Compiler Tools for Mixed-Precision Tuning

Presented at:
ATPESC 2026, July 30, 2026

Ignacio Laguna
Center for Applied Scientific Computing (CASC)

Prepared by LLNL under Contract DE-AC52-07NA27344.

Roadmap

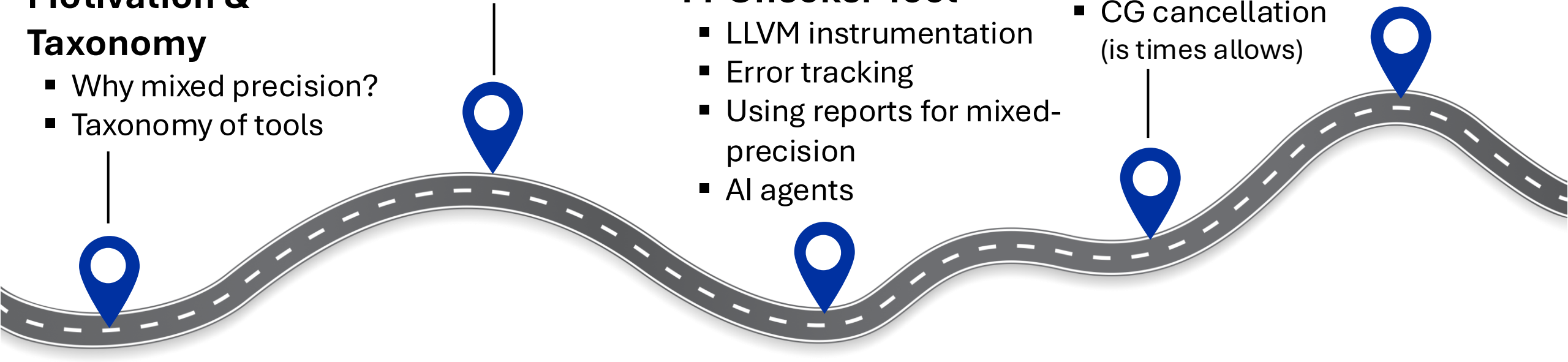
Motivation & Taxonomy
- Why mixed precision?
- Taxonomy of tools

Rounding Error Basics
- Relative error
- Cancellation

FPChecker Tool
- LLVM instrumentation
- Error tracking
- Using reports for mixed-precision
- AI agents

Examples
- LULESH (AI agent)
- CG cancellation (is times allows)

Demo
- Error reports
- Mixed-precision example in small program



Why Reduced Precision?

Example of recent NVIDIA GPUs

	2022	2024	2026
	H100 SXM Hopper	B200 Blackwell	Rubin
FP64	34 TFLOPS	37 TFLOPS	33 TFLOPS
FP64 Tensor Core	67 TFLOPS	37 TFLOPS	
FP32	67 TFLOPS	75 TFLOPS	130 TFLOPS
FP16 Tensor Core	1,979 TFLOPS	2,250 TFLOPS	4,000 TFLOPS
BF16 Tensor Core	1,979 TFLOPS	2,250 TFLOPS	4,000 TFLOPS
INT8 Tensor Core	3,958 TOPS	4,500 TOPS	250 TOPS
Memory Bandwidth	3.35 TB/s	8 TB/s	22 TB/s

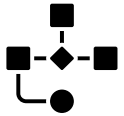
- FP64 FLOP/s not increasing
- Increased FLOP/s for lower precision: FP32, FP16, BF16

Sources:
https://resources.nvidia.com/en-us-hopper-architecture/nvidia-tensor-core-gpu-datasheet?ncid=no-ncid
https://www.spheron.network/blog/nvidia-b200-complete-guide/
https://www.nvidia.com/en-us/data-center/vera-rubin-nvl72/

Advances Needed to Achieve Mixed Precision

Mathematical

- Deeper theoretical foundation



Algorithmic

- Algorithms must be redesigned to be *"precision-aware"*

Tools

- Understand sensitive code regions automatically
- Transform code

} Compiler tools

Mixed Precision Goals

- **Definition:** *storing or computing different parts of an application at different precision levels*
- **Goal:** keep most of the performance benefits of low precision while preserving accuracy where it matters

Mixed-precision (compiler tools)

✓ **Static**
- Code is compiled once
- Runs multiple times

Dynamic
- Code could be *recompiled* at runtime
- JIT

Challenges: *Why is Mixed-Precision Difficult?*

Competing Goals

Performance
- Use memory bandwidth better
- Lower energy consumption

Numerical Stability
- Accuracy
- Trusted numerical results

- Accuracy problems are often:
  - Data dependent
  - Hard to predict from code inspection alone
- Manual tuning is slow
  - Often ends in over-promoting too much code to higher or lower precision (FP32 or FP64)

Compilers can transform the program,
but they need evidence

Questions to Answer When Developing a Mixed-Precision Code

1. What code can be changed (in terms of precision)?
2. What error occurs / appears in real runs?
3. Why is the error growing?
4. Is the code fragile?
5. Can I prove this change is safe?
6. How to ship it (compiler integration)?

Taxonomy of Compiler Tools for Mixed Precision

These tools help us answer the previous questions

Automatic Precision Tunning

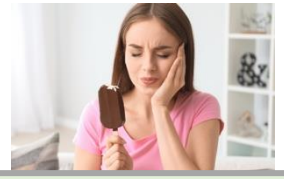
- Change precision of variables, arrays, kernels
- Use compiler **passes** or **annotations** to transform code
- User provides **accuracy** constraints

Precimonious, HiFPTuner, RLTuner

Dynamic Error Measurement

- Instrument code to compare to higher precision
- **Shadow value** & execution
- Track **instabilities** (INF, NaN)
- Increase precision where error is high
- Use LLVM compiler (typically)

FPChecker, Herbgrind, FPSanitizer, EFTSanitizer



Instability and Sensitivity Analysis

- **Monte Carlo Arithmetic** or Stochastic Arithmetic
- Insert **perturbations** to code and see how robust it is
- **AD** to detect code sensitive to changes in the input

MCA: Verificarlo, Verrou, CADNA
AD: Enzyme, Tapenade

Static Error Bounds

- **Static analysis** to identify ***error bounds***
- Interval arithmetic, affine arithmetic, SMT, or Taylor-model;
- Can be applied to math expressions or small kernels only
- Use results to justify **promotion** or **demotion**

FPTaylor, Daisy, Gappa, Fluctuat, Rosa, PRECiSA

Testing Methods

- Test code in different precisions
- Run performance and numerical tests
- Verify code can run at a certain precision level
- Most used in practice today

