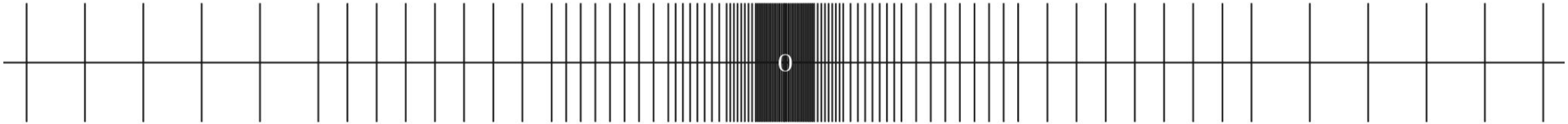
NixNan: A tool for FP exception tracking in NVIDIA GPUs

Supercomputing 2026 Half-Day Tutorial, Tut W184b-c: Monday, 16 November 2026, 1:30pm - 5pm CST

Also featured in the GPU-mode Channel on 11/9/26

Mark S. Baranowski and Ganesh Gopalakrishnan

contact: mark.s.baranowski@gmail.com and ganesh@cs.utah.edu



Supported in part by NSF 2446084, Collaborative Research: CS2: DANUBE:
Formal Methods for Scientific Computing Bridging the Data-Numerics Behavior Tug-of-War

KAHLERT SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

How this "exceptional" talk will go

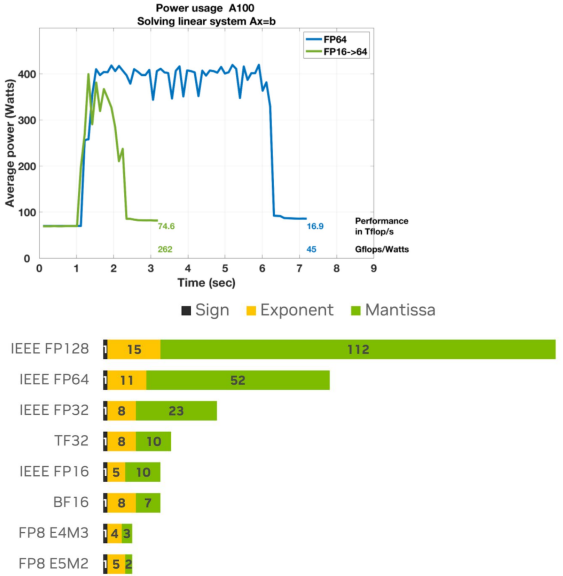
- Floating-point (FP) is used "everywhere"
- It is tricky
  - Non-intuitive Behaviors, Rounding error, Exceptions
- You see more at the binary level
- We help you hunt more using NixNan!
- Case Studies
- How does NixNan Work?
- Tool Complexity Control, Best Practices
- Where to get nixnan
  - https://github.com/parfloat/nixnan - recommended branch now is print_histo_instrn
- Where our examples are (all runs on an A100 - a 3090 GPU)
  - https://github.com/parfloat/SC26-Tutorial - recommended branch now is Flash-Attn
- Conclusions

FP: Move fewer bits … if you can stomach the error (and surprises!)

Lower precision ⇒ Less energy ⇒ but less accuracy

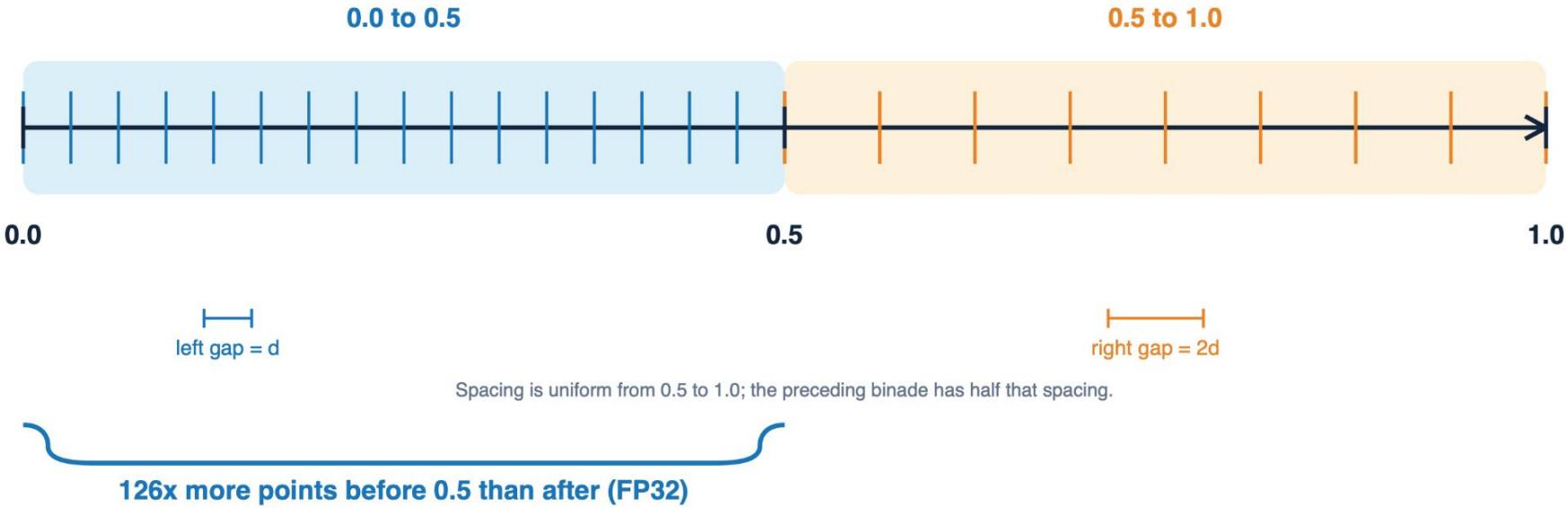
Many Fmts.	Mant.	Exp.	Sm. norm.	Lrgst. Norm.	ULP @ emax	**eql. 2'c bits**	**Redn. factor**
FP16 16 bits	10	5	6.10 E-5	6.55 E4	5	17	1
BF16 16 bits	7	8	1.18 E-38	3.39 E38	120	129	4
TF32 19 bits	10	8	1.18 E-38	3.40 E38	117	129	4
FP32 32 bits	23	8	1.18 E-38	3.40 E38	104	129	4
FP64 **64 bits**	52	11	2.23 E-308	1.80 E308	**971**	**1025 bits**	16

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



FP allows us to get away moving **64 bits** …. instead of moving **1025 bits (in 2's comp.)**

FP spacing doubles at binades!



Increasing Precision may **Increase Confusion :-)**

Float to Double : **10 Iterations** (intuitive) → **11 Iterations** (not)

```
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$ ./
Value: 0.00000000000000000000
(sign: 0, exp: 0, mantissa: 0x000000)
Value: 0.100000001490116119384 8
(sign: 0, exp: 123, mantissa: 0x4CCCCD)
Value: 0.200000002980232238769 5
(sign: 0, exp: 124, mantissa: 0x4CCCCD)
Value: 0.300000011920928955078 1
(sign: 0, exp: 125, mantissa: 0x19999A)
Value: 0.400000005960464477539 1
(sign: 0, exp: 125, mantissa: 0x4CCCCD)
Value: 0.50000000000000000000
(sign: 0, exp: 126, mantissa: 0x000000)
Value: 0.60000002384185791015 62
(sign: 0, exp: 126, mantissa: 0x19999A)
Value: 0.70000004768371582031 25
(sign: 0, exp: 126, mantissa: 0x333334)
Value: 0.80000007152557373046 88
(sign: 0, exp: 126, mantissa: 0x4CCCCE)
Value: 0.90000009536743164062 50
(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 11
(sign: 0, exp: 1019, mantissa: 0x999999999999A)
Value: 0.20000000000000001110 22
(sign: 0, exp: 1020, mantissa: 0x999999999999A)
Value: 0.30000000000000004440 89
(sign: 0, exp: 1021, mantissa: 0x3333333333334)
Value: 0.40000000000000002220 45
(sign: 0, exp: 1021, mantissa: 0x999999999999A)
Value: 0.50000000000000000000
(sign: 0, exp: 1022, mantissa: 0x0000000000000)
Value: 0.59999999999999997779 55
(sign: 0, exp: 1022, mantissa: 0x3333333333333)
Value: 0.69999999999999995559 11
(sign: 0, exp: 1022, mantissa: 0x6666666666666)
Value: 0.79999999999999993338 66
(sign: 0, exp: 1022, mantissa: 0x9999999999999)
Value: 0.89999999999999991118 22
(sign: 0, exp: 1022, mantissa: 0xCCCCCCCCCCCCC)
Value: 0.99999999999999988897 77
(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$
```

Lower-precision may **Increase Trouble**

- Values carry higher exponents
  - Resulting in higher rounding error

When values don't **fit** or **jive**, you generate FP Exceptions

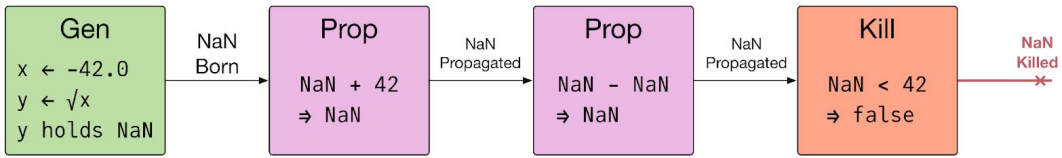
- Values may not fit the format
  - Generation of overflow - leading to +/- **INF**
- Values don't "jive"
  - **+INF + -INF → NaN**
  - **Sqrt(-1) → NaN**
  - **…**
- Calculations involving **INF** / **NaN** create more of them
  - The main "exceptions" (exceptional values) are INF and NaN
  - GPUs don't generate hardware traps for them
    - Thus, exceptions flow and pollute ensuing computations

FP exceptions : Get born, journey, get killed, meanwhile doing mischief

- FP32: (1.9×10^{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 bound to NaN and y bound to 42; answer is 42. Output "clean"
Then switch these assignments. ; answer is NaN. Max must be commutative

- Exceptions often vanish harmlessly (e.g. in ML codes)
  - But it is difficult to know which control-flows they have affected meanwhile



Why Binary Instrumentation (vs. other ways)?

- Source instrumentation
  - Does not reflect code that actually runs
- Intermediate code instrumentation (e.g., LLVM / PTX)
  - Cons: may not be exposed by the manufacturer
  - Lowerings are still not reflected
  - Many libraries are closed-source

NixNan is a binary instrumentation tool that helps track GPU FP exceptions

- We aim to help designers understand these GPU exceptions
  - how they are generated and propagated in GPU codes
  - which FP formats generate which exceptions, and in which binades
  - covers both SIMT cores and Tensor Cores

ML codes often defend against NaNs (and others…)

From https://github.com/Dao-AILab/flash-attention/blob/bc0e4ac01484ffb61ddc694724826bec4d9cf1c2/csrc/flash_attn/src/flash_fwd_kernel.h#L1195)

```
1194        lse_sum = Allreduce<kRowsPerLoadTranspose>::run(lse_sum, sum_op);
1195        // For the case where all local lse == -INFINITY, we want to set lse_logsum to INFINITY. Otherwise
1196        // lse_logsum is log(0.0) = -INFINITY and we get NaN when we do lse_accum(l) - lse_logsum.
1197        ElementAccum lse_logsum = (lse_sum == 0.f || lse_sum != lse_sum) ? INFINITY : logf(lse_sum) + lse_max;
```

