FLiT: Cross-Platform Floating-Point Result Consistency Aid*

Geof Sawaya
University of Utah
sawaya@cs.utah.edu

Michael Bentley
University of Utah
mbentley@cs.utah.edu

Ian Briggs
University of Utah
ianbriggsutah@gmail.com

Ganesh Gopalakrishnan
University of Utah
ganesh@cs.utah.edu

Dong H. Ahn
Lawrence Livermore National Laboratory
ahn1@llnl.gov

ABSTRACT

Result reproducibility in societally important simulations faces the grave risk of becoming voided when code is ported across hardware platforms or optimized using different compiler flags. There are currently no available tools that can help a programmer assess the extent of such non-portability through systematic testing. In this work, we offer a novel testing framework called FLiT that comes with a large collection of predesigned tests, and also test automation facilities, including Makefile generation, distribution of the tests to CPU/GPU platforms, and result amalgamation/query supported through SQL queries. We also develop an intuitive 2D "heat-map" style visualization of how much a given compiler and its flag(s) can affect a given program. FLiT is an easy-to-use and community extensible framework, allowing the tool to grow in versatility as more tests are contributed and support for newer compilers is added. Our results span three architectures (including GPUs), four popular compilers, and dozens of compiler optimization flags. We demonstrate the power of FLiT's test automation and result analysis/visualization support. For the first time, for example, FLiT's systematic exploration was able to discover the following results: (1) when seeking higher performance through flags, compilers may apply unsafe algebraic simplifications, yielding up *six* or more different answers in some test cases; (2) compilers are inconsistent in the number of answer variants; in this ranking, Intel's `icpc` comes first, followed by GCC, then Clang, and finally NVCC.

CCS CONCEPTS

•Software and its engineering → Software testing and debugging; *Compilers; Software maintenance tools; Software evolution;*

KEYWORDS

architecture for high-end computing, programming environment, reproducibility, floating-point arithmetic, compiler flags, compiler optimization, platform variations

ACM Reference format:
Geof Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, and Dong H. Ahn. 2017. FLiT: Cross-Platform Floating-Point Result Consistency Aid. In *Proceedings of ACM Symposium on High-Performance Parallel and Distributed Computing , Washington D.C., June 2017 (HPDC'17),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

*FLiT tool website at http://www.cs.utah.edu/fv/FLiT

1 INTRODUCTION

Trust in computed (floating-point) results must not be left to chance, especially in the light of evidence [28] that critical programs such as the Community Earth Simulation Model have yielded inconsistent results large enough to suggest wrong scientific conclusions when ported across platforms and compilers. Similarly, errors in floating-point calculations have invalidated some of the experimental results from the Large Hadron Collider [3].

This paper takes a well-known problem for which systematic solutions are lacking: *are results reproducible across platforms, and when compilers and their optimization flags are changed?* The following real-world bug story highlights this problem and also may serve as a warning for similar problems waiting to happen in the exascale era where platform and compiler diversity are bound to increase. In a recent project [27], an attempt to port some of the MPI processes to run on the Xeon-Phi architecture while leaving others running on the Xeon architecture caused a curious deadlock that took days to debug. The root cause was the Xeon-Phis calculating the number of messages to be sent (through an expression $\lfloor p/c \rfloor$) differently from how the Xeons calculated the number of messages to be received (also governed by $\lfloor p/c \rfloor$). Unfortunately, the developers had not applied due precautions to their compilation flags, resulting in 63 messages being sent but only 62 attempted to be received, which then caused the deadlock. The following two factors made this error much more difficult to debug: (1) The use of two different hardware platforms to support the same computation. (2) Overlooked differences in floating-point arithmetic between the Intel Xeon and Xeon Phi Coprocessor x100 product [9] and the lack of precautions (either via compiler flags or the nature of the conditional expression used).

In this paper, we focus on *flag-induced variability* and offer a versatile framework called FLiT[1]. By "flag-induced" we mean a combination of two facets: (1) hardware platform variations: for example, CPU, Coprocessor, GPU, and in future FPGA-based custom hardware; compilation flags: for example, scalar and vectorizing compilers,[2] their flags, and subtle differences between flags that might either be not documented or easily glossed over while reading. We focus on the compiler story in greater detail, as there are no

[1]"Floating-point Litmus Tester" and results that may "flit" (flutter) about.
[2]For example, vectorization of reduction loops can build reduction trees instead of sequentially aggregating the results.

hard and fast rules governing compiler optimization flags and what they mean across compilers. Even similarly named flags can have slightly different meanings with compilers and also may apply differently to different programs.

Aren't "small errors" safe? Decades of research in numerical analysis, multigrid methods, and traditional kinds of uncertainty quantification have all helped enhance trust, providing HPC researchers and practitioners with a huge array of tools and techniques to help them trust numerical approximation schemes based on sound scientific principles. But when it comes to real-world experiences where we port code across two platforms or compilers and see different answers, *very little of this knowledge helps.* At this stage, our complacency born out of our daily experience with floating-point math tends to give a sense of reassurance saying "things are OK." For instance, both `(100+(1/3)) - x` and `(100+(1/3)) - y` may yield `100.0` in Python for `x` and `y` being `.333333333333333` and `.333333333333334` respectively, and almost everyone moves on without much worry, attributing things to floating-point rounding. Unfortunately, in a complex parallel program, variability of this magnitude may not just be flag-induced; there could also be many more causes [7]: (1) hardware bugs, (2) compiler bugs [37], (3) departures from ISA spec [16], (4) data races [1], (5) bit-flips, (6) result reassociation (floating-point arithmetic is not associative), or (7) human misinterpretation of documentation. FLiT can (1) either help a scientist be proactive, and document their code with the possibility of non-portability under certain compilers and flags, or (2) upon seeing very little evidence of flag-induced non-portability, suggest to the scientist that these other causes may be at work.

What about parallelism? FLiT is a testing framework that can work with equal ease for both sequential and parallel code. The tool already supports OpenMP and we plan to add support for MPI soon. However, the main focus of FLiT is to target basic compiler optimizations acting on sequential program units. Inconsistencies on sequential algorithms amplify when deployed in parallel across many CPUs.

Parallelism can introduce two additional sources of non-determinism, namely result variation due to operation re-association (e.g., in parallel reductions) and data races. It is therefore highly recommended that before using FLiT, users employ tools that can determinize operations using other tools and techniques (e.g., [5, 13, 19, 30] as well as eliminate data races using tools such as [1]. Our FLiT page has more information, including instructions for obtaining the source code: http://www.cs.utah.edu/fv/FLiT/

1.1 Specific Contributions

Our main contributions are the following:

- A detailed description of how we designed tests that demonstrate flag-induced variability. Our approach enables the community to critically examine and extend our tests, thus allowing FLiT to *grow in its incisiveness and coverage* as newer platforms and compilers emerge.
- We offer the downloadable and easy-to-use FLiT tool that comes with many pre-built tests, test automation support, and result assembly/analysis support.
- The approach we have developed to display at an intuitive level the overall amount of variability exhibited by a compiler across the space of tests and flags.

We now elaborate on these aspects.

