https://fpanalysistools.org/SC26/

SC26 CHICAGO NOV 15–20

Tutorial:

Navigating Mixed-Precision: *Tools and Techniques for Floating-Point Error Profiling*



Ignacio Laguna



Ganesh Gopalakrishnan



Mark S. Baranowski

THE UNIVERSITY of UTAH

LLNL-PRES-2012776

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Previous tutorials

https://fpanalysistools.org





- ISC'26
- SC'25
- SC'24
- SC'19
- PEARC'19

Have any of you attended an earlier tutorial?

Slido poll

Learning Objectives

Tools for Floating-Point Error Profiling → with a focus on Mixed-Precision Tunning



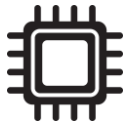
Identify numerical instabilities

- Isolate floating-point exceptions (**NaN** and **Infinity**) and **cancellation**



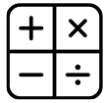
Characterize dynamic range and rounding error

- Generate ***dynamic range profiles & profile error accumulation***



Automate analysis via compiler and binary instrumentation: FPChecker & NixNan

- Lear to compile and run code with the tools



Implement data-driven mixed-precision strategies

- Use numerical error reports to assign lower or higher precision: **FP32** or **FP64**

Accessing Slides

https://fpanalysistools.org/SC26/



Access to AWS instances

1 Add your name to a username here: https://fpchecker.org/usernames

2
- Connect to the AWS instance via ssh
- Ask instructor for the password

```
$ ssh -L 8000:localhost:8000 user[N]@IP_ADDRESS
```

IP Addresses:

X.X.X.X

X.X.X.X



Agenda

Module	Duration (min)	Time (PM)
☑ Welcome	8	1:30 – 1:38
Motivation & Overview	10	1:38 – 1:48
Floating-Point Basics	15	1:48 – 2:03
FPChecker's Workflow & Installation	12	2:03 – 2:15
Guided Practice 1: Exceptions (linear solver)	15	2:15 – 2:30
Guided Practice 2: Dynamic Range Analysis	15	2:30 – 2:45
Guided Practice 3: Mixed-Precision in CG	15	2:45 – 3:00
Break	**30**	
Demo 1: Agentic mixed-precision with FPChecker	15	3:30 – 3:45
Demo 2: Instrumenting a production library (Hypre)	10	3:45 – 3:55
Q&A - FPChecker	5	3:55 – 4:00
NixNaN	60	4:00 – 5:00

Module 1

Motivation & Overview

- Problems the tool solves
- Need for lower & mixed precision
- FPChcker's overview

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

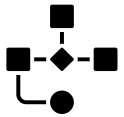
Sources:
https://resources.nvidia.com/en-us-hopper-architecture/nvidia-tensor-core-gpu-datasheet?ncid=no-ncid
https://www.spheron.network/blog/nvidia-b200-complete-guide/
https://www.nvidia.com/en-us/data-center/vera-rubin-nvl72/

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

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

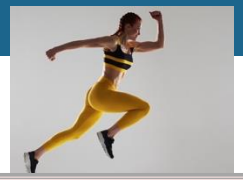
Taxonomy of Compiler Tools



Automatic Precision Tunning

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

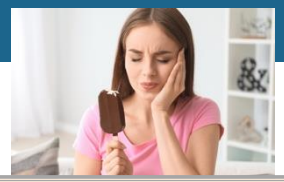
Precimonious, HiFPTuner, RLTuner



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, NixNaN, Herbgrind, 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



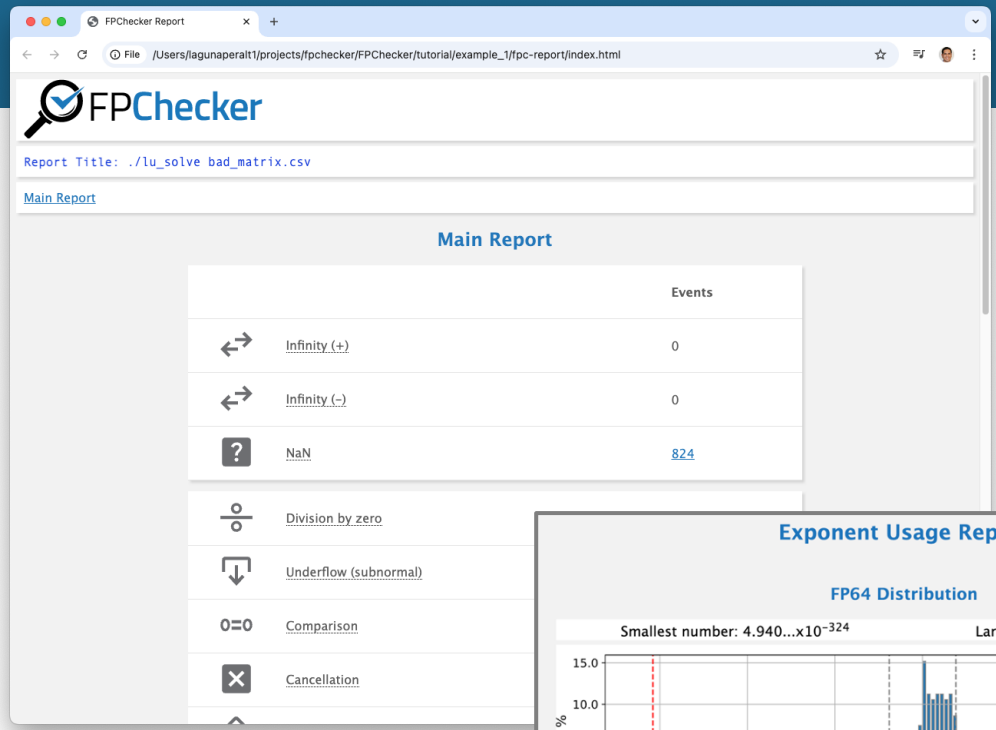
Testing Methods

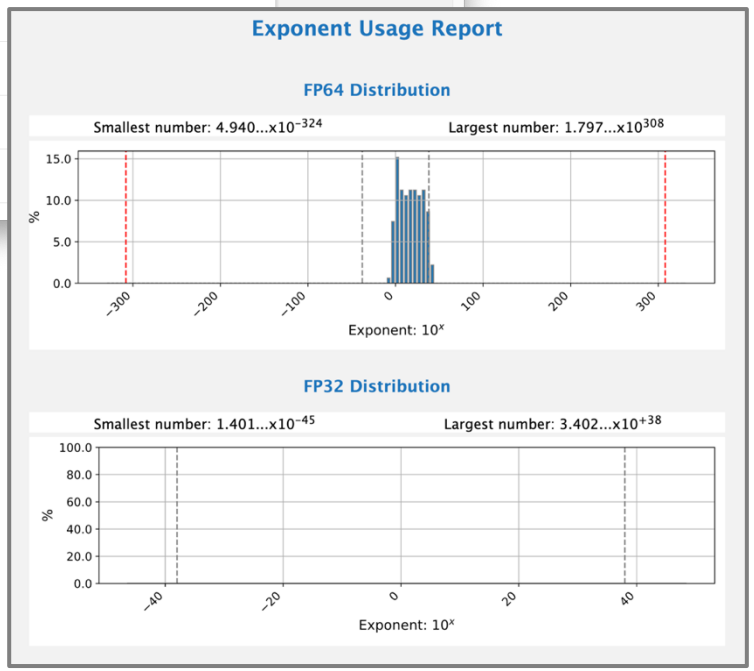
- Test code in different precisions
- Run performance and numerical tests
- Verify code can run at a certain precision level
- Most used in practice today

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

FPChecker Overview

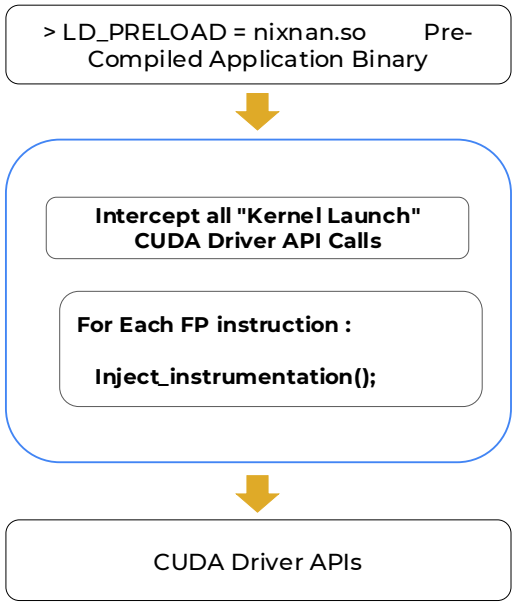
- Detects floating-point exceptions
  - NaN, Infinity
- Shows impacted *lines of code*
- Shows other "code smells"
  - Cancellations, underflows
- Analyzes dynamic range
  - Is FP32 or FP64 enough?
- Computes accumulated rounding error
  - Helpful for *mixed-precision*
- Works with compiler instrumentation
  - Relies on Clang/LLVM
- Documentation: https://fpchecker.org/





NixNan

- Reports Floating-Point Exceptions on GPUs
  - Observes exceptions at the binary level
  - Considers compiler optimizations
  - Many GPU codes are shipped only as binaries
- NixNan
  - Tracks the flow of exceptions in NVIDIA GPUs
    - Process does not rely on hardware traps – not supported on GPUs
  - Slower than source-instrumented code executions




```
#nixnan: ------------ nixnan Report

#nixnan: --- FP16 Operations ---
#nixnan: NaN: 4
#nixnan: Infinity: 3
#nixnan: Subnormal: 2
#nixnan: Division by 0: 0

#nixnan: --- FP32 Operations ---
#nixnan: NaN: 0
#nixnan: Infinity: 0
#nixnan: Subnormal: 0
#nixnan: Division by 0: 0

#nixnan: --- FP64 Operations ---
#nixnan: NaN: 0
#nixnan: Infinity: 0
#nixnan: Subnormal: 0
#nixnan: Division by 0: 0
```

Agenda

	Module	Duration (min)	Time (PM)
☑	Welcome	8	1:30 – 1:38
☑	Motivation & Overview	10	1:38 – 1:48
	Floating-Point Basics	15	1:48 – 2:03
	FPChecker's Workflow & Installation	12	2:03 – 2:15
	Guided Practice 1: Exceptions (linear solver)	15	2:15 – 2:30
	Guided Practice 2: Dynamic Range Analysis	15	2:30 – 2:45
	Guided Practice 3: Mixed-Precision in CG	15	2:45 – 3:00
	Break	**30**	
	Demo 1: Agentic mixed-precision with FPChecker	15	3:30 – 3:45
	Demo 2: Instrumenting a production library (Hypre)	10	3:45 – 3:55
	Q&A - FPChecker	5	3:55 – 4:00
	NixNaN	60	4:00 – 5:00

Module 2

Floating-Point Basics

- Floating point representations
- Rounding error & relative rounding error
- How is relative error bounded?
- Cancellation
- Machine epsilon
- Exceptions

