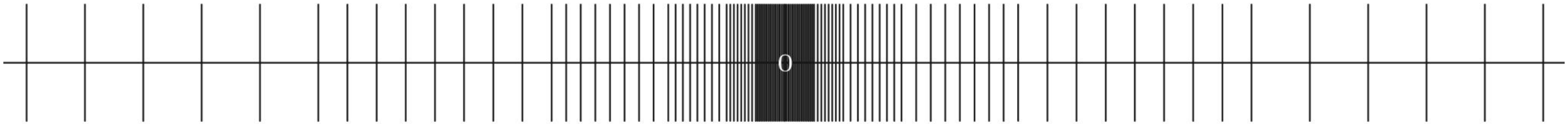
# Tools to Detect and Diagnose Floating-Point Errors in Heterogeneous Computing Hardware and Software

**A Half-Day Tutorial at Supercomputing 2025**

**Tut 122:** Sunday, 16 November 2025, 8:30am - 12:00pm CST : **Location 121**




Lawrence Livermore National Laboratory

KAHLERT SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

UC DAVIS
UNIVERSITY OF CALIFORNIA

# Tools to Detect and Diagnose Floating-Point Errors in Heterogeneous Computing Hardware and Software

**A Half-Day Tutorial at Supercomputing 2025**

## Presenters


Ignacio Laguna

Lawrence Livermore National Laboratory

Mark Baranowski and Ganesh Gopalakrishnan

KAHLERT SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

Dolores Miao and Cindy Rubio-González

UCDAVIS
UNIVERSITY OF CALIFORNIA

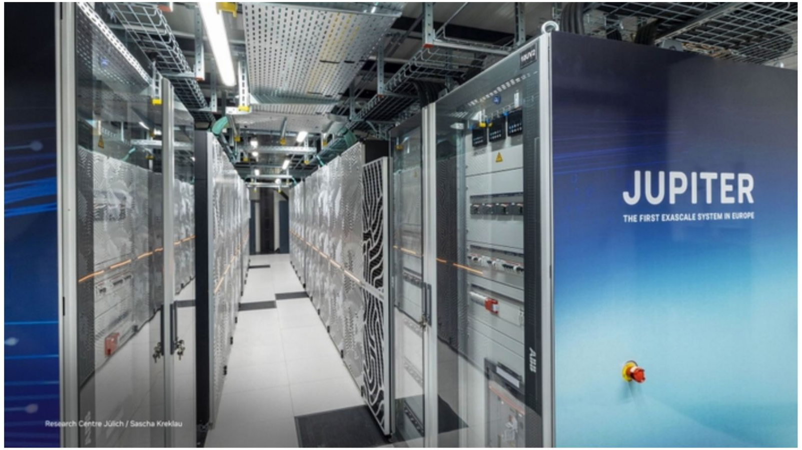
# Agenda, and a single QR code to access all our material

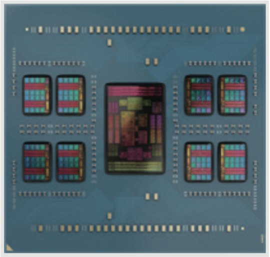
- 08:30 : Introduction
- 08:35 : FPChecker
- 09:35 : NixNan (Part-1)
- 09:55 : Plz Fill Survey
- 10:00 : Coffee Break
- 10:30 : Brief Recap
- 10:35 : NixNan (Part-2)
- 11:05 : FloatGuard
- 11:50 : Closing Remarks
- 11:55 : Plz Fill Survey

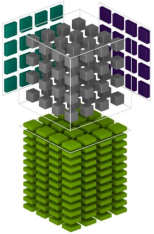
SC 25 Floating-Point Tutorial : All Tools and Benchmarks