Test Design. We have equipped FLiT with multiple classes of tests which were developed through the following approach:

- A class of tests are based on the basics of floating-point rounding, and widely discussed rounding issues [10, 15, 29].
- Another class of tests are based on compiler insights. Some compilers, notably the *open source compiler* GCC, serve as exemplars for other compilers. Specifically, newer compilers (a notable example being Clang) imitate GCC's flag names and internal optimization steps to some extent.[3] Some of these optimizations go beyond what little the IEEE standard says about "unsafe" compiler optimizations [18].

  Therefore, we have designed a class of tests that specifically respond to many of GCC's unsafe flags being turned on. The hope is that these tests will prove to be similarly effective across other compilers, as well.
- We adapted a test called *paranoia* [21] to increase our coverage further. The paranoia test was originally developed by William Kahan in 1983, one of the first leading researchers in floating-point arithmetic. This test has a wide following, and has been ported to multiple platforms for testing the implementation of floating-point arithmetic. We have discovered that this test provides useful insights even for studying flag-induced variability. A key feature of this test is that it establishes "milestones" ranging from 1 to 221. Each milestone is crossed after a series of complicated calculations and assertions. The test has been modified to report the first milestone when a mistake occurred. We found potential infinite loops due to optimization flags and detect this through a robust timeout mechanism. We show that by varying the compiler flags, this test can attain fewer milestones showing various forms of floating-point variability. Such tests provide another excellent metric of how much disruption platforms and flags can cause.
- The extent of floating-point rounding is determined by the magnitude of the operands. Sometimes, it is the overall high magnitudes that matter; at other times, it is the alternation of the magnitudes; at other times, the presence of subnormal numbers matter (e.g., some compilers may abruptly round results). We have come up with methods for input generation that include targeted *fuzzing* methods with user-selectable levels of test generation as well as a choice of randomization methods (random in the floating-point number scale versus in the real-number scale).

Test Administration Automation. Administering a collection of tests on a collection of platforms, administering the flags, finding out and eliminating tests that crash, and gathering the results all requires automation that FLiT provides in an easily extensible manner. FLiT comes with a powerful facility that includes: (1) Automated test dispatch to a cluster of nodes; (2) Makefile generation in a platform-specific manner; (3) Infrastructure to ship results back for analysis.

[3] Less is known about closed-source compilers such as `icpc` although such imitation may be happening there as well.

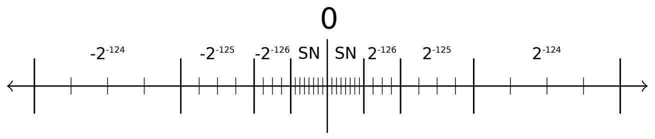

Figure 1: 32-bit floating-point numbers along the real number line. There are 2^{21} values between small tick marks. Subnormals are represented by SN.

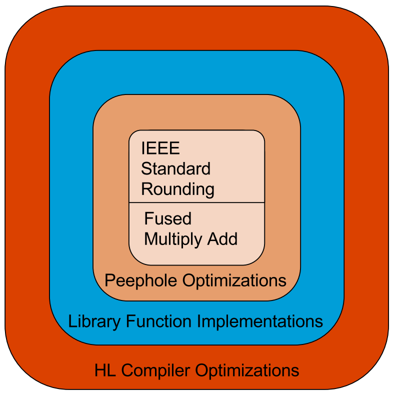

Figure 2: Compiler-Centric View of Floating-Point Optimizations

A feature of FLiT's test architecture is that it extensively uses modern C++ facilities, thus permitting easy test addition and adaptation. As a case in point, GPU tests are written no differently than CPU tests. This addresses a growing community of researchers who are worried about result portability between CPUs and GPUs.

Result Assembly and Analysis Facilities. We equip FLiT with a powerful result amalgamation and query facility through an SQL interface. We also provide a "heat map" visualization of the overall nature of inconsistencies under a given collection of tests and flags. Our results spanning three architectures (including GPUs), four popular compilers, with dozens of compiler optimization flags exercisable indicates that FLiT's test automation can significantly reduce the burden of assessing code non-portability.

Roadmap. We first present an overview of how compiler optimizations treat floating-point expressions, present our testing methodology and FLiT's architecture in §2. We then describe our testing results, including basic tests, GPU tests, compiler insight-based tests, and paranoia tests in §3. Visualizations (2D heat maps) of how four popular compilers (one for GPU) treat floating-point optimizations are presented in §4. Concluding remarks, including related work and future prospects are presented in §5.

2 METHODOLOGY

2.1 Background: Compiler-level Effects

General Overview of Floating-point Arithmetic. Figure 1 depicts the familiar floating-point number scale where the representable numbers are classified into subnormal numbers (the range of numbers not normally representable in the floating-point representation, AKA underflow) and normal numbers (all other representable floating-point numbers). The spacing between normal numbers doubles every successive binade (each increment of the exponent puts us in the next binade). The magnitude difference between 1.0 and the next higher representable floating-point number is called *machine epsilon* and is 2^{-23} for single precision and 2^{-52} for double precision. The *unit in the last place* (ULP) is a function that when given a real value x yields the distance between $|x|$ and the next representable value, going towards inf (see [17] for a full discussion). Whenever a computed value falls on the number line, it is snapped to the closest representable floating-point number, suffering a maximum of a *half ULP* error for *round to nearest.*[4]

There are many hardware-level departures from this rounding model. Given the preponderance of addition and multiplication, many platforms support *fused multiply add* (FMA) where one less rounding step is performed. Almost all these discussions apply equally to scalar instruction sets and vector instruction sets. In new instruction sets, there are beginning to be approximated versions of many more operations. In [2], Intel has introduced fast but approximate reciprocal and square-root vector instructions that operate in seven cycles but guarantee only 38 of the 53 mantissa bits.

Floating-point Arithmetic Viewed wrt Compilation. Viewed from the point of view of a source program that enters a compiler, the overall picture of rounding is much more involved. Figure 2 coarsely portrays the overall machinery so that we can then talk about the flags and their effects more concretely:

A user program enters through the outer-most ring where it is subject to an optional series of optimizations such as vectorization. Here, the floating-point effects of loops may be changed. We have observed the `icpc` compiler turning a linear addition (reduction) tree into a (somewhat) balanced vector addition tree. Whether this optimization is applied or not may depend on the loop trip count and other factors.

Next, the code penetrates the blue ring, where library functions (e.g., for `sin`, `exponentiation`, etc.) may be treated in one of several ways. It may be that `sin` is approximated by a ratio of polynomials; or a specific "fast path" or special case rule (e.g., $sin(\pi)$) may be invoked.

The third ring (peephole optimizations) portrays the kinds of optimizations we have observed where a compiler substitutes pure sub-expressions in lieu of their L-values, triggering algebraic simplifications. For instance, under certain circumstances, x/y may be changed to $x \cdot (1/y)$ to permit the use of the reciprocal instruction.

At the inner gray core, we portray standard instructions, many of which guarantee a half-ULP error bound under the *round to nearest* rule (many more rounding modes are available; see [17] for details). As discussed earlier, the fused multiply-add optimization is supported by many CPU and GPU types. The general rule is that if the `fma` flag is applied and the hardware supports FMA, then the optimization *may* happen depending on whether the optimization is deemed productive-enough (for the source program under consideration), by the compiler.

[4]Approximate division and square-root supported by many platforms does not guarantee this error bound, but runs much faster.