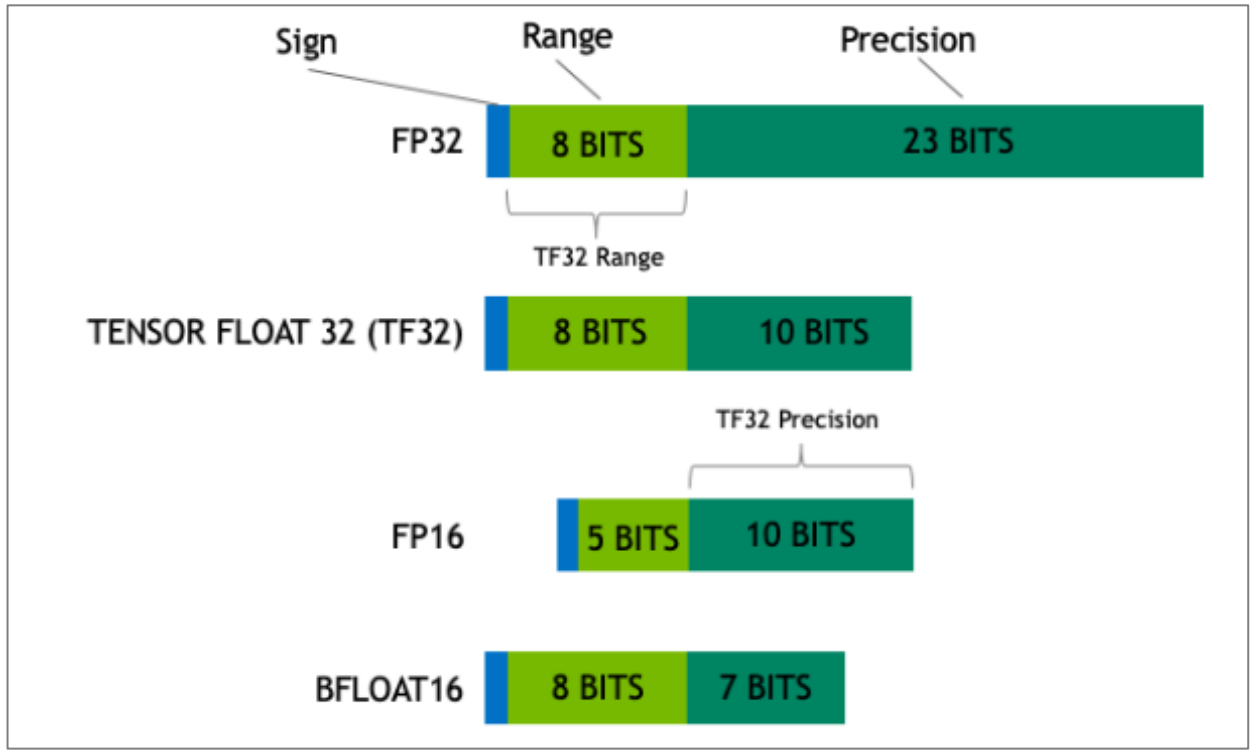
Floating-Point Representations

Standard formula for floating-point representations

$$\text{Value} = (-1)^{\text{sign}} \times \text{Mantissa} \times 2^{\text{Exponent}}$$

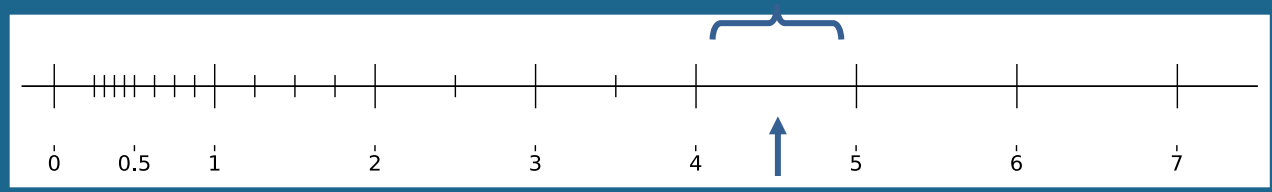
Exponent: determines the ***range***
Mantissa: determines the ***precision***

Bits for different formats



From https://developer.nvidia.com/

Precision Loss



For larger numbers, the steps are larger

Python

```
>>>
>>> (10e8 + 1e-8) - 10e8
0.0
>>>
```

- Mathematically the answer is 10^{-8}

- The computer must add:

- 10^{8} (a massive number)
- 10^{-8} (a tiny fraction)

- In FP32, we only have about **7 decimal digits** of precision to work with.
- When you try to add these two, you are essentially asking the computer to store:

100,000,000.00000001

- **There are not enough bits to store it**, so it must be rounded to 10^{8}

- The subtraction becomes:

$$(10^{8}) - 10^{8} = 0$$

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
1.2345678**9**	1.234568	+0.00000011	8.91×10^{-8}
12345678**9**.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: $\epsilon \approx 10^{-7}$
  - For FP64: $\epsilon \approx 10^{-16}$

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:

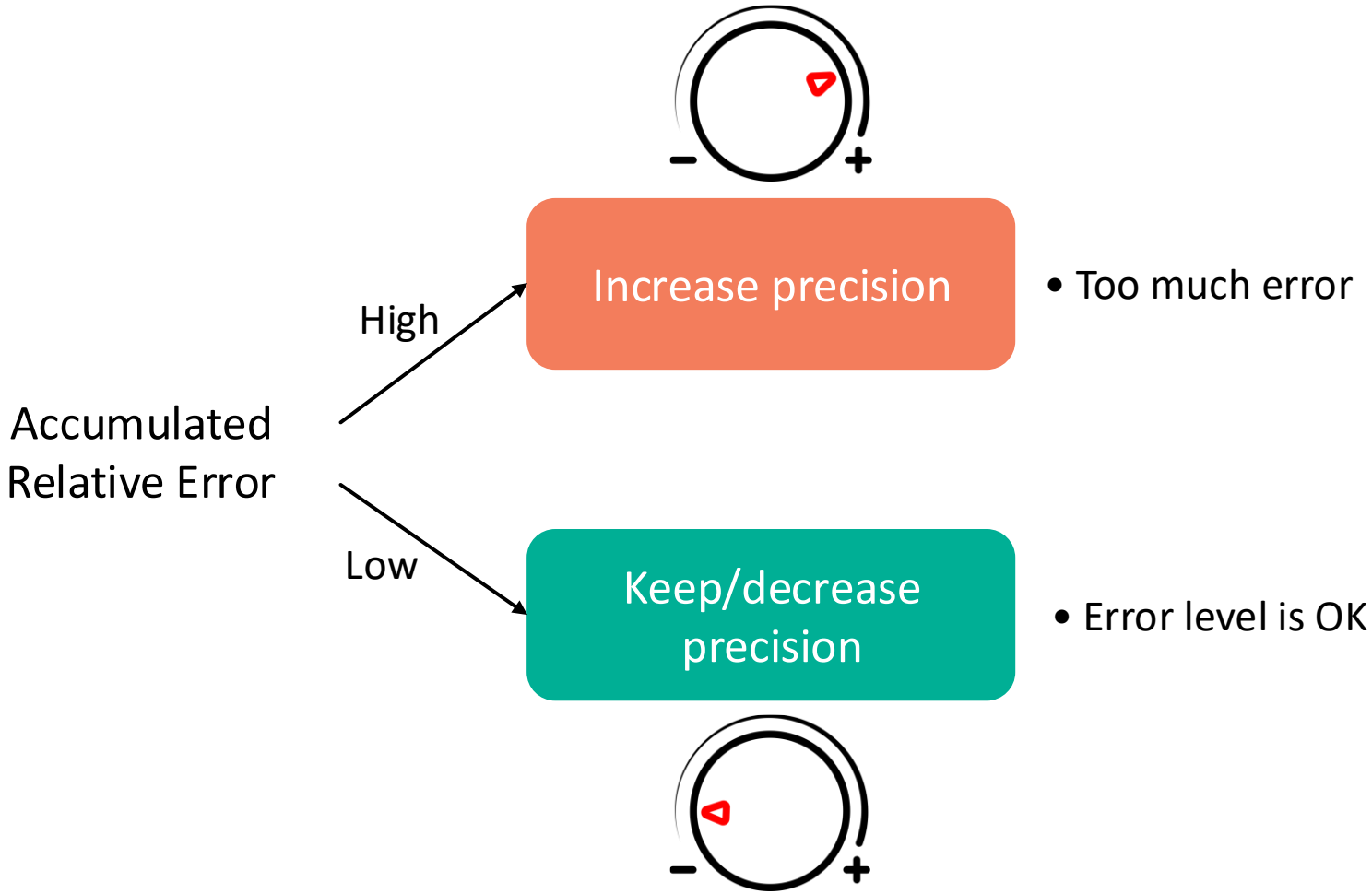
```
x = 1.0000000000000001
y = 1.0000000000000002
```

Mathematically: `y-x = 0.0000000000000001`

In floating-point: `y-x = 2.220446049250313e-16`

Large relative error!

Key Insights From Relative Error Analysis

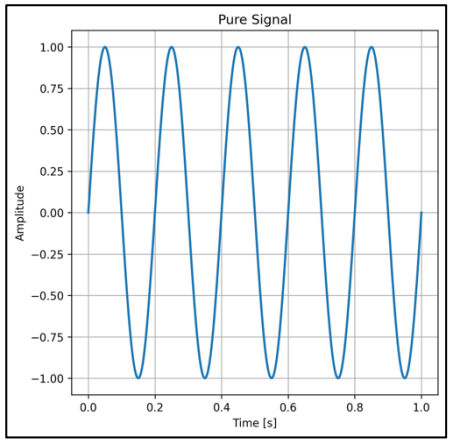


Signal-to-Noise Ratio Analogy: *When to Increase Precision?*

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

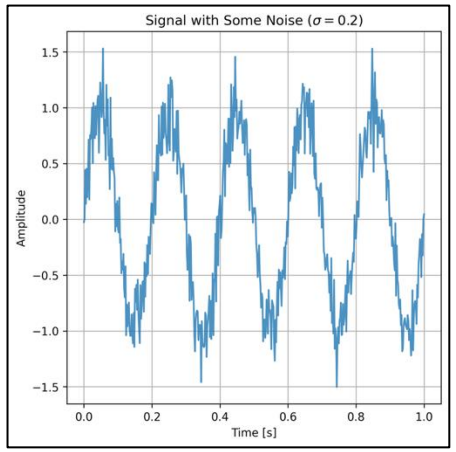


- Correct signal
- Intended math

Case 1

Relative error is small:

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

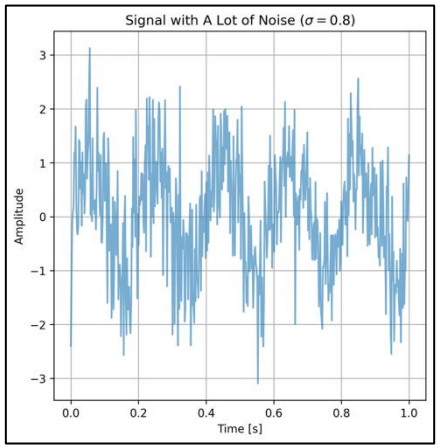


Keep in FP32

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

Floating-Point Exceptions

- Exceptions are the ***smoke before the fire***
- Indicate a computation is about to **"blow up"**

Type	Explanation
Division by Zero	• Result in **INF** (infinity)
Overflow	• The result is too large to be represented • Results in **INF**
Invalid Operation	• Math is undefined • Examples: `sqrt(-1), 0/0` • Result in **NaN**
Underflow	• Result is approaching zero • Smaller than smallest normal number
Inexact	• Result must be rounded • We don't worry about this one in HPC

- Most important
- Result in **INF** or **NAN**

When to Worry about Exceptions?

