juliacon

FlowFPX: Nimble Tools for Debugging Floating-Point Exceptions

Taylor Allred[1], Xinyi Li[1], Ashton Wiersdorf[1], Ben Greenman[1], and Ganesh Gopalakrishnan[1]

[1]University of Utah

ABSTRACT

Reliable numerical computations are central to scientific computing, but the floating-point arithmetic that enables large-scale models is error-prone. Numeric exceptions are a common occurrence and can propagate through code, leading to flawed results. This paper presents FlowFPX, a toolkit for systematically debugging floating-point exceptions by recording their flow, coalescing exception contexts, and fuzzing in select locations. These tools help scientists discover when exceptions happen and track down their origin, smoothing the way to a reliable codebase.

Keywords

Julia, floating-point, debugging

1. Introduction

Reliable numeric computations are central to high-performance computing, machine learning, and scientific applications. Yet the floating-point arithmetic that powers these computations is fundamentally unreliable (section 2). Exceptional values, such as Not a Number (`NaN`) and infinity (`Inf`), can and often do arise thanks to culprits such as roundoff error, catastrophic cancellation, singular matrices, and vanishing derivatives [33, 5, 11, 9, 23, 4]. Developers are responsible for guarding against exceptions, but this task is difficult because many operations can generate and propagate exceptions. Worst of all, operations such as less-than (<) can kill an exceptional value, leaving no trace of the problem. There is little tool support to assist in exception debugging, thus (unsurprisingly) a quick GitHub search reports over 4,000 open issues related to `NaN` or `Inf` exceptions [10].

This paper introduces FlowFPX, a toolkit for debugging floating-point exceptions (section 3) that has helped improve a variety of applications, from ocean simulations to heat modes (section 4):

—The centerpiece of FlowFPX is FloatTracker, a dynamic analysis tool that selectively monitors for exceptions and fuzzes code for vulnerabilities.

—To visualize results, FloatTracker adapts coalesced stack-trace graphs (CSTGs, or stack graphs) [15] to summarize the program contexts that handled exceptional values.

—A companion tool GPU-FPX [24] provides fine-grained insights for GPU kernels.

2. Floating-Point Exception Primer

Floating-point numbers use a finite number of bits to represent a spectrum of points along the real number line (fig. 1). The implementation strategy is essentially that followed in scientific notation. A floating-point number packs a sign bit, an exponent, and a fraction part (also called the “significand” or “mantissa”) into a bitstring. Typical strings are 64 or 32 bits long, but 16-bit and 8-bit formats are on the rise [17, 26]. This representation supports very small and very large numbers in a narrow range of bits:

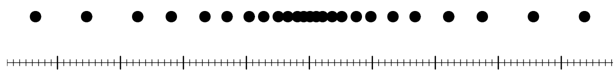

Fig. 1. Floats are spread across the real number line

```
julia> prevfloat(typemax(Float64))
1.7976931348623157e308
julia> nextfloat(typemin(Float64))
-1.7976931348623157e308
```

The flip side is that most real numbers fall into the gaps between floating-point numbers and must be rounded, which introduces error and can lead to surprising results. For example, adding the tiny Planck constant to the large Avogadro number results in Avogadro’s number after rounding:

```
julia> planck = 6.62607015e-34
6.62607015e-34
julia> avogadro = 6.02214076e23
6.02214076e23
julia> avogadro + planck == avogadro
true
```

This is an extreme example, but many operations on floating-point numbers closer in magnitude induce a loss of accuracy. Refer to the literature for more details, e.g., [21, 34, 28].

2.1 Exceptions and Exceptional Values

The IEEE 754 floating-point standard [16] defines exceptions and exceptional values as the outcome of operations that have “no single universally acceptable result” [30]. For example, dividing by zero and exceeding the `Float64` range both lead to exceptions:

```
julia> 0 / 0
NaN
julia> avogadro^avogadro
Inf
julia> log(0)
```

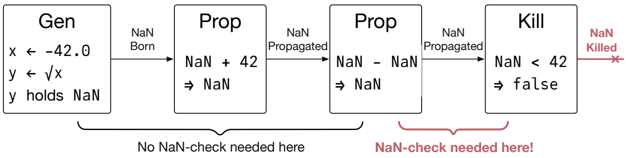

Fig. 2. Gen, Prop, Kill: Lifetime of an exceptional value

```
-Inf
julia> Inf + NaN
NaN
```

IEEE 754 defers the question of how to handle such exceptions to application code. It is up to developers to watch for `NaN`, `Inf`, and subnormal numbers (underflow) and implement an appropriate repair. This is no easy task, however, because of the approximations inherent to floating point. Even an apparently-safe division could result in a `NaN` if its denominator gets truncated to zero. Some `NaN`s might be spurious, others might be fatal, but in any event anticipating the various exceptions is a burden. All too often, exceptional values go unhandled and flow through the code.

