FLiT: Cross-Platform Floating-Point Result-Consistency Tester and Workload

Geof Sawaya*, Michael Bentley*, Ian Briggs*, Ganesh Gopalakrishnan*, Dong H. Ahn†
*University of Utah, †Laurence Livermore National Laboratory

***Abstract*—Understanding the extent to which computational results can change across platforms, compilers, and compiler flags can go a long way toward supporting reproducible experiments. In this work, we offer the first automated testing aid called FLiT (Floating-point Litmus Tester) that can show how much these results can vary for any user-given collection of computational kernels. Our approach is to take a collection of these kernels, disperse them across a collection of compute nodes (each with a different architecture), have them compiled and run, and bring the results to a central SQL database for deeper analysis. Properly conducting these activities requires a careful selection (or design) of these kernels, input generation methods for them, and the ability to interpret the results in meaningful ways. The results in this paper are meant to inform two different communities: (a) those interested in seeking higher performance by considering "IEEE unsafe" optimizations, but then want to understand how much result variability to expect, and (b) those interested in standardizing compiler flags and their meanings, so that one may safely port code across generations of compilers and architectures. By releasing FLiT, we have also opened up the possibility of all HPC developers using it as a common resource as well as contributing back interesting test kernels as well as best practices, thus extending the floating-point result-consistency workload we contribute. This is the first such workload and result-consistency tester underlying floating-point reproducibility of which we are aware.**

I. Introduction

There is ample evidence that result variations caused by compiler flags and architectural heterogeneity lead to a serious productivity cost, in addition to undermining trust in HPC simulations. In the Community Earth Simulation Model [1], a fused-multiply-add introduced by a compiler prevented the scientists from obtaining reliable results. Architectural heterogeneity caused result inconsistency across MPI communications, leading to a deadlock [2] whose root-cause was non-obvious for a week.

In the ideal world, HPC applications enjoy bitwise reproducibility even after being optimized using differing compilation flags or ported across CPU and GPU platforms. Clearly, the presence of parallelism affects the reduction order of routines, and the inherent non-determinism in applications also prevents bit-wise reproducibility. But even assuming fully deterministic and sequential applications, bitwise reproducibility is almost impossible to achieve in practice, unless one is willing to avoid optimizations heavily.

Almost all realistic compilers provide higher optimization levels (e.g., "`-O3`") and also some "IEEE unsafe" optimization flags that can together bring about a $5\times$ factor of speed-up (see Figure 5)—but also change the results. This is too much performance gain to pass up, and so one must embrace result-changes in practice. However, exploiting such compiler flags is fraught with many dangers. A scientist publishing a piece of code with these flags used in the build may not really understand the extent to which the results would change across inputs, platforms, and other compilers. For example, the optimization flag `-O3` does not hold equal meanings across compilers. Also, some flags are exclusive to certain compilers; in those cases, a user who is forced to use a different compiler does not know which substitute flags to use. Clearly, tool-support is needed to automatically check HPC routines for portability—the goal of this work.

In one sense, despite the floating-point numbers in the result changing, what finally matters is the "overall application semantics." Unfortunately, this notion is ill-defined: the extent of result variability tolerated by a social network graph clustering tool may not be the same as that tolerated by a piece of climate simulation code. Thus, anyone building a tool to gauge the extent of result variability across compilers and platforms cannot tie the tool too deeply to a particular application class. Their best hope is also to draw *representative kernels* from different applications, and populate the tool with such kernels. The tool can display the extent of result variability manifested by various compilers, flags, and platforms, and leave the final interpretation of the exact numbers to the domain scientist who cares about the domains from which those kernels come from. The kernel model can also make a tool that gauges variability run faster, not burdened by the pressure of provisioning all the memory and I/O resources needed by actual applications.

Specific Contributions: In this paper, we contribute a tool called FLiT (Floating-point Litmus Tester) that is meant to help those who seek to exploit result variability as well as to avoid it (to the extent possible). Our main contributions are the following:

- A detailed description of how we designed our litmus 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 as well as an initial workload (extensible by the community) to assess result-reproducibility[1].

[1] FLiT tool: pruners.github.io/flit, code at github.com/PRUNERS/FLiT

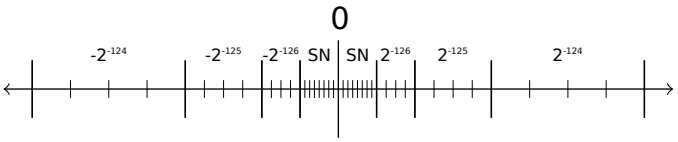


Fig. 1: 32-bit floating-point numbers along the real number line. There are 2^{21} values between small tick marks. Subnormals are represented by SN.

