPCCG (Mantevo benchmark suite)
- 1.2x LULESH

Harshitha Menon et al., ADAPT: Algorithmic Differentiation Applied to Floating-point Precision Analysis. SC'18
https://github.com/LLNL/adapt-fp

Tutorial on Floating-Point Analysis Tools @ SC19

http://fpanalysistools.org/

- Demonstrates several analysis tools
- Hands-on exercises
- Covers various important aspects

- Tutorials
  - SC19, Denver, Nov 17th, 2019
  - PEARC19, Chicago, Jul 30th, 2019

Some Useful References

General Guidance

- P. Dinda and C. Hetland, “Do Developers Understand IEEE Floating Point?”
  - https://doi.ieeecomputersociety.org/10.1109/IPDPS.2018.00068
- Do not use denormalized numbers (CMU, Software Engineering Institute)
  - https://wiki.sei.cmu.edu/confluence/display/java/NUM54-J.+Do+not+use+denormalized+numbers
- The Floating-point Guide
  - https://floating-point-gui.de/
- John Farrier “Demystifying Floating Point” (youtube video)
  - https://www.youtube.com/watch?v=k12BJGSc2Nc&t=2250s
- David Goldberg. “What every computer scientist should know about floating-point arithmetic”. ACM Comput. Surv. 23, 1 (March 1991), 5-48.
  - https://doi.org/10.1145/103162.103163

NVIDIA GPUs & Floating-Point

- Floating Point and IEEE 754 Compliance for NVIDIA GPUs
  - https://docs.nvidia.com/cuda/floating-point/index.html
- Mixed-Precision Programming with CUDA 8
  - https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/

In Summary

- Many factors can affect floating-point results
  - Compilers, hardware, optimizations, precision, parallelism, …
  - Be aware of how compiler optimizations could change results
- Be aware of default behavior of compiler optimizations
- Be aware of language semantics
- Avoid the use subnormal numbers if possible
- Pay attention to floating-point computations on GPUs
- Mixed precision involves correctness and performance analysis

Funding support provided by BSSw and ECP

Contact: ilaguna@llnl.gov

Disclaimer