2.2 Lifetime of an Unhandled Exceptions

Unhandled exceptional values have a lifetime: they are born, or *generated*, by some operation; they *propagate* through other operations; and they either appear in the program output, go out of scope, or get *killed* by a numeric operation. Figure 2 summarizes this *gen-prop-kill* process. The gens and props are straightforward; see above for examples (section 2.1). The kills often arise from numeric comparisons (<, =, etc.), but exponents (`1^NaN`) and over-eager matrix optimizations [5] can kill exceptions as well.

To illustrate the perils of killed exceptions, consider the following two ways of finding the maximum value in a list. The first compares numbers with `<=` while the second uses the built-in `max` function:

```
function max1(lst)
  max_seen = 0.0
  for x in lst
    # swap if x is not too small
    if ! (x <= max_seen)
      max_seen = x
    end
  end
  max_seen
end

function max2(lst)
  foldl(max, lst)
end
```

For lists with a `NaN` inside, the functions can give different results because `<=` kills `NaN`s whereas `max` propagates them:

```
julia> max1([1, 5, NaN, 4])
4.0
julia> max2([1, 5, NaN, 4])
NaN
```

Not only is the result from `max1` problematic for obscuring the fact that there was a `NaN` in the list, the result is arguably wrong! The result from `max2` at least shows that a `NaN` was in the works, though in a realistic setting it may not be clear where the `NaN` came from.

Exn. Input?	⇒	Exn. Output?	=	Event
×	⇒	✓	=	gen
✓	⇒	✓	=	prop
✓	⇒	×	=	kill

Fig. 3. How to classify operations that see exceptions

Both versions would thus benefit from tools that track exceptions across their lifetime. FlowFPX can help.

3. FlowFPX

FlowFPX is a toolkit for tracking down floating-point exceptions. The primary tools in this paper are FloatTracker, which records lifetimes and enables fuzzing, and stack graphs (more precisely, CSTGs), which visualize the flow of exceptions. GPU-FPX is a third component of FlowFPX that tracks floating-point exceptions inside GPU kernels [24].

3.1 FloatTracker

FloatTracker tracks exceptional values across their *gen-prop-kill* lifetime and can fuzz code for vulnerabilities by injecting a `NaN` or `Inf` as the result of an operation. FloatTracker is implemented in Julia and is available on JuliaHub:

`https://juliahub.com/ui/Packages/FloatTracker/dBXig`

3.1.1 Tracking Exceptional Values. FloatTracker monitors `NaN` and `Inf` exceptions by overloading arithmetic operations and logging key events. When the input to an operation is exception-free but the output is exceptional, the operation is a gen event (fig. 3). When the input and output contain exceptions, the operation is a prop event. And when the input contains an exception but the output does not, the operation is a kill event. Put together, the log of all gens, props, and kills sheds light of how various exceptions traveled across the program. The logs also record the call context (stack trace) and the arguments to the operation as a starting point for debugging efforts.

FloatTracker writes logs to three files: one for gen events, one for props, and one for kills. This way, discovering where a `NaN` came from is a matter of sifting through the gen file. Users can also lower the overhead of logging by turning it off for prop events.

The instrumentation works through custom floating-point types: `TrackedFloat64`, `TrackedFloat32`, and `TrackedFloat16`. Developers must opt in to FloatTracker by wrapping numbers in a custom type. From then on, tracking is automatic and extends transitively to all outputs of tracked operations. Multiplying a `Float64` value with a `TrackedFloat64`, for example, yields a `TrackedFloat64` to continue the logging trail.

Crucially, the tracked version of an exception, say a `NaN`, points to the original exceptional value. This enables techniques such as `NaN`-packing, or otherwise using the payload in creative ways. Not all Julia operations currently preserve `NaN`s bit-for-bit,[1] but FloatTracker does its part.

3.1.2 Fuzzing. Operator overloading gives FloatTracker a powerful way to fuzz code from the inside out. Each overloaded function serves as a hook where FloatTracker can decide whether to observe the operation or step in, discard the original result, and return an exception instead. Injecting faults in a random way [12], also known as fuzzing, is a useful way to discover vulnerabilities in a large codebase. Demmel et al. propose essentially the same idea for BLAS and LAPACK [5].

[1] `https://github.com/JuliaLang/julia/issues/48523`

FloatTracker exposes several parameters to let developers control the fuzzer:

—`odds::Int64` inject if `rand(odds) == 1`.

—`n_inject::Int64` upper bound on the number of `NaN`s to inject.

—`active::Bool` fuzz only when set to true.

—`functions::ArrayString` limit fuzzing to the dynamic extent of the listed functions.