- Exceptions propagate as a **virus**
  - Once a **NaN** or **INF** enters a calculation, almost every subsequent operation will also result in **NaN** or **INF**
- By default, CPUs don't stop the program when these happen
  - FPChecker will detect them for you
- Dynamic range is reduced in lower precision formats: FP32, FP16, …
  - Likely to get more **overflows**

Module's Review

- Relative rounding error is bounded by machine epsilon
  - For FP32: $\epsilon \approx 10^{-7}$
  - For FP64: $\epsilon \approx 10^{-16}$
- **Increase** precision if relative error is **large**
- Cancellation relative error is **not bounded** by machine epsilon
  Cancellation error could be 100% or more
- Exceptions can introduce **INF** and **NaN**
  Exceptions matter when reducing precision

Questions

Agenda

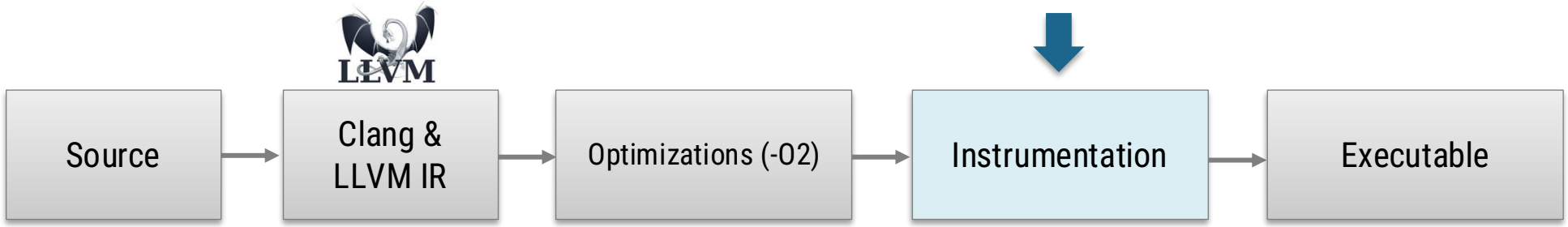
	Module	Duration (min)	Time (PM)
☑	Welcome	8	1:30 – 1:38
☑	Motivation & Overview	10	1:38 – 1:48
☑	Floating-Point Basics	15	1:48 – 2:03
	FPChecker's Workflow & Installation	12	2:03 – 2:15
	Guided Practice 1: Exceptions (linear solver)	15	2:15 – 2:30
	Guided Practice 2: Dynamic Range Analysis	15	2:30 – 2:45
	Guided Practice 3: Mixed-Precision in CG	15	2:45 – 3:00
	Break	**30**	
	Demo 1: Agentic mixed-precision with FPChecker	15	3:30 – 3:45
	Demo 2: Instrumenting a production library (Hypre)	10	3:45 – 3:55
	Q&A - FPChecker	5	3:55 – 4:00
	NixNaN	60	4:00 – 5:00

Module 3

FPChecker's Workflow & Installation

- Tool's workflow
- Compiler instrumentation concepts
- Keeping track of exceptions and rounding error
- Error reports (per line); what does it mean?

LLVM Instrumentation Process



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

Two Ways to Instrument Applications

Method 1

Pass extra flags to the compiler

```
$ clang++ -c file.cpp -O2 \
-g \
-include /path/.../Runtime_cpu.h \
-fpass-plugin=/path/.../libfpchecker_cpu.dylib
```

To find the appropriate flags:

```
$ fpchecker-show
```

Method 2

Use the FPChecker compiler wrappers:

- `clang-fpchecker`
- `clang++-fpchecker`
- `mpicc-fpchecker`
- `mpicxx-fpchecker`

```
$ FPC_INSTRUMENT=1 clang++-fpchecker -c file.cpp -O2
```

Env variables must be enabled to instrument:

- FPC_INSTRUMENT, **or**
- FPC_INSTRUMENT_ERR_TRACKING

Two Classes of Env Variables: *Compile-time & Run-time*

✓ Covered in the Tutorial

Compile-time Variables

Variable	Type	Description
✓ FPC_INSTRUMENT	Compile-time	Instruments the application for exceptions/event checking
✓ FPC_INSTRUMENT_ERR_TRACKING	Compile-time	Instruments the application for rounding error tracking
FPC_ANNOTATED	Compile-time	Indicates that the program is annotated

Run-time Variables

Variable	Type	Description
✓ FPC_EXPONENT_USAGE	Run-time	Profiles exponent usage for FP32/FP64
FPC_TRAP_INFINITY_POS	Run-time	Program exits when Infinity positive is found
FPC_TRAP_INFINITY_NEG	Run-time	Program exits when Infinity negative is found
FPC_TRAP_NAN	Run-time	Program exits when NaN is found
FPC_TRAP_DIVISION_ZERO	Run-time	Program exits when division-by-zero is found
FPC_TRAP_CANCELLATION	Run-time	Program exits when cancellation is found
FPC_TRAP_COMPARISON	Run-time	Program exits when Comparison is found
FPC_TRAP_UNDERFLOW	Run-time	Program exits when underflow is found
FPC_TRAP_LATENT_INF_POS	Run-time	Program exits when Latent Infinity positive is found
FPC_TRAP_LATENT_INF_NEG	Run-time	Program exits when Latent Infinity negative is found
FPC_TRAP_LATENT_UNDERFLOW	Run-time	Program exits when Latent Underflow is found

Dynamic range analysis
from FP64 → FP32 → FP16 …

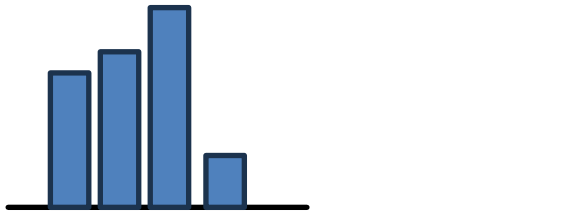
- Identifying **underflow** and **overflow** risks:
  - See if values fit within the format
    - If values ranges between 10^{-20} and 10^{20}, safely use FP32
    - FP32 covers roughly: 1.18×10^{-38} and 1.18×10^{38}

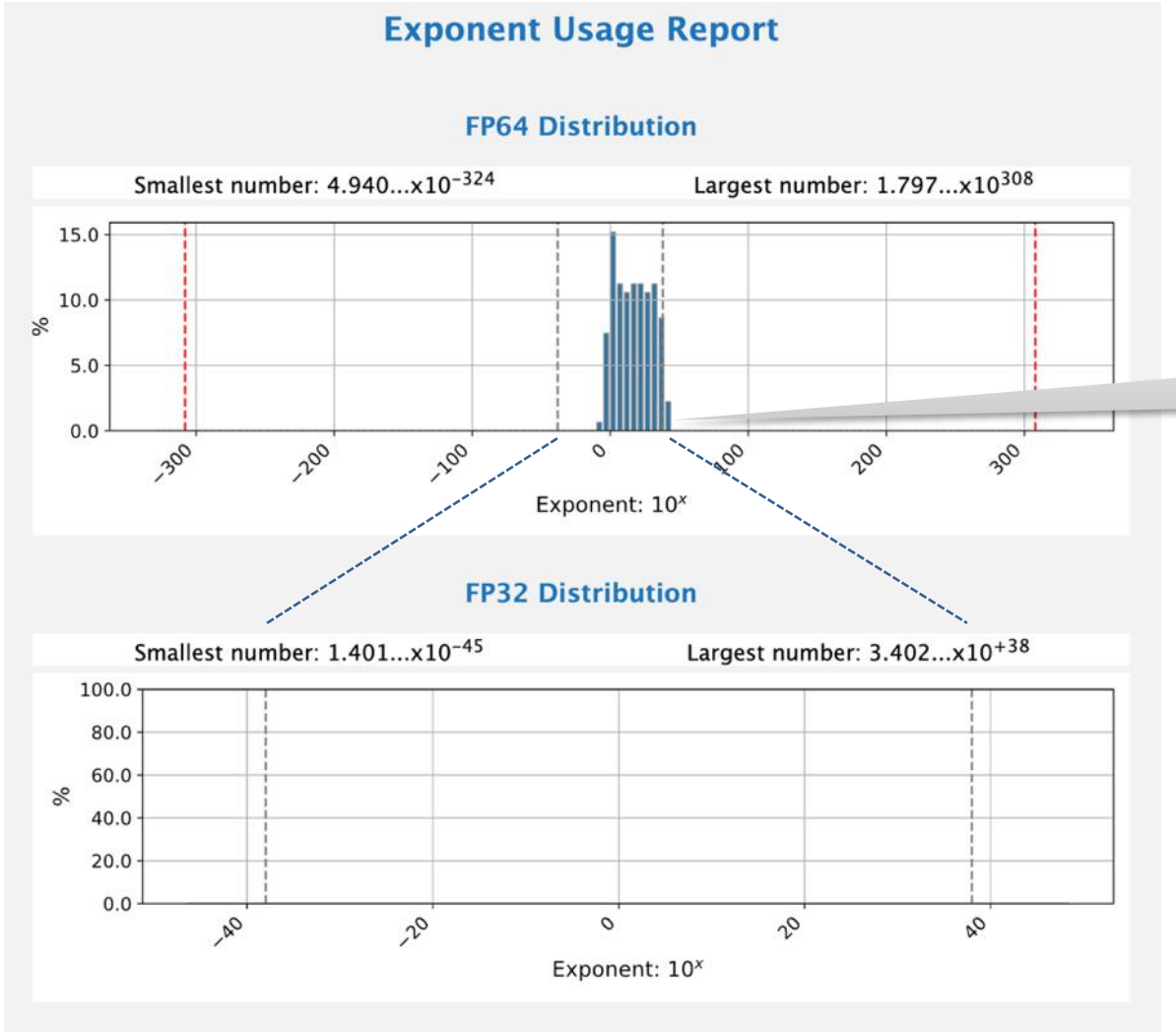
- **Targeted** precision reduction
  - Not every part of your code needs the same level of precision

Exponent Usage Analysis:
Dynamic Range of Computed Numbers

1. Observes computed floating-point numbers
2. FPChecker builds a **histogram** of exponent values

$$variable = 1.23 \times 10^{200}$$





Out of range for FP32

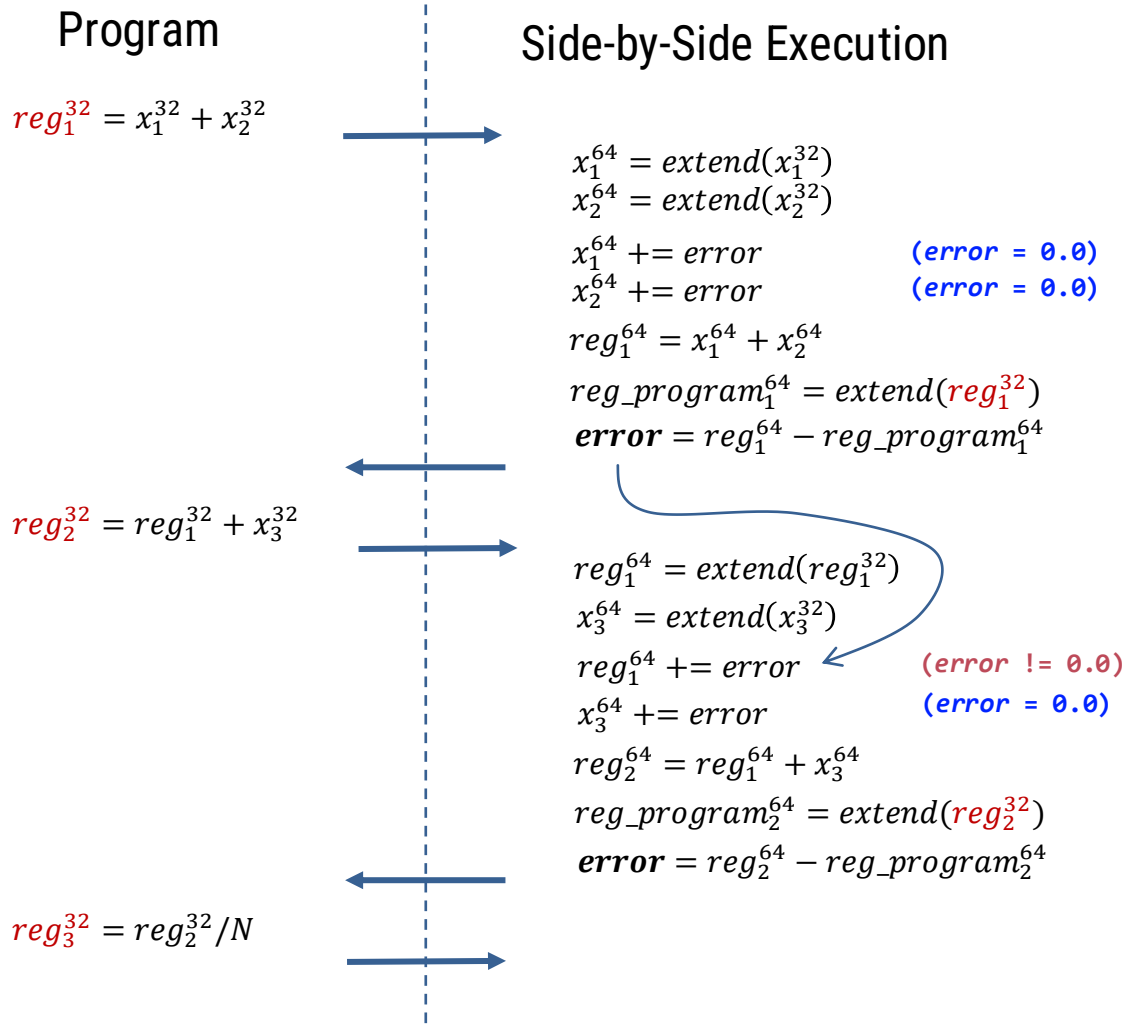
Side-by-Side Execution in Higher Precision