Libraries: MPFR, double-double, C++, Boost.Multiprecision

Roadmap

Motivation & Taxonomy

- Why mixed precision?
- Taxonomy of tools

Rounding Error Basics

- Relative error
- Cancellation

FPChecker Tool

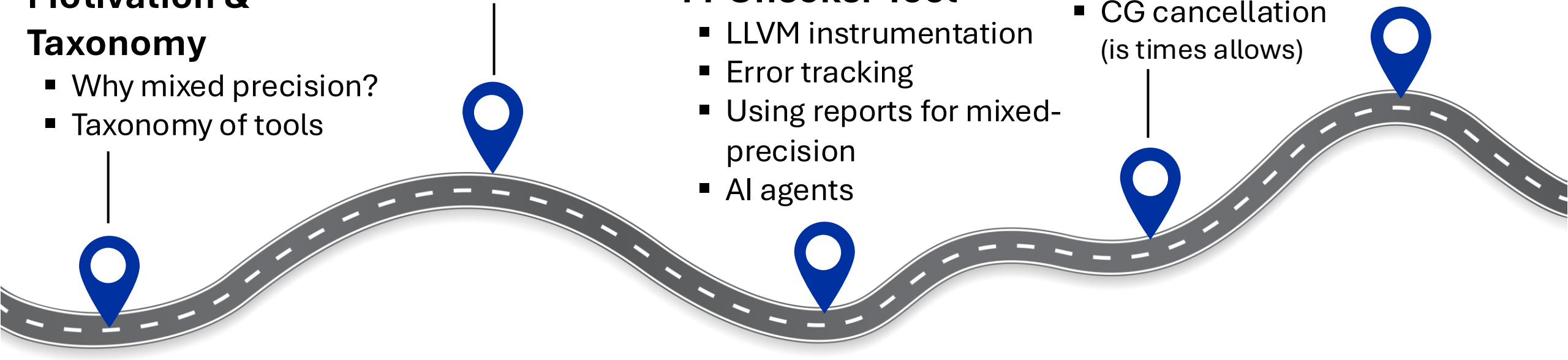
- LLVM instrumentation
- Error tracking
- Using reports for mixed-precision
- AI agents

Examples

- LULESH (AI agent)
- CG cancellation (is times allows)

Demo

- Error reports
- Mixed-precision example in small program

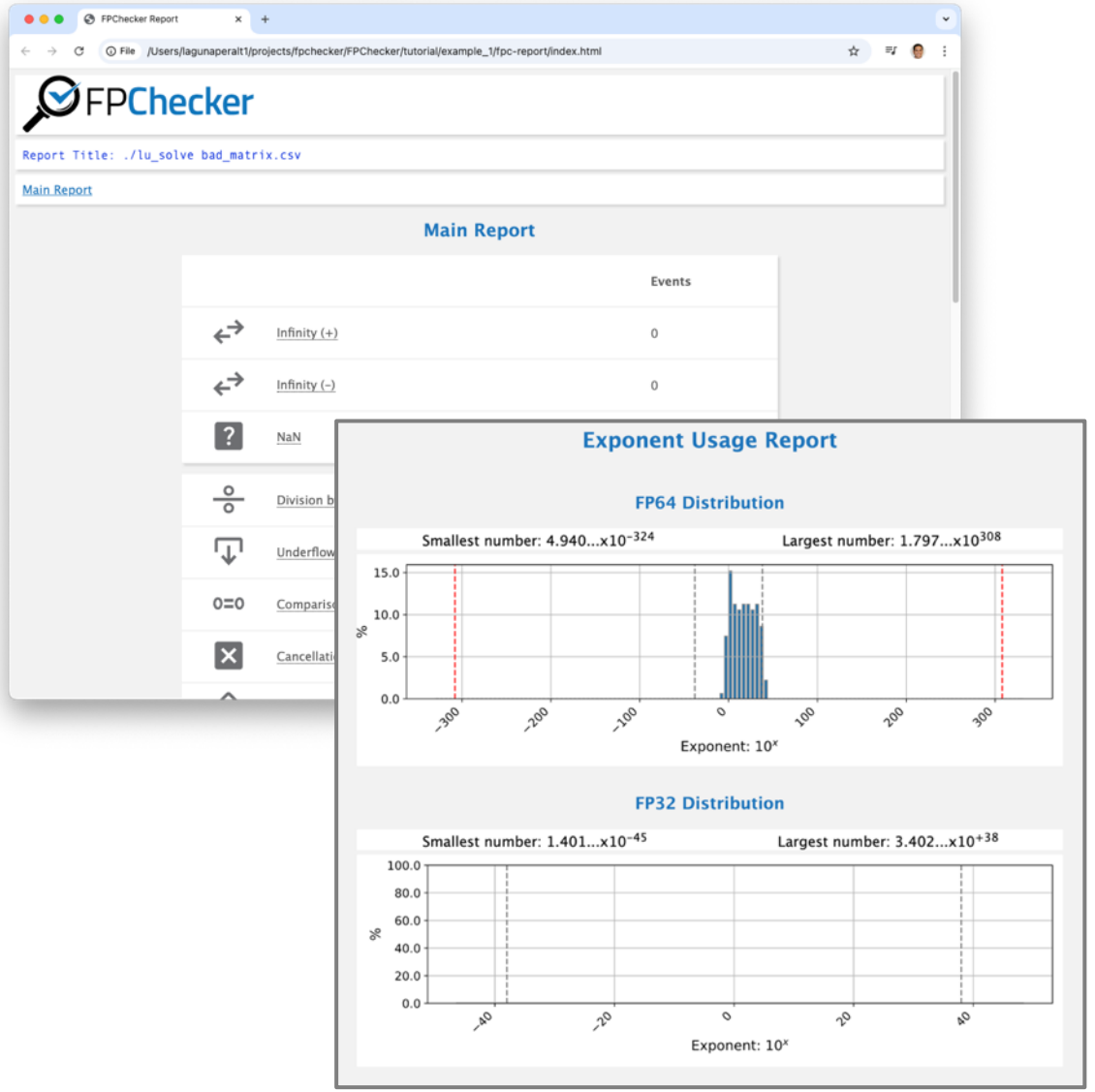


Dynamic Error Measurement with FPChecker

https://fpchecker.org/

- Detects floating-point exceptions
  - `NaN, Infinity`
- Shows impacted *lines of code*
- Shows other "code smells"
  - `Cancellations, underflows`
- Analyzes dynamic range
  - Is FP32 or FP64 enough
- Works with compiler instrumentation
  - Uses Clang/LLVM

- **Round Error Tracking**
  - Helpful for ***mixed-precision tunning***
  - Release 0.6



Relative Rounding Error is Most Useful for Mixed-Precision

$$\text{Relative Error} = \frac{|\text{Experimental} - \text{Theoretical}|}{|\text{Theoretical}|}$$

Absolute error is huge, but **relative error** is the same

Real Math Value	Computer Representation (7-digit limit)	Rounding Error	Relative Rounding Error
1.23456**789**	1.234568	+0.00000011	8.91×10^{-8}
123456**789**.0	123456800.0	+11.0	8.91×10^{-8}

- IEEE 754: relative error is bounded by a constant, **Machine Epsilon** (ϵ)
- Regardless of how big the number is
  - For FP32: $\boldsymbol{\epsilon \approx 10^{-7}}$
  - For FP64: $\boldsymbol{\epsilon \approx 10^{-16}}$

Cancellation Errors

- Occurs when ***subtracting two nearly identical numbers***
- It's the most aggressive way to lose precision
- Can delete entire result in a single subtraction
- Primary reason scientific simulations *"blow up"*

Suppose:

```
x = 1.0000000000000001
y = 1.0000000000000002
```

Mathematically: `y-x = 0.0000000000000001`

In floating-point: `y-x = 2.2204460492503130e-16`

Large relative error!