—`libraries::ArrayString` limit fuzzing to functions from the following libraries

When fuzzing reveals an error, the next step is to craft a regression test to guide repairs. FloatTracker therefore records the sequence of injections that it makes during a fuzzing run and enables a replay of any recording after the fact. Replay runs proceed deterministically so that developers can harden their code and check that the fixes remove the error.

3.1.3 FloatTracker Internals. FloatTracker takes advantage of Julia's operator overloading to track exceptions and fuzz for vulnerabilities. For example, below is a variant of + that is overloaded for `TrackedFloat64`. It calls the basic + (or injects a `NaN`) and checks for exceptions before returning: This lets FloatTracker intercept all floating-point operations involving `TrackedFloat` types.

```
function +(x::TrackedFloat64, y::TrackedFloat64)
  result = run_or_inject(+, x.val, y.val)
  check_error(+, result, x.val, y.val)
  TrackedFloat64(result)
end
```

Every arithmetic operation requires a similar overloading. Thus, the implementation of FloatTracker uses a metaprogramming technique adapted from `Sherlogs` [20] to abstract over the common patterns. For every binary operation, the library creates an overloading similar to the one for + above. Unary operations and others work analogously. This approach saved thousands of lines of code. The implementation weighs in at 218 lines and defines 645 overloaded function; assuming 5 lines of code per function, a handwritten version would require over 3,000 lines.

3.2 Stack Graphs

FloatTracker can produce copious amounts of log files which can be challenging to sift through manually. Coalesced stack trace graphs (CSTGs or stack graphs for short) provide a way to visualize large amounts of stack traces in a compact form [15]. This techinque pairs well with FloatTracker and makes analyzing the log files easier.

Figure 4 illustrates the construction of a stack graph from a collection of stack traces. Each trace on the left contributes nodes and edges to the graph on the right. Repeated edges get emphasized with darker lines and larger counts.

Reading bottom-up, a stack graph based on the *gen* events in a program highlights the contexts that frequently produced exceptional values. Using fig. 4 as an example, `Frame D` produced three exceptions, two of which arose under `Frame C`. In a large program with many exceptions, stack graphs offer a way to prioritize debugging efforts: go for the heavily-trodden paths first.

3.3 GPU-FPX

Many programs offload work onto GPUs, which are no less susceptible to floating-point exceptions than CPUs. In fact, GPU programs are worse off because they lack exception-handling mechanisms from the CPU world [23]. Since FloatTracker instruments Julia programs, it cannot help directly; however, the companion tool GPU-FPX instruments GPU kernels to detect and report floating-point exceptions [24]. Together, FloatTracker and GPU-FPX provide insights for accelerated programs.

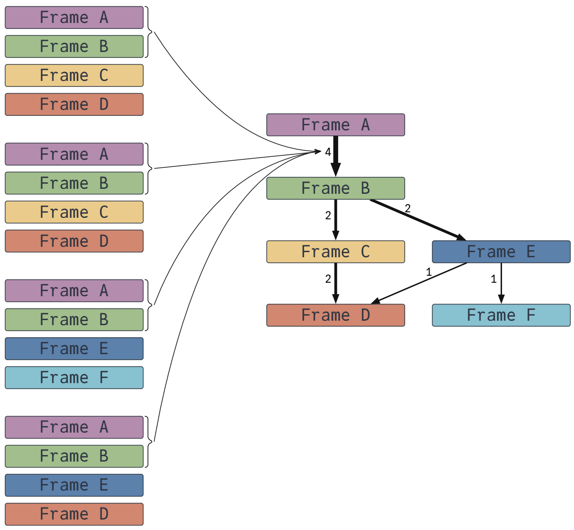

Fig. 4. From stack traces (left) to stack graph (right)

4. Case Studies

FlowFPX has helped to debug exceptions and fuzz for issues in a variety of settings, some synthetic and some realistic. The case studies include a shallow water simulation, the `OrdinaryDiffEq` solver, and a Bayesian inference library.

4.1 ShallowWaters

`ShallowWaters` simulates the flow of water over a seabed [18, 19]. The library has dozens of parameters that a scientist can experiment with. One notable parameter is the Courant-Friedrichs-Lewy (CFL) number, which roughly describes the size of the time step to take in running the simulation. A small CFL number makes the simulation run slowly, but accurately; a large number speeds it up but loses precision because the system does not get enough time to propagate information.

Normal values for the CFL number range between zero and one. With a CFL of 0.9, the shallow water simulation produces the graph on the left half of fig. 5 showing current speed in a rectangular basin. Raising the CFL too high, to 1.6, causes trouble (fig. 5, right) with large, white regions where the current speed is `NaN` instead of a normal value.