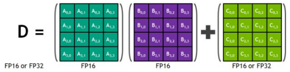
fpanalysistools.org

# Floating-Point Computations are fundamental to CPUs and GPUs

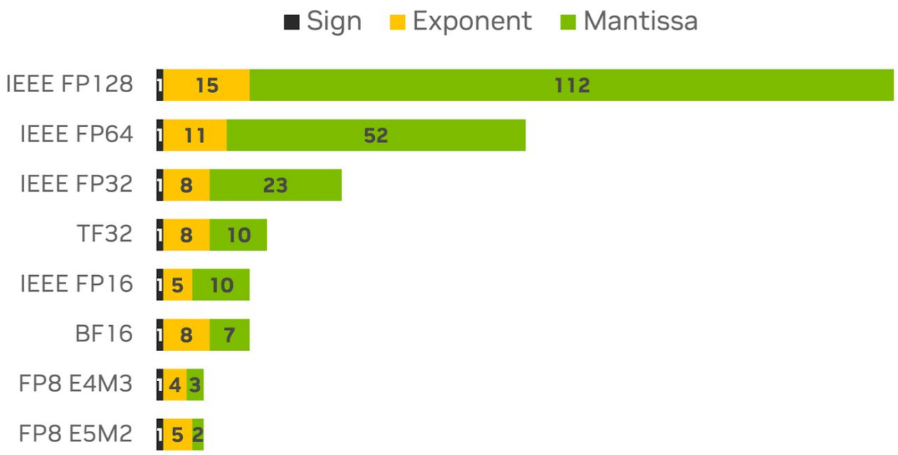




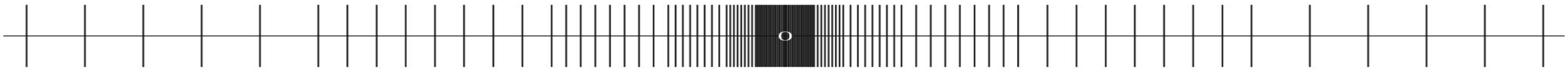
# Floating-Point Numbers : A compromise, with many good properties, that can span a large range with a small number of bits

## Half the representable numbers lie between -1 and +1

| Format | Prec. | Exp. | Sm. norm. | Lrgst. Norm. | ULP @ emax | **eql. 2'c bits** | Redn. factor |
|---|---|---|---|---|---|---|---|
| FP16 16 bits | 10 | 5 | 6.10 E-5 | 6.55 E4 | 5 | 41 | 2.5 |
| BF16 16 bits | 7 | 8 | 1.18 E-38 | 3.39 E38 | 120 | 262 | 16.3 |
| TF32 19 bits | 10 | 8 | 1.18 E-38 | 3.40 E38 | 117 | 265 | 13.9 |
| FP32 32 bits | 23 | 8 | 1.18 E-38 | 3.40 E38 | 104 | 254 | 7.9 |
| FP64 **64 bits** | 52 | 11 | 2.23 E-308 | 1.80 E308 | **971** | **2099 bits** | 32.7 |



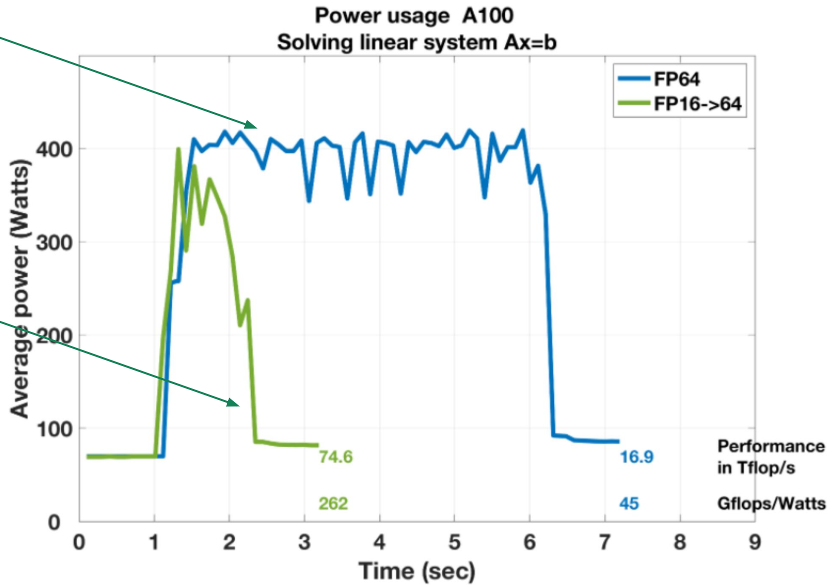

FP allows us to get away moving **64 bits** …. instead of moving **2099 bits (in 2's comp.)**



# Lower precision Floating-Point saves **energy** (and **time**)

High-Precision
LU Solving

Low-Precision
LU Solving




From "Hardware Trends Impacting Floating-Point Computations In Scientific Applications," Dongarra et al.