Because of Cancellation Relative Error Can Be 100% (or More)

$$Relative\ error = \frac{|Approximation\ - True\ Value|}{|True\ Value|}$$

a_{true} = 1234567.891
b_{true} = 1234567.890

$c_{true} = a_{true} - b_{true}$
c_{true} = 0.001

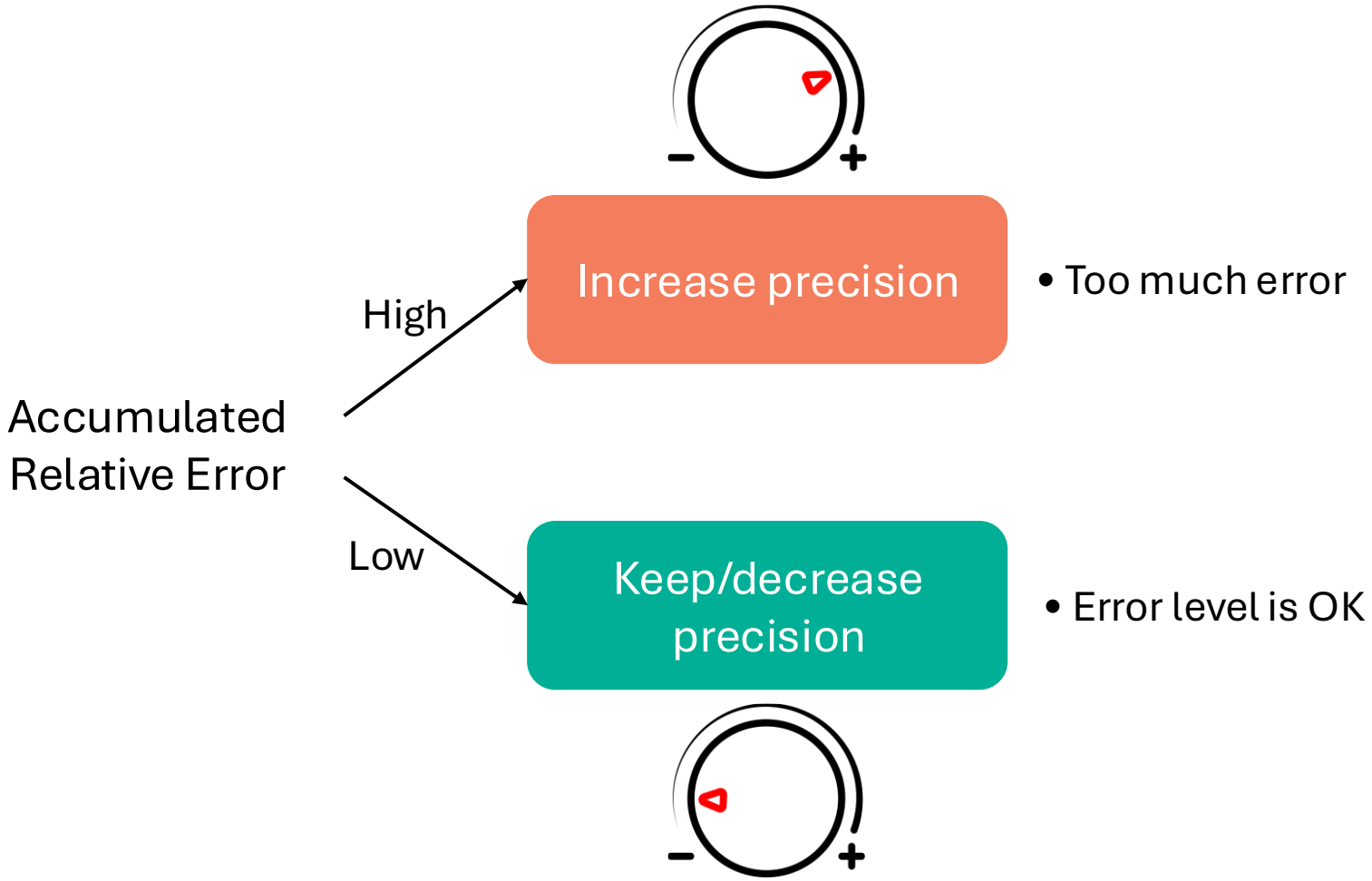
- **Single precision** had about 7 decimal digits of precision
- Suppose numbers **would be rounded**:

a = 1234568.0
b = 1234568.0
c = a − b = 0.0

$$Relative\ error = \frac{|0.0\ - 0.001|}{|0.001|} = 1.0$$

Cancellation is not bounded by ϵ

Key Insights From Relative Error Analysis

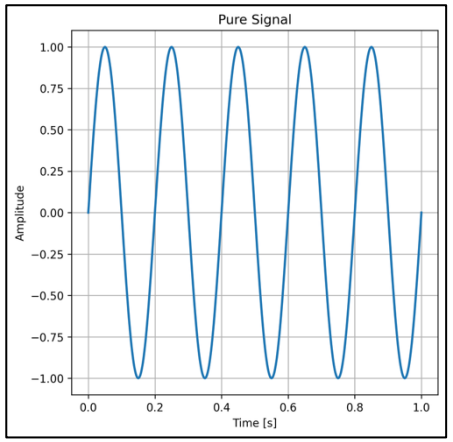


Signal-to-Noise Ratio Analogy: *When to Increase Precision?*

Think of your calculation as a ***radio broadcast***

- **The Signal:** The actual result you are trying to compute
- **The Noise:** The rounding error introduced

FP32 precision

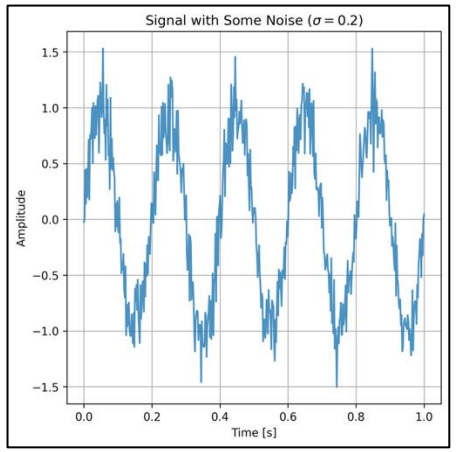


- Correct signal
- Intended math

Case 1

Relative error is small:

- Signal is **much louder** than the noise
- Your result is *"mostly correct"*

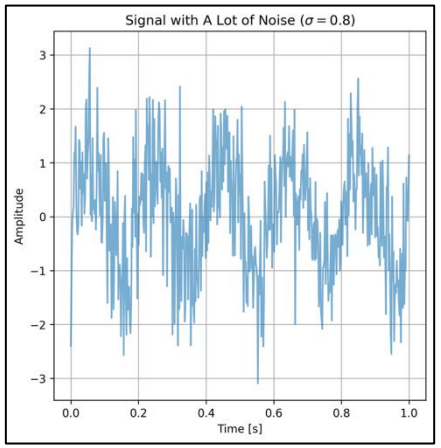


Keep in FP32

Case 2

Relative error is large:

- The "*noise*" has completely **drowned out** "*signal*"
- Digits you are seeing are essentially random
- Values no longer represent the math you intended



Increase to FP64

Roadmap

Motivation & Taxonomy

- Why mixed precision?
- Taxonomy of tools

Rounding Error Basics

- Relative error
- Cancellation

FPChecker Tool

- LLVM instrumentation
- Error tracking
- Using reports for mixed-precision
- AI agents

