The Grass is Really Green on the Other Side

Empirical vs. Rigorous in Floating-Point Precision Analysis, Reproducibility, Resilience

Ganesh Gopalakrishnan
School of Computing, University of Utah, Salt Lake City, UT 84112

www.cs.utah.edu/fv

pruners.github.io

collaborations with students, Utah colleagues and LLNL (Dong Ahn), PNNL (Sriram Krishnamoorthy)
On Floating-Point Errors

Numerical errors are rare, rare enough not to care about them all the time, but yet not rare enough to ignore them.

— William M. Kahan
Empirical vs. Rigorous Methods

- **Empirical (testing-based) methods are hugely important**
  - “Beware of bugs in the above code; I have only proved it correct, not tried it.”
    -- Knuth

- **But do not give us insights into code behavior across all inputs**
  - “Program testing can be used to show the presence of bugs, but never to show their absence!”
    -- Dijkstra

- **Rigorous methods can help develop Better Empirical Methods**
  - “We should continually be striving to transform every art into a science: in the process, we advance the art.”
    -- Knuth

*Empirical*: “based on, concerned with, or verifiable by observation or experience rather than theory or pure logic”
Some Current Challenges in Floating-Point

• Floating-point code seldom carries rounding bounds
  • a rigorous guarantee of rounding error across intervals of inputs
  • Inferred specifications (rounding bounds extracted from code) can be useful

• Precision allocation is often done without rigorous guarantees
  • the resulting code may prove to be brittle

• Non-reproducibility (due to numerics) is a net productivity loss
  • code outlives hardware/compilers; answers may change after porting

• Soft errors can skew the numerics
  • long-running codes may harbor silent data corruptions

• Compiler bugs can also result in aberrant numerical results
  • compilers that reschedule operations are complex and have exhibited bugs
Unified Handling of Challenges

• We have developed rigorous approaches for roundoff error analysis

• These methods have proven valuable/promising to address “unrelated” challenges in detecting
  • soft errors, (published)
  • profiling precision, (in progress)
  • guarding against compiler bugs, (published)
  • and hopefully also reproducibility (TBD)
This Talk

• Scalable Rigorous Precision Estimation methods
  • Compute roundoff errors - tool SATIRE
    • Scalable Abstraction-guided Technique for Inferring Rounding Errors
  • Analytically bound roundoff errors - tool FPDetect

• Rigorous ways to catch soft errors (“bit flips”)
  • FP Detect results in soft-error detectors that come with guarantees
  • Empirical ways to guard against polyhedral compiler bugs
    • FP Detect detectors can also help catch compiler bugs

• Empirical methods for reproducibility
  • Our FLiT tool can help isolate submodules thwarting reproducibility
  • Rigorous methods can help FLiT advance

Conclusions
Coauthor and Funding credits

- **SATIRE:** Arnab Das (PhD stud), Ian Briggs (PhD stud), Sriram Krishnamoorthy (PNNL), Pavel Panchekha (U of U)

- **FLiT:** Michael Bentley (PhD stud), Ian Briggs (U), Dong H Ahn, Ignacio Laguna, Gregory L. Lee, Holger E. Jones (LLNL)

- **FPDetect:** Arnab Das, Ian Briggs, Sriram Krishnamoorthy, Ramakrishna Tipireddy (PNNL)

- **Funding:** NSF and DOE Contract (PNNL, LNNL)
  - NSF CCF – Awards: 1704715, 1817073, 1918497
Publications

• SATIRE

• FPDetect
  • making its way through TACO (ask us for a copy)

• FLiT
  • In HPDC’19: https://dl.acm.org/doi/10.1145/3307681.3325960
  • To appear in CACM (ETA this year?)
  • https://github.com/PRUNERS
Let’s begin with roundoff error estimation
Nobody likes to use the actual Rounding error

Rounding error while adding $x$ and $y$ in $[-1, 2]$
Background:
One seeks Tight and Rigorous upper bounds

```c
double jetEngine(double x1, double x2) {
    double t = 3 * x1 * x1 + 2 * x2 - x1
    double q = x1 * x1 + 1
    double p = t / q
    double s1 = 2 * x1 * p * (p - 3)
    double s2 = x1 * x1 * (4 * p - 6)
    double s3 = 3 * x1 * x1 * x1
    double s4 = x1 * x1 * x1
    double s5 = 3 * p
    return x1 + ((s1 + s2) * q + s3 + s4 + x1 + s5)
}
```

Deriving such error functions for very large Floating-point expressions can be quite handy! How to do this?

Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamaric, Ganesh Gopalakrishnan:
Question 1: Onto Scalable Roundoff Estimation

- Rigorous roundoff analysis methods have had limited scalability
- **SATIRE** offers enhanced scalability
  - to a point that more meaningful insights can be obtained
  - Could have helped our mixed-precision work on FPTuner (POPL’17) scale

- Has many strengths over today’s empirical alternatives
  - Shadow-value based
  - Statistical

- Exceeds the best tools in its class in terms of scalability
Question 2: Scalable Reproducibility

- The numerical behavior of a program can change upon porting
- Sources:
  - Different IEEE rounding, presence of FMA, ...
  - Non-IEEE optimizations
  - Falling into C-undefined behaviors
  - Truly ill-conditioned code
  - Compiler bugs
- Need: A tool that can help root-cause non-reproducibility
- Our tool FLiT is promising in this regard
  - It is a search-based tool that is empirical
    - finds repro issues for the test inputs chosen
Question 3: What are the Resilience Concerns?

- Bit-flips can destroy results
  - So can polyhedral compiler bugs

- But what is a “golden answer”? 

- Is there a way to detect bit-flips / compiler bugs without knowing these golden answers?

- Traditional approaches
  - DMR: code duplication

- Our approach
  - Predict “virtually roundoff-free values” to appear “T steps ahead”
  - FPDetect is promising in this regard
Organization of the rest of the talk

• High-level details of
  • Satire,
  • FLiT, and
  • FPDetect

• How to bring the communities closer...
  • Papers on FP analysis appear in “PL conferences” and “HPC conferences”
  • They have a different take on things, different criteria for rigor, etc.
Scalable Abstraction-guided Tool for Incremental Reasoning of floating-pt Error
Tools similar to SATIRE

- Gappa ("error = diff between less accurate and more accurate" - G. Melquiond - courtesy F. de Dinechin)
- Rosa
- Real2Float
- Precisa
- FPTaylor (best in the class)
- ... (see FPTaylor TOPLAS’19 paper for others)

- SATIRE focuses on first-order error (sensitivity study in our arxiv)
  - Allows it to scale to > 1M operator nodes
  - Comparable rigorous tools handled only about 100 nodes
  - Satire’s bounds are almost always tighter (ignoring second-order error)
SATIRE in one slide

A Large Expression DAG

Intervals of values For each Input of the Expression DAG

Maximum Absolute Error across all points in the input intervals

“not just interval analysis” (more like “symbolic affine”)

Example DAGs

Fig. 1. Simplified 1D stencil over 6 time steps
Examples studied using SATIRE

- Unrolled loops coming from
  - PDE solvers
    - Stencils
  - Others
    - Mat-Mat
  - Of special interest
    - Scan
    - FFT -- SATIRE’s bounds are better than published analytical bounds

- Tricky ones
  - Lorenz equations

- Why unroll loops?
  - Finding tight loop invariants is very hard (for FP code)
  - Unrolled loop analysis can lend insights
  - Just to obtain meaningful large expression DAGS (and face the scaling challenges!)
How SATIRE works

- Symbolic reverse-mode A/D
  - Derivative-strength of Out wrt n
- Keep expressions canonicalized
- Multiply forward error at n
- Compute incrementally
- Abstract when Err_n becomes large
Scalability-Related Lessons Learned

- Scalability is a function of
  - Amount of Non-linearity
  - Number of reconvergent forward paths
  - Examples
    - FDTD goes to ~200K operators (without abstractions)
    - Lorenz system: Bottlenecked at ~200 operators (without abstractions)

- Good canonicalization is key (we win over other tools due to this)
- Good abstraction heuristics are key (Shannon-Info measure used by us)

- Symbiotic uses of SATIRE with Empirical tools is a promising path
  - Have developed a promising method to estimate relative error

- Ability to extract and publish specifications can be a good practice!
A Floating-point Litmus Tester
FLiT helps keep Science on Keel when Software Moves

- Yesterday’s results \leftrightarrow \text{SAME ???} \rightarrow Results a decade later

- Compilers \quad \text{Evolution}

- Hardware \quad \text{Evolution}
A Medium-Sized Example: MFEM

- Open-source finite element library
  - Developed at LLNL
  - [https://github.com/mfem/mfem.git](https://github.com/mfem/mfem.git)
- Provides many example use cases
- Represents real-world code

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>source files</td>
<td>97</td>
</tr>
<tr>
<td>average functions per file</td>
<td>31</td>
</tr>
<tr>
<td>total functions</td>
<td>2,998</td>
</tr>
<tr>
<td>source lines of code</td>
<td>103,205</td>
</tr>
</tbody>
</table>
MFEM Results displayed after FLiT-search