The flags we administer in our experiments try to exercise as many of the options in these rings shown in Figure 2. Tools such as FLiT will become even more important when compiler versions and associated flags that activate the aforesaid types of fast and approximate instructions become available.

2.2 Defining Litmus Tests

By a litmus test, we mean a specific program fragment and associated data, i.e., a pair (p, d) such that given a compiler and a platform, a *deterministic* answer is produced (we do not include non-deterministic tests). As testing is the crux of FLiT, we cover this aspect under three headings: (1) classification of litmus-tests (the types of programs/idioms p we choose) and the scoring method; (2) kernels included (the actual programs p chosen for each type of program) and reasons; and (3) input fuzzing approach (the amount and spread of data values d that are administered).

2.2.1 Classification of Litmus-tests and Scoring. The crux of all testing is to generate inputs meaningfully and have an oracle (ground truth) that judges the success of testing. Generally, our preference is to come up with tests that demand the least amount of effort from users while also maximizing impact. This means (1) ask users for as few (or no) data inputs, and (2) provide an easy means to obtain the *ground truth*. These goals are not easy to achieve while also offering tests that offer sufficient coverage. Below, we describe three classes of tests: *fixed parameter tests*, *fixed input tests*, and *fuzzed-input tests* that progressively ask more of the user.

- Some tests are fixed parameter tests, meaning that we fix the input once and for all using some rigid parameters, and let the tool generate instances under this class. Examples in this space are triangles of a given base (b) and height (h) whose area is the known ground-truth ($(b \cdot h)/2$). FLiT then automatically generates a family of triangles by "*shearing*" a reference triangle of this base and height into a family of triangle variants. It then uses various standard formulae for calculating the area (e.g., Heron's formula $\sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}$). We then assign a natural *score* by calculating the area of the sheared triangles and finding out the error with respect to the ground-truth.
- A few tests are designed with fixed inputs. For example, we take a convex hull calculation program and feed it a polygon with a large set of vertices that were precalculated with a random input generator. A good property of such tests is that there is a discrete answer (namely, the number of vertices in the computed convex polygon) that serves as a score.
- We also have tests that fuzz inputs until a result-difference is manifested. These tests again avoid examining the whole space of inputs, and rather focus their attention on an objective function. One example is: for a given flag, generate a collection of K inputs $i_1, \ldots, i_K$ such that under these inputs, the output scores are all pairwise distinct. These tests are generated using one of two input-randomization methods elaborated in §2.2.3.

The nearly fifty tests that FLiT comes with fall into the aforesaid categories. We now detail some of the specific tests included and our reasons for choosing them.

2.2.2 Kernels Included, and Reasons. As discussed earlier, our list of litmus tests include fixed parameter tests, fixed input tests and fuzzed input tests. We now provide some examples of these tests included in our collection.

Test Name	Uniform Reals	Uniform FP
DistributivityOfMultiplication	0	4,896
DoHariGSBasic	0	197
DoHariGSImproved	0	134
TrianglePHeron	0	12,888
TrianglePSylv	0	12,637

Figure 3: Divergent inputs count of 100,000 attempts for floats comparing flag `-O0` with `-O3 -funsafe-math-optimizations` using GCC. Uniform reals are uniform random samples on the real number line projected onto FP space. Uniform FP is where each FP is equally probable.

Fixed Parameter Tests. These include the following:

- A test *TrianglePHeron* employs Heron's formula itemribed on Page 4.
- The test *TrianglePSylvie* is contributed in [6]. In our testing, this test exhibited variability different from Heron's approach in terms of flags (detailed in §4).

Fixed Input Tests: Many of our tests are of the fixed input variety where input fuzzing does not play a significant role in exhibiting variability. Some examples include these:

- *SimpleCHull* implements the convex hull test with points a, b, c, d, e described on Page 4. This test can be run with a well-chosen set of initial points a, b, c, d, e.
- In *DoOrthoPerturbTest*, we choose two vectors that are orthogonal, with one vector along the x axis and another along the y axis (based on a fixed input). We then rotate one of the vectors by small increments, computing the dot products along the way, and sum the dot-products. The vectors are rotated by 200 ULPs per dimension.
- In *DoHariGSBasic/DoHariGSImproved*, we employ an implementation of the Graham Schmidt orthonormalization process.[34] Again, we seed these tests with a fixed input.

Fuzzed input tests. In these tests, we fuzz the inputs as described in §2.2.3. Specific tests in this family include *DistributivityOfMultiplication_idxN* for various N. This family of tests exercise the distributivity property often exercised by compilers that transform an expression of the form $(a \cdot b) + (a \cdot c)$ to an expression $a \cdot (b + c)$. Exhibiting a test outcome due to flags required fuzzing the inputs.

2.2.3 Input Fuzzing Approach. In order to exhibit a sufficient amount of result variability, we must vary the data component d of a test (p, d) to cover all relevant regions of the floating-point number scale. For example, if the variability is due to the *abrupt underflow* [17] rule that flushes all subnormals to zero,[5]then clearly we must have one input that is a subnormal and another that is a normal number.

In order to assist with testing, we created an input generator tool that generates floating-point inputs randomly and executes selected tests until a specified number of inputs have been found to cause divergent outputs. The way this is done in FLiT is to compile two versions of the test, one into the executable and the

[5]In [12], a discussion of when abrupt rounding matters is provided.

other into a shared library. These binaries can be run within the same process to compute the result from the test, given random inputs in both versions. This mode of testing can be applied when comparing two different flag choices as to whether they cause a result difference—for instance `-O0` and some set of optimization flags (e.g. `-O3 -funsafe-math-optimizations`).

This testing method is driven by two user-given inputs: (1) the desired number of divergence-causing inputs D, and (2) the maximum number of random inputs generated before the testing process gives up, R. The tool then generates random inputs $d_1, d_2, \ldots$ and keeps checking the scores generated corresponding to these inputs for two compiler flag variants. Whenever the score changes due to a new input d_k (across the flag variants), then a divergence-causing input has been obtained, thus meeting one of the desired Ds. This process repeats till D diverging inputs are generated or D exceeds R (a cap on the randomization process).

The main question to resolve now is how the d_i values are chosen. The first thing that comes to mind is to generate random samples *uniformly over the real number line*, but restricted within the representable range of positive floating-point values (e.g., 0 to 1.17549×10^{38} for float). (It should be noted that common random number generators, such as *std::rand()*, distribute uniformly over the real number line.] This "random according to the reals" has the property that many more higher values of reals get snapped to the same floating-point number as magnitudes grow. Thus, from the floating-point number perspective, the coverage is *highly skewed* toward higher magnitudes of floating-point numbers (from Figure 1, it can be seen that the spacing between consecutive floating-point values increases as we get further away from zero). Figure 3 shows that this method of randomization is poor for exhibiting divergence, the reason being the very low number of low-magnitude floating-point numbers it manages to generate.

We therefore choose the (natural) alternative, which is to choose floating-point values in an equally likely manner. Projected back to real numbers, this would mean that many more low-magnitude reals will be chosen. The way this is done is to generate a random bit pattern, reinterpret that bit pattern as a floating-point number, and if the number is not within the desired range, we do it again. We discard NaNs in the process.

Figure 3 shows that the equiprobable floating-point number approach of randomization is much more effective than the approach of equiprobable reals in a given range. Through this method, we generate a healthy dose of low-magnitude floats and also a sufficient number of "medium" values that prove to be "high enough." Previous works have studied different randomization approaches to floating-point value generation (e.g., [14]); however, this study in the context of flag-induced variability detection is believed to be new.