- 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.

Roadmap: We first present an overview of how compiler optimizations treat floating-point expressions along with our testing methodology and FLiT's architecture in §II. We then describe our testing results, including basic tests, GPU tests, compiler insight-based tests, and Paranoia tests in §III. Paranoia is one of the six tests highlighted in the authoritative compilation Handbook of Floating-Point Arithmetic [3]. Visualizations (2D heat maps) of how four popular compilers (one for GPU) treat floating-point optimizations are presented in §IV. Concluding remarks, including related work and future prospects are presented in §V.

II. Methodology

A. 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 (a fixed-point representation at the smallest exponent supported by floating-point), 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 ∞ (see [4] for a full discussion). Whenever a computed value falls on the number line, it is snapped to either the nearest smaller or nearest larger representable floating-point number, suffering a maximum of a *half ULP* error for *round to nearest*.[2]

There are many hardware-level departures from this rounding model. Given the extensive use of addition and multiplication operations, many platforms support *fused multiply add* (FMA) where hardware implementation allows an add and a multiply to be done in a single step, resulting in one less rounding step. Almost all these discussions apply equally to scalar instruction sets and vector instruction sets. In new instruction sets, there are introduced approximated performant versions of many more operations. In [5], 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 of accuracy.

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

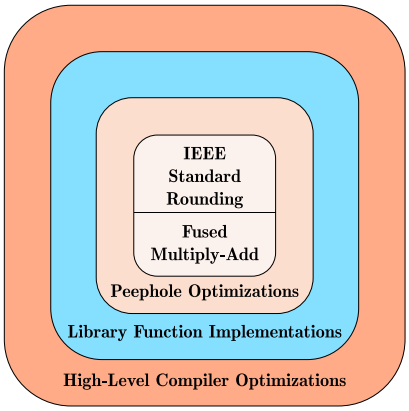


Fig. 2: Compiler-Centric View of Floating-Point Optimizations. The compiler moves from the outer layer to the inner layer, first performing higher-level optimizations, then acting on small groups of instructions, and finally optimizing single instructions. The library function implementation layer stands out because it does not modify the user's code.

Compiler Perspective of Floating-point Arithmetic: Viewed from the point of view of a source program that enters a compiler, the full picture of rounding is much more involved. Figure 2 coarsely portrays the overall machinery of compiler optimizations.

A user program enters through the outer-most layer 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 library function implementation layer where standard library functions (e.g., `sin` or `exp`) may be treated in a special way or may be replaced with a completely different implementation. For example, `sin` may be approximated by a ratio of polynomials; or a specific "fast path" special case rule (e.g., `sin(`π`)`) may be invoked.

The third layer (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 central section (IEEE Standard Rounding and Fused Multiply-Add), 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 [4] 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.

The flags we administer in our experiments try to exercise as many of the options in these layers 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.

B. 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).

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 test 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 ($(b \cdot h)/2$). The test 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. This score is the value returned to FLiT and stored in the database for later analysis.
- 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.

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

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.

Fixed Parameter Tests: These include the following:

- A test *TrianglePHeron* employs Heron's formula described on Page 3.
- 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 §IV).

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 3. 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 Gram-Schmidt orthoganalization [7]. Again, we seed these tests with a fixed input.

Fuzzed input tests: In these tests, we fuzz the inputs. 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.

C. Putting it all together: FLiT Tool

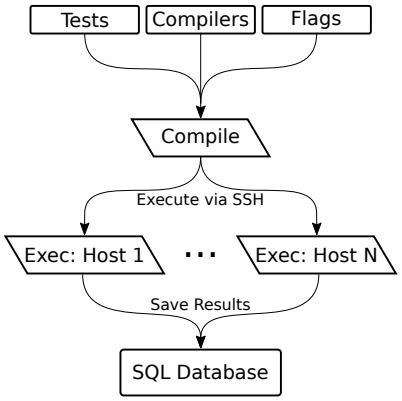


Fig. 3: FLiT tool workflow

Figure 3 shows the overall organization of FLiT. As described earlier, the FLiT tool receives a list of compilers, flags, and hosts, and a collection of tests. The results of running

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

TABLE I: All used floating-point flags

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 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 Table I. In this work, we chose to avoid combining multiple compiler flags. This choice allows (1) the search space to be significantly decreased and (2) the effect of each flag to be isolated to simplify analysis.[3]

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. There are stored routines and python scripts that help users plot and analyze the data stored in the database. However, the database is organized simply and an experienced SQL user can mine data directly from the database.