<table>
<thead>
<tr>
<th>compiler</th>
<th>optl</th>
<th>switches</th>
<th>comparison</th>
<th>nanosec</th>
</tr>
</thead>
<tbody>
<tr>
<td>clang++-6.0 -O3</td>
<td></td>
<td>-ffast-math</td>
<td>0.0</td>
<td>2857386994</td>
</tr>
<tr>
<td>clang++-6.0 -O3</td>
<td></td>
<td>-funsafe-ma</td>
<td>0.0</td>
<td>2853588952</td>
</tr>
<tr>
<td>clang++-6.0 -O3</td>
<td></td>
<td>-mfma</td>
<td>0.0</td>
<td>2858789982</td>
</tr>
<tr>
<td>g++-7</td>
<td>-O3</td>
<td>-ffast-math</td>
<td>0.0</td>
<td>2841191528</td>
</tr>
<tr>
<td>g++-7</td>
<td>-O3</td>
<td>-funsafe-ma</td>
<td>0.0</td>
<td>2868636192</td>
</tr>
<tr>
<td>g++-7</td>
<td>-O3</td>
<td>-mfma</td>
<td>193.007351</td>
<td>2797305220</td>
</tr>
</tbody>
</table>

One compilation had 193% relative error (also ran faster!)
Other compilations had no error.
Which site in the source-code is responsible?
The code you run is not what you "see"
We offer FLiT, a tool to study FP optimizations.

FLiT is a reproducibility test framework in the PRUNERS toolset (pruners.github.io).

Hundreds of compilations are compared against a “ground-truth” compilation.
FLiT Workflow

1. User Code → Create FLiT tests
2. Deterministic?
   - Yes → Run FLiT Tests
   - No → Determinize
3. Run FLiT Tests
4. Is the fastest repro sufficient?
   - Yes → FLiT Bisect, then Done
   - No → Reproducibility and Performance
5. Reproducibility and Performance
   - Yes → FLiT Bisect, then Done
   - No → Library, Source, and Function Blame
6. Library, Source, and Function Blame
   - Debug Issue using standard tools
   - Done

Note: The workflow is iterative and may require multiple passes through some steps.
FLiT Bisect: File or Symbol

File Bisect

\[ \begin{array}{c|c}
\text{fn}_1 & \text{fn}_2 \\
\text{fn}_3 & \text{fn}_4 \\
\text{fn}_5 & \text{fn}_6 \\
\end{array} \]

Symbol Bisect

\[ \begin{array}{c|c}
\text{fn}_1 & \text{fn}_2 \\
\text{fn}_3 & \text{fn}_4 \\
\text{fn}_5 & \text{fn}_6 \\
\end{array} \]
FLiT Bisect: File or Symbol

File Bisect
\[ \begin{array}{c}
\text{fn}_1 \\
\text{fn}_2 \\
\text{fn}_3 \\
\text{fn}_4 \\
\text{fn}_5 \\
\text{fn}_6 \\
\end{array} \]

Symbol Bisect
\[ \begin{array}{c}
\text{fn}_1 \\
\text{fn}_2 \\
\text{fn}_3 \\
\text{fn}_4 \\
\text{fn}_5 \\
\text{fn}_6 \\
\end{array} \]

THESE ARE FRANKEN-BINARIES!!
FLiT-Bisect under Singleton Minimal Set

<table>
<thead>
<tr>
<th>Speed</th>
<th>Files / symbols and their optimization levels (shaded = optimized)</th>
<th>Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Speed" /></td>
<td><img src="image" alt="Files/Symbols" /></td>
<td><img src="image" alt="Correctness" /></td>
</tr>
<tr>
<td><img src="image" alt="Speed" /></td>
<td><img src="image" alt="Files/Symbols" /></td>
<td><img src="image" alt="Correctness" /></td>
</tr>
<tr>
<td><img src="image" alt="Speed" /></td>
<td><img src="image" alt="Files/Symbols" /></td>
<td><img src="image" alt="Correctness" /></td>
</tr>
<tr>
<td><img src="image" alt="Speed" /></td>
<td><img src="image" alt="Files/Symbols" /></td>
<td><img src="image" alt="Correctness" /></td>
</tr>
<tr>
<td><img src="image" alt="Speed" /></td>
<td><img src="image" alt="Files/Symbols" /></td>
<td><img src="image" alt="Correctness" /></td>
</tr>
</tbody>
</table>