2.3 Putting it all together: FLiT Tool

Figure 4 shows the overall organization of FLiT. As described earlier, it takes a specification of compilers, flags, and hosts, and a collection of tests. The results of running tests is reflected in scores. The FLiT tool performs many compilations from all possible combinations of compilers and compiler flags to generate a set of compiled binaries

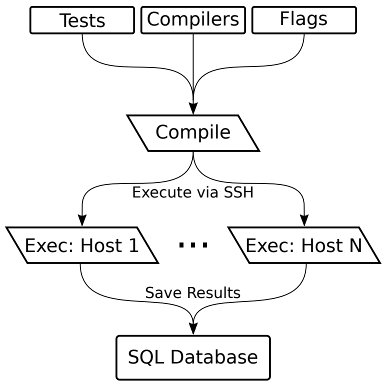

Figure 4: FLiT tool workflow

for each host, with each binary containing all tests. The set of possible compiler flags is limited to all combinations of an optimization level (one of -O0, -O1, -O2, or -O3) and a single floating-point flag from Figure 5.[6]

After this compilation phase, the binaries are copied to the hosts. After executing each binary on their respective host, the results are saved in an SQL database. This database can later be mined through a versatile set of queries.

3 EXPERIMENTAL RESULTS

In this Section, we present our experimental results to show the effectiveness of FLiT in detecting compiler-induced variability on an x86_64 Intel architecture. These experiments were performed using GCC 5.2, Clang 3.8, Intel compiler 16.0 and CUDA 7.5

3.1 Summation/Dot-product

Summation and dot-product are two centerpieces of HPC, involved in one way or the other in a large number of routines. We introduce two wrinkles: choose those algorithms that have in-built error compensation steps which compilers are known to remove by employing algebraic simplifications that are true of integers and real numbers but not floating-point numbers [15]. We study the Shewchuk algorithm for accurate summation of floating-point values [32] that offers this possibility for compiler optimization.

Dot-product invites the possibility to involve fused multiply-add (FMA), raising another set of interesting possibilities.[7] These tests [24] demonstrate the efficacy of FMA for dot-product computation. We choose three dot-product algorithm variants: (1) *langDotFMA* that is a naïve FMA based dot-product, (2) *langCompDot* that is a compensating dot-product without FMA, and finally (3) *langCompDotFMA* which employs compensation and FMA.

3.1.1 Shewchuk Summation Algorithm. The Shewchuk summation algorithm (Algorithm 1) was developed specifically to implement arbitrary precision numbers using floating-point numbers. It

[6]In fact, *applying multiple flags* can sometimes make a difference (and in some cases, the order in which certain flags are applied can make a difference). Given the huge number of flags, we do not currently go through the powerset of flag combinations. This is left for future work.

[7]Incidentally, the error in climate simulation mentioned in §1 resulted from FMA that, while known to increase accuracy, curiously caused departure from previously trusted simulation results obtained without FMA.

Figure 5: All used floating-point flag combinations

Flag	GCC	Clang	Intel	NVCC
`-fassociative-math`	X	X		
`-fcx-fortran-rules`	X			
`-fcx-limited-range`	X		X	
`-fexcess-precision=fast`	X	X		
`-fexcess-precision=standard`		X		
`-ffinite-math-only`	X	X		
`-ffloat-store`	X	X	X	
`-ffp-contract=on`	X	X		
`-fma`			X	
`-fmerge-all-constants`	X	X	X	
`-fno-trapping-math`	X	X		
`-fp-model fast=1`			X	
`-fp-model fast=2`			X	
`-fp-model=double`			X	
`-fp-model=extended`			X	
`-fp-model=precise`			X	
`-fp-model=source`			X	
`-fp-model=strict`			X	
`-fp-port`			X	
`-freciprocal-math`	X	X		
`-frounding-math`	X	X	X	
`-fsignaling-nans`	X	X		
`-fsingle-precision-constant`		X	X	
`-ftz`			X	
`-funsafe-math-optimizations`	X	X		
`-march=core-avx2`		X	X	
`-mavx`	X	X	X	
`-mavx2 -mfma`	X	X	X	
`-mfpmath=sse -mtune=native`	X	X	X	
`-mp1`			X	
`-no-fma`			X	
`-no-ftz`			X	
`-no-prec-div`			X	
`-prec-div`			X	
`--fmad=false`				X
`--fmad=true`				X
`--ftz=true`				X
`--prec-div=false`				X
`--prec-div=true`				X
`--prec-sqrt=false`				X
`--prec-sqrt=true`				X
`--use_fast_math`			X	X

has been proven to be precise within the range of representable numbers of the floating-point type that is used [32]. Because 32-bit floating-point can store decimal values with up to 7.2 digits of precision, performing 10^7 additions of number 1 is guaranteed to result in 10^7. In order to demonstrate the difference between Shewchuk and naïve summation, we chose to the sequence $[10^8, 1, -10^8, 1]$. Under real arithmetic, the answer should be 2. The Shewchuk algorithm successfully shows the sum to be 2.0 while the naïve summation yields 0.0.

Using the FLiT tool on a x86_64 architecture, the Shewchuk summation algorithm was tested against the Intel compiler, GCC, and Clang with combinations of optimization levels with a single chosen flag. Of all of those combinations, only one compiler flag made

Algorithm 1 Shewchuk Summation Algorithm

```
procedure SHEWCHUKSUM(values)
    $partials \leftarrow []$
    for each $x$ in values do
        $newPartials \leftarrow []$
        for each $y$ in partials do
            $y, x \leftarrow$ smallerFirst($x, y$)
            $hi \leftarrow x + y$
            $lo \leftarrow y - (hi - x)$
            if $lo \neq 0.0$ then
                $newPartials$.append($lo$)
            end if
            $x \leftarrow hi$
        end for
        if $x \neq 0.0$ then
            $newPartials$.append($x$)
        end if
        $partials \leftarrow newPartials$
    end for
    return exactSum($partials$)
end procedure
```

a difference. When compiled with flag `-funsafe-math-optimizations` applied under GCC, the algorithm becomes as bad as the naïve implementation. Neither the Intel compiler nor the Clang compiler upset the Shewchuk algorithm.

When compiled with GCC with an optimization level higher than `-O0` and with the flag `-funsafe-math-optimizations`, the algorithm is ruined and returns the incorrect value of zero, just like the naïve sum. Not only does the algorithm return the same performance as the naïve approach, but even after optimization, it proved to be far more inefficient during execution. Details of this situation are discussed in §3.3.1 where, without compiler insight and reading the binary, we were unable to understand what happened.

3.1.2 Langlois Compensated Dot-Product. The algorithms in this section employ *Error Free Transformations* studied in past work (e.g., TwoSum [23] and TwoProd [11]). This approach generates compensation terms (ϵ), in addition to the sum and product. We chose these tests for their importance in the HPC community and to demonstrate that they are fragile to some degree and vulnerable to the effects of different compiler configurations. Figure 6 summarizes the extent of variability for these algorithms (we present the number of equivalence classes of scores).