While increasing precision almost always improves accuracy, there are rare cases where this does not happen.

- The loop in FP32 iterates 10 times (as per real semantics)
- In FP64, it iterates 11 times

```
File Edit Options Buffers Tools C Help
#include <stdio.h>

int main() {
  int cnt=0;
  for (float i = 0.0f; i < 1.0f; i += 0.1) {
    cnt++;
    printf("%.1f ", i);
  }
  printf("\n");
  printf("cnt=%d\n",cnt);
  return 0;
-UU-:---  F1  floatCnt.c     Top   L5     (C/*l Abbrev) ------------------------------
#include <stdio.h>

int main() {
  int cnt=0;
  for (double i = 0.0f; i < 1.0f; i += 0.1) {
    cnt++;
    printf("%.1f ", i);
  }
  printf("\n");
  printf("cnt=%d\n",cnt);
  return 0;
-UU-:---  F1  doubCnt.c      Top   L1     (C/*l Abbrev) ------------------------------
```

```
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ gcc -o fc floatCnt.c
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ gcc -o dc doubCnt.c
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ ./fc
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
cnt=10
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ ./dc
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
cnt=11
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$
```

# Non-intuitive behavior caused by Rounding!

```
File Edit Options Buffers Tools C Help
#include <stdio.h>

int main() {
  int cnt=0;
  for (float i = 0.0f; i < 1.0f; i += 0.1) {
    cnt++;
    printf("%.1f ", i);
  }
  printf("\n");
  printf("cnt=%d\n",cnt);
  return 0;
-UU-:---  F1  floatCnt.c      Top   L5     (C/*l Abbrev) ------------------------------
#include <stdio.h>

int main() {
  int cnt=0;
  for (double i = 0.0f; i < 1.0f; i += 0.1) {
    cnt++;
    printf("%.1f ", i);
  }
  printf("\n");
  printf("cnt=%d\n",cnt);
  return 0;
-UU-:---  F1  doubCnt.c       Top   L1     (C/*l Abbrev) ------------------------------
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ gcc -o fc floatCnt.c
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ gcc -o dc doubCnt.c
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ ./fc
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
cnt=10
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ ./dc
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
cnt=11
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$
```

```
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ gc
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ ./
Value: 0.00000000000000000000
(sign: 0, exp: 0, mantissa: 0x000000)
Value: 0.10000000149011611938
(sign: 0, exp: 123, mantissa: 0x4CCCCD)
Value: 0.20000000298023223877
(sign: 0, exp: 124, mantissa: 0x4CCCCD)
Value: 0.30000001192092895508
(sign: 0, exp: 125, mantissa: 0x19999A)
Value: 0.40000000596046447754
(sign: 0, exp: 125, mantissa: 0x4CCCCD)
Value: 0.50000000000000000000
(sign: 0, exp: 126, mantissa: 0x000000)
Value: 0.60000002384185791016
(sign: 0, exp: 126, mantissa: 0x19999A)
Value: 0.70000004768371582031
(sign: 0, exp: 126, mantissa: 0x333334)
Value: 0.80000007152557373047
(sign: 0, exp: 126, mantissa: 0x4CCCCE)
Value: 0.90000009536743164062
(sign: 0, exp: 126, mantissa: 0x666668)
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$ ./
Value: 0.00000000000000000000
(sign: 0, exp: 0, mantissa: 0x0000000000000)
Value: 0.10000000000000000555
(sign: 0, exp: 1019, mantissa: 0x999999999999A)
Value: 0.20000000000000001110
(sign: 0, exp: 1020, mantissa: 0x999999999999A)
Value: 0.30000000000000004441
(sign: 0, exp: 1021, mantissa: 0x3333333333334)
Value: 0.40000000000000002220
(sign: 0, exp: 1021, mantissa: 0x999999999999A)
Value: 0.50000000000000000000
(sign: 0, exp: 1022, mantissa: 0x0000000000000)
Value: 0.59999999999999997780
(sign: 0, exp: 1022, mantissa: 0x3333333333333)
Value: 0.69999999999999995559
(sign: 0, exp: 1022, mantissa: 0x6666666666666)
Value: 0.79999999999999993339
(sign: 0, exp: 1022, mantissa: 0x9999999999999)
Value: 0.89999999999999991118
(sign: 0, exp: 1022, mantissa: 0xCCCCCCCCCCCCC)
Value: 0.99999999999999988898
(sign: 0, exp: 1022, mantissa: 0xFFFFFFFFFFFFF)
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$
(base) ganesh@ganesh-XPS-17-9710:~/parfloat-cs6961/c-prec-expts$
```