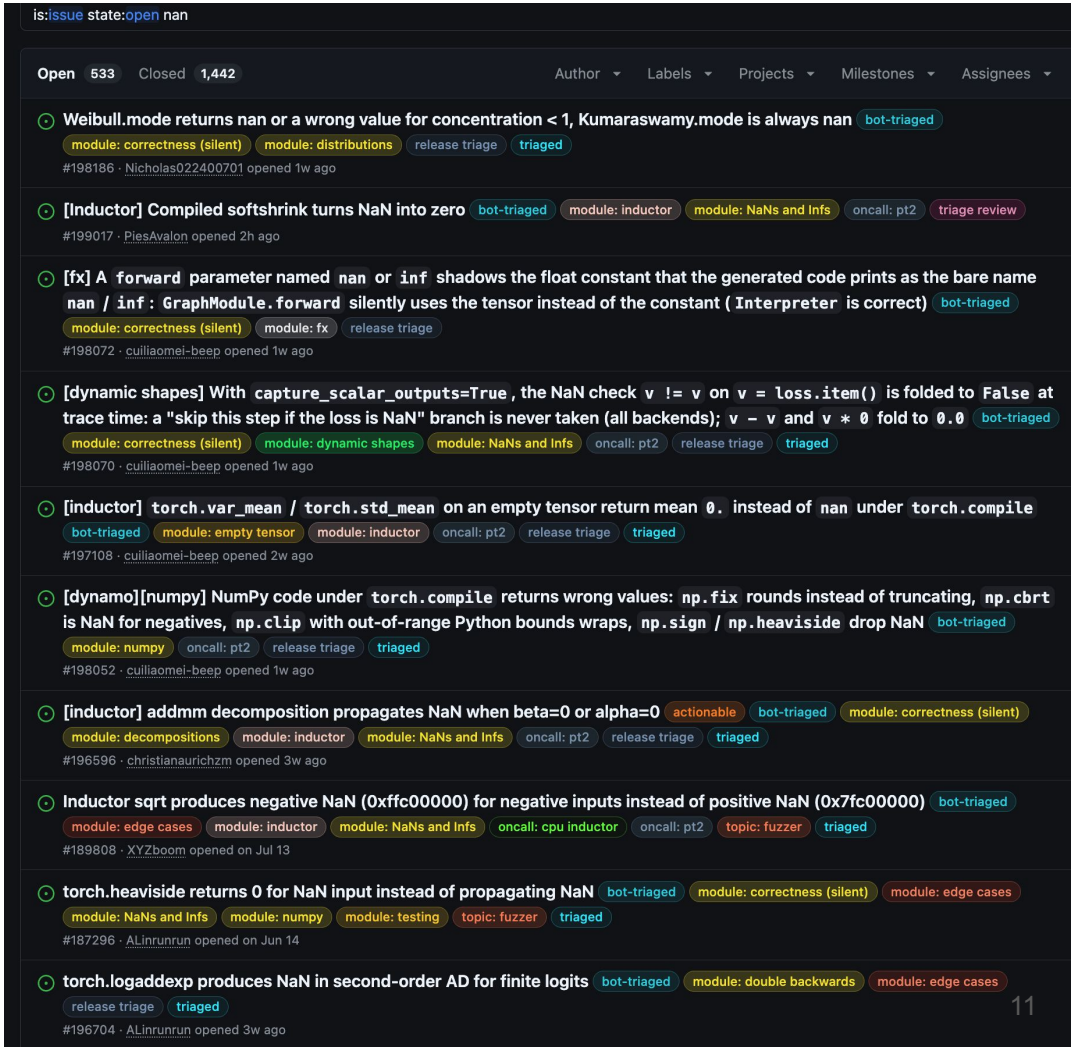


A NaN-Check

Exceptions can still leak…

PyTorch issues containing 'nan'

9/29/26 snapshot



HPC codes also often defend against exceptions

`KSPCheckNorm(ksp, beta)` is a safety macro used by PETSc's Krylov linear solvers (KSP).

It checks whether beta—typically a residual norm or another scalar quantity that should be finite—is NaN, +∞, or −∞:

```
// include/petsc/private/kspimpl.h, lines 604–622
#define KSPCheckNorm(ksp, beta) \
  do { \
    if (PetscIsInfOrNanReal(beta)) { \
      ...
      ksp->reason = KSP_DIVERGED_NANORINF;
      ksp->rnorm = beta;
      PetscFunctionReturn(PETSC_SUCCESS);
    } \
  } while (0)
```

While these checks are found in CPU codes of solvers, we would like to be armed with tools that can read closed-source GPU versions of such solvers, as well.

GPUs can't imitate CPU exception-checking methods

- CPUs use Trap-Handlers
  - When trap-handling is enabled, handle it in software
- **NVIDIA GPUs do not generate traps!**
  - Hence, we need to look for Exceptional Values!
- What about other GPUs?
  - AMD GPUs “sort of “ record exceptions in hardware (see Miao et.al., HPDC 2025)
    - With AMD GPUs, more effort is needed to find out
      - which threads generated exceptions
      - and at which source locations
  - Some GPUs do report NaNs
    - E.g., Trainium, Meta, … (anything using SIMD-like FP)

What is NixNan? : It is a tool for detecting FP exceptions in NVIDIA GPUs

Detects exceptions in
SIMT Cores as well as Tensor Cores



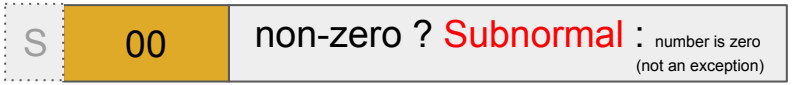
- NVIDIA GPUs do not generate any traps when an FP exception occurs
- Thus, we have to detect exceptions in software
  - By decoding the exponent and mantissa of the computed answer

NixNan works by decoding *Exceptional Values*

If Exp is FF, then decode Mantissa to discern INF vs. NaN



If Exp is 00, then decode Mantissa to discern Subnormal



A Demo of NixNan

- Clone https://github.com/parfloat/nixnan
  - and go to branch print_histo_instrn
- Go to test/first_test
- LD_PRELOAD= ../external/nixnan/nixnan.so ./hello_nixnan
- Look at hello_nixnan.cu and the results

```
#nixnan: error [NaN] detected in operand 2 of instruction DFMA R12, R12, R14, R12 ; in function hello of type f64
#nixnan: error [NaN] detected in operand 3 of instruction DFMA R12, R12, R14, R12 ; in function hello of type f64
#nixnan: error [NaN] detected in operand 0 of instruction DFMA R12, R12, R14, R12 ; in function hello of type f64
#nixnan: error [infinity] detected in operand 2 of instruction DMUL R14, R6, R4 ; in function hello of type f64
#nixnan: error [NaN] detected in operand 0 of instruction DMUL R14, R6, R4 ; in function hello of type f64
#nixnan: error [NaN] detected in operand 1 of instruction STG.E.64 [R10.64+0x8], R14 ; in function hello of type f64
#nixnan: error [div0] detected in operand 1 of instruction @!P0 MUFU.RCP R6, R0 ; in function hello of type f32
#nixnan: error [infinity] detected in operand 0 of instruction @!P0 MUFU.RCP R6, R0 ; in function hello of type f32
#nixnan: error [infinity] detected in operand 0 of instruction HMUL2 R0, R0.H0_H0, 65504, 65504 ; in function hello of type f16
#nixnan: error [infinity] detected in operand 2 of instruction FMUL R11, R8, R5 ; in function hello of type f32
#nixnan: error [NaN] detected in operand 0 of instruction FMUL R11, R8, R5 ; in function hello of type f32
#nixnan: error [NaN] detected in operand 1 of instruction STG.E [R6.64+0x4], R11 ; in function hello of type f32
#nixnan: error [infinity] detected in operand 1 of instruction HMUL2 R9, R0, R4.H0_H0 ; in function hello of type f16
#nixnan: error [NaN] detected in operand 0 of instruction HMUL2 R9, R0, R4.H0_H0 ; in function hello of type f16
#nixnan: error [subnormal] detected in operand 0 of instruction FMUL R13, R2, 1.17549435082228750 8e-38 ; in function hello of t
ype f32
#nixnan: error [NaN] detected in operand 1 of instruction STG.E [R2.64+0x4], R9 ; in function hello of type f16
#nixnan: error [infinity] detected in operand 1 of instruction HFMA2.BF16_V2 R7, R5.H0_H0, 3.0040552704739098727 6e+38, 3.004055
27047390987276e+38, -RZ.H0_H0 ; in function hello of type bf16
#nixnan: error [infinity] detected in operand 0 of instruction HFMA2.BF16_V2 R7, R5.H0_H0, 3.0040552704739098727 6e+38, 3.004055
27047390987276e+38, -RZ.H0_H0 ; in function hello of type bf16
#nixnan: error [infinity] detected in operand 1 of instruction HFMA2.BF16_V2 R15, R7, R8.H0_H0, -RZ.H0_H0 ; in function hello o
f type bf16
#nixnan: error [NaN] detected in operand 0 of instruction HFMA2.BF16_V2 R15, R7, R8.H0_H0, -RZ.H0_H0 ; in function hello of typ
e bf16
#nixnan: error [NaN] detected in operand 1 of instruction STG.E [R4.64+0x4], R15 ; in function hello of type bf16
#nixnan: running kernel [tc] ...
#nixnan: error [infinity] detected in operand 1 of instruction HMMA.16816.F32 R16, R8.reuse, R12, RZ ; in function tc of type f
16
#nixnan: error [NaN,infinity] detected in operand 0 of instruction HMMA.16816.F32 R16, R8.reuse, R12, RZ ; in function tc of ty
pe f32
#nixnan: error [infinity] detected in operand 1 of instruction HMMA.16816.F32 R12, R8, R14, RZ ; in function tc of type f16
#nixnan: error [NaN] detected in operand 0 of instruction HMMA.16816.F32 R12, R8, R14, RZ ; in function tc of type f32
#nixnan: error [NaN] detected in operand 1 of instruction STG.E.64 [R8.64], R16 ; in function tc of type f32
#nixnan: error [NaN] detected in operand 1 of instruction STG.E.64 [R8.64+0x20], R12 ; in function tc of type f32
hello_nixnan done: d=inf,-nan f=inf,nan,5.87747e-39 D[0]=inf
#nixnan: Finalizing GPU context...
```

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

#nixnan: --- FP16 Operations ---
#nixnan: NaN:                    1 (0 repeats)
#nixnan: Infinity:               4 (0 repeats)
#nixnan: -Infinity:              0 (0 repeats)
#nixnan: Subnormal:              0 (0 repeats)
#nixnan: Division by 0:          0 (0 repeats)

#nixnan: --- BF16 Operations ---
#nixnan: NaN:                    1 (0 repeats)
#nixnan: Infinity:               3 (0 repeats)
#nixnan: -Infinity:              0 (0 repeats)
#nixnan: Subnormal:              0 (0 repeats)
#nixnan: Division by 0:          0 (0 repeats)

#nixnan: --- FP32 Operations ---
#nixnan: NaN:                    3 (0 repeats)
#nixnan: Infinity:               3 (0 repeats)
#nixnan: -Infinity:              0 (0 repeats)
#nixnan: Subnormal:              1 (0 repeats)
#nixnan: Division by 0:          1 (1 repeats)

#nixnan: --- FP64 Operations ---
#nixnan: NaN:                   17 (0 repeats)
#nixnan: Infinity:               2 (0 repeats)
#nixnan: -Infinity:              0 (0 repeats)
#nixnan: Subnormal:              0 (0 repeats)
#nixnan: Division by 0:          1 (1 repeats)