Algorithm 2 LangDotFMA (naïve FMA based dot-product)

```
procedure LANGDOTFMA(a,b)
    $sum \leftarrow []$
    $sum[0] \leftarrow a[0] \times b[0]$
    for $x$ in $0..N$ do
        $sum[x] \leftarrow fma(a[x], b[x], sum[x-1])$
    end for
    return $sum$
end procedure
```

Algorithm 3 LangCompDot (compensating dot-product, without FMA)

```
1: procedure LANGCOMPDOT(a,b)
2:     $sum \leftarrow [], comp \leftarrow []$
3:     $\epsilon_{sum} \leftarrow 0, \epsilon_{prod} \leftarrow 0, prod \leftarrow 0$
4:     $sum[0], comp[0] \leftarrow TwoProd(a[0], b[0])$
5:     for $x$ in $0..N$ do
6:         $prod, \epsilon_{prod} \leftarrow \text{TwoProd}(a[x], b[x])$
7:         $sum[x], \epsilon_{sum} \leftarrow \text{TwoSum}(prod, sum[x-1])$
8:         $comp[x] \leftarrow comp[x-1] + (\epsilon_{prod} + \epsilon_{sum})$
9:     end for
10:    return $sum[N-1] + comp[N-1]$
11: end procedure
```

Algorithm 4 TwoSum Algorithm

```
1: procedure TwoSum(a, b)
2:     $sum \leftarrow a + b$
3:     $T \leftarrow sum - a$
4:     $\epsilon \leftarrow (a - (sum - T)) + (b - T)$
5:     return $sum, \epsilon$
6: end procedure
```

Algorithm 5 TwoProd Algorithm

```
1: procedure TwoProd(a, b)
2:     $product \leftarrow a \cdot b$
3:     $\epsilon \leftarrow \text{FMA}(a, b, -product)$
4:     return $product, \epsilon$
5: end procedure
```

Figure 6: FMA / Compensating Dot-Product Equivalence Class Count by Precision

Test Name	float	double	long double
langDotFMA	1	1	3
langCompDot	2	4	1
langCompDotFMA	6	5	4

3.2 Preliminary Study of Variability in GPUs

3.2.1 Support for GPU Testing in FLiT. Modern HPC relies on a variety of heterogeneous platforms to perform computation, and GPU coprocessing is a popular choice. While GPUs provide thousands of cores per unit, developers must decide whether they may trust results provided by these alternate computation mechanisms. An early study in this regard [36] sheds light on the difficulties of porting critical medical imaging software from CPUs to GPUs.

We provide GPU versions of most of our litmus-tests, along with some basic math support (for vector and matrix math, for instance). Additionally, adding a GPU test requires the developer to override a virtual method which is a CUDA kernel. After this, the test distribution and collection facilities will populate results in the FLiT database.

3.2.2 Optimizations Supported. NVidia [35] supports four optimizations in floating-point computation, with one 'fast math'. These are denormal handling: *discard and flush to zero*, division and square root: *use fast approximations*, fma contraction: *use FMA instruction* and IEEE round to nearest.

	fast math library	flush to zero	precise divide	precise square root	fused multiply-add	IEEE round to nearest
`ftz=true`		X				
`ftz=false`						
`prec-div=true`			X			X
`prec-div=false`						
`prec-sqrt=true`				X		X
`prec-sqrt=false`						
`fmad=true`					X	
`fmad=false`						
`use-fast-math`	X				X	

Clearly, the `fast-math` flag causes the most variability, as borne out by our experiments.

3.3 Tests Designed Through Compiler Insights

In order to make more targeted tests that leverage compiler effects, we utilized the open source nature of gcc. Internally gcc has an optimization structure whereby operations are handled one at a time and each handler decides, based on global flags and static analysis, whether a given optimization is allowable.

For instance, if the `hypot` function is used in the source to calculate the hypotenuse of a triangle, there are three classes of optimizations that can be used. (1) If there are sign altering functions applied to the arguments, such as `-` or `abs`, they are always removed; (2) If the compiler can know that one of the arguments is `0`, then the function always is replaced with `fabs` of the other argument. (3) If the compiler is allowed unsafe math optimizations and it knows that both arguments are equal, then the value is replaced with $|x|\sqrt{2}$.

The ability for the compiler to know these conditions can change according to inlining and other optimization enabled. Also, altering flags that control inlining and other optimization can affect floating-point results, even though these flags are not inherently directly related to floating-point arithmetic.

We now discuss how the insights we drew from `GCC` helped us understand why Shewchuk's algorithm discussed in §3.1.1 was rendered incorrect by GCC's optimization.

3.3.1 Unintended Optimization by `GCC`. Given that hi is assigned $x + y$ and lo is assigned $y - (hi - x)$, `GCC` first transformed lo to $hi - hi$. However, we were very surprised that lo was not replaced by `0.0` on line 8 in Algorithm 1. Had this been done, compiler could have eliminated the statement `if` $lo \neq$ `0.0`.

The reason was that since associative math is turned on, the resulting value of lo can become `NaN` if the value of hi becomes `inf` (infinity). The net effect is that the compensation in the algorithm is removed. Based on this investigation, we concluded that with `-funsafe-math-optimizations`, the algorithm was rendered functionally equivalent to the naïve sum in returning `0`, and yet performed twice the number of operations!

3.3.2 Summary of Tests Based on Compiler Insights. We now summarize all tests that were derived thanks to our insights into `GCC`.

Reciprocal Math. We discovered that under many circumstances, `GCC` will look for code sequences of the following kind, and upon seeing three uses of division by `m`, will first compute the reciprocal of `m` and then multiply it with `a` through `d`. Thus, we can derive a score as follows (in figure 6), and observe a difference with respect to `-O0`.

Compile-time Versus Runtime Evaluation. The following test (Figure 7) reveals our understanding of how a compiler's behavior may be affected by constant propagation, as well as compile-time and run-time decisions.

Algorithm 6 ReciprocalMath Test

```
1: procedure ReciprocalMath(a,b,c,d)
2:     $a \leftarrow a/m$
3:     $b \leftarrow b/m$
4:     $c \leftarrow c/m$
5:     $d \leftarrow d/m$
6:     return $a + b + c + d$
7: end procedure
```

Algorithm 7 SinInt Test

```
1: procedure SinInt
2:     $zero \leftarrow (rand()\%10)/99$
3:     $pi \leftarrow 3.14159265358979...L$
4:     return $sin(pi + zero)/sin(pi) - 1$
5: end procedure
```

Under IEEE assumptions, this should always be zero. Under most non-IEEE assumptions this should be zero, since both calls to sine are given the same number.

However, the difference arises from constant propagation, allowing `std::sin(pi)` to be evaluated at compile time, but the other call to `sin` is left to the runtime!

Any difference between the compile time and runtime implementations of `sin` near *pi* will therefore manifest in the test result. A result we have obtained from this code is $-3.30261141e-05$ for the `double` type.

3.4 Paranoia Tests

Figure 7: How many flag combinations stopped at each milestone in the Paranoia test.

Milestone	ICPC	GCC	CLANG
10	0	9	0
30	279	0	0
50	4	11	18
121	0	2	2
150	55	64	70
Goal: 221	106	127	138
Total	444	213	228

3.4.1 Overview. Paranoia is a full suite of tests developed by William Kahan in 1983 to verify floating-point arithmetic specified by the IEEE floating-point standard draft at the time. We have inserted this test suite as a single test inside of the FLiT framework. Although these tests were originally intended to test implementation against the IEEE standard, we use this well established suite to look for compiler-induced variability due to optimizations.