Running FloatTracker on the high-CFL simulation reveals where the simulation drifted from numbers to `NaN` exceptions. The first step in applying FloatTracker is to convert relevant floats to tracked floats, e.g., `Float32` to `TrackedFloat32`. For `ShallowWaters`, this step is easy because the simulation is parameterized by a floating-point type for internal use. Swapping in a tracked type is enough:

```
run_model(T=TrackedFloat32,
          cfl=10,
          nx=100,
          Ndays=100,
          L_ratio=1,
```

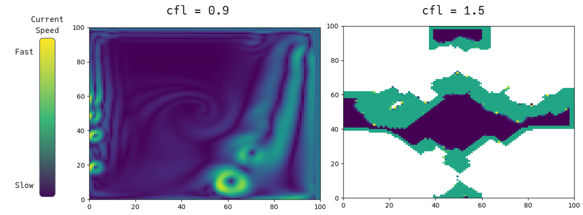

Fig. 5. Raising the CFL number creates white gaps due to `NaN`s

```
        bc="nonperiodic",
        wind_forcing_x="double_gyre",
        topography="seamount")
```

With tracking, the simulation runs as before, producing an ugly graph. It additionally outputs logs for all gen, prop, and kill events as they happen. Below is one event from the gen logs:

```
-([-Inf, -Inf])      FT/TrackedFloat.jl:106
momentum_u!          SW/rhs.jl:246
rhs_nonlinear!       SW/rhs.jl:50
rhs!                 SW/rhs.jl:14 [inlined]
time_integration     SW/time_integration.jl:77
run_model            SW/run_model.jl:37
top-level
```

We can see that the `NaN` appeared as the result of subtracting two infinities (`-Inf - -Inf`). The trace further shows that this subtraction happens inside the function `momentum_u!` on line 246 of the `rhs.jl` file.

This solves the mystery of where one `NaN` came from, but raises a new question about the source of the `Inf` value. The logs for `Inf` gens have an answer:

```
^([-1.515f31, 2])              FT/TrackedFloat.jl:138
literal_pow()                  intfuncs.jl:325
...
materialize(^)                 broadcast.jl:860
top-level getproperty(...)     examples/sw_nan_tf.jl:14
..
```

This `Inf` came from an exponent that overflowed the float type (`-1.515e31^2`). FloatTracker has shown exactly which numeric values in which operation caused exceptions to occur.

4.1.1 Stack Graphs for a Bigger Picture. While the logs for `ShallowWaters` contain useful information, there is an overwhelming amount of it. There are over ten thousand lines in the gen file alone. Converting these logs to a stack graph gives a quick overview of the most common paths to exceptional values.

Figure 6 presents the stack graph for `NaN` gen events in `ShallowWaters` with the high CFL number. Reading bottom-up, every `NaN` came from calls to the '-' and '+' operations. Calls to '+' account for most of the `NaN`s. These `NaN`s arose in two different contexts: a small number (30) occurred within a momentum calculation, while the rest (162) occurred within a continuity step. Moving to the top of the graph, it shows that the function `run_model` drove the entire simulation. With this overview of the program, a promising next step is to guard against `NaN`s in the momentum and continuity functions.