Examples

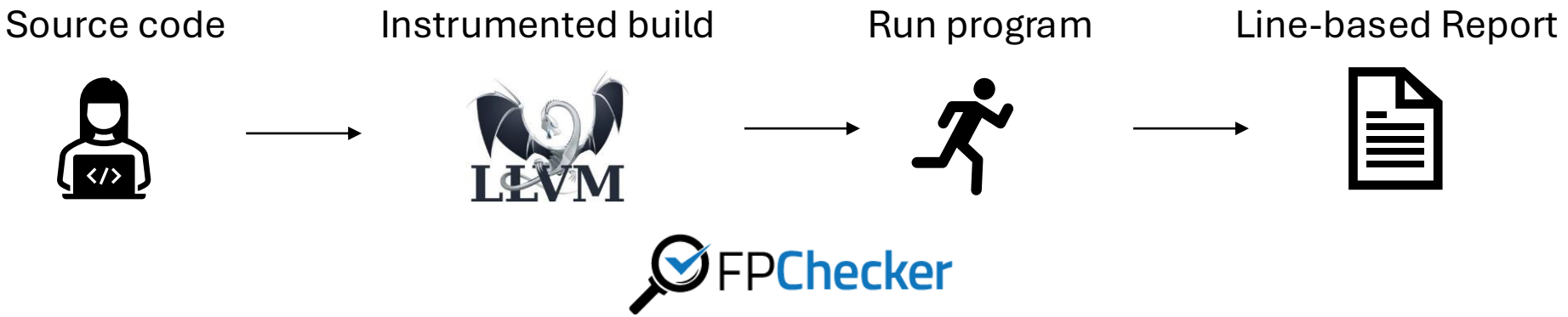
- LULESH (AI agent)
- CG cancellation (is times allows)

Demo

- Error reports
- Mixed-precision example in small program

FPChecker's New Capability

- FPChecker now supports rounding-error tracking for FP32 programs
- Instruments floating-point operations
  - Recomputes a **higher-precision reference** during execution
- Reports ***absolute*** and ***relative*** error back at source-line granularity



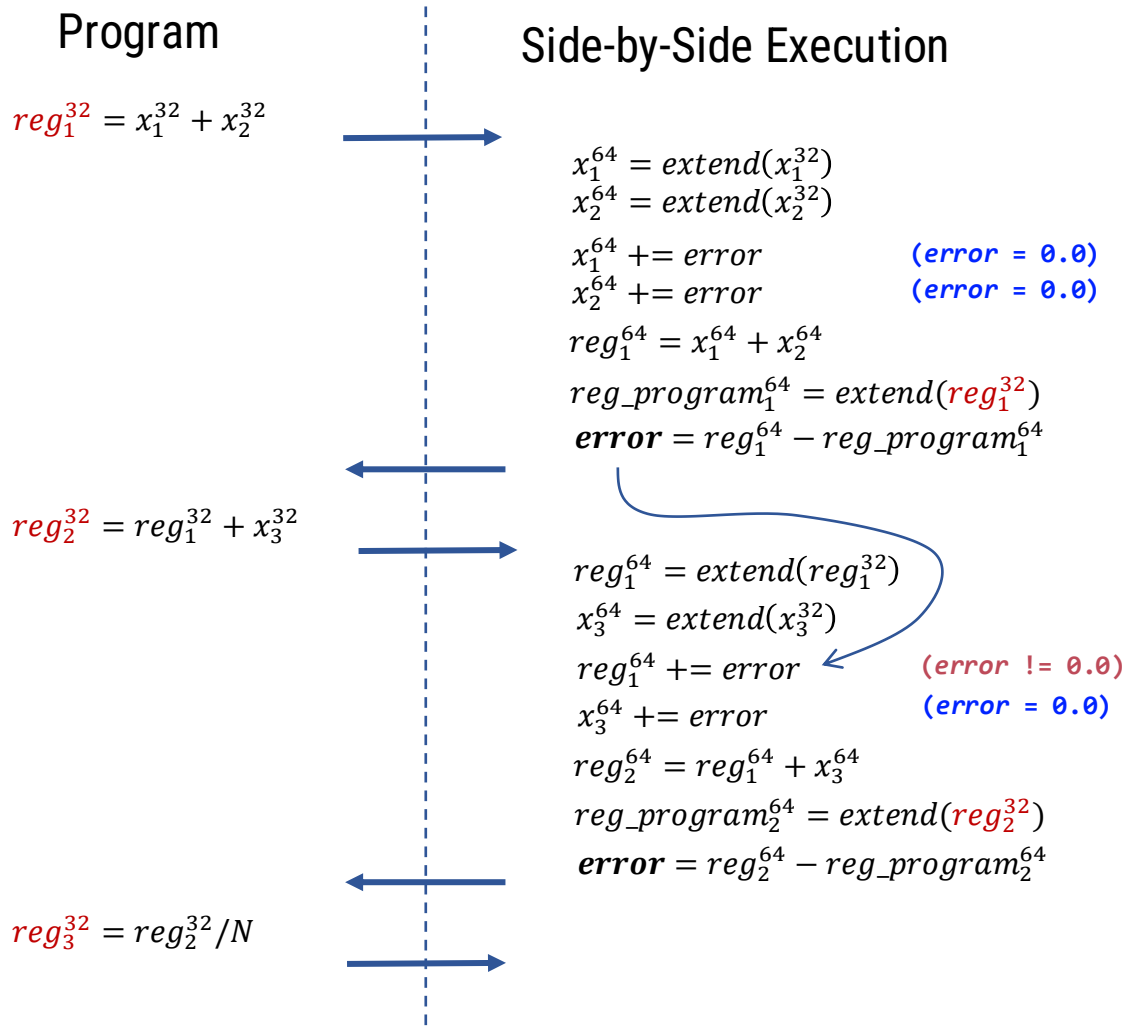
Side-by-Side Execution in Higher Precision