Because of its size, the Paranoia test suite has been separated into milestones that range from 1 to 221. The test was modified to stop at the milestone of the first detected test point failure. These milestones serve as stumbling blocks for the optimizer. We focus only on these stumbling block milestones which can bee seen in Figure 7 for each of the tested compilers.

3.4.2 Results. Each milestone in Figure 7 represents a different type of test failure. The failure on milestone 10 indicates a loop construct executing infinitely due to unsafe optimizations performed by GCC with the `-funsafe-math-optimizations` flag. The Intel compiler was the only one blocked at milestone 30, which is symptomatic of inconsistencies induced by using higher precision representations for intermediate computations. Stoppage at milestone 50 was caused by a large number of flags. The assertion at milestone 50 checks the so-called "sticky bit" and asserts $(1.75 - \epsilon) + (\epsilon - 1.75) = 0$ where ϵ is machine epsilon. Milestone 121 fails if $(x \neq z)$ and $(x - z = 0)$. Milestone 150 has a series of complicated computations involving logarithms, the pow function, multiplication, and division. There are many optimizations that break this assertion in different ways.

This test coupled with the FLiT tool allow us to gain valuable insights into compilers and their numerous flags. Certainly optimizations are desired for improved performance, yet often we unknowingly sacrifice reproducibility or necessary guarentees.

4 VISUALIZING FLIT RESULTS

Figures 8, 9, 10, and 11 present our visualizations for NVCC, Clang, GCC and ICPC (respectively). These plots were automatically generated by running queries against the SQL database of results populated after we run FLiT on a collection of tests using these compilers and flags.

We now explain our conventions in these plots, taking ICPC (Figure 11) as an example. This figure consists of flags on the y axis running from `-O0` going through the list of flags, which are listed alphabetically. The x axis labeling at the top mentions some of the tests in FLiT (the ones that had more than one result under the current compiler).

We now present the logic followed in each column (for definiteness, consider the column `TrianglePSylv`, figure 11. At `-O0`, the color starts at black and ranges through white (a 'hot' color scheme). At `-O3` (the second row from the top), the color changes to red (hotter), indicating that for the test `TrianglePSylv`, the result changed. Then going down, the color stays red until `-fmerge-all-constants -O3`, indicating that thisnew test result holds till there. When we switch to `-fp-model=double -O3`, a new test result (hotter, yellow) is obtained; this result holds also for `-fp-model=extended -O3` at yellow. At `-fp-model fast=1 -O2`, the color reattains red (the same test result as with the earlier reds is obtained, so we reuse the color).

At `-fp-model=precise -O3`, another test result is obtained (hotter, white), and this stays till `-fp-model=strict -O3`. The

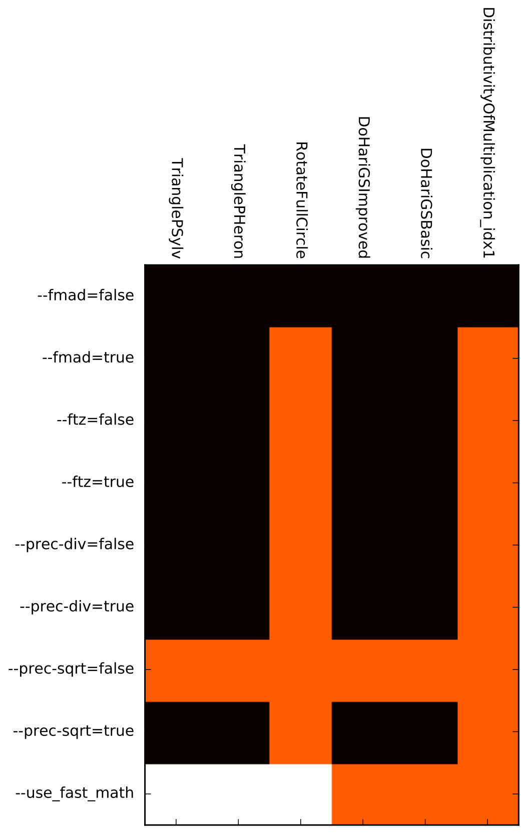

Figure 8: Visualization of test divergences under NVCC

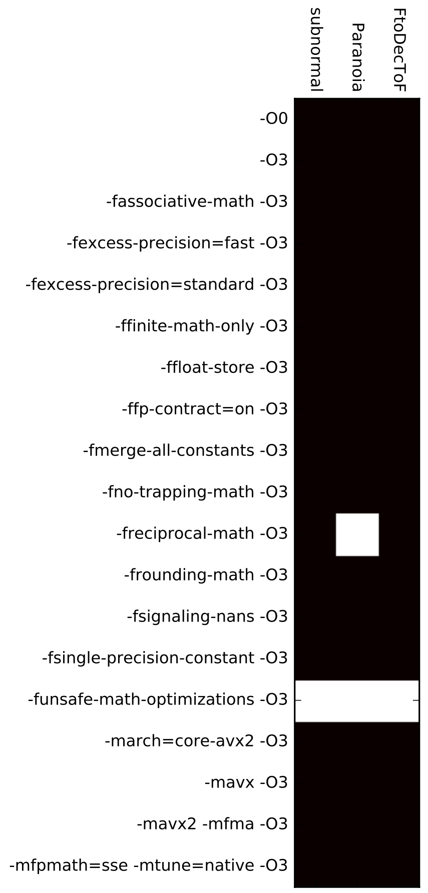

Figure 9: Visualization of test divergences under Clang

rest of this column may be read similarly. Altogether, this column contains four colors (thus the flags present managed to express four different results).

Across columns, there is no relationship, except that by staring at the color at one row across two columns, one can tell which test (in which column) is more incisive. If, at row y_k, if the column at x_i has a color that is hotter than the color at column at x_j, we can then say that the number of flags administered from y_0 till y_k caused the color to get hotter at test i than at test j—meaning that test i has so far shown more accumulated extent of variability.

5 CONCLUDING REMARKS

We now present a few key related pieces of work and also sketch some ideas for future work.

Related Work. In general, reproducibility has received a significant amount of attention [25, 26, 33]. This paper and its ideas were greatly inspired by the excellent empirical study called *Deterministic cross-platform floating-point arithmetics* by Seiler [31], a study done in 2008 but does not seem to have been continued.

The possibility of cross-platform portability leading to wrong scientific conclusions being drawn is raised in [28]. They also release a tool KGEN [22] that extracts *computation kernels* from a program in order to study them in isolation from the larger system. A complementary approach that the CESM team took was using *Ensemble-based consistency* [4]. This involves using an ensemble, or collection of runs that simulate the same climate mode, using randomly perturbed initial conditions. This way, a signature of the model is generated, represented as a distribution of the observed model states. This signature is compared to the state of subsequent runs where the validity is unknown, such as after switching platforms or adding features to the code base. We envisage our work helping with efforts such as KGEN in helping prioritize one's explorations based on flags known to cause the highest amounts of variability.

