Tools and Techniques for Floating-Point Analysis

Ignacio Laguna
Lawrence Livermore National Laboratory

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

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).
What You will Learn

1. Some interesting areas of floating-point analysis in HPC
2. Potential issues when writing floating-point code
3. Some tools (and techniques) to help programmers

Focus on high-performance computing applications
# A Hard-To-Debug Case

Hydrodynamics mini application

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Optimization Level</th>
<th>Error Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang</td>
<td>-O1</td>
<td></td>
</tr>
<tr>
<td>clang</td>
<td>-O2</td>
<td></td>
</tr>
<tr>
<td>clang</td>
<td>-O3</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>-O1</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>-O2</td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td>-O3</td>
<td></td>
</tr>
<tr>
<td>xlc</td>
<td>-O1</td>
<td></td>
</tr>
<tr>
<td>xlc</td>
<td>-O2</td>
<td></td>
</tr>
<tr>
<td>xlc</td>
<td>-O3</td>
<td></td>
</tr>
</tbody>
</table>

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

It took several weeks of effort and many methods to debug it
IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008)

- **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, etc.
Do Programmers Understand IEEE Floating Point?


- 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-complaint behavior?
Myth: It’s Just Floating-Point Error...Don’t Worry

Many factors are involved

Optimizations (be careful with –O3)
Floating-point precision
Compiler (proprietary vs. open-source)
Architecture (CPU != GPU)
Language semantics (FP is underspecified in C)
Other factors

Incorrect number 144174.9336

http://fpanalysis.tools.org/
What Floating-Point Code Can be Produce Variability?
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

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

IBM Power9, V100 GPUs (LLNL Lassen)

c clang –O3

c clang –O3

c nvcc –O3

c nvcc –O3

$ ./test-nvcc
-2.313909330000002e-188

http://fpanalyistools.org/
Example 2: Can –O0 hurt you?

Random tests

```c
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;
}
```

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
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;

http://fpanalysistools.org/
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

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 as possible)

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

- Programming checks are available in CUDA:

```c
__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

Compilation phase

- CUDA Program
- LLVM Compiler
- Runtime
- Instrumentation
  - Runtime device code
  - host code
- Binary

Execution phase

- Input
- Binary
- Runtime
- Exceptions Report

http://fpanalysistools.org/
Example of Compilation Configuration for FPChecker

Use clang instead of NVCC

```plaintext
#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

http://fpanalysisistools.org/
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

How to debug it?

http://fpanalysisistools.org/
Root-Cause Analysis Process

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

http://fpanalysistools.org/
Delta Debugging

- Identifies **input** that makes problem manifest
  - **Input** for us: *file & function*
- Identifies minimum input
- Iterative algorithm
  - Average case: $O(\log N)$
  - Worst case: $O(N)$
Delta Debugging Example

**Input:** \( \text{func}_1, \text{func}_2, \text{func}_3, \text{func}_4, \text{func}_5, \text{func}_6, \text{func}_7, \text{func}_8 \)

**Bug:** Wrong results when:
1. \( \text{func}_3 \) and \( \text{func}_7 \) are compiled with high optimization
2. Remaining functions compiled low optimization

**Step 1**
Split input

\[ \text{func}_1, \text{func}_2, \text{func}_3, \text{func}_4 \quad / \quad \text{func}_5, \text{func}_6, \text{func}_7, \text{func}_8 \]

**Step 2**

- **chunk 1 → low optimization**
  - \( \text{func}_1, \text{func}_2, \text{func}_3, \text{func}_4 \)
  - \( \text{func}_5, \text{func}_6, \text{func}_7, \text{func}_8 \)

- **chunk 1 → high optimization**
  - \( \text{func}_1, \text{func}_2, \text{func}_3, \text{func}_4 \)
  - \( \text{func}_5, \text{func}_6, \text{func}_7, \text{func}_8 \)

- **chunk 2 → high optimization**
  - \( \text{func}_5, \text{func}_6, \text{func}_7, \text{func}_8 \)

- **chunk 2 → low optimization**
  - \( \text{func}_5, \text{func}_6, \text{func}_7, \text{func}_8 \)
Delta Debugging Example

Step 3  use chunks of finer granularity

chunk 1 → low optimization  
chunk 2, 3, 4 → high optimization

- 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

http://fpanalyistools.org/
Results: File & Function Isolated

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

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

<table>
<thead>
<tr>
<th>Optimization level</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O2</td>
<td>$</td>
</tr>
<tr>
<td>-O3</td>
<td>$</td>
</tr>
<tr>
<td>-O3 (except rUpdateQuadratureData2D)</td>
<td>$</td>
</tr>
</tbody>
</table>
Other Problems: Subnormal Numbers

- Subnormal numbers + -O3 = **bad results**

- Suggestion: **Do not use subnormal numbers!**
  - **Reason 1:** may impact performance
  - **Reason 2:** you lose too much precision
Subnormal Numbers are 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
How to Avoid Subnormal Numbers?

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

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

Bisection Method

baseline (e.g., g++ -00)
under test (e.g., g++ -03)
final executable (mixed)
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
How can we take advantage of floating-point mixed-precision?