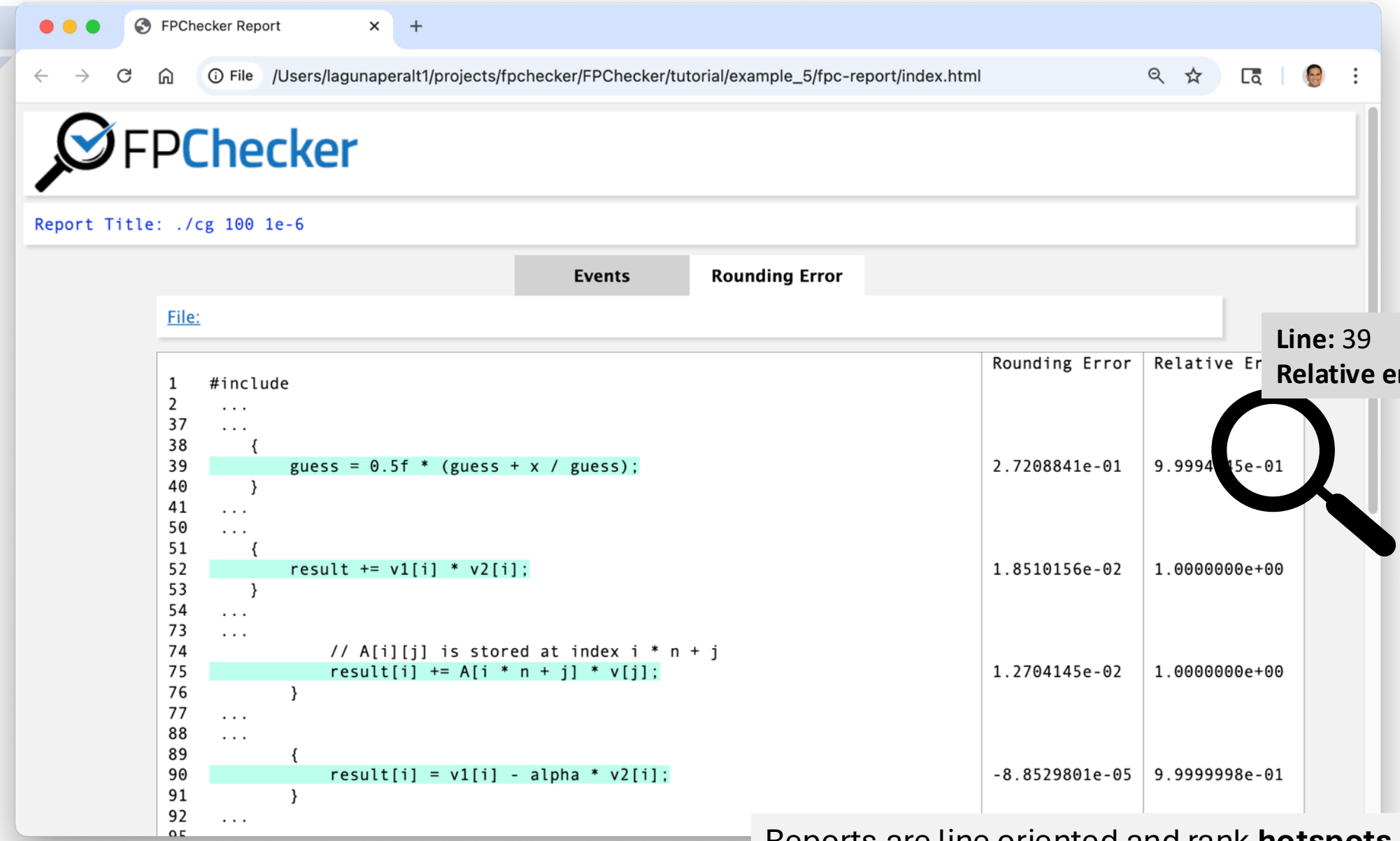
Example (average): $y = \frac{x_1 + x_2 + x_3}{N}$

```
1 void average(float *array, int N, float &average)
2 {
3     float temp = 0.0f;
4     for (int i = 0; i < N; ++i)
5     {
6         temp += array[i];
7     }
8     average = sum_f / N;
9 }
```

```
1 void average(float *array, int N, float &average)
2 {
3     float temp = 0.0f;                          0.000000
4     for (int i = 0; i < N; ++i)
5     {
6         temp += array[i];                       7.12845e-7
7     }
8     average = sum_f / N;                        7.12345e-9
9 }
```

Relative Rounding Error





Line: 39
Relative error: 9.99e-01

Reports are line oriented and rank **hotspots** by error magnitude

Accumulated Rounding Error Per Line:
"Last Seen" Error

```
1   float dot_product(
2     const vector<float> &v1, const vector<float> &v2)
3   {
4     float result = 0.0f;
5     for (size_t i = 0; i < v1.size(); ++i)
6     {
7       result += v1[i] * v2[i];
8     }
9     return result;
10  }
```

Example: v1 = {0.1, 0.2}
v2 = {0.3, 0.4}

Question:
How many arithmetic operations will we observe at runtime for line 7? *(assuming no FMA)*

(a) 2 (b) 4 (c) 6 (d) 7

At runtime:

$$\text{temp}_1 = v_{1,1} \times v_{2,1} \quad \text{Error 1}$$

$$\text{result} = \text{temp}_1 + 0.0 \quad \text{Error 2}$$

$$\text{temp}_2 = v_{1,2} \times v_{2,2} \quad \text{Error 3}$$

$$\text{result} = \text{temp}_2 + \text{result} \quad \underline{\text{Error 4}}$$

"Last error we saw" for line 7

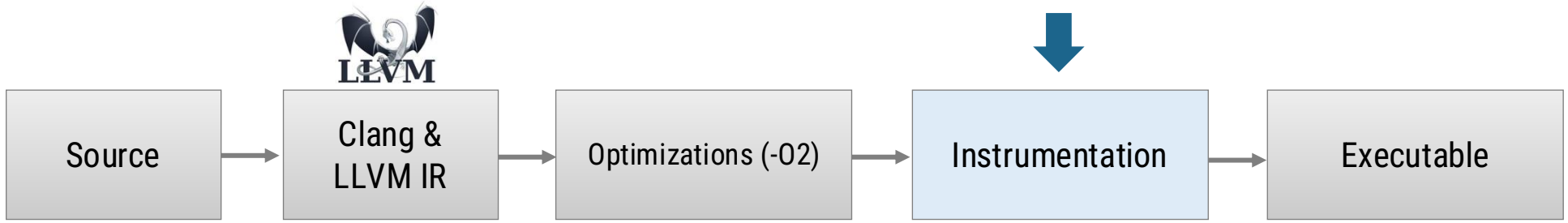
How to Read the Report

Assume we analyze an FP32 execution

Relative error	Approx. trusted digits	Recommendation	Interpretation
$< 10^{-5}$	≥ 5	Keep precision	• Usually safe to keep in FP32 • Unless the application is extremely sensitive
10^{-5} to 10^{-2}	about 2 to 5	Warning / watch it	• Worth monitoring • Candidate for FP64 if this line is on a critical path or the error accumulates repeatedly
$> 10^{-2}$	< 2	Increase precision	• Strong candidate for FP64 promotion • Numerical quality is likely at risk

- Relative error near 1.0 (or 100%) often indicates a **severe hotspot**:
  - Signals **cancellation** or unstable arithmetic
- Absolute error still matters when application scale is important

LLVM Instrumentation Process



1. Make changes to your Makefile or build system (Cmake)
   - Add instrumentation pass to your CFLAGS and/or CXXFLAGS
2. Compile code *(instrument code)*
3. Run executable
4. Create report

LLVM Instrumentation & Runtime

Most important instrumented LLVM IR instructions:

ADD: %r = fadd float %a, %b

SUB: %r = fsub float %a, %b

MUL: %r = fmul float %a, %b

DIV: %r = fdiv float %a, %b

Math call: %r = call float @sqrtf(float %x)

PHI: %x = phi float [%a, %then], [%b, %else]

Branch: br i1 %cond, label %then, label %else

Call: %r = call float @my_func(float %x, float %y)

Invoke: %r = invoke float @may_throw(float %x) to label %normal unwind label %handler

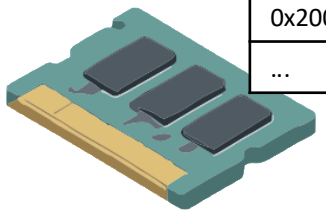
Store: store float %x, ptr %p

Load: %x = load float, ptr %p

At Runtime:

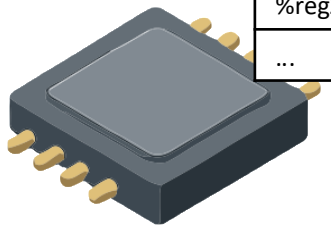
- Keep ***global tables*** for memory addresses and registers
- A register and address is associated with an error