Blame
Significant “finds” using FLiT

• Test-13 within the MFEM library
• Compiler optimization that involved
  • AVX2, FMA, Higher precision intermediate floating-point values
• Result has 190% relative error
• Symbol-based bisection narrows down problem
  • Single function calculating $M = M + a \cdot A \cdot A^T$
• Conversation with developers underway
• Significant Recent Milestone:
  • Recently a 1M LOC critical export-control code was analyzed by FLiT and we found issues (root-caused to a small number of LOC)
  • FLiT recently deployed on LLNL machines (contact Mike Colette, Olga Pearce, …)
Interesting Find: Non-Portability Between IBM XLC and GCC

```c
#define xsw(a, b) a^=b^=a^=b.
```

Nan in IBM XLC ; “OK value” in GCC 😊
Compilation, Repro, Speedup (Two MFEM Tests)

- **Fastest bitwise equal:**
  - `g++ -O3`
  - Speedup: 1.128

- **Fastest variable:**
  - `g++ -O3 -mavx2 -mfma`
  - Speedup: 1.044
  - Variability: 2.99e-13

- **Fastest bitwise equal:**
  - `clang++ -O3`
  - Speedup: 1.094

- **Fastest variable:**
  - `icpc -O3 -fp-model fast=1`
  - Speedup: 1.396
  - Variability: 7.78e-14

Graphs show:
- **Compilation**
- Blue dots represent bitwise equal to baseline.
- Red crosses indicate variability.
FPDetect:
Using the computed tight rounding-error bound to detect “unexpected numerics”
System Resilience: Need


**Figure 1**

Taxonomy of derating terms.

**Figure 2**

POWER6 test system mounted in beamline.
Why is System Resilience (research/deployment) Stymied?

- Nobody wants slowdown ("bit-flips == fake-news")
  - Myopic when facing end-of-Moore
    - P. Bose of IBM, Engelmann, Cappello, .... many other other studies
- No good error detector!
  - High overhead
  - Omissions
  - False positives
- Way forward
  - Focus on specific aspects
  - Cover that well with
    - **Rigorous Guarantees**, High Detection, Low Overhead, **No False Positives**
Our contributions

• What is FP Detect?
  • Method to detect errors in the “data space”
    • Currently for Stencil code
    • Distinguishes “normal data” from SDC-laden data

• How does FP Detect work?
  • Infers round-off error as a specification
  • If observed value aberration is more than inferred, then “something else is going on”, which could be
    • Bit flips
    • Compiler bugs
FPDetect Mental Model

• “Skate where the puck will be”
  • W. Gretsky
Predict “almost Real” value at t+6

• When “computation reaches” t+6, check modulo rounding loss
  (“Error = diff between less accurate and more accurate” - G. Melquiond, courtesy F. de Dinechin)
  • We have omitted details that show that we can fire the detectors infrequently and still guarantee coverage across a whole protected volume

Fig. 1. Simplified 1D stencil over 6 time steps
Obtaining “almost Real” value T-steps ahead

• Compute the “Expected Answer” based on full path-strength
  • evaluated through “unfolding”,
  • implemented using Kahan’s sum
  • vectorization to minimize overheads

<table>
<thead>
<tr>
<th>Set Number</th>
<th>Coefficients at Offsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset index →</td>
<td>-3</td>
</tr>
<tr>
<td>Set₁</td>
<td>0</td>
</tr>
<tr>
<td>Set₂</td>
<td>0</td>
</tr>
<tr>
<td>Set₃</td>
<td>0.015625</td>
</tr>
</tbody>
</table>

Table 1. Evolution of Coefficient values as we unroll the stencil
Error Detection in the Abstract

• Challenges overcome in our analysis
  • tightly estimate error across all reconvergent paths
  • eliminate paths that have been dominated by others
    • Passes our empirical checks ... but ...
    • Need: Formal Verification tool that can check our “ferocious” hand calculations
      • Why3?

Fig. 2. Illustration of path dominance
FPDetect for Logical Bug Detection

We found that comparable soft-error detectors (SSD and AID) do not serve as effective logical-bug detectors.
Summary

- **Rigorous** method Satire (see arxiv paper) for error analysis
  - May be used symbiotically with Empirical tools for relative-error estimation, informing where to tune, etc.

- **Empirical** tool FLiT (see HPDC’19, CACM forthcoming)
  - Needs Rigorous methods to define coverage to be above “just luck”

- **Rigorous** tool FPDetect (TACO, accepted)
  - Uses rigorous methods, and has spawned empirical precision profiling methods
    - Learn by how much the FPDetect detector is pessimistic wrt. the shadow-value
    - Learn how much precision exists in a bounding volume
    - At run-time, do not use any shadow-value (only the detector)
    - Give report of “true” precision in bounding volume
Plea

• More rigorous tools needed
  • More bridges between Rigorous and Empirical needed
    • Helps scale
    • Helps tech-transfer

More workshops like this needed!