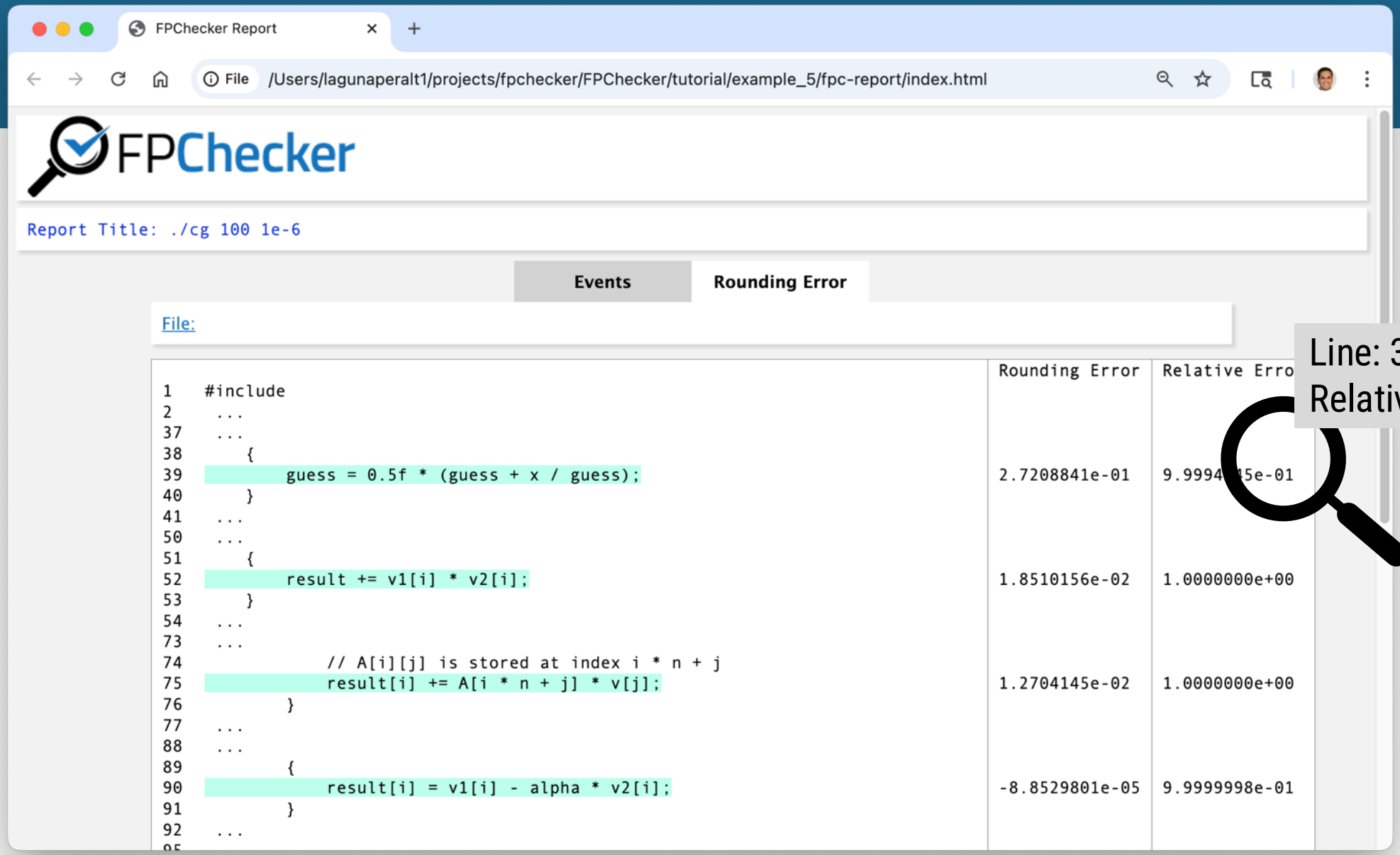
Example (average): $y = \frac{x_1 + x_2 + x_3}{N}$

```
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 }
```

Relative Rounding Error

```
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 }
```




FPChecker Report
File /Users/lagunaperalt1/projects/fpchecker/FPChecker/tutorial/example_5/fpc-report/index.html
FPChecker
Report Title: ./cg 100 1e-6
Events
Rounding Error
File:
Rounding Error
Relative Erro
1 #include
2 ...
37 ...
38 {
39 guess = 0.5f * (guess + x / guess);
40 }
41 ...
50 ...
51 {
52 result += v1[i] * v2[i];
53 }
54 ...
73 ...
74 // A[i][j] is stored at index i * n + j
75 result[i] += A[i * n + j] * v[j];
76 }
77 ...
88 ...
89 {
90 result[i] = v1[i] - alpha * v2[i];
91 }
92 ...
2.7208841e-01
9.9994 5e-01
1.8510156e-02
1.0000000e+00
1.2704145e-02
1.0000000e+00
-8.8529801e-05
9.9999998e-01
Line: 39
Relative error: 9.99e-01

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 in line 7? *(assume no FMA)*

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

At runtime:

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

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

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

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

"Last seen" error for line 7

Auto-Vectorizaiton is disabled

- We disable auto-vectorize
- Added flags: -fno-vectorize -fno-slp-vectorize
- This keeps source-line attribution accurate

Software Requirements

- Linux, Mac OS
- LLVM/Clang 19.1.7
- Cmake
- Recent Python (e.g., python 3.12 or higher)
- Matplotlib
- Optional for parallel code:
  - MPI
  - OpenMP

See installation process using Conda at the end of the presentation

Access to AWS instances

1 Add your name to a username here: https://fpchecker.org/usernames

2
- Connect to the AWS instance via ssh
- Ask instructor for the password

```
$ ssh -L 8000:localhost:8000 user[N]@IP_ADDRESS
```

IP Addresses:

X.X.X.X

X.X.X.X



Get FPChecker Source

This tutorial is based on release v0.6



AWS instance

```
$ cd /home/user1

  # Clone FPChecker
$ git clone https://github.com/LLNL/FPChecker.git
$ cd FPChecker
```

FPChecker is installed in /opt/fpchecker for a all users

fpchecker-show

```
$  fpchecker-show
   ========================================
            FPChecker Configuration
   ========================================

   Installation path: /opt/fpchecker

   Add the following to CFLAGS and/or CXXFLAGS:

   (1) For exceptions checking:
   -g -include /opt/fpchecker/src/Runtime_cpu.h -fpass-plugin=/opt/fpchecker/lib/libfpchecker_cpu.so

   (2) For rounding error tracking:
   -g -fno-vectorize -fno-slp-vectorize -include /opt/fpchecker/src/Runtime_error.h -fpass-
   plugin=/opt/fpchecker/lib/libfpchecker_error.so

   Wrappers are located here:
   /opt/fpchecker/bin/clang-fpchecker
   /opt/fpchecker/bin/clang++-fpchecker
   /opt/fpchecker/bin/mpicc-fpchecker
   /opt/fpchecker/bin/mpicxx-fpchecker
```

Agenda

	Module	Duration (min)	Time (PM)
☑	Welcome	8	1:30 – 1:38
☑	Motivation & Overview	10	1:38 – 1:48
☑	Floating-Point Basics	15	1:48 – 2:03
☑	FPChecker's Workflow & Installation	12	2:03 – 2:15
	Guided Practice 1: Exceptions (linear solver)	15	2:15 – 2:30
	Guided Practice 2: Dynamic Range Analysis	15	2:30 – 2:45
	Guided Practice 3: Mixed-Precision in CG	15	2:45 – 3:00
	Break	**30**	
	Demo 1: Agentic mixed-precision with FPChecker	15	3:30 – 3:45
	Demo 2: Instrumenting a production library (Hypre)	10	3:45 – 3:55
	Q&A - FPChecker	5	3:55 – 4:00
	NixNaN	60	4:00 – 5:00

Module 4

Guided Practice 1: Exceptions

Location: tutorial/example_1

- The first win: easy to follow example
- Exceptions in simple linear solver (NaN, Infinity)
- Creating traces (JSON)
- Generating reports
- Show first report of errors

First, build the *common* library
It will be used in other examples

- Common library:
  - BLAS operations
  - Linear solvers (LU, CG)
  - Matrix & vector printing
  - Other functionalities



In AWS instance:

```
   # Compile library
$  cd /home/user[NUMBER]/FPChecker/tutorial/common/
$  FPC_INSTRUMENT=1 make
```

Example 1:
NaN & Infinity exception in linear solver

- Location:

  `tutorial/example_1/lu_solve.cpp`

- Program description
  - Linear solver $Ax = b$
  - Solves $Ax = 1$
  - LU decomposition: $PA = LU$
  - Partial pivoting
  - Solve by forward/backward substitution

FPChecker's Use Case

- Use an ill-condition problem (matrix)
- Produces NaN and Infinity
  - U factor ends up with zero diagonal
  - Division by zero
- FPChecker locates the exceptions

JSON Trace Files

After an instrumented program runs, it should generate a trace file in .fpc_logs

```
$ ./application input1 input1
  ...
  ...

$ ls .fpc_logs/
  fpc_king01_27794.json
```

JSON file format:

`fpc_[hostname]_[PID].json`

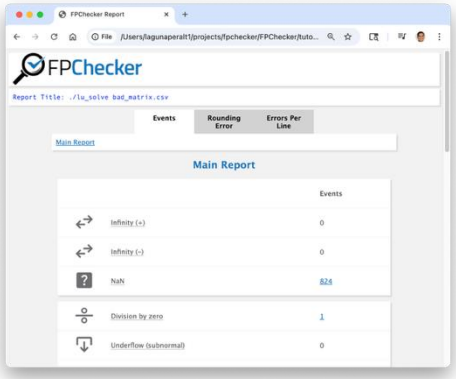
Generating Reports

- To generate an HTML report, run **`fpc-create-report`**
- The parameter –t is optional to define a title

```
$ fpc-create-report -t "report title"
  Generating FPChecker report...
  ...
  Report created: ./fpc-report/index.html
```

- Reports are created in **`./fpc-report`**
- In Mac, the report can be open easily:

```
$ open ./fpc-report/index.html
```



- The –s option shows a report in the shell
- It does not create an HTML report

```
$ fpc-create-report -s
  Generating FPChecker report...

  ========= Main Report ===========
  positive_infinity                0
  negative_infinity                0
  nan                              824
  division_by_zero                 1
  cancellation                     0
  comparison                       0
  underflow                        0
  latent_positive_infinity         0
  latent_negative_infinity         0
  latent_underflow                 0

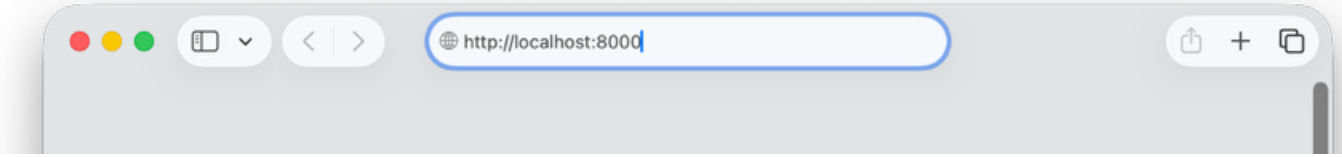
  ===== Rounding Error Report =====
  No files with rounding errors.
```

How to see the HTML reports from your laptop

- Start a lightweight, built-in web server using Python
- Remember to ssh with

  -L 8000:localhost:8000
- Then, open http://localhost:8000 in your local browser

```
$ cd /home/user1
$ python3 -m http.server 8000 &
  ...
  Serving HTTP on 0.0.0.0 port 8000
  (http://0.0.0.0:8000/) ...
```



Answer these Questions

5 minutes

Follow the README.md

- Which code lines produce NaN values?
- Where does division by zero (or near-zero pivot) occur?