Memory



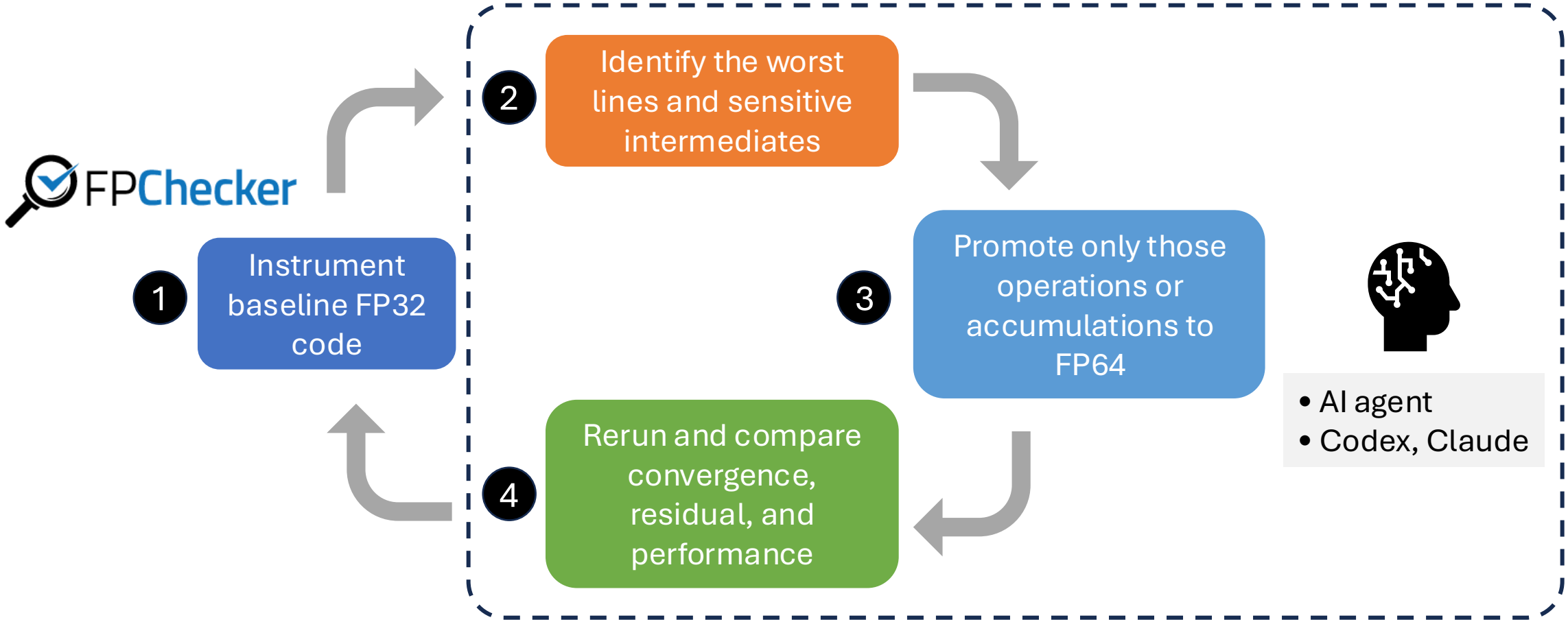
Address	Error	File:Line
0x1000	1.23×10^{-8}	file.cpp:20
0x2000	2.34×10^{-9}	file.cpp:22
…	…	…

CPU



Register	Error	File:Line
%reg1	5.23×10^{-8}	file.cpp:25
%reg2	8.34×10^{-9}	file.cpp:30
…	…	…

Workflow for Users: *Practical Tuning Loop*



Limitations

- Performance aspects not considered
- Currently can’t analyze FP64 or mixed-precision versions
- Tool only shows the error for a given run and input
  - Cannot guarantee error accumulation for ***all inputs***
  - The tool provides **useful evidence**, not an oracle
- Doesn’t replace algorithmic analysis or domain knowledge

Roadmap

Motivation & Taxonomy

- Why mixed precision?
- Taxonomy of tools

Rounding Error Basics

- Relative error
- Cancellation

FPChecker Tool

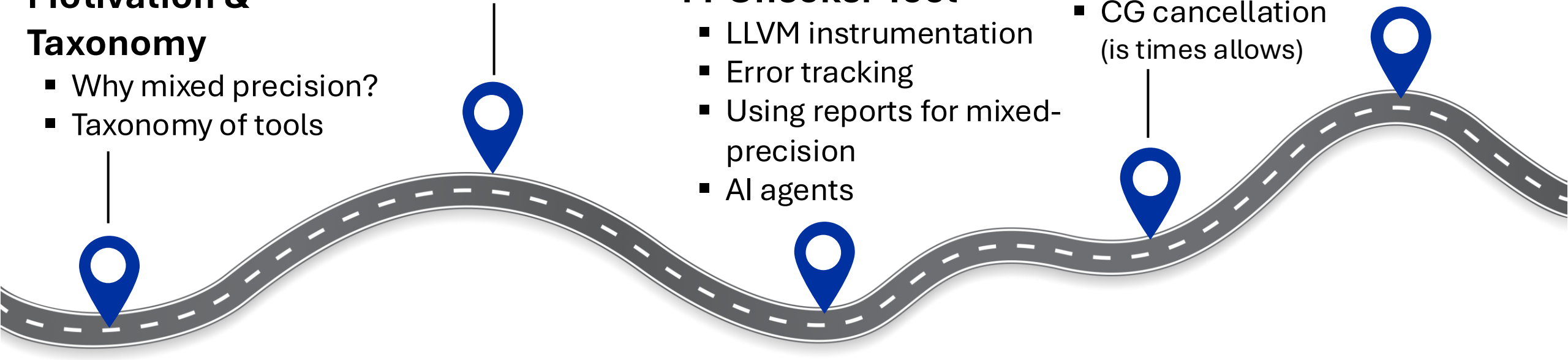
- LLVM instrumentation
- Error tracking
- Using reports for mixed-precision
- AI agents

Examples

- LULESH (AI agent)
- CG cancellation (is times allows)

Demo

- Error reports
- Mixed-precision example in small program



Case Study: LULESH & AI Agent

- GPT 5.4
- Prompt asked:
  - Instrument LULESH with FPChecker and produce a report
  - Use the report to identify **high-error hotspots**
  - Find a **mixed-precision configuration** by promoting to FP64 based on the hotspots
  - Report FOM speedup, and accuracy
- For most questions from GPT, I answered “go ahead” with suggestions
- ~ 1 hour of work

LULESH Mixed Precision Configurations

Mixed version	Code Changes	Error Report Help Rating	Why?
mixed_hydro_volume	• Promotes precision in **hydro update paths** and **volume-related calculations** • This is where instability tends to accumulate and propagate to energy	★★★★★ (5/5)	This version is strongly guided by high-error hotspots and code that the report highlights.
mixed_volume	• Promotes mainly **volume/geometry update computations** • Most hydro logic in lower precision.	★★★★☆ (4/5)	The report is still central (volume lines are clear hotspots), but the scope is narrower than hydro_volume.
mixed_reference	• Promotes reference/baseline-style quantities used for comparisons or normalization, with less broad kernel coverage.	★★☆☆☆ (2/5)	The report gives only hints, but few hotspots map directly to this grouping.