There have been publications that describe the general lack of portability across architectures and platforms such as from Intel [10] and Microsoft [15] and also pertaining to specific programming languages such as Java [20]. Intel reports on compiler reproducibility [10] as it relates to performance and cross CPU compatibility. They characterize their own fast math flag as follows: "The variations implied by *unsafe* are usually very tiny; however, their impact on the final result of a longer calculation may be amplified if the algorithm involves cancellations." This extends to Fortran and MPI, with ANSI Fortran allowing re-association that obeys parentheses, and MPI having routines that are not guaranteed to be exact or reproducible.

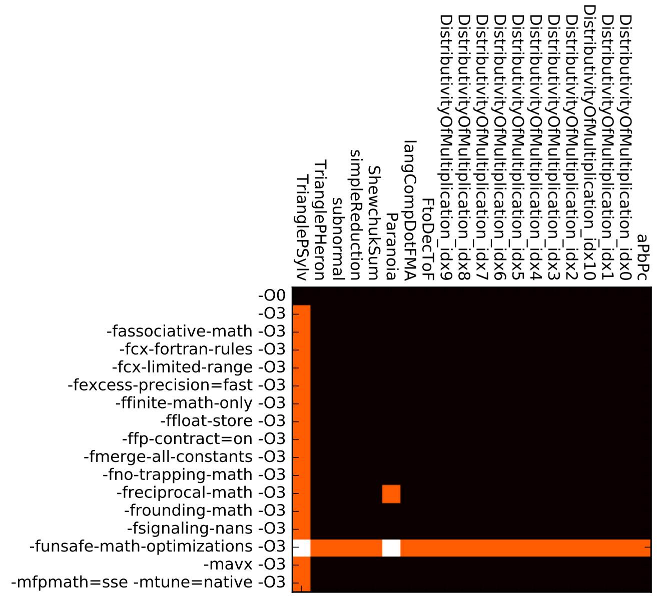

Figure 10: Visualization of test divergences under GCC

For transcendentals, Intel makes a modest effort toward compatibility with the flag `no-fast-transcendentals`, but they have no real guarantees relating to its use cross core or cross system. In the end, Intel recommends that if reproducibility is desired, the best one can do is have portability across different processor types of the same architecture. While the use of *fp model precise* is more reproducible, even here there are no guarantees.

There have been many efforts that address the general lack of robustness in floating-point computations. Bailey's high precision math [3] work addresses ill-conditioned situations by offering multiple higher precisions. The MPFR library [29] is another effort offering higher precision when necessary.

Conclusions and Future Work. We offer the first empirical testing tool FLiT that can check for cross-platform portability of user codes. Given that floating-point arithmetic is counterintuitive for most application scientists, and given that there is a lack of compiler standardization, the need for a tool such as FLiT in forewarning users is believed to be a facility that will be increasingly valuable as compiler and platform diversity increases during the exascale era.

The testing method described in §3.3 depended on our being able to read the source of a compiler and then design tests with this insight. This is undoubtedly tedious and error prone. A much more convenient and scalable approach will be for compiler writers to provide such tests, especially given that reproducibility is growing in importance.

Our future plans include aggressively launching a collection of kernels from HPC researchers so that we can maintain a well calibrated collection of kernels that exhibit variability. It may then become possible to pattern-match a given user program and flag the presence of these kernels to provide an earlier warning to application scientists as to the portability of their code.

We envisage another interesting practical application for our 2D visualizations, given that compilers keep evolving, and the newer version of a compiler may render a users' delicately hand-crafted floating-point expressions ineffective as to its rounding-error control. If the users produce a 2D visualization across two compilers and produce a "diff," they may be able to quickly tell whether some of their idioms are likely to be broken.

REFERENCES

[1] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016.* IEEE Computer Society, 53–62. DOI: http://dx.doi.org/10.1109/IPDPS.2016.68

[2] avx-512 2015. Intel Architecture Instruction Set Extensions Programming Reference. (2015). https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf

[3] David H Bailey and Jonathan M Borwein. 2013. High-Precision Arithmetic: Progress and Challenges. (2013). www.davidhbailey.com.

[4] A.H. Baker, D.M. Hammerling, M.N. Levy, H. Xu, J.M. Dennis, B.E. Eaton, J. Edwards, C. Hannay, S.A. Mickelson, R.B. Neale, D. Nychka, J. Shollenberger, J. Tribbia, M. Vertenstein, and D. Williamson. 2015. A new ensemble-based consistency test for the community earth system model. 8 (2015), 2829fi?!2840. doi:10.5194/gmd-8-2829-2015.

[5] Pavan Balaji and Dries Kimpe. 2013. On the reproducibility of MPI reduction operations. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on.* IEEE, 407–414.

[6] Sylvie Boldo. 2014. *Deductive Formal Verification: How To Make Your Floating-Point Programs Behave.* Thèse d'habilitation. Université Paris-Sud. http://www.lri.fr/~sboldo/files/hdr.pdf

[7] F. Cappello, Emil M. Constantinescu, Paul D. Hovland, T. Peterka, C. L. Phillips, M. Snir, and S. M. Wild. 2015. Improving the Trust in Results of Numerical Simulations and Scientific Data Analytics. (2015).

[8] Michelle Connolly (Ed.). 2016. *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.* Procedia Computer Science, Vol. 80. Elsevier. http://www.sciencedirect.com/science/journal/18770509/80

[9] M Corden. 2013. *Differences in floating-point arithmetic between Intel R Xeon R processors and the Intel R Xeon PhiTM coprocessor.* Technical Report. Technical report, Intel.

[10] Martyn J Corden and David Kreitzer. 2009. *Consistency of floating-point results using the intel compiler or why doesnfit my application always give the same answer.* Technical Report. Technical report, Intel Corporation, Software Solutions Group. https://software.intel.com/sites/default/files/article/164389/fp-consistency-102511.pdf.

[11] Theodorus J. Dekker. 1971. A Floating-Point Technique for Extending the Available Precision. *Journal of Numerical Mathematics* 18, 3 (1971), 224–242.

[12] James Demmel. 1984. Underflow and the reliability of numerical software. *SIAM J. Sci. Stat. Comput.* 5 (1984), 887–919.

[13] James Demmel and Hong Diep Nguyen. 2015. Parallel Reproducible Summation. *IEEE Trans. Computers* 64, 7 (2015), 2060–2070. DOI: http://dx.doi.org/10.1109/TC.2014.2345391

[14] Allen Downey. 2007. Generating Pseudo-random Floating-Point Values. (2007). http://allendowney.com/research/rand/.

[15] Eric Fleegal. 2004. Microsoft Visual C++ Floating-Point Optimization. *Microsoft Corp* (2004). https://msdn.microsoft.com/en-us/library/aa289157(v=vs.71).aspx.

[16] Patrice Godefroid and Ankur Taly. 2012. Automated synthesis of symbolic instruction encodings from I/O samples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 441–452. DOI: http://dx.doi.org/10.1145/2254064.2254116

[17] David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.* 23, 1 (March 1991), 5–48. DOI: http://dx.doi.org/10.1145/103162.103163

[18] IEEE Standard for Floating-Point Arithmetic 2008. IEEE Standard for Floating-Point Arithmetic. (2008). https://standards.ieee.org/findstds/standard/754-2008.html.

[19] Intel MPI Library Conditional Reproducibility 2012. Intel MPI Library Conditional Reproducibility. (2012). https://goparallel.sourceforge.net/wp-content/uploads/2015/06/PUM21-3-Intel_MPI_Library_Conditional_Reproducibility.pdf.