Example 1: Script (Good Matrix)

```
  # Go back to example 1
$ cd ../example_1/

  # Compile and run problem
$ FPC_INSTRUMENT=1 make
$ ./lu_solve matrix.csv

  # List trace files (there is one)
$ ls -l .fpc_logs/

  # Create report
$ fpc-create-report -t "./lu_solve matrix.csv"
$ open fpc-report/index.html


$
```

Nothing interesting in the report

Optional: Transfer HTML Report to your Laptop

```
$  scp -r user1@54.152.122.0:/home/user2/tutorial/FPChecker/tutorial/example_1/fpc-report .
   ...
   ...

$  open fpc-report/index.html
```

Example 1: Script (Bad Matrix)

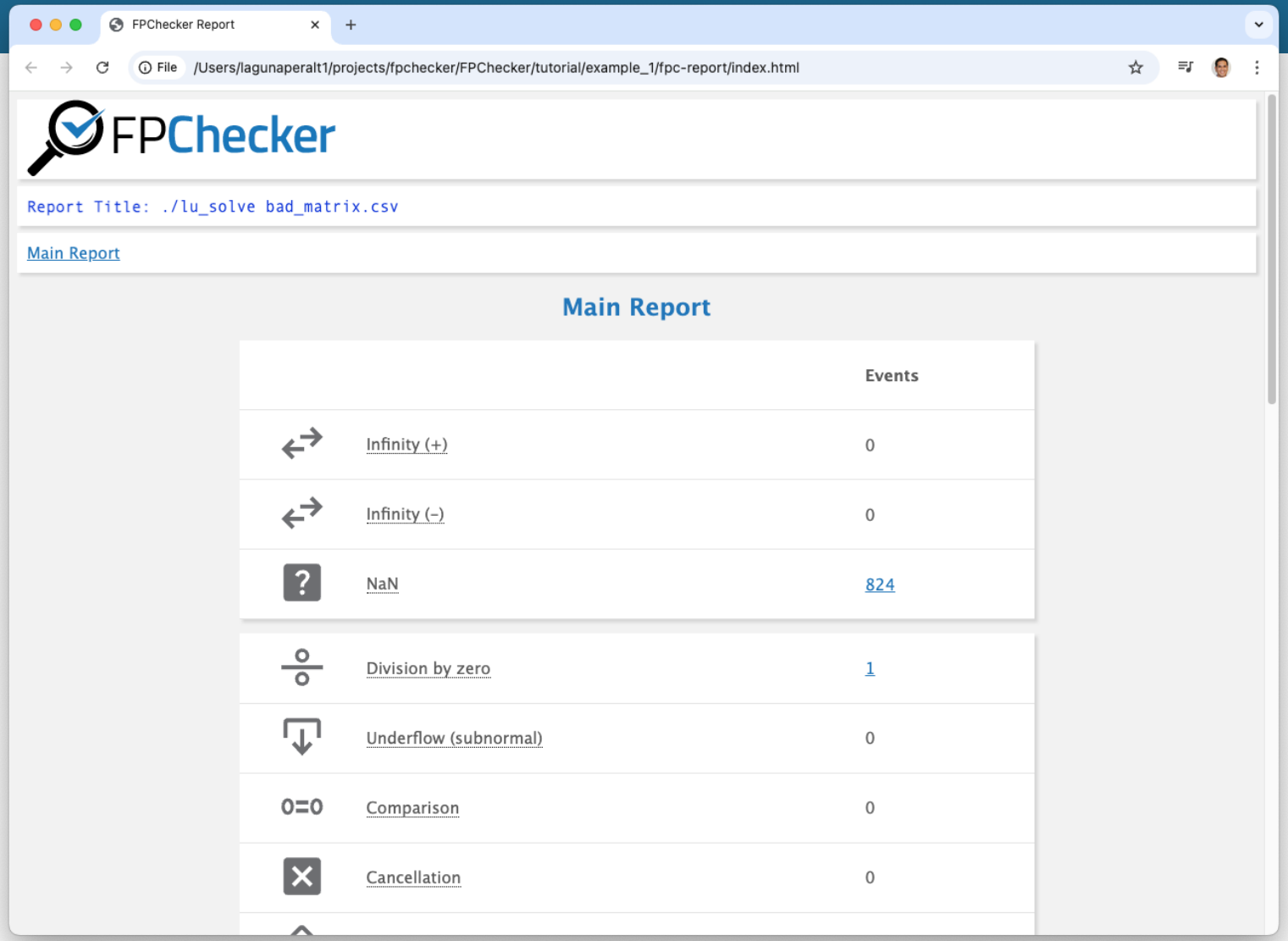
```
  # Clear logs and remove report
$ fpc-create-report -rc

  # Run problem
$ ./lu_solve bad_matrix.csv

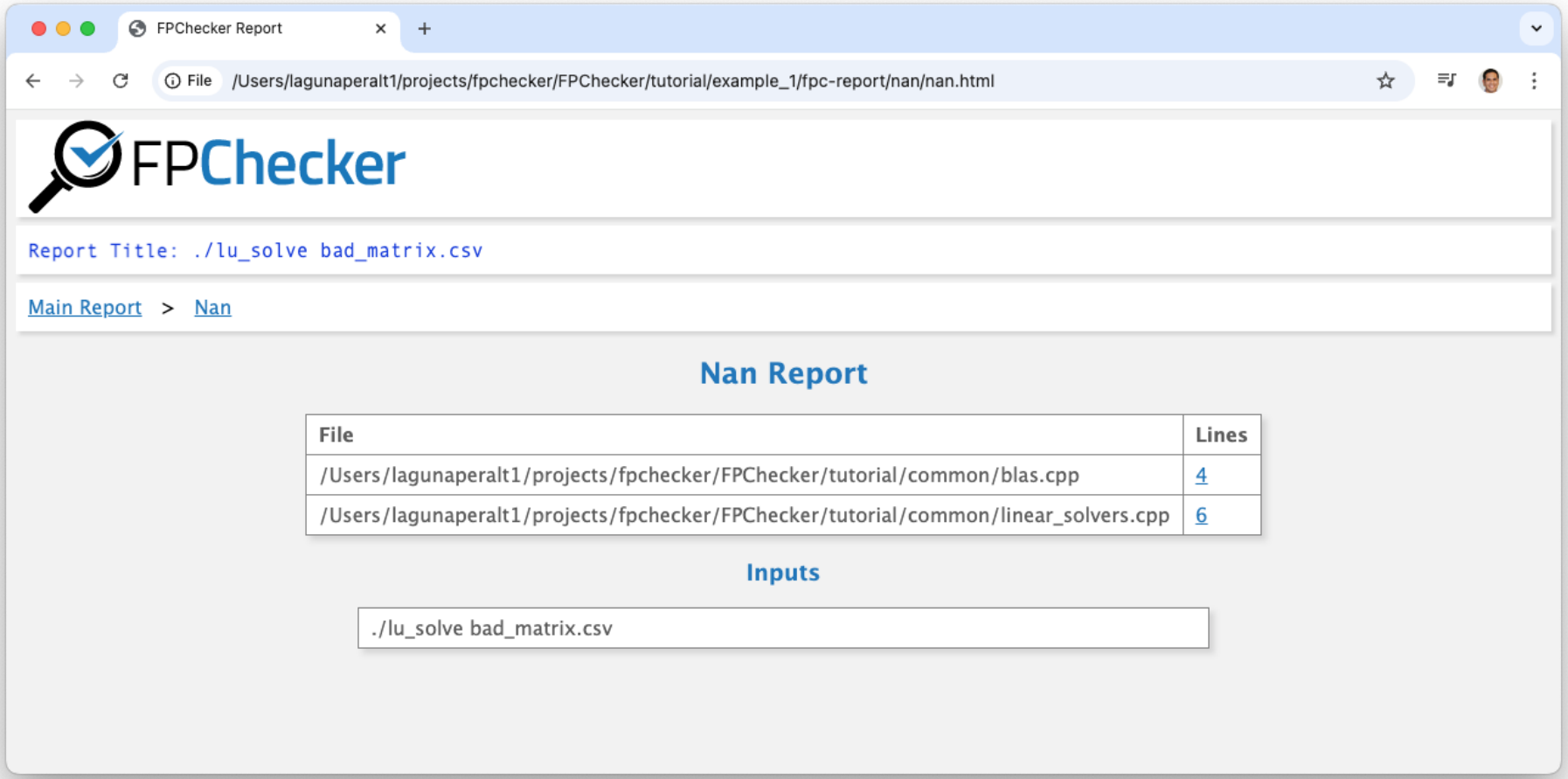
  # Create report
$ fpc-create-report -t "./lu_solve bad_matrix.csv"
$ open fpc-report/index.html
```

- NaN
- Division by zero

Report of Example 1



Report of Example 1



Module 5

Guided Practice 2: Dynamic Range Analysis

Location: tutorial/example_2

- Dynamic range analysis for FP32 and FP64
- Example with small PDE solver
- FP32 goes out of range
- Requires higher precision

Example 2:
Exponent Usage on FP64-to-FP32 porting

- Location:

  `tutorial/example_2/reaction_diffusion.cpp`

- Program description
  - 1D linear **reaction-diffusion equation**
  - PDE: $\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} + \lambda u$
  - Explicit finite difference method
  - Large λ provides positive feedback
    - Over time, it leads to **exponential growth**

FPChecker's Use Case

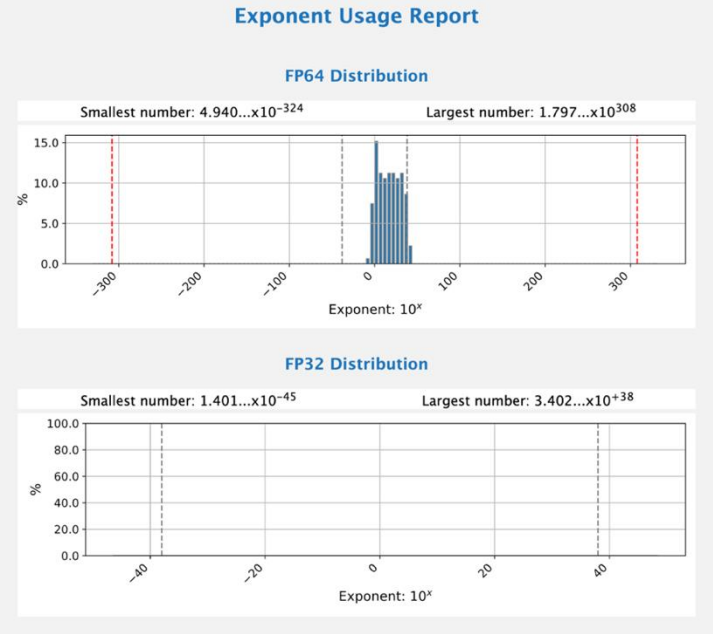
- Run simulation in FP64 and FP32
- Visualize exponent usage
- In FP32, values are "*out-of-range*"
  - Produce exceptions
  - *Should increase precision to FP64*

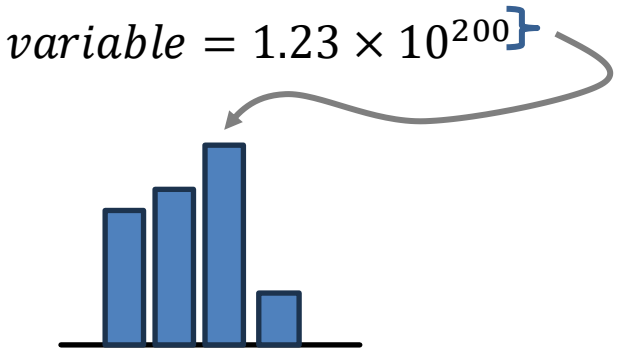
Runtime Variable to Allow Dynamic Range Profiling

- Enable this variable at **runtime**:

  FPC_EXPONENT_USAGE=1

- Allows keeping track of magnitudes (exponents)





Answer these Questions

5 minutes

Follow the README.md

1. At what magnitude does FP32 become unsafe in this example?
2. Which report sections indicate FP32 overflow risk?
3. Does this code and input require FP64?