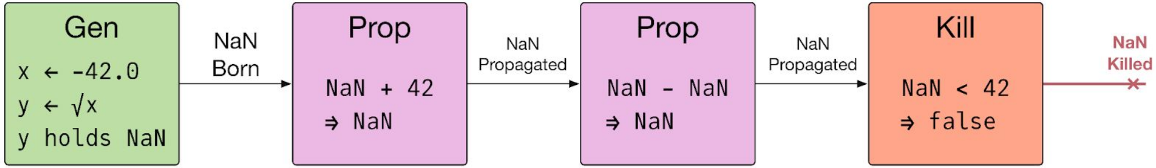
# Focus of this tutorial

- Excessive Rounding Error  (not addressed in this tutorial)
- Exhausting the range **(**addressed**)**
    - **Detection of Exceptions  (main focus)**

# FP Exceptions (Exceptional Values) are Undesirable (*often* are bugs)

- FP32: ( $1.9 \times 10^{19}$ $)^2$ → INF … ( in E4M3, $16^2$ = INF ) ⇐ a popular ML low-prec format
  - Easily generated during dot-product
- Of the FP Exceptions, we focus on INF and NaN
  - NaNs can skew control-flow
- This macro is buggy
  - #define MAX(x,y) ( (x >= y) ? x : y )

  hint : consider x == NaN or y == NaN
- Exceptions often vanish harmlessly (e.g. in ML codes)
  - But it is difficult to know which control-flows they have affected meanwhile



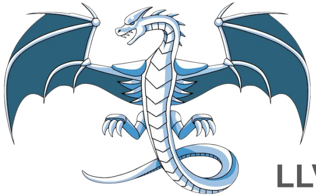
# Tools Presented

FPChecker

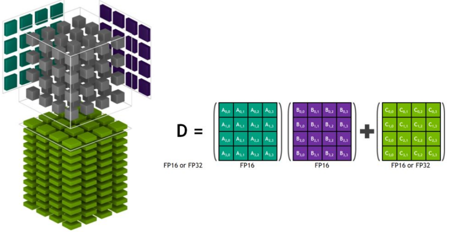
GPU-FPX + NixNaN

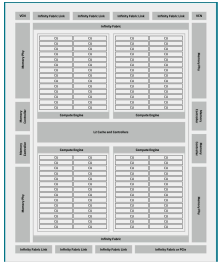
FloatGuard

NVIDIA GPUs



LLVM



AMD GPUs

Lawrence Livermore National Laboratory

KAHLERT SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

UCDAVIS
UNIVERSITY OF CALIFORNIA

# Agenda

- 08:30 : Introduction
- **08:35 : FPChecker**
- 09:35 : NixNan (Part-1)
- 09:55 : Fill Survey
- 10:00 : Coffee Break
- 10:30 : Brief Recap
- 10:35 : NixNan (Part-2)
- 11:05 : FloatGuard
- 11:50 : Closing Remarks
- 11:55 : Fill Survey