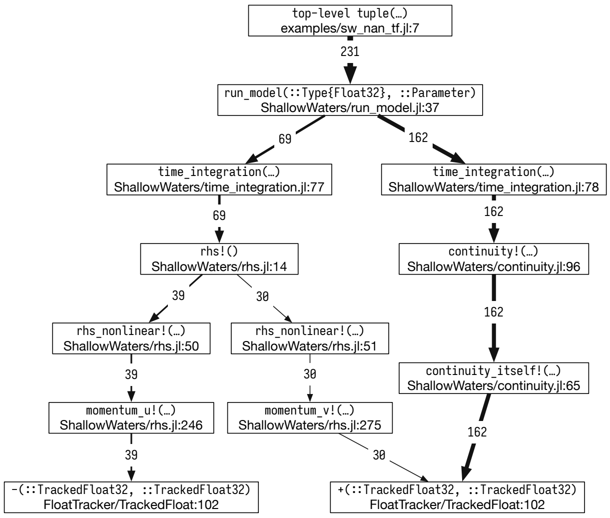

Fig. 6. Stack graph for `NaN` gens in `ShallowWaters`

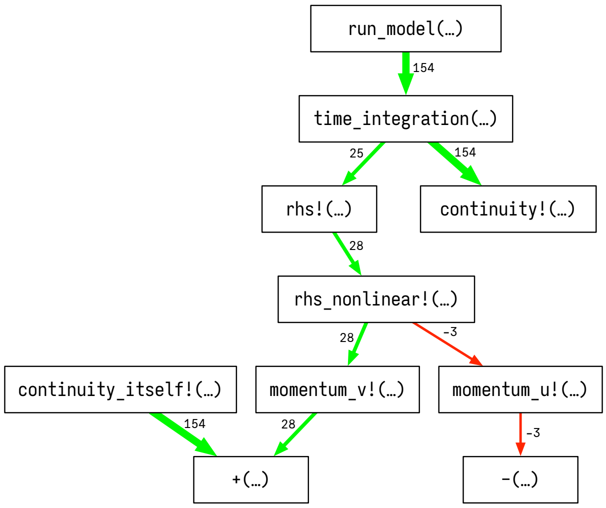

Fig. 7. Stack graph diff

4.1.2 Stack Graph Differences. Graph diffing works well for stack graphs; it shows how flows in the program evolved from one stage to another. Figure 7 illustrates one diff in the context of `NaN` gens for `ShallowWaters`: it compares the first 10% of gen events to the latter 90% of gens. The positive numbers and green lines indicate flows that are new in the latter part, and the negative numbers and red lines show flows that disappeared in the latter part. A domain expert might use these clues to find where an instability started in the first part of the program. In this case, the function `momentum_u!` showed up only in the beginning of the logs—it may be an effective point to check for `NaN`s.

4.2 OrdinaryDiffEq

With the targeted fuzzing abilities of FloatTracker, we set focus on the `NBodySimulator` package from the SciML team.[2] We only

[2] `https://github.com/SciML/NBodySimulator.jl`

wanted to test the robustness of the `NBodySimulator` package, so we configured the fuzzer to inject `NaN`s *only* when inside of a function of that library. This way, we could avoid injecting in the standard library or other dependencies of `NBodySimulator` and concentrate on improving this one library.

However, Fuzzing discovered zero bugs and zero exceptional events. In fact, fuzzing injected *zero* exceptional values whatsoever, even with adjusting the odds to *force* an injection at the first available opportunity. The reason for this lack of `NaN` injection, it turns out, is that the nbody simulation merely creates a problem for the `OrdinaryDiffEq` solver.[3] All the floating-point operations were in another library.

Fuzzing on `OrdinaryDiffEq` led to a curious situation. The library itself reported a `NaN` and printed a message that it would exit. However, after printing that message, the program went into an infinite loop. Using stack graphs to guide our search, we found a `NaN` kill that manifested repeatedly in the logs. The stack traces for that kill originated from inside the file `solve.jl`:

```
<([NaN, 3.0e6])  at FT/TrackedFloat.jl:193
solve!           at ODE/solve.jl:515
...
```

The relevant part of `solve.jl` contains a pair of loops. With injection, the variable `tdir` holds a `NaN`, stopping all productivity:

```
while !isempty(time_stops)
  while tdir * t < first(time_stops)
    # do integration work
    # pop_off(time_stops)
  end
end
```

In more detail, the `NaN` for `tdir` propagates though the multiplication and gets killed by the `<` comparison. Hence the condition for inner `while` loop is always false, which means the outer loop never ends up with an empty list.

This is a real-world example of how `NaN` kills can affect control flow, and we filed an issue for it.[4] Fortunately, the problem in this case is benign as the code was already trying to halt.

4.3 Finch

Finch is a domain-specific language for specifying PDEs [14].[5] In the spirit of FEniCS [2] and related tools [13, 25, 7, 32], Finch helps scientists quickly convert math into code. What sets Finch apart is its flexibility. It supports multiple discretization methods (finite element and finite volume) and multiple backends (Julia, C++, DENDRO [8]). Furthermore it strives to output code that humans can easily fine-tune.

Fuzzing with FloatTracker revealed two places where Finch needed protection against user input. The first was when reading an input mesh.[6] A `NaN` injected in the mesh led to a crash further on:

```
  BoundsError: attempt to access 1-element
   Vector{Int64} at index [2]
```

[3] `https://github.com/SciML/OrdinaryDiffEq.jl`
[4] `https://github.com/SciML/OrdinaryDiffEq.jl/issues/1939`
[5] Not to be confused with the Finch loop optimizer [1].
[6] `https://github.com/paralab/Finch/issues/16`

```
-- FP64 Operations --       -- FP32 Operations --
Total NaN:          2       Total NaN:          1
Total INF:          1       Total INF:          0
Total subnormal:    0       Total subnormal:    0
Total div0:         2       Total div0:         0
                  -- Other Stats --
                  Kernels:   21
```

Fig. 8. Example GPU-FPX output

Before accessing the input, Finch needed to check for `NaN`s. The second place was in setting bounds for the solver.[7] Here, a `NaN` could leave bounds uninitialized, leading to a bounds error. Additionally, FloatTracker and stack graphs have been useful for identifying `NaN`s that appear in unstable heat simulations written in Finch.