#nixnan: --- FP16 Memory  Operations ---
#nixnan: NaN:                    1 (0 repeats)
#nixnan: --- BF16 Memory  Operations ---
#nixnan: NaN:                    1 (0 repeats)
#nixnan: --- FP32 Memory  Operations ---
#nixnan: NaN:                    3 (4 repeats)
#nixnan: --- FP64 Memory  Operations ---
#nixnan: NaN:                    1 (0 repeats)
```

History of NixNan

- NixNan is an **NVBit** tool
  - NVBit is the dynamic binary instrumenter released by NVIDIA
- It helps locate
  - Exceptions at different precision
    - Generated by SIMT and Tensor Cores
  - It provides
    - A reasonable amount of debugging info
    - Indication of exceptional values being written into memory
- More tool history
  - It is the latest in a series of tools developed by these researchers
    - Dr. Mark Baranowski - NixNan's creator
    - Dr. Xinyi Li - created the precursor to NixNan, namely GPU-FPX
    - Dr. Ignacio Laguna - created BinFPe and advised us on all our tools
    - Prof. Ganesh Gopalakrishnan - supervision, (often) undeserved attention

Thanks, NVIDIA, for giving us a **solid base**!



Need this for all basic functionality

Courtesy of NVLabs (NVIDIA)

* Basics of FP instruction instrumentation (which registers are used, etc)

* Reporting FP exceptions generated by SASS instructions

* Detect exceptions

* Then Analyze to determine flow of exceptions

* Sampling

* Black/White listing

* Fixes and enhancements

* Gather all exceptions from various threads

* Support 16-bit formats

* Support Tensor Cores

* Histogramming FP values produced

* Various shell-scripts that drive the core NixNan infra with flag combos that extract info gradually

- plain autonixnan Determine kernels, runtimes, and extreme-value tendencies; runs top prospects

- iter / ksel-autonixnan : run (selected) kernels one by one under instrumentation

Two example uses - one in HPC and one in ML

HPC Linear Solver on Good and Bad Matrices

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

#nixnan: --- FP16 Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: Infinity:                 0 (0 repeats)
#nixnan: -Infinity:                0 (0 repeats)
#nixnan: Subnormal:                0 (0 repeats)
#nixnan: Division by 0:            0 (0 repeats)

#nixnan: --- BF16 Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: Infinity:                 0 (0 repeats)
#nixnan: -Infinity:                0 (0 repeats)
#nixnan: Subnormal:                0 (0 repeats)
#nixnan: Division by 0:            0 (0 repeats)

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

#nixnan: --- FP64 Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: Infinity:                 0 (0 repeats)
#nixnan: -Infinity:                0 (0 repeats)
#nixnan: Subnormal:                0 (0 repeats)
#nixnan: Division by 0:            0 (0 repeats)

#nixnan: --- FP16 Memory  Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: --- BF16 Memory  Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: --- FP32 Memory  Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: --- FP64 Memory  Operations ---
#nixnan: NaN:                      0 (0 repeats)
```

```
--- Solution X Vector (10)
1.12338
0.799707
7.82785
-1954.76
-0.000778511
1.07278
-0.535868
-3.61902
8.83203
0.585185
```

Good

In lu_solve_gpu/ do this:

```
LD_PRELOAD=../external/nixnan/nixnan.so
./lu_solve_gpu good_matrix.csv
```

```
LD_PRELOAD=../external/nixnan/nixnan.so
./lu_solve_gpu bad_matrix.csv
```

```
--- Solution X Vector (10)
1.4665
inf
2.15945
0.895458
10.9171
1.11917
inf
24.0914
0.421027
-1.39342
```

Bad

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

#nixnan: --- FP16 Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: Infinity:                 0 (0 repeats)
#nixnan: -Infinity:                0 (0 repeats)
#nixnan: Subnormal:                0 (0 repeats)
#nixnan: Division by 0:            0 (0 repeats)

#nixnan: --- BF16 Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: Infinity:                 0 (0 repeats)
#nixnan: -Infinity:                0 (0 repeats)
#nixnan: Subnormal:                0 (0 repeats)
#nixnan: Division by 0:            0 (0 repeats)

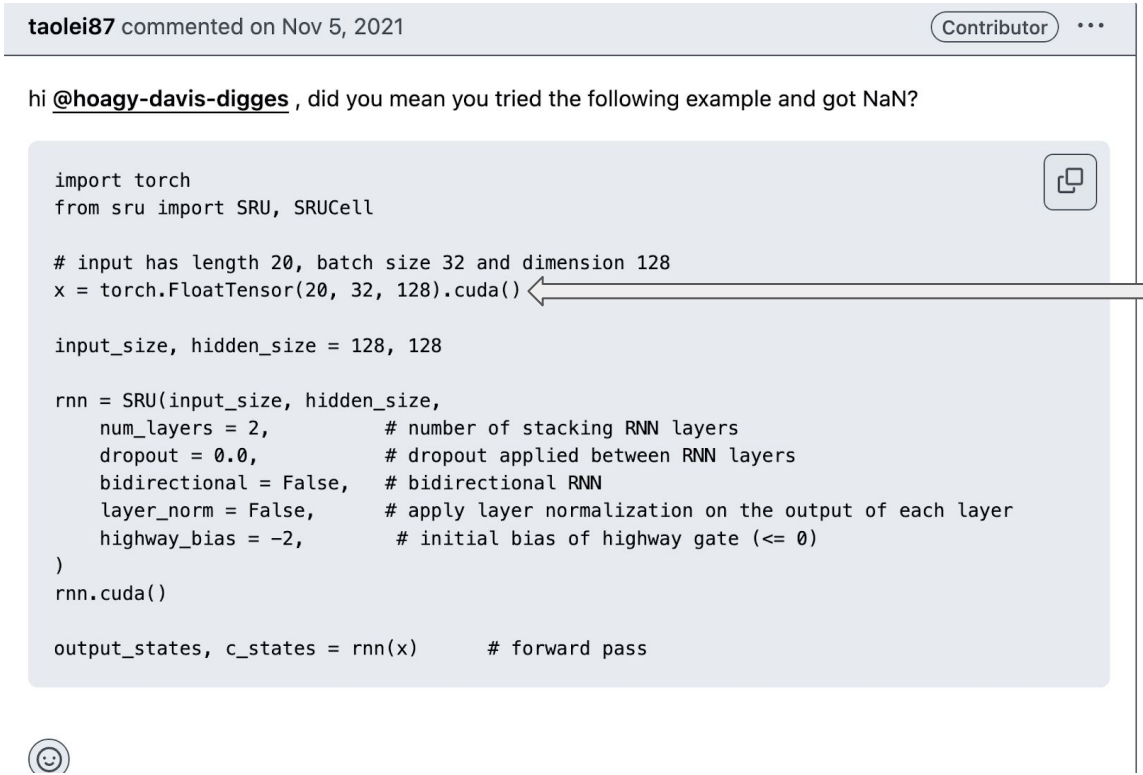
#nixnan: --- FP32 Operations ---
#nixnan: NaN:                      4 (0 repeats)
#nixnan: Infinity:                 0 (0 repeats)
#nixnan: -Infinity:                0 (0 repeats)
#nixnan: Subnormal:                0 (0 repeats)
#nixnan: Division by 0:            0 (0 repeats)

#nixnan: --- FP64 Operations ---
#nixnan: NaN:                     48 (0 repeats)
#nixnan: Infinity:                 2 (0 repeats)
#nixnan: -Infinity:                0 (0 repeats)
#nixnan: Subnormal:                0 (0 repeats)
#nixnan: Division by 0:            2 (2 repeats)

#nixnan: --- FP16 Memory  Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: --- BF16 Memory  Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: --- FP32 Memory  Operations ---
#nixnan: NaN:                      0 (0 repeats)
#nixnan: --- FP64 Memory  Operations ---
#nixnan: NaN:                      0 (0 repeats)
```

NixNan helped understand a real PyTorch issue wrt "SRU"

- Link: https://github.com/asappresearch/sru/issues/193



taolei87 commented on Nov 5, 2021 — Contributor

hi **@hoagy-davis-digges** , did you mean you tried the following example and got NaN?

```
import torch
from sru import SRU, SRUCell

# input has length 20, batch size 32 and dimension 128
x = torch.FloatTensor(20, 32, 128).cuda()

input_size, hidden_size = 128, 128

rnn = SRU(input_size, hidden_size,
    num_layers = 2,          # number of stacking RNN layers
    dropout = 0.0,           # dropout applied between RNN layers
    bidirectional = False,   # bidirectional RNN
    layer_norm = False,      # apply layer normalization on the output of each layer
    highway_bias = -2,        # initial bias of highway gate (<= 0)
)
rnn.cuda()

output_states, c_states = rnn(x)      # forward pass
```

Hmm…

This issue was root-caused manually by observing NixNan Traces

Exception was present very early

Reason: User forgot to initialize input-tensor!

In SC26-Tutorial/sru_study LD_PRELOAD=../external/nixnan/nixnan.so python reproducer.py (also try the other reproducers)

Exceptions written into memory : shows that the user is bound to see a noticeable behavioral impact

Before Fixing SRU

```
#nixnan: --- FP16 Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: Infinity:                  0 (0 repeats)
#nixnan: -Infinity:                 0 (0 repeats)
#nixnan: Subnormal:                 0 (0 repeats)
#nixnan: Division by 0:             0 (0 repeats)

#nixnan: --- BF16 Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: Infinity:                  0 (0 repeats)
#nixnan: -Infinity:                 0 (0 repeats)
#nixnan: Subnormal:                 0 (0 repeats)
#nixnan: Division by 0:             0 (0 repeats)

#nixnan: --- FP32 Operations ---
#nixnan: NaN:                     843 (2418829 repeats)
#nixnan: Infinity:                 13 (15635 repeats)
#nixnan: -Infinity:                 0 (0 repeats)
#nixnan: Subnormal:               746 (251293 repeats)
#nixnan: Division by 0:             0 (0 repeats)

#nixnan: --- FP64 Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: Infinity:                  0 (0 repeats)
#nixnan: -Infinity:                 0 (0 repeats)
#nixnan: Subnormal:                 0 (0 repeats)
#nixnan: Division by 0:             0 (0 repeats)

#nixnan: --- FP16 Memory  Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: --- BF16 Memory  Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: --- FP32 Memory  Operations ---
#nixnan: NaN:                       3 (5373 repeats)
#nixnan: --- FP64 Memory  Operations ---
#nixnan: NaN:                       0 (0 repeats)
```

Our fix was applied

After Fixing (by initializing tensor)

```
#nixnan: --- FP16 Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: Infinity:                  0 (0 repeats)
#nixnan: -Infinity:                 0 (0 repeats)
#nixnan: Subnormal:                 0 (0 repeats)
#nixnan: Division by 0:             0 (0 repeats)

#nixnan: --- BF16 Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: Infinity:                  0 (0 repeats)
#nixnan: -Infinity:                 0 (0 repeats)
#nixnan: Subnormal:                 0 (0 repeats)
#nixnan: Division by 0:             0 (0 repeats)

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

#nixnan: --- FP64 Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: Infinity:                  0 (0 repeats)
#nixnan: -Infinity:                 0 (0 repeats)
#nixnan: Subnormal:                 0 (0 repeats)
#nixnan: Division by 0:             0 (0 repeats)