Example 2: Script

FP64 Version

```
   # Compile and run problem
$  FPC_INSTRUMENT=1 make
$  FPC_EXPONENT_USAGE=1 ./reaction_diffusion

   # List trace files (there are two)
$  ls -l .fpc_logs/

   # Create report
$  fpc-create-report
$  open fpc-report/index.html

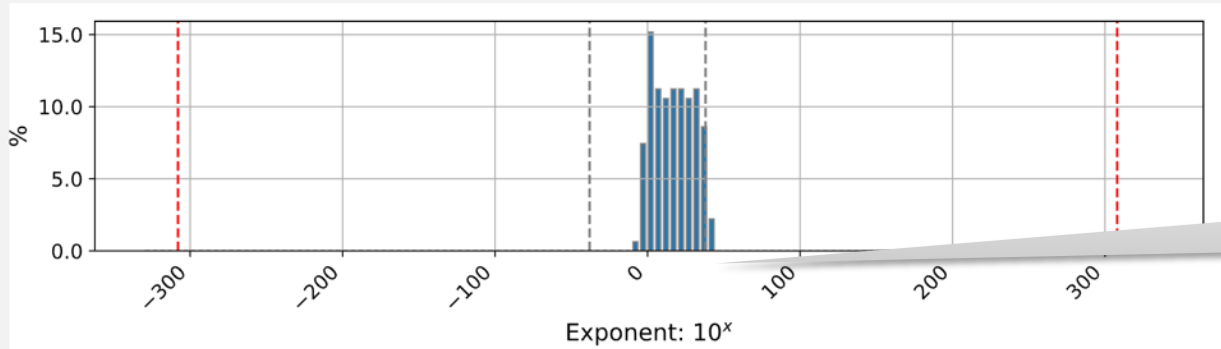
   # Clear logs and remove report
$  fpc-create-report -rc
```

Report (Example 1, FP64)

Exponent Usage Report

FP64 Distribution

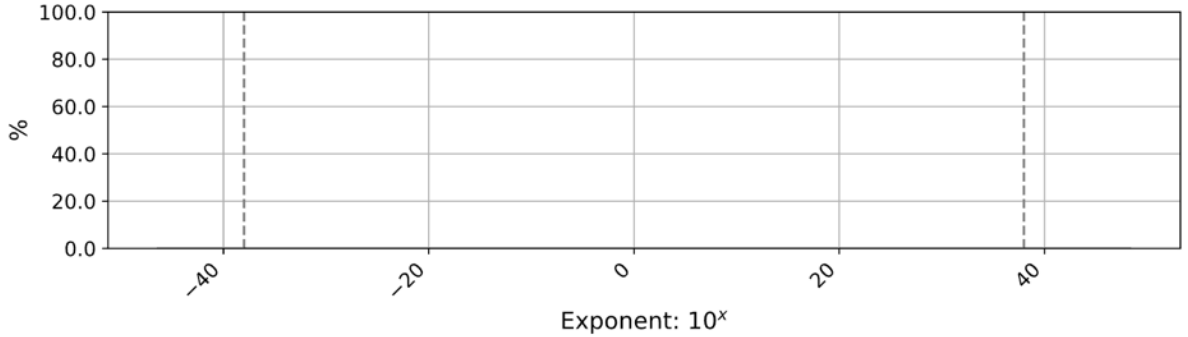
Smallest number: $4.940...x10^{-324}$ Largest number: $1.797...x10^{308}$



Out of range for FP32

FP32 Distribution

Smallest number: $1.401...x10^{-45}$ Largest number: $3.402...x10^{+38}$



Example 2: Script

First Step:

- Open `reaction_diffusion.cpp`
- Modify lines 7-8 to use FP32

```
typedef double Real_t;
// typedef float Real_t;
```

FP32 Version

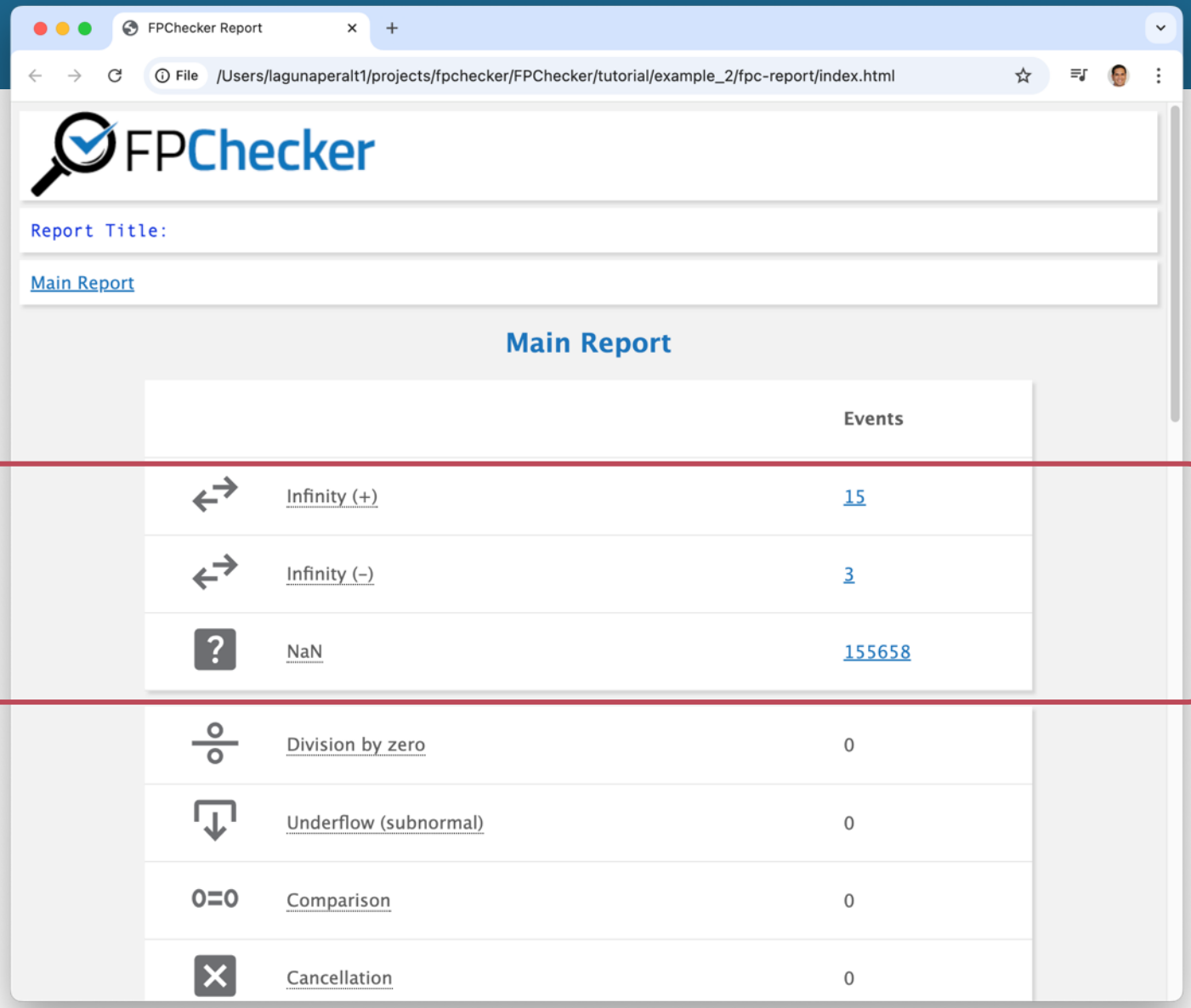
```
   # Compile and run problem
$  FPC_INSTRUMENT=1 make
$  FPC_EXPONENT_USAGE=1 ./reaction_diffusion

   # List traces files (there are two)
$  ls -l .fpc_logs/

   # Create report
$  fpc-create-report
$  open fpc-report/index.html

   # Clear logs and remove report
$  fpc-create-report -rc
```

Report (Example 2, FP32)



Agenda

	Module	Duration (min)	Time (PM)
☑	Welcome	8	1:30 – 1:38
☑	Motivation & Overview	10	1:38 – 1:48
☑	Floating-Point Basics	15	1:48 – 2:03
☑	FPChecker's Workflow & Installation	12	2:03 – 2:15
☑	Guided Practice 1: Exceptions (linear solver)	15	2:15 – 2:30
☑	Guided Practice 2: Dynamic Range Analysis	15	2:30 – 2:45
	Guided Practice 3: Mixed-Precision in CG	15	2:45 – 3:00
	Break	**30**	
	Demo 1: Agentic mixed-precision with FPChecker	15	3:30 – 3:45
	Demo 2: Instrumenting a production library (Hypre)	10	3:45 – 3:55
	Q&A - FPChecker	5	3:55 – 4:00
	NixNaN	60	4:00 – 5:00

Module 6

Guided Practice 3: Data-Driven Mixed-Precision Conjugate Gradient

Location: tutorial/example_7

- CG example code (functions, precision, etc.)
- Rounding error analysis for a given input
- Data-driven mixed-precision version

Example Objectives

1. Run the baseline FP32 solver with instrumentation
2. Identify high-error lines from the report
3. Compare FP32, mixed, and FP64 behavior
4. Explain why targeted mixed precision improves convergence and residual

Algorithm in FP32

Algorithm 1 Conjugate Gradient (FP32 baseline)

Require: SPD matrix A, RHS b, tolerance τ, max iterations $k_{\max}$

$x_0 \leftarrow 0,\ r_0 \leftarrow b - Ax_0,\ p_0 \leftarrow r_0$
$\rho_0 \leftarrow r_0^T r_0,\ b_n \leftarrow \sqrt{b^T b}$
for $k = 0, 1, \ldots, k_{\max} - 1$ **do**
  $q_k \leftarrow Ap_k$
  $\alpha_k \leftarrow \rho_k / (p_k^T q_k)$
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  $r_{k+1} \leftarrow r_k - \alpha_k q_k$
  $\rho_{k+1} \leftarrow r_{k+1}^T r_{k+1}$
  if $\sqrt{\rho_{k+1}}/b_n < \tau$ **then**
    return x_{k+1}
  end if
  $\beta_k \leftarrow \rho_{k+1}/\rho_k$
  $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
end for
return $x_{k_{\max}}$

- No preconditioner
- Example solves $Ax = 1$
- Script to generate matrices provided
- Tridiagonal Toeplitz SPD matrices
- Varying condition numbers

What the example includes

`cg.cpp`	FP32 solver (will be instrumented with FPChecker rounding-error tracking)
`cg_mixed.cpp`	Mixed-precision solver (targeted FP64 intermediates)
`cg_FP64.cpp`	FP64 reference solver
`generate_cg_matrices.py`	Matrix generation utility
`matrices/`	Pre-generated SPD matrices used in the tutorial
`Makefile`	Build rules for all variants

Challenge & Questions

5 minutes

- Follow README.md
- Run FP32 and inspect rounding errors

Questions:

1. Which lines have the largest relative error?
2. Which operations are likely causing cancellation or unstable accumulation?
3. Which parts can remain FP32?
4. Which variables should move to FP64?