FP64 (double precision)

LULESH NVIDIA P100 GPU

Run 1

Run 2

FP64 (double precision) and Mixed-Precision (FP64 & FP32)

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

http://fpanalysisistools.org/
Floating-Point Precision Levels in NVIDIA GPUs Have Increased

FP64:FP32 Performance Ratio

- **Tesla** (FP64:FP32 1:8)
- **Fermi** (FP64:FP32 1:8)
- **Kepler** (FP64:FP32 1:24)
- **Maxwell** (FP64:FP32 1:32)
- **Pascal** (FP64:FP32 1:2)
- **Volta** (FP64:FP32 1:2)

FP32, FP64, Compute capability 1.3

http://fpanalysistools.org/
Mixed-Precision Programming is Challenging

- Scientific programs have many variables
- \{\text{FP32, FP64}\} precision: \(2^N\) combinations
- \{\text{FP16, FP32, FP64}\} precision: \(3^N\) combinations
Example of Mixed-Precision Tuning

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

```c
__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;
    }
}
```

Error of particle position

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

\((x,y,z)\): baseline position

\((x_0,y_0,z_0)\): new configuration

http://fpanalysistools.org/
GPUMixer: Performance-Driven Floating-Point Tuning

define their own metric for error, however, for this illustrative case, we define the relative error introduced by mixed-precision as:

$$error = \left( \frac{|x - x_0|}{x} + \frac{|y - y_0|}{y} + \frac{|z - z_0|}{z} \right) \times 100,$$

where $x, y, z$ are the particle positions for the baseline, and $x_0, y_0, z_0$ are the particle positions for a new configuration.

```c
__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;
    }
}
```

Table 1 shows the particle values, error, and performance speedup of four configuration with respect to the baseline, case 1. Case 2 shows the configuration where all variables in the kernel are declared as FP32, i.e., as `float`. We observe that while the speedup is significant, 53%, the error is high, 15.19. Case 3 shows the case where only variable `invDist3` is declared as FP32 and the rest as FP64—in this case the error decreases, but the speedup is not too high, only 5%. Case 4 shows an interesting case: when the variable `invDist3` is the only one declared as FP32, the error is very low, but the speedup is negative, i.e., performance degrades. Case 5 shows the best we found when the `distSqr, invDist` and `invDist3` variables are declared as FP32: the error is lower than as in case 4 while the speedup is about 11%. This example illustrates that some configurations can produce low performance speedup or even performance degradation; the goal of our approach is to find via static analysis configurations such as 3 and 5 that improve performance and discard cases such as 4.

3.2 Configurations

While mixed-precision configurations can be expressed in terms of the precision of variable declarations (as in the previous example), a more precise approach is to express configurations in terms of the precision of floating-point operations. The reason behind this is that a variable can be used in multiple floating-point operations; the precision of each of these operations can be decreased/increased.

**Example of Mixed-Precision Tuning (2)**

<table>
<thead>
<tr>
<th>No.</th>
<th>Variables in FP32</th>
<th>Error</th>
<th>Speedup(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All</td>
<td>15.19</td>
<td>53.70</td>
</tr>
<tr>
<td>2</td>
<td>invDist3</td>
<td>4.08</td>
<td>5.78</td>
</tr>
<tr>
<td>3</td>
<td>distSqr</td>
<td>1.93</td>
<td>-43.35</td>
</tr>
<tr>
<td>4</td>
<td>invDist3, invDist, distSqr</td>
<td>1.80</td>
<td>11.69</td>
</tr>
</tbody>
</table>
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”

Dynamic Analysis for Floating-Point Precision Tuning

Annotated with error threshold

Source Code
Test Inputs

Precimonious

Less Precision

Type Configuration
Modified Program

Modified program in executable format

http://fpanalysistools.org/

Cindy Rubio González
University of California, Davis
**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 \( x=a \)

\[
y = f(x) = f(a) + f'(a)(x - a) + \frac{f''(a)}{2!}(x - a)^2 + \ldots \\
\approx f(a) + f'(a)(x - a).
\]

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

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

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

http://fpanalysisistools.org/
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
Correctness 2019: Third International Workshop on Software Correctness for HPC Applications @ SC19

- November 18, 2019 (full day), at SC19 (Denver)
- There is a session on floating-point mixed precision
- URL: https://correctness-workshop.github.io/2019/
Some Useful References

General Guidance
  - [https://doi.ieeecomputersociety.org/10.1109/IPDPS.2018.00068](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](https://wiki.sei.cmu.edu/confluence/display/java/NUM54-J.+Do+not+use+denormalized+numbers)
- The Floating-point Guide
  - [https://floating-point-gui.de/](https://floating-point-gui.de/)
- John Farrier “Demystifying Floating Point” (youtube video)
  - [https://www.youtube.com/watch?v=k12BjGSc2Nc&t=2250s](https://www.youtube.com/watch?v=k12BjGSc2Nc&t=2250s)
  - [https://doi.org/10.1145/103162.103163](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](https://docs.nvidia.com/cuda/floating-point/index.html)
- Mixed-Precision Programming with 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
- Avoid the use subnormal numbers (you lose too much precision)
- Pay attention to floating-point computations on GPUs
- Mixed precision involves correctness and performance analysis
- Tools community is your friend

Contact: ilaguna@llnl.gov

Funding support provided by BSSw and ECP

http://fpanalysisistools.org/
Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.