Speedup and Final Energy Error for LULESH

$$Relative\ error = \frac{|Energy_{FP64} - Energy_{Mixed}|}{|Energy_{FP64}|}$$

Input Size	Code version	Relative Error (Final Energy)	FOM Speedup
96	mixed_hydro_volume	3.246888182366020 6e-07	6.2810%
128	**mixed_hydro_volume**	**1.3697812021090248e-07**	**10.9144%**
128	mixed_volume	1.3697812021090248e-07	7.1530%
128	mixed_reference	1.3697812021090248e-07	6.8581%

Roadmap

Motivation & Taxonomy

- Why mixed precision?
- Taxonomy of tools

Rounding Error Basics

- Relative error
- Cancellation

FPChecker Tool

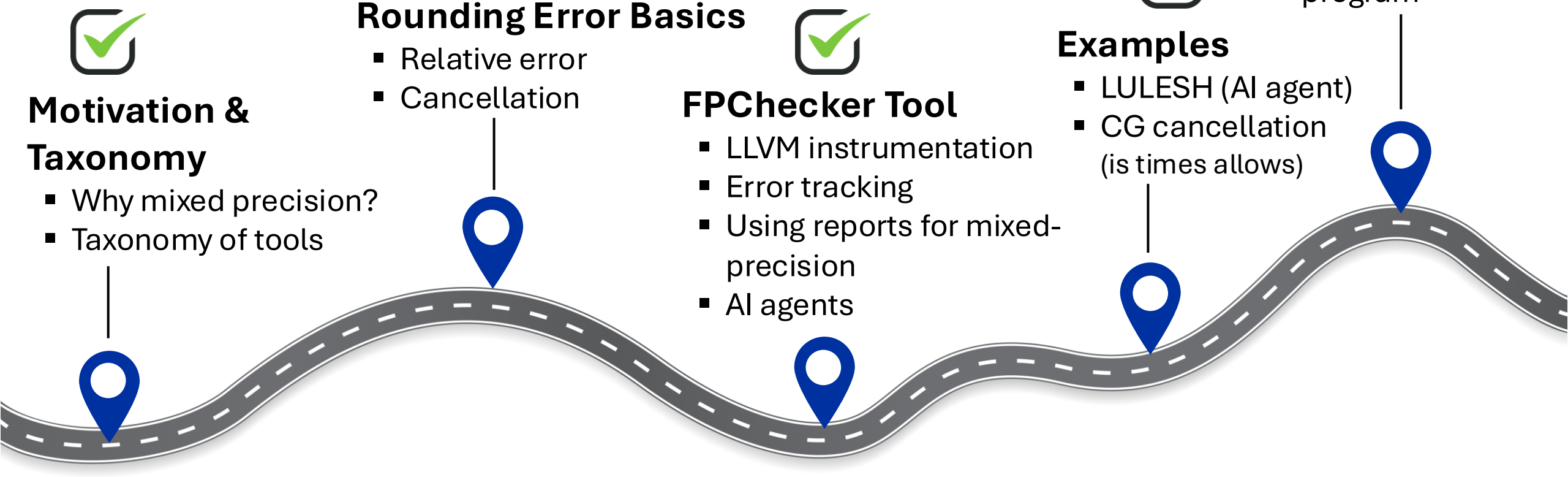
- LLVM instrumentation
- Error tracking
- Using reports for mixed-precision
- AI agents

Examples

- LULESH (AI agent)
- CG cancellation (is times allows)

Demo

- Error reports
- Mixed-precision example in small program



Demo:
Report-Driven Mixed-Precision Workflow

1. Run a **baseline FP32** implementation with FPChecker rounding-error instrumentation
2. Use the report to identify **numerically unstable lines**
3. Promote only the sensitive parts to FP64 (mixed precision)
4. Compare FP32 and mixed-precision against FP64 reference accuracy

Tutorial Slides:
https://fpanalysistools.org/ISC26/

What the Example Includes

Where to find the example:
https://github.com/llnl/FPChecker/tree/master/tutorial/example_6

`fp32.cpp`	Baseline FP32 implementation
`mixed.cpp`	Selective FP64 promotion and algorithmic improvements
`fp64.cpp`	FP64 reference implementation
`compare_accuracy.py`	Compares numerical error for FP32 and mixed vs FP64

Program Computes Two Quantities

Quantity 1

Small root of quadratic equation

$$ar^2 + br + c = 0$$

FP32 baseline (unstable form):

$$\Delta = b^2 - 4ac,$$

$$r_{\text{small}}^{(\text{fp32})} = \frac{-b + \sqrt{\Delta}}{2a}.$$

When $b \gg c$ and $a \approx 1$, we get $\sqrt{\Delta} \approx b$.

Cancellation in $-b + \sqrt{\Delta}$

Quantity 2

Variance of data

$$S_1 = \sum_{i=0}^{N-1} x_i\,, \qquad S_2 = \sum_{i=0}^{N-1} x_i^2\,, \qquad \mu = \frac{S_1}{N}$$

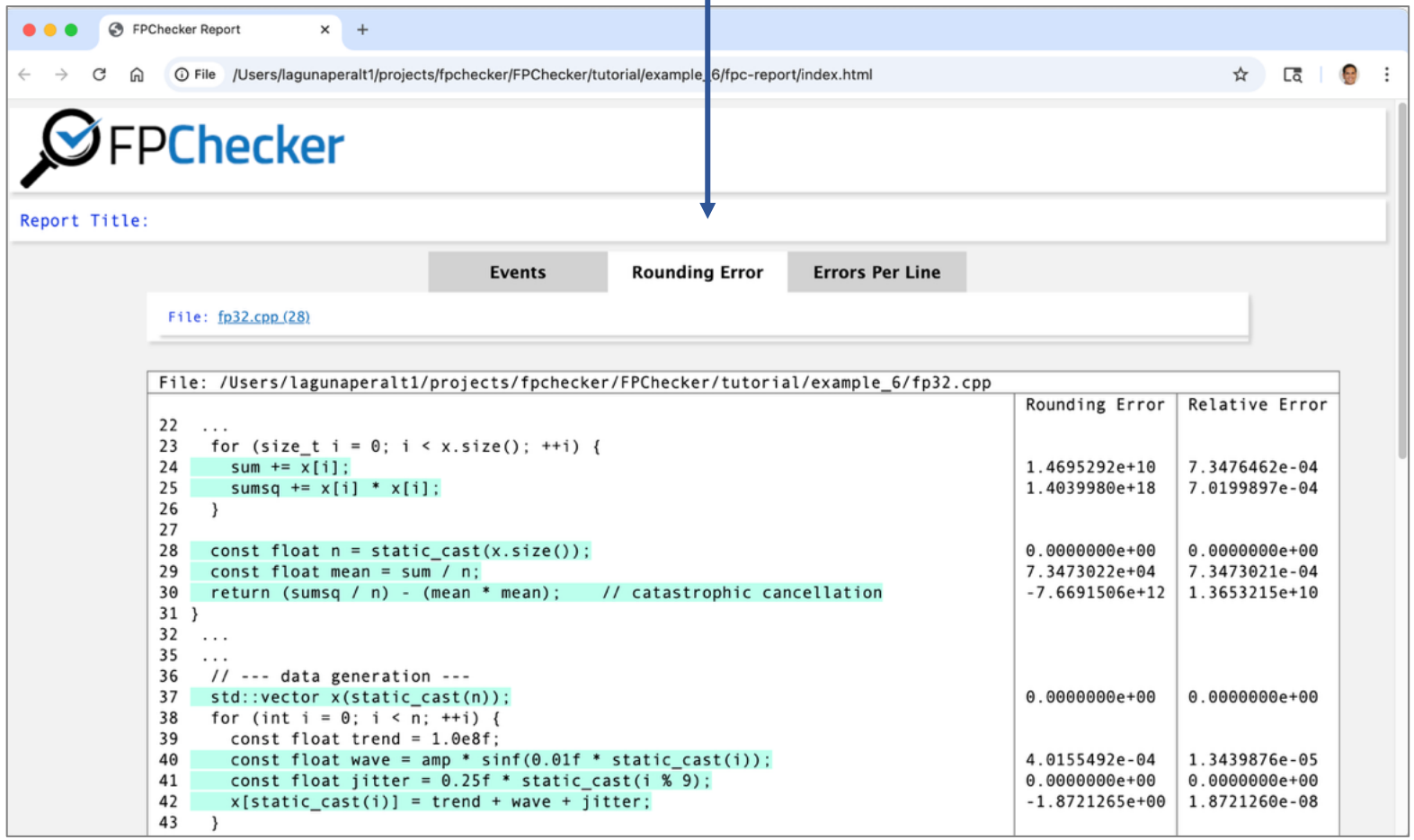
The FP32 baseline returns:

$$\sigma^2_{\text{one-pass}} = \frac{S_2}{N} - \mu^2$$

- For large-magnitude data (e.g., 10^8), both terms are large and close
- Subtracting them causes severe cancellation