4.4 Oceananigans

`Oceananigans` [31] is simulation package for incompressible fluid dynamics that can generate code for Nvidia GPUs. For example, the following program (from the project readme) simulates turbulance:

```
using Oceananigans
grid = RectilinearGrid(GPU(),
  size=(128, 128), x=(0, 2π), y=(0, 2π),
  topology=(Periodic, Periodic, Flat))
model = NonhydrostaticModel(; grid,
  advection=WENO())
ϵ(x, y, z) = 2rand() - 1
set!(model, u=ϵ, v=ϵ)
simulation = Simulation(model;
  Δt=0.01, stop_time=4)
run!(simulation)
```

GPU-FPX provides detailed feedback on this program. The output in fig. 8 shows that 21 kernels appear and generate six floating-point exceptions. There are three `NaN`s, one `Inf`, and two division by zero errors. The report is a starting point for further investigation of the reliability of the example.

4.5 RxInfer

We discovered an open issue in the `RxInfer.jl` library related to `NaN` detection and suggested FloatTracker to the developers.[8] Within a day, they were able to track down the location of an important `NaN` gen. This success came with little input from our end; in fact, the application program was closed-source. We merely explained how to use FloatTracker by wrapping inputs in a tracked type.

5. Related Work

5.1 Error Analysis

Demmel et al.[5] examine floating-point exceptional value handling in BLAS and LAPACK, identify several inconsistencies, and propose an API for debugging and adding determinism. The proposal includes an extension to the `INFO_ARRAY` parameter with fields that record *gen-prop-kill* information. It also includes fuzzing, which we realize in FloatTracker.

Toronto and McCarthy [34] propose a test-driven method for detecting numeric error: plot the results of an expression on a range of inputs and look for sharp deviations, or *badlands*. They point out

[7] `https://github.com/paralab/Finch/issues/17`
[8] `https://github.com/biaslab/RxInfer.jl/issues/116`

several ways to rewrite code to avoid badlands by rewriting code in a semantics-preserving way.

Herbie [29] automatically rewrites arithmetic expressions to reduce floating-point error. This would combine well with FloatTracker: first identify the location of a `NaN`, ask Herbie to find a repair. The Odyssey[27] workbench provides an interactive interface to Herbie.

5.2 Diagnosing floating-point exceptions

The Julia library `Sherlogs` [20] inspired our use of a custom number type to intercept operations on a number. In contrast to FloatTracker which monitors for exceptional values and logs stack traces at interesting points in their lifetime, `Sherlogs` tracks and reports the range of values seen over the course of a computation. This is intended to provide insight into whether or not a library could tolerate a lower-precision floating-point format.

FPSpy [6] is an `LD_PRELOAD` shared library that works on unmodified x86 binaries. It monitors a program during execution for operations that generate an exception, such as division by zero, underflow, and overflow. By contrast to FloatTracker, it does not track prop or kill events. FPSpy has the advantage of being lightweight enough to run on production code for certain loads.

5.3 Stack Graphs

Our stack graphs utilize the CSTG library for coalesced stack trace graphs [15]. In turn, CSTGs build on the STAT tool from LLNL [3]. STAT collects, analyzes, and visualizes stack traces from concurrent processes to highlight anomalies. It produces visualizations similar to those of CSTG, thought CSTG offers more compact views and supports diffs.

5.4 GPU Exception Tracking

FPChecker [22] is a tool to report floating-point exceptions occurring on the GPU. FPChecker relies on LLVM-level instrumentation of GPU kernels, and so cannot run on the plethora of closed-source GPU kernels in usage today. BinFPE [23] is another tool in the same space; BinFPE performs SASS-level analysis of GPU kernels, but is limited in that it is slow and does not catch errors that alter control flow. The latter deficiency is particularly worrisome, as we have seen, silent `NaN` kills can invalidate results without the user noticing. GPU-FPX [24] improves on the work of FPChecker and BinFPE by being more performant and catching a wider set of errors, including those that alter control flow. GPU-FPX is a shared library that, like FPSpy uses `LD_PRELOAD` to work on unmodified binaries. GPU-FPX runs on CUDA cores from NVIDIA and reports total numbers of exceptional values. Like FloatTracker, GPU-FPX can catch `NaN` gens and kills, but it does this on the GPU where FloatTracker doesn't apply. Despite these improvements, GPU-FPX is limited by the closed-source nature of common GPU cores, and cannot report at the rich level of source detail that FloatTracker can.

6. Discussion

Lightweight tools for error analysis that can quickly identify floating-point problems and suggest repairs are an important topic. The number of scale of scientific application has grown tremendously over the years. For small teams that cannot afford a full-time analyst, tools like FlowFPX fill a critical role.

FlowFPX it itself an evolving toolkit. Below we discuss some topics for future work.

6.1 Performance

FloatTracker incurs significant overhead on the order of 100x slower than a non-instrumented run of the same program. It is a debugging tool, not a production tool. To put this number into context, Valgrind runs with a similar level of slowdown.