Script 1: Getting Rounding Errors in FP32

- Check in Makefile we are using: FPC_INSTRUMENT_ERR_TRACKING=1

```
   # Compile and run problem
$  make clean
$  make cg
$  ./cg matrices/5_matrix_3.998e+04.csv 500 1e-6

   # Create report in the terminal
$  fpc-create-report -s rounding
```

Rounding Error Report

Line	Code	Error	Rel. Error
39	guess = 0.5f * (guess + x / guess);	2.148692e+03	9.999707e-01
52	result += v1[i] * v2[i];	1.154287e+06	1.000000e+00
75	result[i] += A[i * n + j] * v[j];	-9.567699e+01	1.010557e+00
90	result[i] = v1[i] - alpha * v2[i];	9.567699e+01	9.999952e-01
97	result[i] = v1[i] + alpha * v2[i];	-2.400936e+02	1.448971e+00
111	result[i] = r[i] + beta * p[i];	2.366245e+03	1.000000e+00
146	row.push_back(atof(token));	0.000000e+00	0.000000e+00
165		0.000000e+00	0.000000e+00
227		0.000000e+00	0.000000e+00
229	float alpha = rs_old / dot_product(p, Ap);	8.886324e-01	4.678308e+05
239	float relative_residual = my_sqrt(rs_new) / relative_b;	9.806319e+01	1.000000e+00
248	float beta = rs_new / rs_old;	2.993113e-01	2.947664e-01
265		-7.821318e+01	1.000000e+00
287	float tolerance = (argc > 3) ? atof(argv[3]) : 1e-6f;	0.000000e+00	0.000000e+00
645	(line not found)	0.000000e+00	0.000000e+00

Lines 39:
FP32 Newton-style sqrt routine

Line 52:
Dot product accumulation

Lines 75:
Matrix-vector row accumulation

Lines 90, 97:
Vector fused updates

Lines 111:
Search-direction update

Line 229:
Division with high error

Line 239:
Division with high error

Errors might be slightly different between AWS instances and Mac

Changes Performed for Mixed-Precision

Observation: FP32 accumulation and sensitive divisions amplify error

Main changes:

- We target FP64 intermediate operations to reduce rounding error propagation
- No need to convert all storage to FP64 (***most storage remains FP32***)
- We promote to FP64 only ***sensitive arithmetic***
- *No changes performed to the algorithm*

Script 2: Mixed and FP64

```
   # Compile and run mixed and FP64
$  make cg_mixed cg_FP64

$  printf "\n=== FP32 ===\n"
$  ./cg matrices/5_matrix_3.998e+04.csv 500 1e-6

$  printf "\n=== Mixed ===\n"
$  ./cg_mixed matrices/5_matrix_3.998e+04.csv 500 1e-6

$  printf "\n=== FP64 ===\n"
$  ./cg_FP64 matrices/5_matrix_3.998e+04.csv 500 1e-6
```

Numerical Results for FP32, Mixed, and FP64

=== FP32 ===

```
Converged in 148 iterations. Residual Norm: 7.414996e-06
Average time per iteration: 2.775208e-02 seconds
Execution time: 4.167169e+00 seconds
Final Residual Norm (||Ax - b||): 3.148735e-02
```

=== Mixed ===

```
Converged in 77 iterations. Residual Norm: 7.945719e-06
Average time per iteration: 6.839299e-06 seconds
Execution time: 5.606250e-04 seconds
Final Residual Norm (||Ax - b||): 5.171927e-03
```

=== FP64 ===

```
Converged in 50 iterations. Residual Norm: 1.907349e-06
Average time per iteration: 1.892252e-05 seconds
Execution time: 1.064250e-03 seconds
Final Residual Norm (||Ax - b||): 1.907349e-06
```

	FP32	Mixed	FP64
Iterations	148	77	50
||Ax - b||	3.148735e-02	5.171927e-03	1.907349e-06

Questions

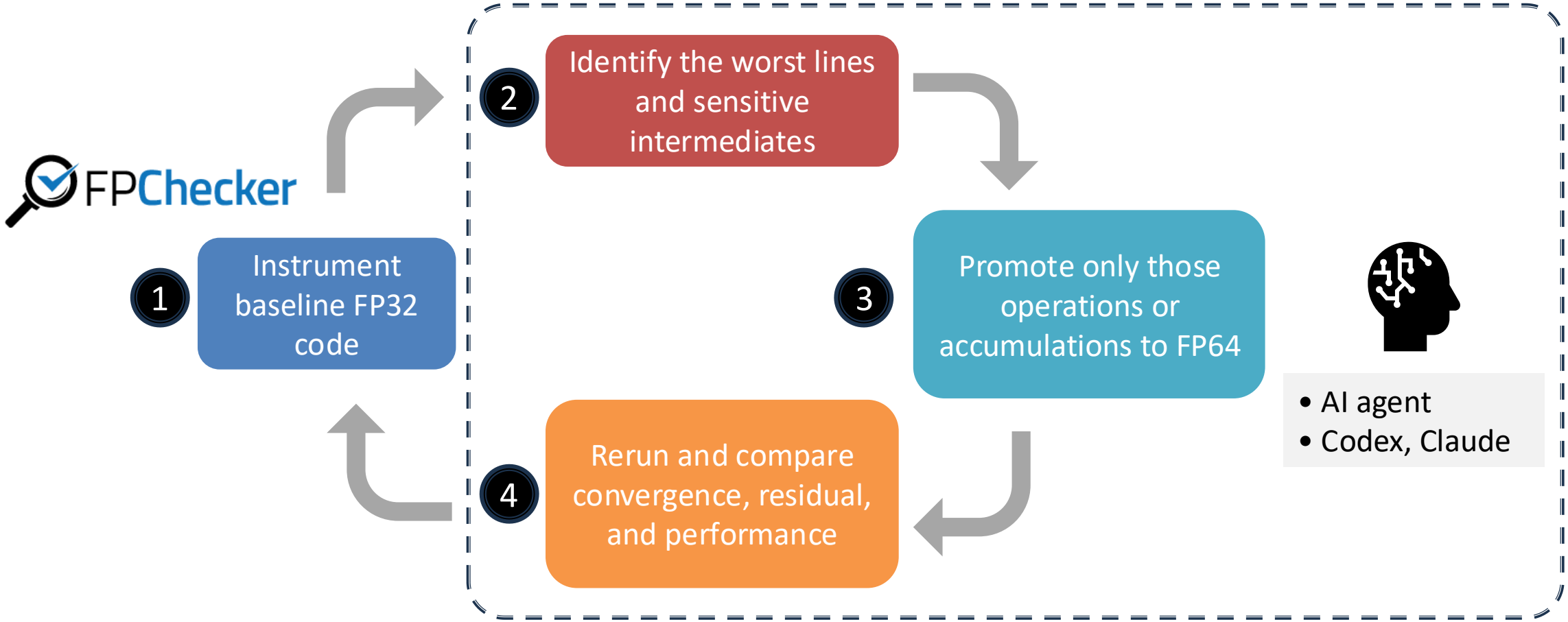
Agenda

	Module	Duration (min)	Time (PM)
☑	Welcome	8	1:30 – 1:38
☑	Motivation & Overview	10	1:38 – 1:48
☑	Floating-Point Basics	15	1:48 – 2:03
☑	FPChecker's Workflow & Installation	12	2:03 – 2:15
☑	Guided Practice 1: Exceptions (linear solver)	15	2:15 – 2:30
☑	Guided Practice 2: Dynamic Range Analysis	15	2:30 – 2:45
☑	Guided Practice 3: Mixed-Precision in CG	15	2:45 – 3:00
☑	**Break**	**30**	
	Demo 1: Agentic mixed-precision with FPChecker	15	3:30 – 3:45
	Demo 2: Instrumenting a production library (Hypre)	10	3:45 – 3:55
	Q&A - FPChecker	5	3:55 – 4:00
	NixNaN	60	4:00 – 5:00

Module 7

Demo 1: Agentic mixed-precision with FPChecker

Workflow for Users: *Practical Tuning Loop*



Case Study: LULESH & AI Agent

- GPT 6 (Sol)
- Prompt summary:
  - 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

 # Find Mixed precision configuration in LULESH with FPChecker

1. Build FPChecker using clang 19
A version of clang 19 is here:
```
$ /opt/anaconda3/envs/tutorial_env/bin/clang++ --version
clang version 19.1.7
Target: arm64-apple-darwin25.5.0
Thread model: posix
InstalledDir: /opt/anaconda3/envs/tutorial_env/bin
```

2. Instrument LULESH with FPChecker (FPC) and produce an error-tracking report
(a) LULESH should be compiled in “float” precision.
(b) Use a small input and a few steps to generate the FPC report.
(c) To test the mixed precision version against the FP64 version, input sizes in the order of 96 and 128 are OK.
(d) Use “fpchecker-show” to see the instrumentation flags for the FP32 error tracking runtime
(e) LULESH should run in CPU, with -O2, no OpenMP, no MPI.

3. Use the report to identify high-error hotspots:
(a) Text reports can be generated running “fpc-create-report -s”

4. Find a mixed-precision configuration by promoting to FP64 based on high-error hotspots
(a) Identify places where error is high and require using higher precision (FP64)
(b) To decide if the error of the mixed-precision configuration is acceptable, look at the final energy (relative to the FP64 energy). We would like an error in the order of 10^-6 and 10^-7

5. Report FOM speedup, and accuracy
(a) Ideally we want an FOM speedup (relative to the FP64 FOM) of 10% or more.
(b) Save the findings in a report in .md format

Video
(Codex running)

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	3.2468881823660206e-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%

Module 8

Demo 2:
Hypre linear solver package

Location: tutorial/example_8

- Practice with Hypre
- Build Hypre with FPChecker
- Show error report

Hypre



- Library of scalable linear solvers: AMG, PCG, ILU, …
- Designed to handle discretized partial differential equations (PDEs) at extreme scales
- Optimized to run the world's fastest supercomputers like **El Capitan** and **Frontier**
  - Uses MPI
- Supports FP32, FP64 and mixed-precision

Example Objectives

1. Download and build Hypre without FPChecker
   - Run a Hypre test with an input
2. **Instrument** Hypre and the test with FPChecker
   - Run an instrumented Hypre test with an input
3. Produce a **rounding error report**

What the Example Includes

`README.md`	Instructions to build Hypre and test without FPChecker
`Makefile`	Makefile to build the test (hypre_test.c)
`hypre_test.c`	Test that runs three solvers: PCG, AMG, and PCG with AMG preconditioner
`matrix.csv`	Input matrix
`README_solution.md`	Solution instructions (do not read immediately)
`Makefile.solution`	Solution Makefile (do not read immediately)