#nixnan: --- FP16 Memory  Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: --- BF16 Memory  Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: --- FP32 Memory  Operations ---
#nixnan: NaN:                       0 (0 repeats)
#nixnan: --- FP64 Memory  Operations ---
#nixnan: NaN:                       0 (0 repeats)
```

Why care about NixNan's internals?

- Explains its (often) "glacial" speed :)
- If/when NixNan breaks, you are empowered to fix (or report)
  - Reason: The process of creating NixNan is not "perfect"
    - Plenty of "reverse-engineering"
    - When the tool deadlocked, we found work-arounds…
  - Lessons
    - GPU correctness tool creation isn't straightforward
      - Manufacturers only say so much
    - Do Try and Fail using NixNan on YOUR code!
      - So, we can learn and improve it !!
    - More NVBit-tool interest may help NVIDIA put more help behind it

Please read Villa et al, Micro 2019 !! (to understand NixNan)

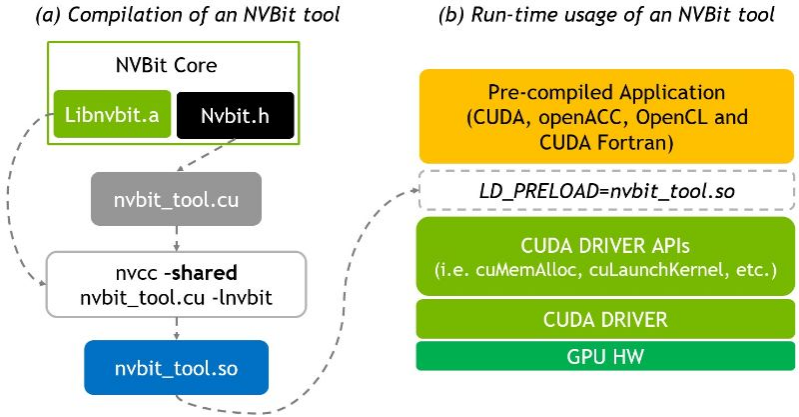


Figure 2: NVBit framework overview. An NVBit tool is first compiled as a shared library and then injected (preloaded) at run-time using *LD_PRELOAD*.

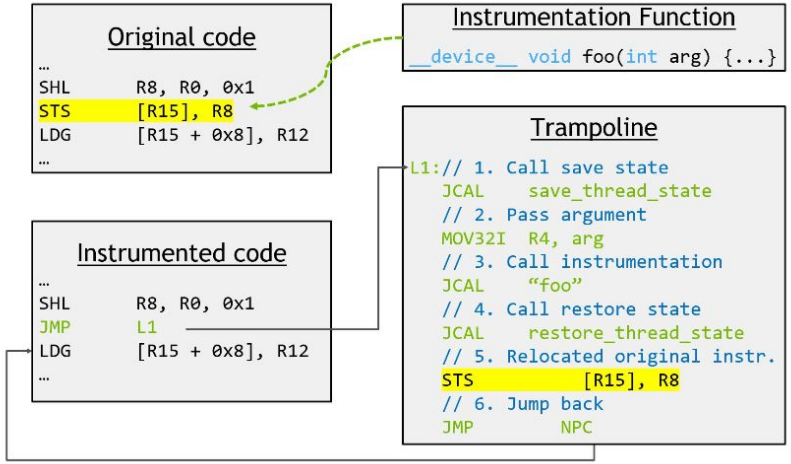


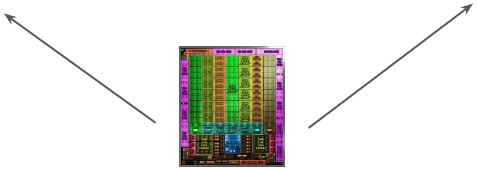
Figure 4: Instrumented code generation process.

Getting more inside **NixNan ...**

Detects exceptions in

SIMT Cores as well as Tensor Cores





> LD_PRELOAD = nixnan.so
> Pre-Compiled Application Binary

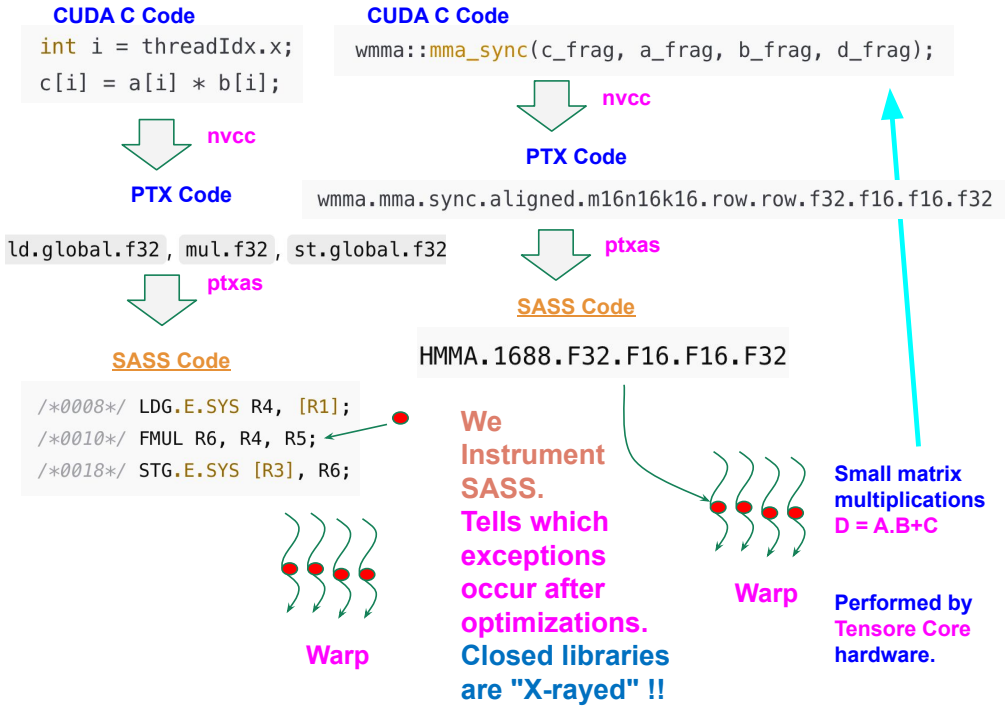
Intercept all "Kernel Launch" CUDA Driver API Calls

For Each FP instruction :

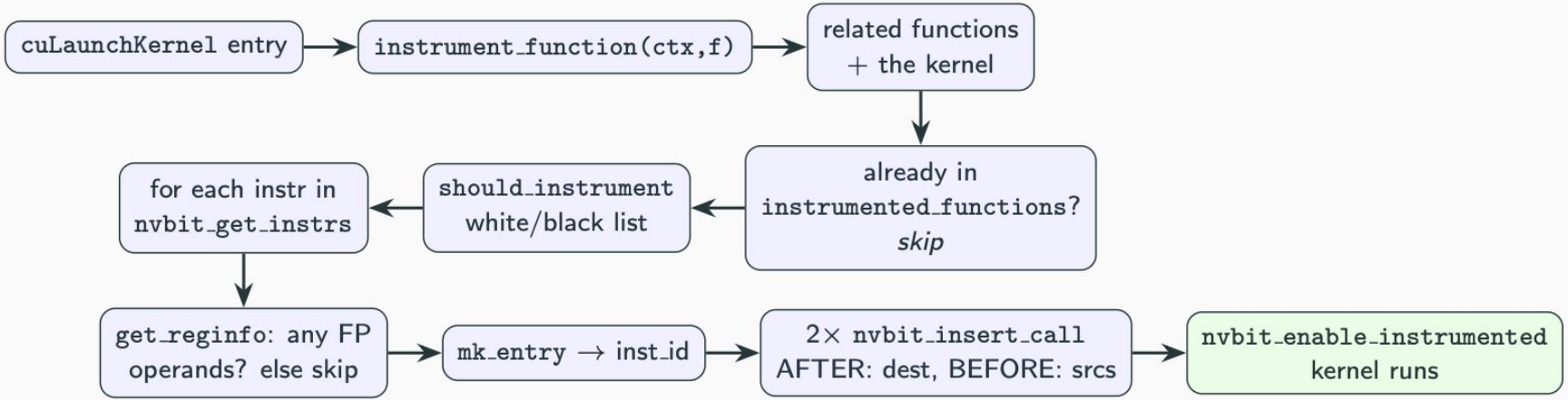
Inject_instrumentation();

CUDA Driver APIs

The CUDA ⇒ PTX ⇒ SASS pipeline



From trapping a kernel … to its instrumented run



`mk_entry` → `inst_id`
Host table, one row per instrumented instruction: SASS text, file:line, function, operand types. `inst_id` = row number (0, 1, 2, . . .), passed to the GPU as the instruction's only identity; the host looks the row up when printing.

2× call, HMMA example
`HMMA.16816.F16 R22, R16, R22, R8`
BEFORE reads in: A R16–R19, B R22–R23, C R8–R9
AFTER reads out: D R22–R23
(R22 is both B and D)

`nvbit_enable_instrumented`
NVBit keeps two copies of each function: the *original* and the *instrumented* one (with the injected calls). Each launch selects which runs: `true` → instrumented, `false` → original.

Nixnan Instruction Record : use varargs

- This requires a new approach
  - Instructions can operate on different types
  - Operands can point to arrays of registers
    - A register, R1, with 4 fp32 values is actually R1, R2, R3 and R4
  - Registers are 32 bits
    - A register with fp16 has two values in it
- Use varargs in the inspection function to allow sending variable amounts of data
- Documents the format of the instruction
  - SASS is minimally documented by Nvidia



Database of Instructions and Errors

- Add a database of instructions that describes what errors to look for in the registers of the instruction

```
bool is_FP32_instruction(std::string inst) {
  if (inst.find("FADD") != std::string::npos ||
      inst.find("FADD32I") != std::string::npos ||
      inst.find("FFMA32I") != std::string::npos ||
      inst.find("FFMA") != std::string::npos ||
      inst.find("FMUL") != std::string::npos ||
      inst.find("MUFU") != std::string::npos ||
      inst.find("FMNMX") != std::string::npos ||
      inst.find("FSEL") != std::string::npos ||
      inst.find("FSET") != std::string::npos ||
      inst.find("FCHK") != std::string::npos ||
      inst.find("FSETP") != std::string::npos) {
    if (inst.find("64H") == std::string::npos) {
      return true;
    }
  }
  return false;
}
```

```
"HMMA.16816.F32":
  {"registers":
    [
      {
        "type": "f32",
        "count": 4
      },
      {
        "type": "f16",
        "count": 4
      },
      {
        "type": "f16",
        "count": 4
      },
      {
        "type": "f16",
        "count": 8
      }
    ],
    "description": "Half-precision matrix multiply accumulate into single-precision accumulator"
  },
```

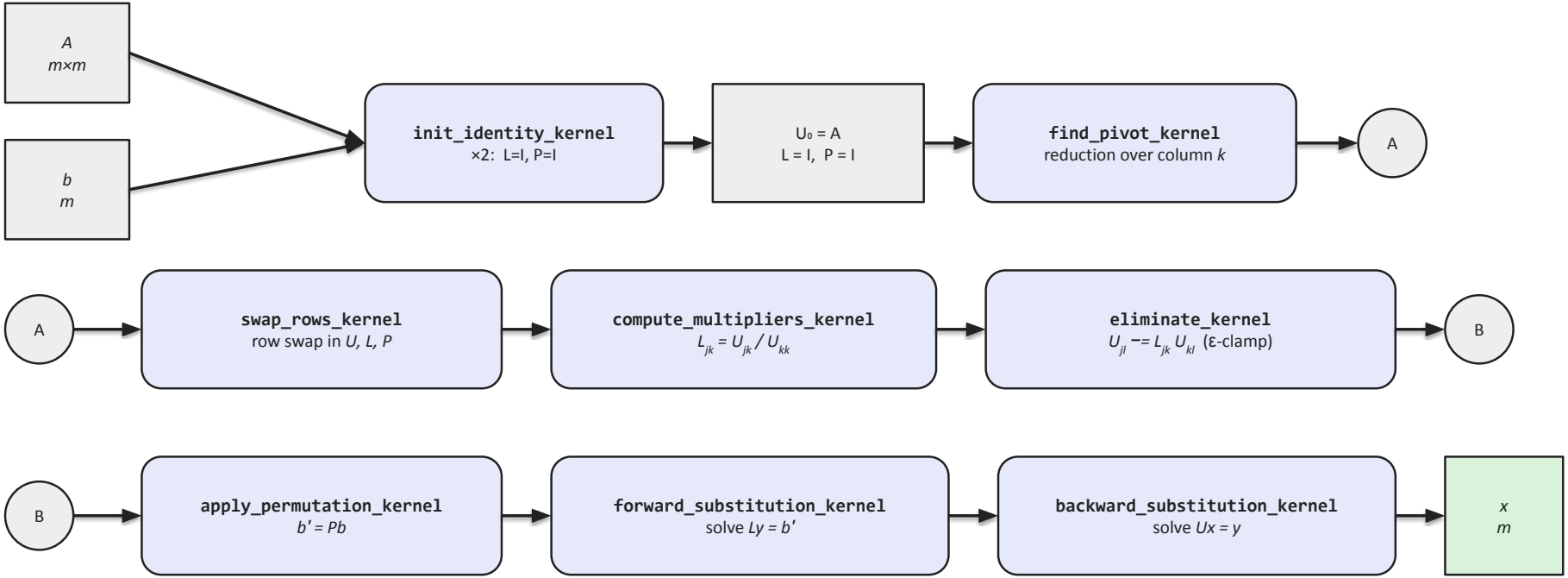
Experience running an LU-Solver under NixNan

- SIMT version first
- Tensor-Core Version has Panel Factorization

We will skip many slides, stopping to discuss only salient details!

Algorithm: LU with partial pivoting, solve $Ax = b$

$PA = LU$ pivot: $p = argmax_{i \geq k} |U_{ik}|$ multiplier: $L_{jk} = U_{jk} / U_{kk}$
eliminate: $U_{jl} \mathrel{-}= L_{jk} U_{kl}$ $(j>k,\ l \geq k)$ forward: ***Ly = Pb*** backward: ***Ux = y***



NixNan Knobs: Environment Variables

INSTR_BEGIN / END	instr. index window
TOOL_VERBOSE	setup diagnostics
ENABLE_FUN_DETAIL	name on every rerun
PRINT_ILL_INSTR	print culprit SASS
SAMPLING	every N-th launch
INSTRUMENT_EXCEPTIONS	master switch
INSTR_MEM	also check STG (excn → mem)
HISTOGRAM	exponent histogram
BIN_SPEC_FILE	ranges to watch

NIXNAN_TIMEOUT	self-terminate, secs
LOGFILE	redirect output
LINE_INFO	source file:line (YMMV)
LOG_KERNELS	log launches only
MAX_ERRORS	stop after N reports
TIME_KERNELS	per-kernel time
FUNCTION_WHITELIST	only these kernels
FUNCTION_BLACKLIST	all but these

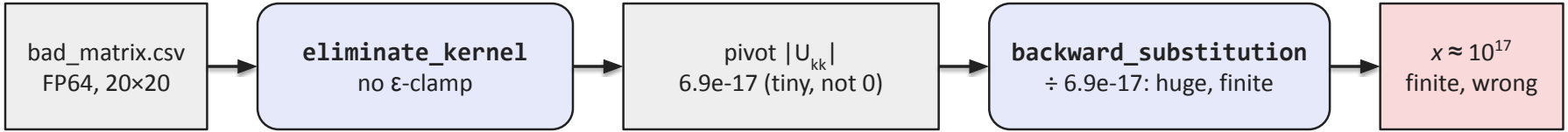
For example, ksel-autonixnan version of NixNan scripts (using the above env vars)
the collection of kernel names and execution times
so users can then white- or black-list exactly what to observe in detail.
See https://github.com/parfloat/nixnan/blob/main/Tutorial.md for details

LU Stats: TIME_KERNELS: the first-launch cost

Kernel	Launches	First launch (µs)	Median later (µs)	Ratio
`init_identity_kernel`	2	312,451	21	≈ 14,900×
`find_pivot_kernel`	19	298,100	20	≈ 14,900×
`swap_rows_kernel`	19	290,205	25	≈ 11,600×
`compute_multipliers_kernel`	19	295,432	20	≈ 14,800×
`eliminate_kernel`	19	278,808	20	≈ 13,900×
`apply_permutation_kernel`	1	300,320	—	—
`forward_substitution_kernel`	1	326,736	—	—
`backward_substitution_kernel`	1	324,597	—	—

- First launch of every kernel: ≈280–330 ms. Later launches: ≈20–25 µs, four to five orders of magnitude less.
- Last three rows involve kernels that were launched only once!
- Library-heavy binaries pay far more:
  the cuSOLVER-based pytorch_slogdet run was killed at 120 s during instrumentation.

LU w/o clamping: If small values are not clamped to 0.0 → a segfault



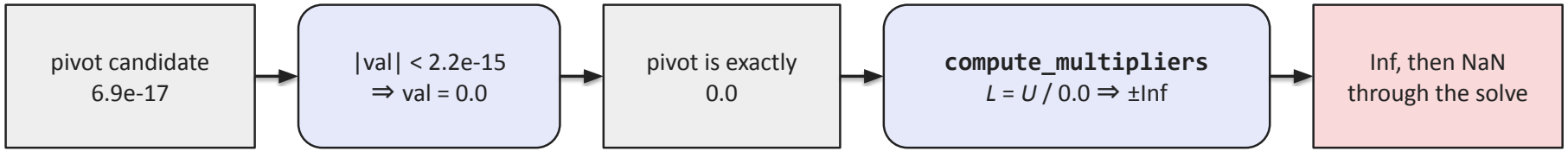
	matrix.csv	bad_matrix.csv
Smallest pivot $\lvert U_{kk}\rvert$ (read back from the GPU)	1.445	6.9e-17
Residual $\lVert Ax - b\rVert_2$	9.7e-16	33.9
Solution x	≈ 1, finite	up to 1.7e17, finite
nixnan exceptions (all precisions, memory)	none	none

All details nicely revealed by running ./run_clamp_story.sh inside the lu_solve_gpu/ directory

Clamping Details: Tiny values set to zero