In addition to the cost incurred by intercepting floating-point operations, gathering stack traces is expensive. We observe a 10x slowdown on `ShallowWaters` with logging disabled. We recommend that users of FloatTracker make use of the `maxLogs` and `exclude_stacktrace` configuration options to limit the number and kind of logs gathered, and thereby reduce the number of calls to `stacktrace()`. Stack traces are essential to decide where to inject a fault when fuzzing, but we defer them as late as possible to maximize performance.

6.2 Enhanced Fuzzing

While fuzzing is useful for discovering issues, its success rate is low because every floating-point operation is a candidate for injection. Even operations that are already well-defended against `NaN`s are candidates. FloatTracker could use two sorts of tools for improving injection. First, fine-grained control to let users decide where not to inject. Second, tools for understanding the context of an injection point after the fact. Program slicing is especially relevant to the latter point and effectively what we did by hand when fuzzing Finch (section 4.3). For each operation, an expert needs to study the values that feed into it to decide whether they are protected or not.

6.3 Tracking Exceptions in External Libraries

FloatTracker is limited to Julia code. It cannot track the lifetime of exceptional values in external libraries, such as GPU kernels or C programs. For GPUs it relies on GPU-FPX, through the connection between these tools is loose. In the future, FloatTracker would benefit from an API to plug in tools for external libraries, gather their output, and present a comprehensive view of exceptions in a multi-language program.

6.4 Interface Concerns

FloatTracker is capable of monitoring all sorts of events and floating-point values beyond exceptions. For example, FloatTracker could monitor large or small values with guidance from the user about which values are worth logging. On a similar note, the interface to FloatTracker is primitive: users express interest in a number by wrapping it in a constructor such as `TrackedFloat64`. Helper functions that track data structures and choose a default size would make it easier to experiment with FloatTracker.

7. Acknowledgments

Thanks to the `Sherlogs.jl` developers for inspiring the architecture of FloatTracker. Thanks to the `RxInfer.jl` developers for testing FloatTracker and for feedback on its user interface. Thanks to Alex Larsen and Rob Durst for comments on an early draft. This work is supported by NSF grant 2030859 to the CRA for the CIFellows project, and DOE ASCR Award Number DE-SC0022252 and NSF CISE Awards 1956106 and 21241.

8. References

[1] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. Looplets: A language for struc-

tured coiteration. In *CGO*, pages 41–54. ACM, 2023. doi:10.1145/3579990.3580020.

[2] M. S. Alnaes, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3, 2015. doi:10.11588/ans.2015.100.20553.

[3] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Stack Trace Analysis for Large Scale Debugging. In *IPDPS*, pages 1–10. IEEE, 2007. doi:10.1109/IPDPS.2007.370254.

[4] Dorra Ben Khalifa, Xinyi Li, Ignacio Laguna, Matthieu Martel, and Ganesh Gopalakrishnan. Toward increasing trust in exascale simulations. In *XLOOP*, pages 26–31. IEEE, 2022. doi:10.1109/XLOOP56614.2022.00010.

[5] James Demmel, Jack J. Dongarra, Mark Gates, Greg Henry, Julien Langou, Xiaoye S. Li, Piotr Luszczek, Weslley S. Pereira, E. Jason Riedy, and Cindy Rubio-González. Proposed consistent exception handling for the BLAS and LAPACK. In *Correctness@SC*, pages 1–9. IEEE, 2022. doi:10.1109/Correctness56720.2022.00006.

[6] Peter Dinda, Alex Bernat, and Conor Hetland. Spying on the Floating Point Behavior of Existing, Unmodified Scientific Applications. In *HPDC*, pages 5–16. ACM, 2020. doi:10.1145/3369583.3392673.

[7] Dune. DUNE numerics, 2023. `https://www.dune-project.org/`. Accessed 2023-06-06.

[8] Milinda Fernando, David Neilsen, Eric W. Hirschmann, and Hari Sundar. A scalable framework for adaptive computational general relativity on heterogeneous clusters. In *SC*, page 1–12. ACM, 2019. doi:10.1145/3330345.3330346.

[9] FPChecker. Open source reports, 2023. `https://fpchecker.org/open-source-reports.html`. Accessed 2023-06-16.

[10] GitHub. Issue search: `NaN+infinity`, 2023. `https://github.com/search?q=NaN+infinity++state%3Aopen&type=issues&ref=advsearch`. Accessed 2023-06-16.

[11] Ganesh Gopalakrishnan, Ignacio Laguna, Ang Li, Pavel Panchekha, Cindy Rubio-González, and Zachary Tatlock. Guarding numerics amidst rising heterogeneity. In *Correctness@SC*, pages 9–15. IEEE, 2021. doi:10.1109/Correctness54621.2021.00007.

[12] Richard Hamlet. Random testing. *Encyclopedia of Software Engineering*, 2:971–978, 1994. doi:10.1002/0471028959.

[13] F. Hecht. New development in FreeFem++. *Journal of Numerical Mathematics*, 20(3-4):251–265, 2012. doi:10.1515/jnum-2012-0013.