Independent Practice

- **10 minutes:** build Hypre and test (follow README.md)
- **10 minutes:** Instrument Hypre and test, build error report
- **5 minutes:** Instructor reveals the solution

Do no immediately look at
- README_solution.md, and/or
- Makefile.solution

Recap & Next Steps

Things We didn't Cover…

- Multiple inputs
  - How is error impacted?
- Performance aspects
- Dynamic mixed-precision
  - Switch between FP32 and FP64 dynamically
- Other floating-point formats: FP16, bfloat16, …
- Instrumenting FP64 or mixed code
- GPUs
- Fortran

Learning Objectives

Tools for Floating-Point Error Profiling → with a focus on Mixed-Precision Tunning



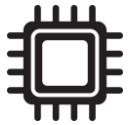
Identify numerical instabilities

- Isolate floating-point exceptions (**NaN** and **Infinity**) and **cancellation**



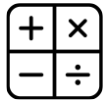
Characterize dynamic range and rounding error

- Generate ***dynamic range profiles & profile error accumulation***



Automate analysis via compiler and binary instrumentation: FPChecker & NixNan

- Lear to compile and run code with the tools



Implement data-driven mixed-precision strategies

- Use numerical error reports to assign lower or higher precision: **FP32** or **FP64**

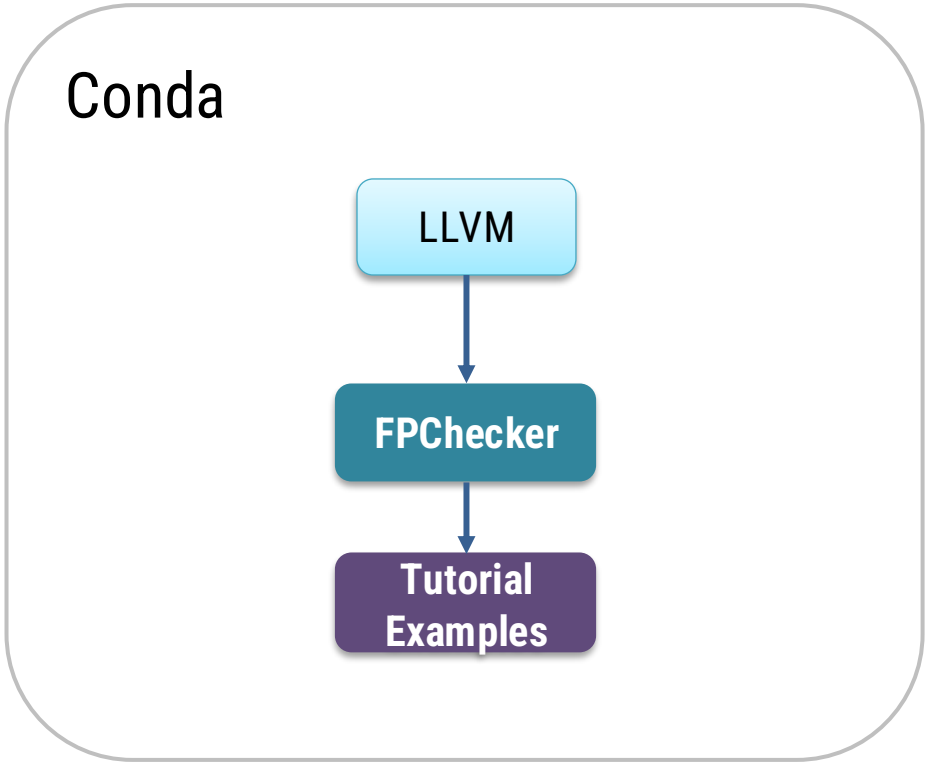
Agenda

	Module	Duration (min)	Time (PM)
☑	Welcome	8	1:30 – 1:38
☑	Motivation & Overview	10	1:38 – 1:48
☑	Floating-Point Basics	15	1:48 – 2:03
☑	FPChecker's Workflow & Installation	12	2:03 – 2:15
☑	Guided Practice 1: Exceptions (linear solver)	15	2:15 – 2:30
☑	Guided Practice 2: Dynamic Range Analysis	15	2:30 – 2:45
☑	Guided Practice 3: Mixed-Precision in CG	15	2:45 – 3:00
☑	**Break**	**30**	
☑	Demo 1: Agentic mixed-precision with FPChecker	15	3:30 – 3:45
☑	Demo 2: Instrumenting a production library (Hypre)	10	3:45 – 3:55
☑	Q&A - FPChecker	5	3:55 – 4:00
	NixNaN	60	4:00 – 5:00

Installing LLVM and FPChecker with Conda

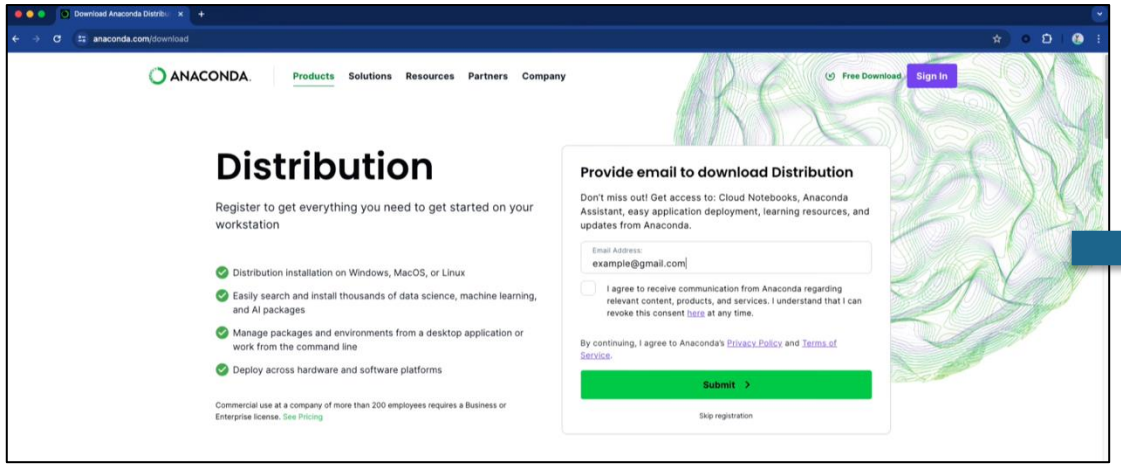
Install Conda

- Why Conda?
  - **Easy to deploy clang/LLVM** for the tutorial (a couple of minutes)
- Alternative method: **build clang/LLVM from source**
  - Not recommended for the tutorial
  - Best approach for production installation

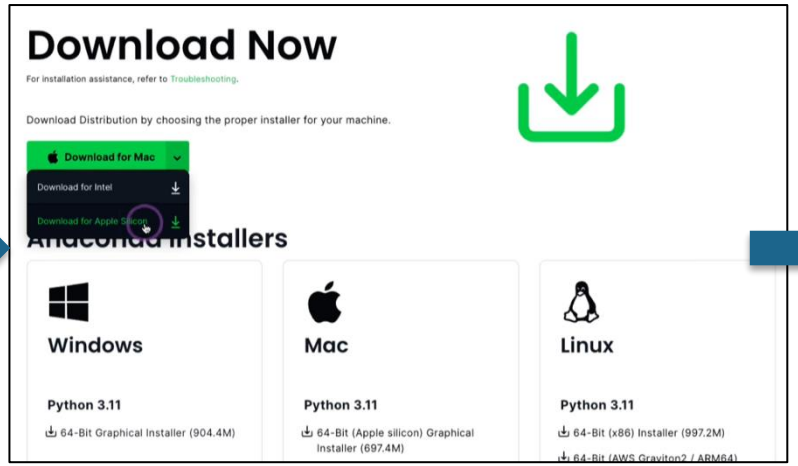


Steps to Install Conda in Mac

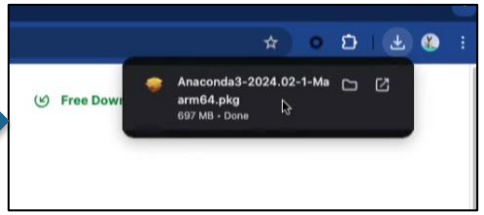
Got https://www.anaconda.com/ -> Download



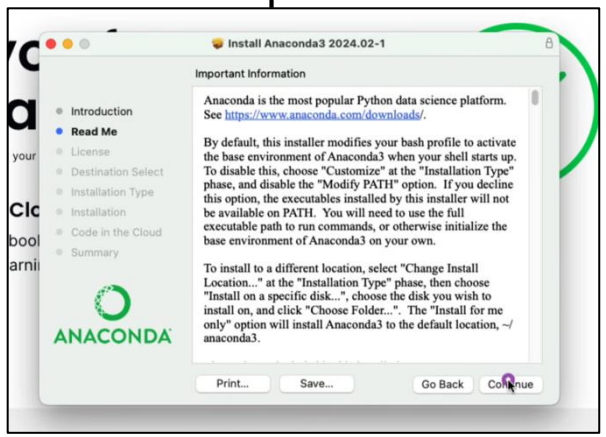
Download



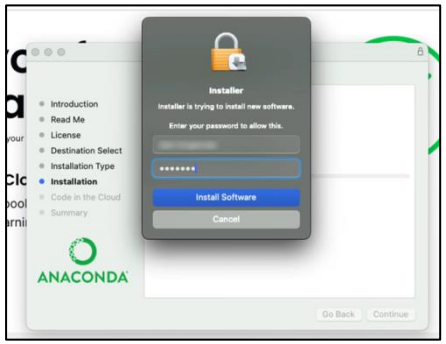
Open Installer



Follow the steps



Provide password if asked



In the terminal, you should see the word **base**:

```
(base) [user@system] $
```

Install LLVM and Python with Conda

Not needed for AWS instances

```
   # Create a conda env
$  conda create --name tutorial_env
$  conda activate tutorial_env

   # Install dependencies: cmake, LLVM, clang++, python
$  conda install cmake
$  conda install make -c conda-forge
$  conda install llvmdev=19.1.7 -c conda-forge
$  conda install clangxx=19.1.7 -c conda-forge
$  conda install python=3.12.9
$  conda install -c conda-forge git

   # Install matplotlib. Not required to build FPChecker, but needed for reports
$  pip3 install matplotlib

   # MPI for some examples, but not required to build for FPChecker
$  conda install openmpi=5.0.7 -c conda-forge
```

Get FPChecker

This tutorial is based on release v0.6



Laptop

```
    # We will install all in /tmp
$   cd /tmp
$   mkdir tutorial
$   cd tutorial

    # Clone FPChecker
$   git clone https://github.com/LLNL/FPChecker.git
$   cd FPChecker
```

Install FPChecker



Laptop

```
$ mkdir build
$ cd build/
$ cmake -DCMAKE_INSTALL_PREFIX=../../install ..
$ make && make install
```

```
  # Export installation path
$ export PATH=/tmp/tutorial/install/bin:$PATH
```

fpchecker-show

```
$ fpchecker-show
  ========================================
           FPChecker Configuration
  ========================================

  Installation path: /home/user1/tutorial/install

  Add the following to CFLAGS and/or CXXFLAGS:

  (1) For exceptions checking:
  -g -include /home/user1/tutorial/install/src/Runtime_cpu.h -fpass-
  plugin=/home/user1/tutorial/install/lib/libfpchecker_cpu.so

  (2) For rounding error tracking:
  -g -fno-vectorize -fno-slp-vectorize -include /home/user1/tutorial/install/src/Runtime_error.h
  -fpass-plugin=/home/user1/tutorial/install/lib/libfpchecker_error.so

  Wrappers are located here:
  /home/user1/tutorial/install/bin/clang-fpchecker
  /home/user1/tutorial/install/bin/clang++-fpchecker
  /home/user1/tutorial/install/bin/mpicc-fpchecker
  /home/user1/tutorial/install/bin/mpicxx-fpchecker
```



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.