[20] William Kahan. 2004. How Javafis Floating-Point Hurts Everyone Everywhere. (2004). https://people.eecs.berkeley.edu/~wkahan/JAVAhurt.pdf.

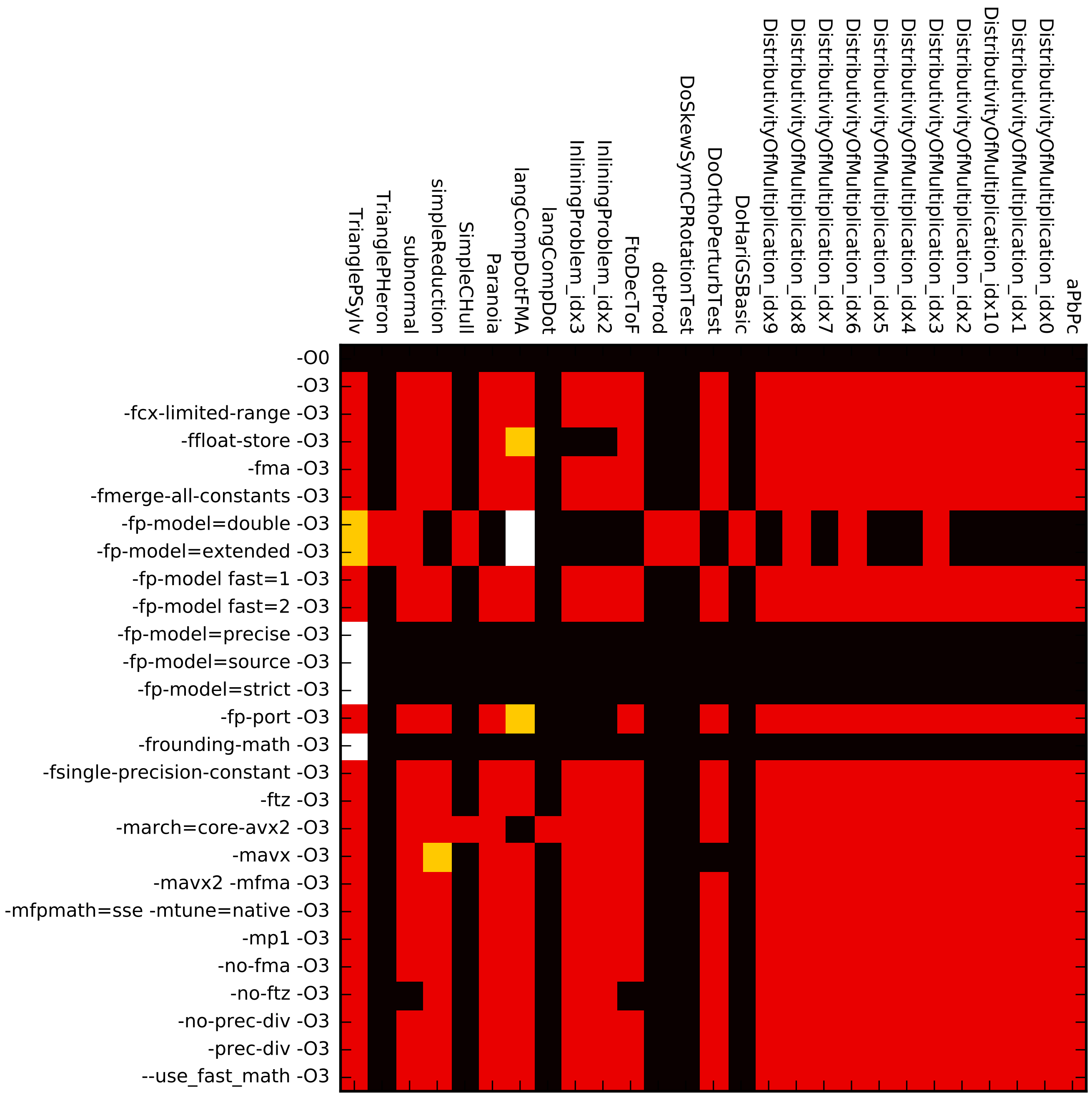

Figure 11: Visualization of test divergences under ICPC

[21] Richard Karpinski. 1985. Paranoia-A Floating-Point Benchmark. (1985).

[22] Youngsung Kim, John M. Dennis, Christopher Kerr, Raghu Raj Prasanna Kumar, Amogh Simha, Allison H. Baker, and Sheri A. Mickelson. 2016. KGEN: A Python Tool for Automated Fortran Kernel Generation and Verification, See [8], 1450–1460. DOI: http://dx.doi.org/10.1016/j.procs.2016.05.466

[23] Donald E. Knuth. 1998. *The Art of Computer Programming: Seminumerical Algorithms.* Addison-Wesley.

[24] Philippe Langlois and Nicolas Louvet. 2006. Operator dependant compensated algorithms. In *Scientific Computing, Computer Arithmetic and Validated Numerics, 2006. SCAN 2006. 12th GAMM-IMACS International Symposium on.* IEEE, 2–2.

[25] Miriam Leeser, Saoni Mukherjee, Jaideep Ramachandran, and Thomas Wahl. 2014. Make it real: Effective floating-point reasoning via exact arithmetic. In *DATE 2014.* 1–4.

[26] Miriam Leeser and Michela Taufer. 2016. Panel on Reproducibility at SC'16. (2016). http://sc16.supercomputing.org/presentation/?id=pan109&sess=sess177.

[27] Qingyu Meng, Alan Humphrey, John Schmidt, and Martin Berzins. 2013. Preliminary Experiences with the Uintah Framework on Intel Xeon Phi and Stampede. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery (XSEDE).* 48:1–48:8.

[28] Daniel Milroy, Allison H. Baker, Dorit Hammerling, John M. Dennis, Sheri A. Mickelson, and Elizabeth R. Jessup. 2016. Towards Characterizing the Variability

of Statistically Consistent Community Earth System Model Simulations, See [8], 1589–1600. DOI: http://dx.doi.org/10.1016/j.procs.2016.05.489

[29] MPFR 2016. The GNU MPFR Library. (2016). www.mpfr.org.

[30] Kento Sato, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, and Martin Schulz. 2015. Clock Delta Compression for Scalable Order-replay of Non-deterministic Parallel Applications. In *SC*. 62:1–62:12.

[31] Christian Seiler. 2008. (2008). http://christian-seiler.de/projekte/fpmath/.

[32] Jonathan Richard Shewchuk. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry* 18, 3 (1997), 305–363.

[33] M. Taufer, O. Padron, P. Saponaro, and S. Patel. 2010. Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs. In *IPDPS*. 1–9.

[34] The Graham-Schmidt Process 2006. The Graham-Schmidt Process. (2006). http://mathworld.wolfram.com/Gram-SchmidtOrthonormalization.html.

[35] Nathan Whitehead and Alex Fit-Florea. 2012. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. (2012). Presented at GTC 2012.

[36] Devon Yablonski. 2011. *Numerical accuracy differences in CPU and GPGPU codes*. Master's thesis. Northeastern University. http://www.coe.neu.edu/Research/rcl/theses/yablonski_ms2011.pdf.

[37] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. DOI: http://dx.doi.org/10.1145/1993498.1993532