[14] Eric Heisler, Aadesh Deshmukh, and Hari Sundar. Finch: Domain Specific Language and Code Generation for Finite Element and Finite Volume in Julia. In *ICCS*, volume 13350, pages 118–132. Springer, 2022. doi:10.1007/978-3-031-08751-6_9.

[15] Alan Humphrey, Qingyu Meng, Martin Berzins, Diego Caminha B. De Oliveira, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Systematic Debugging Methods for Large-Scale HPC Computational Frameworks. *Computing in Science & Engineering*, 16(3):48–56, 2014. doi:10.1109/MCSE.2014.11.

[16] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20, 1985. doi:10.1109/IEEESTD.1985.82928.

[17] Milan Klöwer. *Low-Precision Climate Computing: Preserving Information despite Fewer Bits*. http://purl.org/dc/dcmitype/Text, University of Oxford, 2021.

[18] Milan Klöwer, Peter D. Düben, and T. N. Palmer. Number Formats, Error Mitigation, and Scope for 16-Bit Arithmetics in Weather and Climate Modeling Analyzed With a Shallow Water Model. *Journal of Advances in Modeling Earth Systems*, 12(10), 2020. doi:10.1029/2020MS002246.

[19] Milan Klöwer, Peter D. Düben, and Tim N. Palmer. Posits as an alternative to floats for weather and climate models. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, pages 1–8. ACM, 2019. doi:10.1145/3316279.3316281.

[20] Milan Klöwer and OnButtonUp. Milankl/Sherlogs.jl, 2021. doi:10.5281/ZENODO.5115765.

[21] Donald Ervin Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997.

[22] Ignacio Laguna. FPChecker: Detecting floating-point exceptions in GPU applications. In *ASE*, pages 1126–1129. IEEE, 2019. doi:10.1109/ASE.2019.00118.

[23] Ignacio Laguna, Xinyi Li, and Ganesh Gopalakrishnan. BinFPE: accurate floating-point exception detection for GPU applications. In *SOAP*, pages 1–8. ACM, 2022. doi:10.1145/3520313.3534655.

[24] Xinyi Li, Ignacio Laguna, Katarzyna Swirydowicz, Bo Fang, Ang Li, and Ganesh Gopalakrishnan. Design and evaluation of GPU-FPX: A low-overhead tool for floating-point exception detection in NVIDIA GPUs. In *ACM HPDC 2023*, 2023. doi:10.11578/dc.20230713.4.

[25] Sandra Macià, Pedro J. Martínez-Ferrer, Sergi Mateo, Vicenç Beltran, and Eduard Ayguadé. Assembling a high-productivity DSL for computational fluid dynamics. In *PASC*. ACM, 2019. doi:10.1145/3324989.3325721.

[26] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart F. Oberman, Mohammad Shoeybi, Michael Y. Siu, and Hao Wu. FP8 formats for deep learning. *CoRR*, abs/2209.05433, 2022. doi:10.48550/arXiv.2209.05433.

[27] Edward Misback, Caleb Chan, Brett Saiki, Eunice Jun, Zachary Tatlock, and Pavel Panchekha. Odyssey: An interactive workbench for expert-driven floating-point expression rewriting. *CoRR*, abs/2305.10599, 2023. doi:10.48550/arXiv.2305.10599. arXiv:2305.10599.

[28] Jean-Michel Muller, Nicolas Brunie, Florent De Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Springer, 2018. doi:10.1007/978-3-319-76526-6.

[29] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI*, pages 1–11. ACM, 2015. doi:10.1145/2737924.2737959.

[30] David J. Priest. Handling IEEE 754 invalid operation exceptions in real interval arithmetic, 1997. `https://j3-fortran.org/doc/year/97/97-172.pdf`. Accessed 2023-08-16.

[31] Ali Ramadhan, Gregory LeClaire Wagner, Chris Hill, Jean-Michel Campin, Valentin Churavy, Tim Besard, Andre Souza,

Alan Edelman, Raffaele Ferrari, and John Marshall. Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs. *Journal of Open Source Software*, 5(53):2018, 2020. doi:10.21105/joss.02018.

[32] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *Transactions on Mathematical Software*, 43(3), 2016. doi:10.1145/2998441.

[33] Kasia Świrydowicz, Eric Darve, Wesley Jones, Jonathan Maack, Shaked Regev, Michael A Saunders, Stephen J. Thomas, and Slaven Peleš. Linear solvers for power grid optimization problems: A review of GPU-accelerated linear solvers. *Parallel Computing*, 111:102870, 2022. doi:10.1016/j.parco.2021.102870.

[34] Neil Toronto and Jay McCarthy. Practically Accurate Floating-Point Math. *Computing in Science & Engineering*, 16(4):80–95, 2014. doi:10.1109/MCSE.2014.90.