```
val = U[j][l] - L[j][k] * U[k][l];
if (fabs(val) < 10.0 * DBL_EPSILON)   // 2.2e-15
    val = 0.0;                        // set to zero if close to machine epsilon
U[j][l] = val;
```

in eliminate_kernel (lu_solve_gpu.cu, lines 210–211)



Intent: manifest div-by-small by turning it into div-by-0.

All details would become clear if you ran **./run_nixnan_experiment.sh** inside **lu_solve_gpu/**

Stage 1 results: well-conditioned vs ill-conditioned input

	matrix.csv	bad_matrix.csv
Residual $\|\|Ax - b\|\|_2$ (uninstrumented)	9.68e-16	nan
Binade-threshold reports	99	63
FP64 NaN (repeats)	0 (0)	316 (2171)
FP64 Infinity (repeats)	0 (0)	4 (0)
FP64 Division-by-zero (repeats)	0 (0)	4 (4)

matrix.csv
clean solution, no exceptions

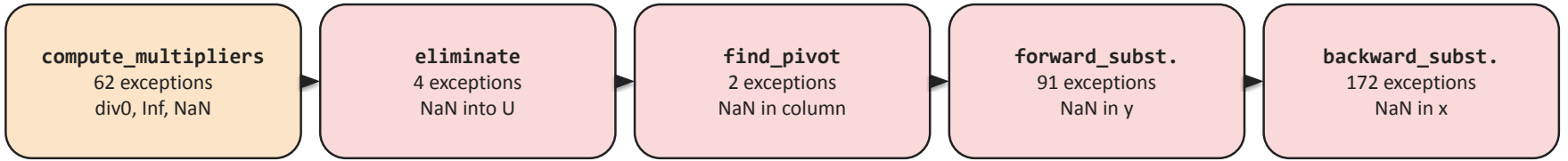
bad_matrix.csv
near-zero pivot → division by zero →
NaN through the whole solve

Useful Env Var Settings

PRINT_ILL_INSTR=1 ENABLE_FUN_DETAIL=1 (FP64, clamp on, bad_matrix.csv); the first report:

```
error [div0] detected in operand 1 of instruction MUFU.RCP64H R3, R5
in function compute_multipliers_kernel at lu_solve_gpu_clamp_study.cu:216  of type f64
```

Line 216 is the division $L_{jk} = U_{jk} / U_{kk}$: the reciprocal of the zeroed pivot is the first thing that goes wrong.



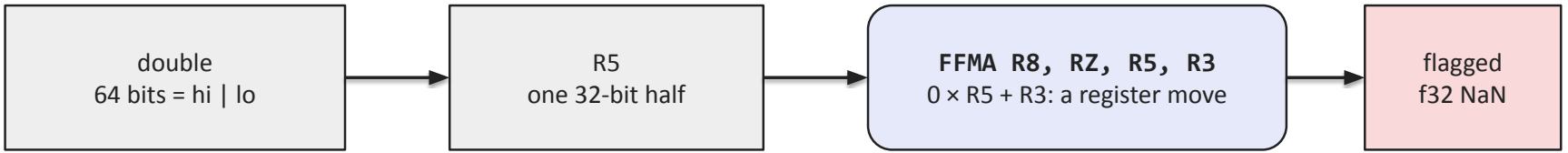
- The first report names the kernel, the instruction and the source line; the rest is propagation.
- The cause is upstream of it: the clamp wrote 0.0 into the pivot. NixNan shows where the zero is divided by, not why it is zero.
- Counts are per kernel, from 331 detail lines (run run_clamp_story.sh step2).

Eh? I saw an FP32-exception when I ran an FP64-code – what happened?

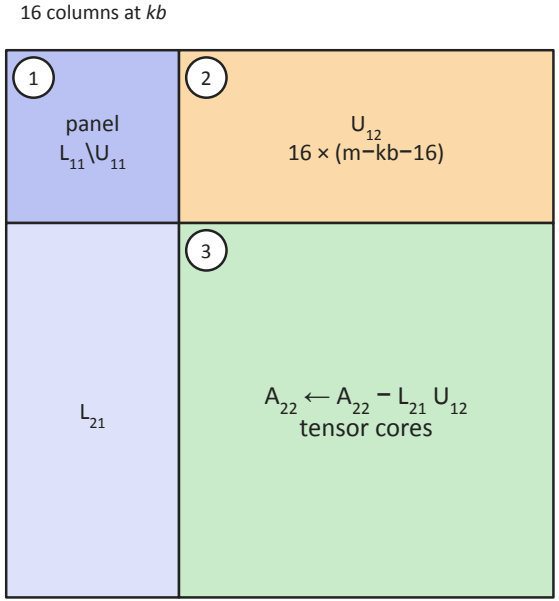
- The source computes only in double, yet bad_matrix.csv also reports 6 FP32 NaN detections (93 repeats).
- Cause: ptxas uses the idiom FFMA R8, RZ, R5, R3 (0 × R5 + R3 = R3) as a register move,

  shuffling the 32-bit halves of a double.

  NixNan sees an f32 FFMA whose operand R5 decodes as NaN.
- Observation: a SASS-level detector reports what the compiled code does (hence, may be confusing).

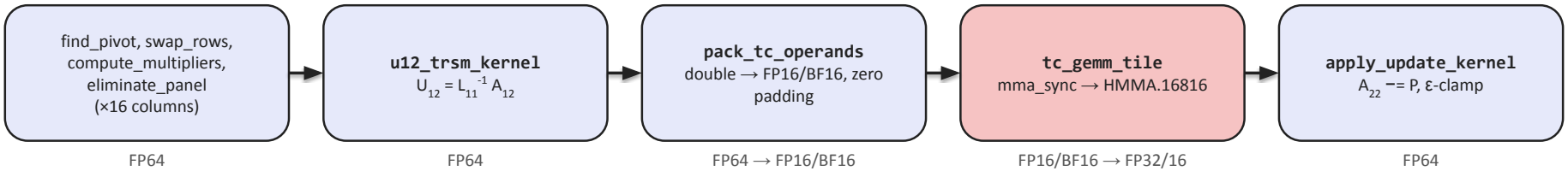


LU Adaptation to Tensor Cores Involves Panel Factorization



1. panel, FP64: pivot search, row swap, multipliers, elimination on the 16 panel columns
2. $U_{12} = L_{11}^{-1} A_{12}$, FP64 triangular solve
3. $A_{22} \mathrel{-}= L_{21}\, U_{12}$: round to FP16/BF16, WMMA on tensor cores, subtract in FP64

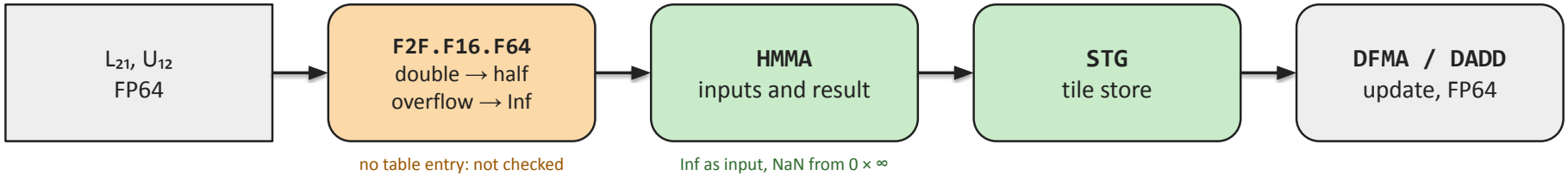
Repeat for kb = 0, 16, 32, …; a last block narrower than 16 columns is panel-only.



All details nicely revealed by running lu_solve_gpu_tc inside the lu_solve_gpu/ directory

More surprises : we didn't instrument it all … (can't !!)

(we don't know what all SASS instructions actually exist :-) – nowhere written down!)



Not all instructions are documented upfront, so we had to discover, ad hoc and whenever possible, which ones NixNan actually checks. Thus, we missed instrumenting F2F.F16.F64

Input	fp16acc32	fp16acc16	bf16acc32	FP64 original
matrix.csv residual	1.1e-4	2.0e-4	4.4e-4	9.7e-16
bad_matrix.csv	NaN	NaN	NaN	NaN
rand:64:100000	NaN	NaN	0.17	–

Who needs a tool similar to NixNan?

- All users must have FP familiarity
- For HPC folks
  - You must have seen one instance where a solver generates NaN/INF-laden solutions
  - You are **tracking down the consequences of lowering precision**
- For ML folks
  - You have experienced a NaN-issue
  - You are developing something new

With this as a starting point, what can one expect of a tool such as NixNan?

PyTorch helps diagnose … but can't rely always (believable ChatGPT summary below)

2 What PyTorch does automatically

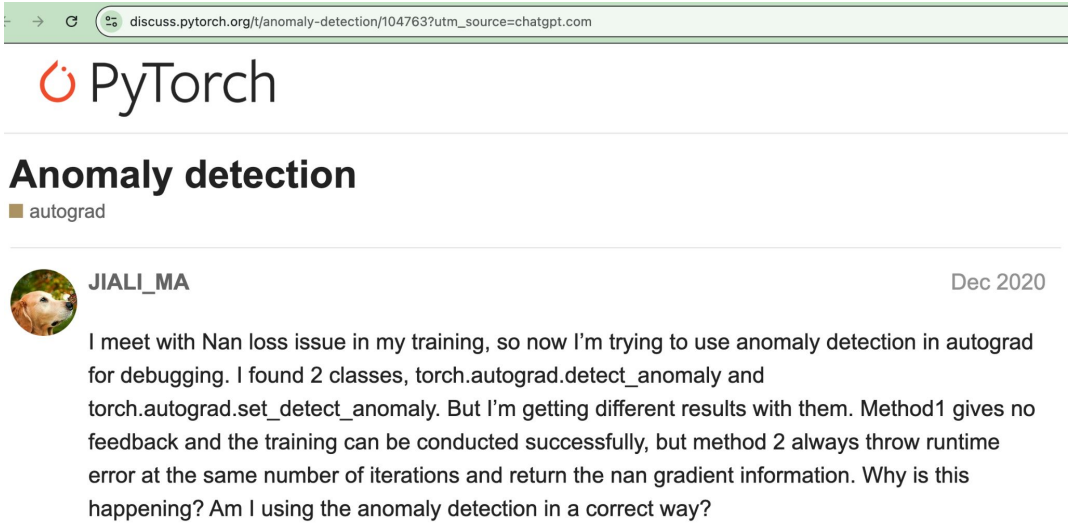
Layer	What happens internally	Key API / behaviour
Forward kernels	Produces INF/NaN normally (IEEE "masked exceptions")—no trap.	`torch.isfinite`, `tensor.isnan()` etc. for explicit checks.
autograd anomaly detection	Wrap graph execution; throws when the *first* backward op creates a non-finite grad and prints the forward traceback.	`torch.autograd.detect_anomaly()` or `set_detect_anomaly(check_nan=True)` PyTorch Forums PyTorch
Mixed-precision scaler	`GradScaler` un-scales grads, then runs a fused `isfinite` reduction **on-GPU**. If any grad is non-finite the *optimizer step is skipped* and the loss-scale is halved.	`scaler.step(optim)` / `scaler.update()` ; see AMP docs PyTorch PyTorch Forums
High-level trainers	Lightning, DeepSpeed, Accelerate all keep a "gradient overflow" flag; a skipped step just logs **OVERFLOW** and carries on.	DeepSpeed fp16 messages "Overflow detected. Skipping step." GitHub
Linalg ops	PyTorch's `torch.linalg` wrappers pass inputs to cuSOLVER/CPU LAPACK **without extra guards**; docs explicitly warn to pre-check with `torch.isfinite`. PyTorch	



PyTorch Issue 160016 tells that this can be "porous"

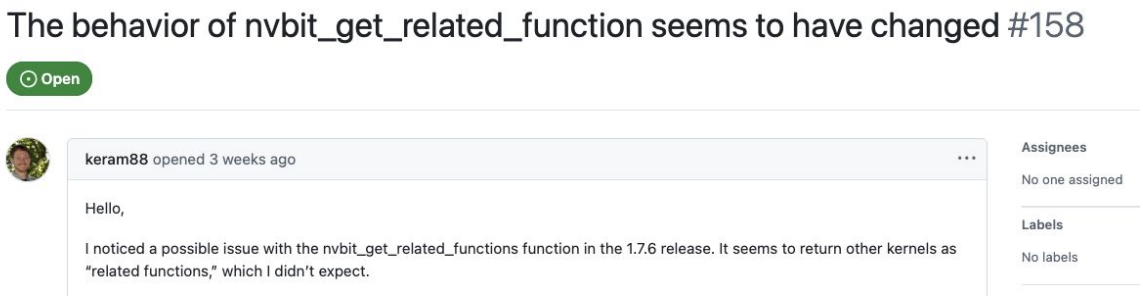
Existing framework solutions may be "best-effort"

- Experts have told me how they deal with show-stopper exceptions
  - Try a bag of tricks (similar to those present in PyTorch - listed shortly)
- Thousands of reports that go like this



NVIDIA, … please help the community do better !!

- We are dependent on NVIDIA's binary instrumentation framework
  - NVBit
  - Our work "merely" extends NVBit
- NVBit issues keep popping up
  - We are at present waiting for a fix (w/o which we can't use their latest release)
    - earlier releases have issues
- Without NVBit, we have (and the community has) **no tools whatsoever**
  - **Other issues: which instructions does nvcc generate?**
    - **and have we covered it?**



Exceptions in ML : no shortage!

SRU

https://github.com/asappresearch/sru/issues/193

- Throws NaN
- User unable to diagnose

  We traced it to Misunderstood Pytorch allocation ; does not clear Memory as the user thought

Julia Bug

https://discourse.julialang.org/t/neuralpde-on-gpu-throws-nans-when-i-use-a-source-term-elevated-to-some-power/114048

- User has NaN issue during training in Julia that launches to GPUs; claims they have a fix

  We found the NaN still coming, (albeit much later)

BatchNorm1d

https://github.com/pytorch/pytorch/issues/162489

- User posted Sep’25
- Finds CPU / GPU differences wrt NaNs in BatchNorm1d

  GPT finds a fix when fed our Tool’s traces.

  "need more precision"

  This is a "newbie bug"

Some NixNan "successes" (issues we reproduced and shone more light on)

NixNan is unique in that it can shed more light

At present, certain NVBit issues are making NixNan less effective

Original developers may have benefited a lot more, than us!

Selected Examples Tried	Type of Issue	Conclusions on Issue - following NixNan run
Karpathy's simple LLM	NaN during training	Does not affect training
Lossy compressor PyBlaz	NaN during compression	Does not affect operation
Issue 125674	Nan in grad. of scaled dotprod attention	Can produce traces that may help customer
Issue 156020	Value too high	Trace suggests NaN written into memory
Issue 156707	Issue with MPS	Tracing on GPUs possible
Issue 152737	NaN prop in Conv1d	Two NaN Stores into memory observed
Issue 157272	Reciprocal NaN issue	Bug in CPU; ran GPU code for comparison (traces)
Issue 159333	Foreach.copy bug	NaN store into memory present. Traces informative?
Issue 68425	Can run min instance	NaN due to predicated execution known
Issue 160016	PyTorch's own anomaly detection can be "porous" (does not report in the cases shown here)	Reproduces issues pointed out
BioMistral	None; runs	Lots of NaN/INF observed; harmless?

Typical uses, Realities

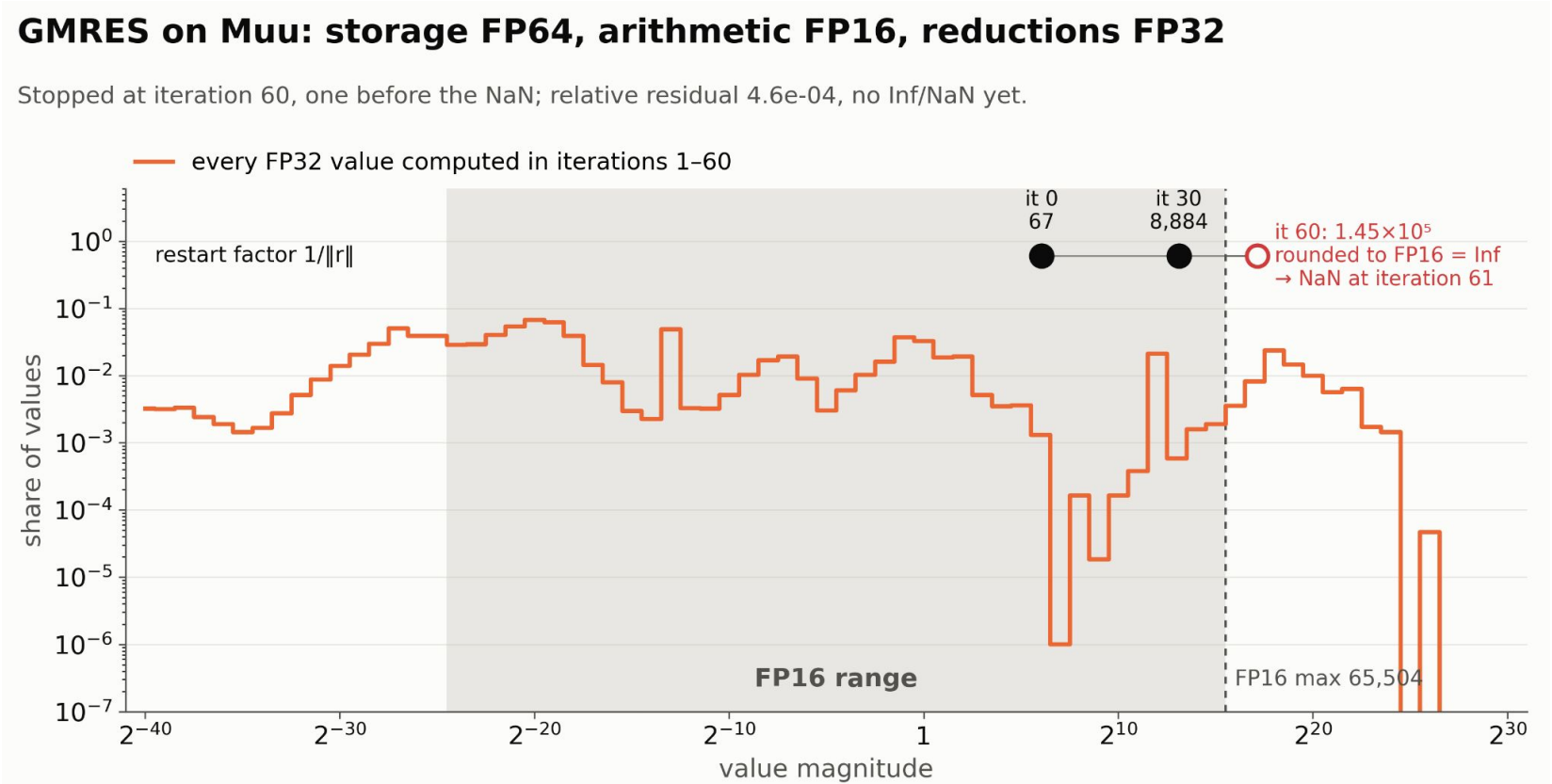
- ML
  - **Exceptions are a function of the particular tool versions**
    - When PyTorch moves, the bug may go away (look at Issue Pages)
  -
- HPC
  - **No real benchmark suites, or even people reporting exceptions as frequently**
    - Again, shareable universal solutions are difficult (everyone is busy)

- With NixNan
  - You can produce more insights
  - **You can perhaps get more out of LLMs when you feed it NixNan traces**

- What I find the most productive approach to be
  - **Run NixNan under Claude!**
  - Extend the tool while you run !!
    - Claude figures out the sampling flags and such
    - the *-autonixnan variants were Claude-created

Binade Histograms are a cool NixNan Feature!

(this pretty drawing isn't part of NixNan … but, certainly you can generate such a plot out of NixNan outputs!)



Bug types: Too easy, Too large, Could have used NixNan during design!



Easy bugs : can often spot and use (with manual effort)	Short and Tricky : Basic Logic of Kernel can be broken!	Too Large to Debug : Don't wait till this time!
BatchNorm1d : Switch from FP32 to FP64 with better accumulation	#160876: silu(x) = x * sigmoid(x) returns NaN; x = -inf (used to be 0)	Many issues are “too large to be posted” – very little can be done
Evident that non-experts suffer. Still gotta help them (non-uppity)	Naive implementation: -inf * 0 L’Hopital’s rule: 0	Solution: Avoid inserting bugs into simpler APIs by having unit-testing

Concluding Remarks, Current Focus

- Exceptions are an under-studied topic in HPC and ML
- Binary-level checking of exceptions is in its infancy
  - Check fpanalysistools.org , talk on FloatGuard (https://fpanalysistools.org/SC25/slides/FloatGuard.pdf)
  - The authors describe their journey building FP Exception Checking for AMD GPUs
- Our current use of NixNan
  - **We are using NixNan to observe exceptions during HPC Mixed-Precision Tuning**

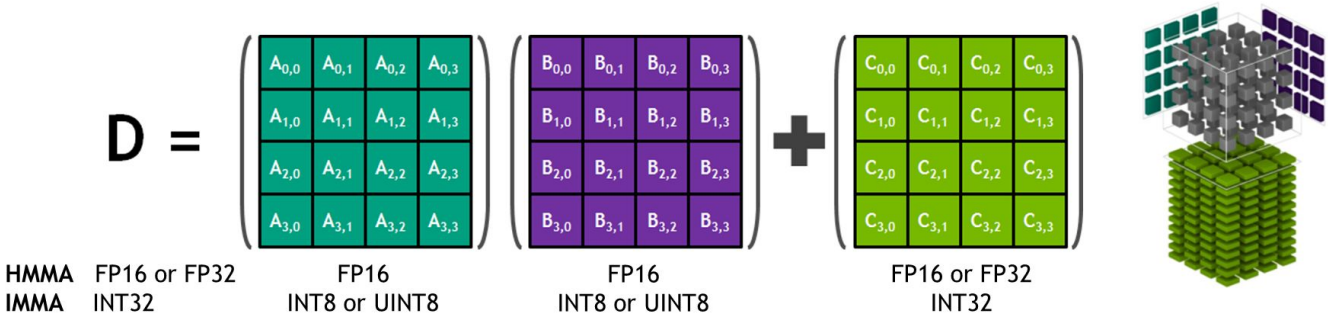
Details: Tensor-Core Instrumentation

Tensor cores

- “Tensor” is a generalization of a matrix
- Tensor cores really just compute matrix-multiplication and accumulation (MMA)
- Increasingly present in new hardware
  - Nvidia tensor cores, Google TPUs, Apple AMX, Intel AMX, AMD
- Accelerates matrix multiplication
- Often uses reduced precision

Tensor cores

- Computes D=A*B+C for small matrices
- Natural fit when matrices are broken into pieces for multiplication
- Nvidia Ampere can compute the product of two 16x16 and 16x8 half-precision matrices in 16 cycles



Tensor cores

- MMA is performed as a warp synchronous operation
  - Nvidia GPUs have simple execution units called threads
  - Each thread is part of a group of 32 threads called a warp
  - For an MMA, all 32 threads need to participate
  - All threads in a warp execute the same instruction

Tensor core exceptional values

- Can generate all three kinds of floating-point exceptional values
- NaN and infinity are the most relevant in the context of matrix multiplication
- Both will propagate during multiplication
  - Especially NaN since basic operations using NaN have NaN results

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & \mathrm{NaN} & 1 \end{pmatrix} = \begin{pmatrix} 5 & \mathrm{NaN} & 3 \\ 9 & \mathrm{NaN} & 4 \\ 3 & \mathrm{NaN} & 1 \end{pmatrix}$$

Tensor core register layout

Each thread holds part of the matrices in its registers

T1

R0	R1	R2	R3	R4	R5	R6	R7
A0	*A1*	*A2*	*A3*	*B0*	*B1*	*C0*	*C1*

T2

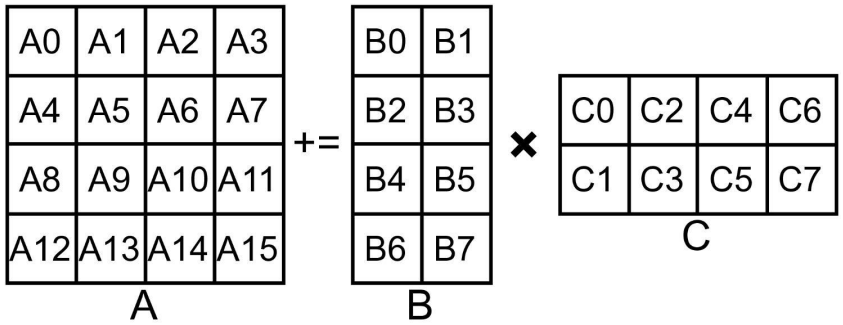
R0	R1	R2	R3	R4	R5	R6	R7
A4	*A5*	*A6*	*A7*	*B2*	*B3*	*C2*	*C3*

T3

R0	R1	R2	R3	R4	R5	R6	R7
A8	*A9*	*A10*	*A11*	*B4*	*B5*	*C4*	*C5*

T4

R0	R1	R2	R3	R4	R5	R6	R7
A12	*A13*	*A14*	*A15*	*B6*	*B7*	*C6*	*C7*

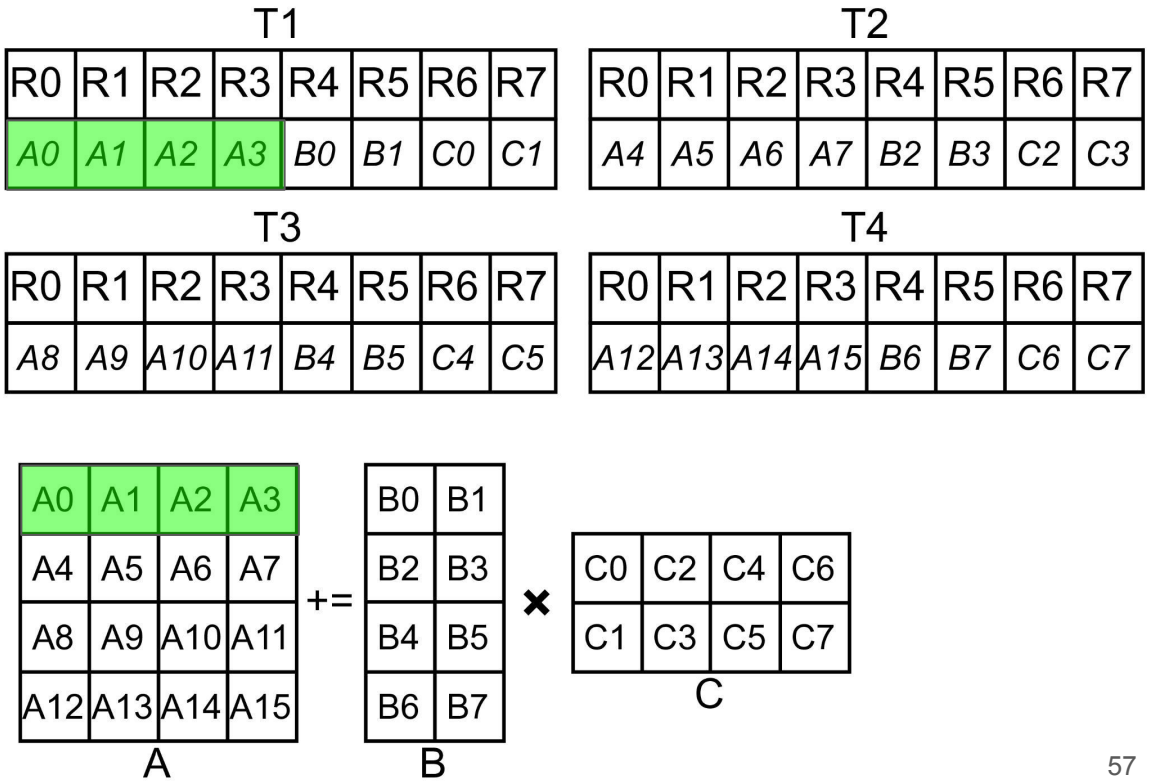


Tensor core register layout

Each thread holds part of the matrices in its registers

Thread T1 holds:

- The first four elements of A

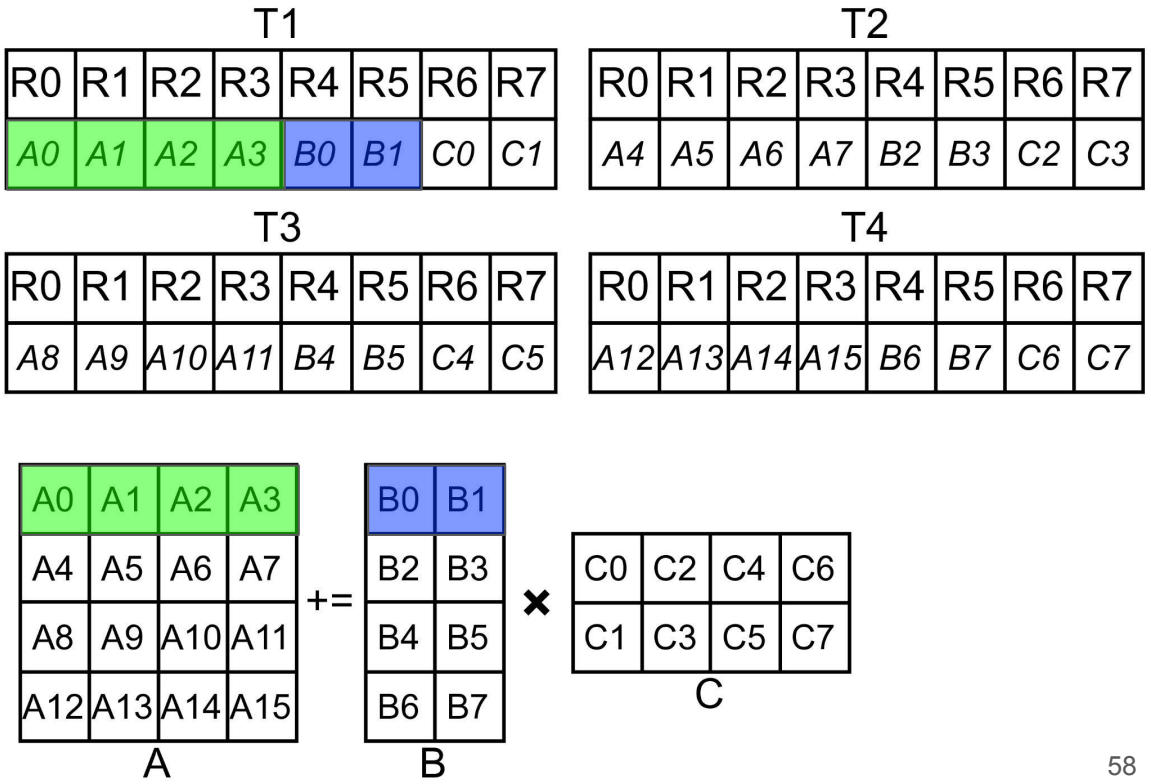


Tensor core register layout

Each thread holds part of the matrices in its registers

Thread T1 holds:

- The first four elements of A
- The first two elements of B

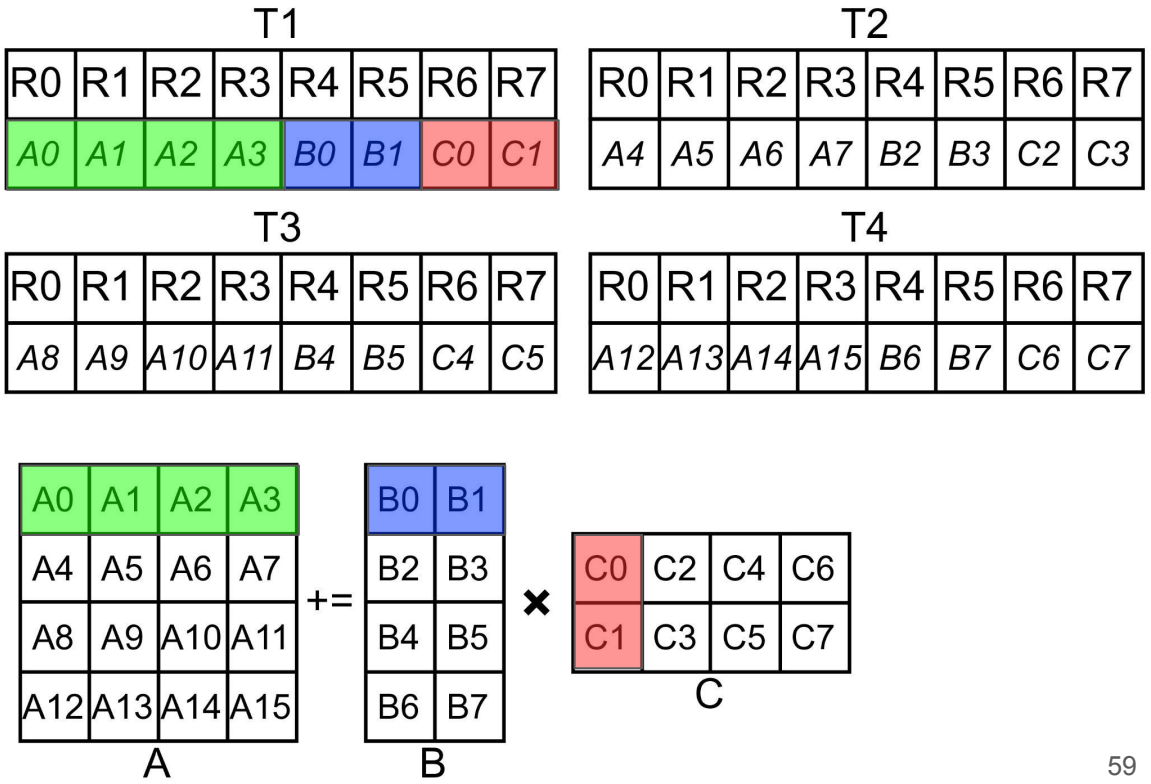


Tensor core register layout

Each thread holds part of the matrices in its registers

Thread T1 holds:

- The first four elements of A
- The first two elements of B
- The first two elements of C



Exceptional AB+C example