Rounding Error Reports

Click on “Rounding Error”



Alternatively, generate report in the terminal:

```
$ fpc-create-report -s rounding
```

Run the Baseline FP32 & Inspect Rounding Errors

```
   # Compile and run problem
$  make clean
$  make fp32
$  ./fp32 200000 32 1.0 1.0e8 1.0

   # Create report
$  fpc-create-report -s rounding
```

← Can take 2-3 minutes in the AWS instance

Run demo and show report

High-error hotspot: 24-30

High-error hotspot: 49-52

```
--- File: /Users/lagunaperalt1/projects/fpchecker/FPChecker/tutorial/example_6/fp32.cpp
Line   | Code                                                                    |         Error |    Rel. Error
-------------------------------------------------------------------------------------------------------------
24     | sum += x[i];                                                            |  1.469529e+10 |  7.347646e-04
25     | sumsq += x[i] * x[i];                                                   |  1.403998e+18 |  7.019990e-04
28     | const float n = static_cast<float>(x.size());                           |  0.000000e+00 |  0.000000e+00
29     | const float mean = sum / n;                                             |  7.347302e+04 |  7.347302e-04
30     | return (sumsq / n) - (mean * mean);    // catastrophi...               | -7.669151e+12 |  1.365321e+10
37     | std::vector<float> x(static_cast<size_t>(n));                           |  0.000000e+00 |  0.000000e+00
40     | const float wave = amp * sinf(0.01f * static_cast<flo...                |  4.015549e-04 |  1.343988e-05
41     | const float jitter = 0.25f * static_cast<float>(i % 9);                 |  0.000000e+00 |  0.000000e+00
42     | x[static_cast<size_t>(i)] = trend + wave + jitter;                      | -1.872127e+00 |  1.872126e-08
46     | const float a = coeff.a;                                                |  0.000000e+00 |  0.000000e+00
47     | const float b = coeff.b;                                                |  0.000000e+00 |  0.000000e+00
48     | const float c = coeff.c;                                                |  0.000000e+00 |  0.000000e+00
49     | const float disc = b * b - 4.0f * a * c;                                | -2.725642e+08 |  2.725642e-08
50     | const float sqrt_d = sqrtf(disc);                                       | -1.490116e-08 |  1.490116e-16
51     | const float numer = -b + sqrt_d;                                  /...   | -1.490116e-08 |  1.000000e+00
52     | const float root_small = numer / (2.0f * a);                            | -7.450581e-09 |  1.000000e+00
55     | const float variance = variance_one_pass_fp32(x);                       |  0.000000e+00 |  0.000000e+00
57     | return {root_small, variance};                                          |  0.000000e+00 |  0.000000e+00
62     | const float  amp = (argc > 2) ? std::strtof(argv[2], ...                |  0.000000e+00 |  0.000000e+00
65     | coeff.a = (argc > 3) ? std::strtof(argv[3], nullptr) ...                |  0.000000e+00 |  0.000000e+00
66     | coeff.b = (argc > 4) ? std::strtof(argv[4], nullptr) ...                |  0.000000e+00 |  0.000000e+00
67     | coeff.c = (argc > 5) ? std::strtof(argv[5], nullptr) ...                |  0.000000e+00 |  0.000000e+00
69     | const Results r = run_problem_fp32(n, amp, coeff);                      |  0.000000e+00 |  0.000000e+00
73     | std::cout << "n=" << n << " amp=" << amp << "\n";                       |  0.000000e+00 |  0.000000e+00
74     | std::cout << "root_small=" << r.root_small << "\n";                     |  0.000000e+00 |  0.000000e+00
75     | std::cout << "variance="   << r.variance   << "\n";                     |  0.000000e+00 |  0.000000e+00
137    | (line not found)                                                        |  0.000000e+00 |  0.000000e+00
1305   | (line not found)                                                        |  0.000000e+00 |  0.000000e+00
```

Mixed-Precision Algorithmic Changes

Quantity 1

Small root of quadratic equation

Mixed and FP64 programs compute the large root first:

$$r_{\text{large}} = \frac{-b - \sqrt{\Delta}}{2a},$$

Then use Vieta's product relation: $r_{\text{small}}\, r_{\text{large}} = c/a$:

$$r_{\text{small}} = \frac{c}{a\, r_{\text{large}}}$$

- Avoids the unstable subtraction: $(-b + \sqrt{\Delta})$
- Numerically much more robust

Quantity 2

Variance of data

First pass:

$$\mu = \frac{1}{N}\sum_{i=0}^{N-1} x_i$$

Second pass:

$$\sigma^2_{\text{two-pass}} = \frac{1}{N}\sum_{i=0}^{N-1} (x_i - \mu)^2$$

- Better conditioned for the centered residuals
- Avoids subtracting two large nearly equal moments

Example Combines Code + Algorithmic Changes

- Precision promotion:
  - Cancellation identification
  - Precision promotion at sensitive operations
- Algorithmic changes:
  - Algorithmic improvement
  - Numerically stable formulas

Compile mixed and FP64 versions

```
   # Compile mixed and FP64 versions
$  make mixed fp64
$  make compare
```

Accuracy Improvements

```
./compare_accuracy.py ./fp32 ./mixed ./fp64 200000 32 1.0 1.0e8 1.0
=== FP32 output ===
mode=fp32
n=200000 amp=32
root_small=0
variance=7.669150646e+12

=== Mixed precision output ===
mode=mixed
n=200000 amp=32
root_small=-1e-08
variance=530.58767247211722

=== FP64 reference output ===
mode=fp64
n=200000 amp=32
root_small=-1e-08
variance=512.5014460013125

=== Relative error (fp32 vs fp64) ===
root_small: 1.000000e+00
variance: 1.496415e+10

=== Relative error (mixed vs fp64) ===
root_small: 0.000000e+00
variance: 3.529010e-02
```

Thanks to Our Collaborators, Contributors, Sponsors

ASCR Project (DANUBE)

Daniel Osei-Kuffuor
LLNL

Ganesh Gopalakrishnan
University of Utah

Sreepathi Pai
University of Rochester

Code Contributors

Shaila Sharmin
Iowa State University

Tanmay Tirpankar
NVIDIA

Funding provided by:

U.S. DEPARTMENT of ENERGY
Office of Science

National Nuclear Security Administration

Summary

- Compiler tools can help in mixed-precision tuning
  - Transform code automatically
  - Identify high-error hotspots, numerical instabilities
  - Find error bounds
- Presented approach to identify ***high-error hotspots*** (FPChecker)
- Can inform users when searching mixed-precision configurations
- Uses LLVM instrumentation
  - Keep track of error accumulation
  - Identifies **sensitivity** of the code to floating-point error

Contact:
ilaguna@llnl.gov