```
----- Test NaN: A[0,0]=inf, B[0,0]=1, C[0,0]=-inf -----
A[0,0] = inf (0x7c00), B[0,0] = 1.000000 (0x3c00), C[0,0] = -inf (0xfc00)
Computing D = A * B + C with Tensor Cores...
#nixnan: Initializing GPU context...
#nixnan: Could not open kernel_whitelist.txt!
#nixnan: Could not open kernel_blacklist.txt!
#nixnan: instrumenting all kernels
#nixnan: running kernel [WMMAF16TensorCore] ...
#nixnan: error [infinity] detected in operand 1 of instruction HMMA.16816.F16 R20, R4.reuse, R16, RZ ; in function WMMAF16TensorCore at line 0 of type f16
#nixnan: error [NaN,infinity] detected in operand 0 of instruction HMMA.16816.F16 R20, R4.reuse, R16, RZ ; in function WMMAF16TensorCore at line 0 of type f16
#nixnan: error [NaN] detected in operand 0 of instruction HMMA.16816.F16 R22, R4, R18, RZ ; in function WMMAF16TensorCore at line 0 of type f16
#nixnan: error [infinity] detected in operand 1 of instruction HMMA.16816.F16 R22, R4, R18, RZ ; in function WMMAF16TensorCore at line 0 of type f16
#nixnan: error [infinity] detected in operand 1 of instruction HMMA.16816.F16 R10, R4, R18, R10 ; in function WMMAF16TensorCore at line 0 of type f16
#nixnan: error [infinity] detected in operand 3 of instruction HMMA.16816.F16 R8, R4.reuse, R16, R8 ; in function WMMAF16TensorCore at line 0 of type f16
#nixnan: error [infinity] detected in operand 1 of instruction HMMA.16816.F16 R8, R4.reuse, R16, R8 ; in function WMMAF16TensorCore at line 0 of type f16
#nixnan: error [NaN] detected in operand 0 of instruction HMMA.16816.F16 R8, R4.reuse, R16, R8 ; in function WMMAF16TensorCore at line 0 of type f16
D[0,0]=nan (0x7fff)
|nan nan nan nan nan nan nan nan| |inf 0.0 0.0 0.0 0.0 0.0 0.0 0.0| |1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0| |-inf 0.0 0.0 0.0 0.0 0.0 0.0 0.0|
|0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0| |0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0| |0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0| |0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0|
|0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0| |0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0| |0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0| |0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0|
|0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0| |0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0| |0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0| |0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0|
|0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0|=|0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0|*|0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0|+|0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0|
|0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0| |0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0| |0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0| |0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0|
|0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0| |0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0| |0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0| |0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0|
|0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0| |0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0| |0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0| |0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0|
--------------------------------------------------------------------------------
#nixnan: Finalizing GPU context...

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

#nixnan: --- FP16 Operations ---
#nixnan: NaN:                    4 (60 repeats)
#nixnan: Infinity:               6 (90 repeats)
#nixnan: Subnormal:              0 (0 repeats)
#nixnan: Division by 0:          0 (0 repeats)

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

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

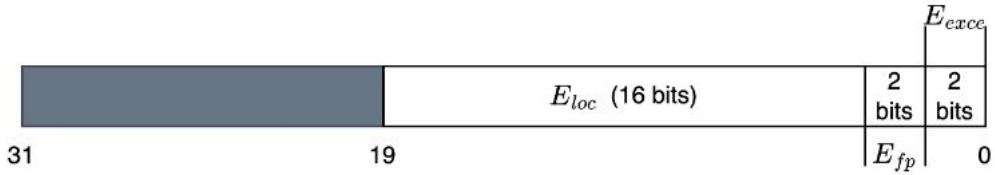
More Nixnan Internals

Nixnan error detection

- Create a more general error detection function
- Since we are sending variable amounts of data, we store pairs of register information and the the data from the registers for checking
  - The register information says the type and number of registers of that type are to be checked
  - Additionally whether it is a check for 0, in the case of division

Nixnan error recording

- Preserve more detailed information about which exceptions occurred in a warp
- GPU-FPX stores information to its table as follows:
  - Exceptions are denoted by an integer, 0, 1, 2 or 3
  - This shifts an index into the global table
  - The index is the id of the instruction (such as 9 for the 10th instruction)



Nixnan error recording

- Preserve more detailed information about which exceptions occurred in a warp
- GPU-FPX stores information to its table as follows:
  - Exceptions are denoted by an integer, 0, 1, 2 or 3
  - This shifts an index into the global table
  - The index is the id of the instruction (such as 9 for the 10th instruction)
- Nixnan uses two more bits for error recording
  - Each exception makes one bit in the exception recording field
  - Allows to record combinations of errors

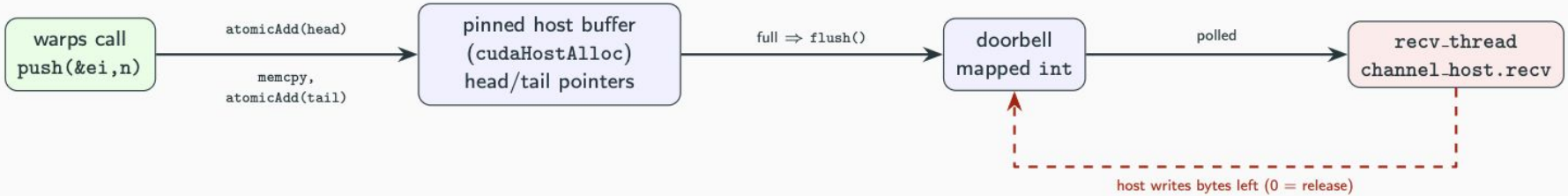


Wishlist

- In progress work to create a more adaptable GPU-FPX like framework
- Add a database of instructions that describes what errors to look for in the registers of the instruction
  - GPU-FPX has fixed code to identify instructions
- Create a more general error detection function
  - GPU-FPX creates a new function for new classes of instruction
- Preserve more detailed information about which exceptions occurred in a warp
  - GPU-FPX just records whether the exception occurred, such as infinity
  - Nixnan can report that an infinity and a NaN for instance were produced by a warp
- Independent error tables
  - Allows for the possibility of applying more than one instrumentation pass

More Nixnan Details, including Studies in Progress

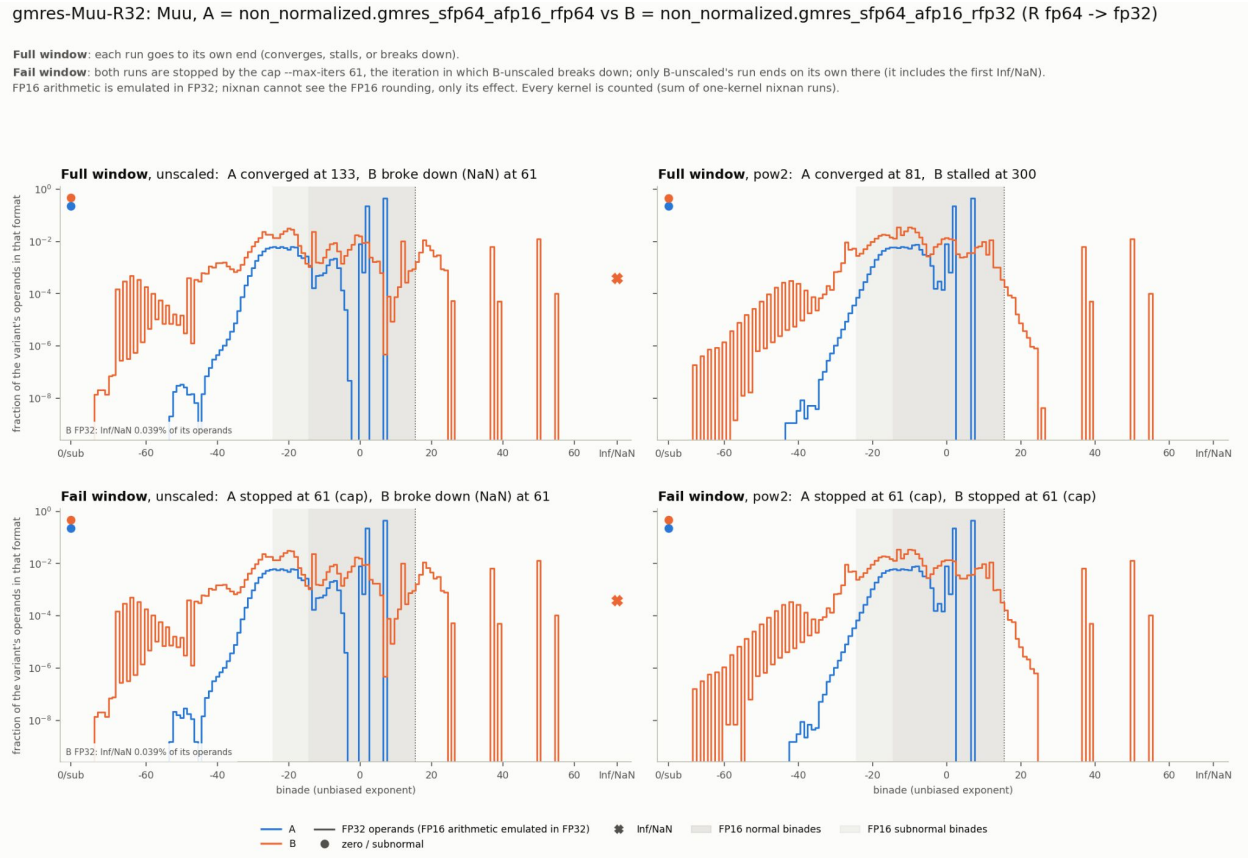
GPU -> Host Communication



push: reserve space with `atomicAdd` on the head; if it runs past `buff_end`, the first warp to notice waits for tail==its pointer and calls `flush()`; others spin until head is reset.

flush: `__threadfence_system()`; `*doorbell = nbytes`; **spin until host sets doorbell to 0**; reset tail then head. Corner-cases to avoid deadlocks are incorporated.

Illustration of Histograms from NixNan: FP32 -> FP16



- Study of GMRES on the Muu matrix (suiteSparse)
- Fp32-emulates-fp16: so non-normalized fp32 results are put into fp16.
- Variant A runs well, but not Variant B
- We show comparison between reduction fp64 (Variant A) vs fp32 (Variant B)
- The inputs are unscaled input (left charts) vs normalized by highest existing power of 2 to fit <= 1 (right charts)
- **Conclusion:** Histograms captured by NixNan help designer take decisions