The FLiT tool intends to find reproducibility problems due to variability introduced by compiler optimizations or differences in architecture. It is important to note that variability can have other sources such as race conditions, randomization, or even hardware failures. The FLiT tool is intended to run on *deterministic* code. It is the responsibility of the user of FLiT to first ensure that the code run by FLiT is deterministic, perhaps using other tools. If this assumption of deterministic code fails, then problematic decisions may be made based on misleading information from FLiT.

[3] In fact, in some cases, the order in which certain flags are applied can affect the compilation.

III. Experimental Results

In this Section, we present our experimental results to show the effectiveness of FLiT in detecting compiler-induced variability on one CPU architecture and one GPU architecture. The hardware used an Intel Xeon CPU E5645 (Intel's x86_64 Westmere microarchitecture) and an NVidia GM200 graphics card (NVidia's Maxwell architecture). These experiments were performed using GCC 5.2, Clang 3.8, Intel compiler 16.0 and CUDA 7.5.

A. 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 [8]. We study the Shewchuk algorithm for accurate summation of floating-point values [9] that offers this possibility for compiler optimization.

Algorithm 1 Shewchuk Summation Algorithm

```
1:  procedure SHEWCHUKSUM(values)
2:      partials ← []
3:      for each x in values do
4:          newPartials ← []
5:          for each y in partials do
6:              y, x ← smallerFirst(x, y)
7:              hi ← x + y
8:              lo ← y − (hi − x)
9:              if lo ≠ 0.0 then
10:                 newPartials.append(lo)
11:             end if
12:             x ← hi
13:         end for
14:         if x ≠ 0.0 then
15:             newPartials.append(x)
16:         end if
17:         partials ← newPartials
18:     end for
19:     return exactSum(partials)
20: end procedure
```

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

numbers. It has been proven to be precise within the range of representable numbers of the floating-point type that is used [9]. 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 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.

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.

2) Langlois Compensated Dot-Product: The algorithms in this section employ *Error Free Transformations* studied in past work (e.g., TwoSum [10] 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. Table II 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)

```
1: procedure LANGDOTFMA(a,b)
2:     sum ← 0
3:     for i from 1 to N do
4:         sum ← FMA(a[i], b[i], sum)
5:     end for
6:     return sum
7: end procedure
```

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

TABLE II: Three dot product implementations along with how many different answers were obtained by varying the compilation, separated by precision.

B. Preliminary Study of Variability in GPUs

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 [12] 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.

2) Optimizations Supported: NVidia [13] 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.

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

Algorithm 3 LangCompDot (compensating dot-product)

```
1: procedure LANGCOMPDOT(a,b)
2:     sum, ε_sum, prod, ε_prod, comp ← 0
3:     for i from 1 to N do
4:         prod, ε_prod ← TWOPROD(a[i], b[i])
5:         sum, ε_sum ← TWOSUM(prod, sum)
6:         comp ← comp + (ε_prod + ε_sum)
7:     end for
8:     return sum + comp
9: end procedure
10: procedure TWOSUM(a, b)
11:     sum ← a + b
12:     T ← sum − a
13:     ε ← (a − (sum − T)) + (b − T)
14:     return sum, ε
15: end procedure
16: procedure TWOPROD(a, b)
17:     product ← a · b
18:     ε ← FMA(a, b, −product)
19:     return product, ε
20: end procedure
```

C. 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 $|x|\sqrt{2}$.

1) 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 algorithm 4), and observe a difference with respect to `-O0`.

Compile-time Versus Runtime Evaluation: The following test (Algorithm 5) 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 4 ReciprocalMath: cause reciprocal optimization

procedure RECIPROCALMATH(a,b,c,d,m)
  $a \leftarrow a/m$
  $b \leftarrow b/m$
  $c \leftarrow c/m$
  $d \leftarrow d/m$
  return $a + b + c + d$
end procedure

Algorithm 5 SinInt Test

procedure SININT
  $zero \leftarrow (\text{RAND}()\%10)/99$
  return $\text{SIN}(\pi + zero)/\text{SIN}(\pi) - 1$
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.

D. Paranoia Tests

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

TABLE III: How many flag combinations stopped at each milestone in the Paranoia test.

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 be seen in Table III for each of the tested compilers.

2) Results: Each milestone in Table III 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.

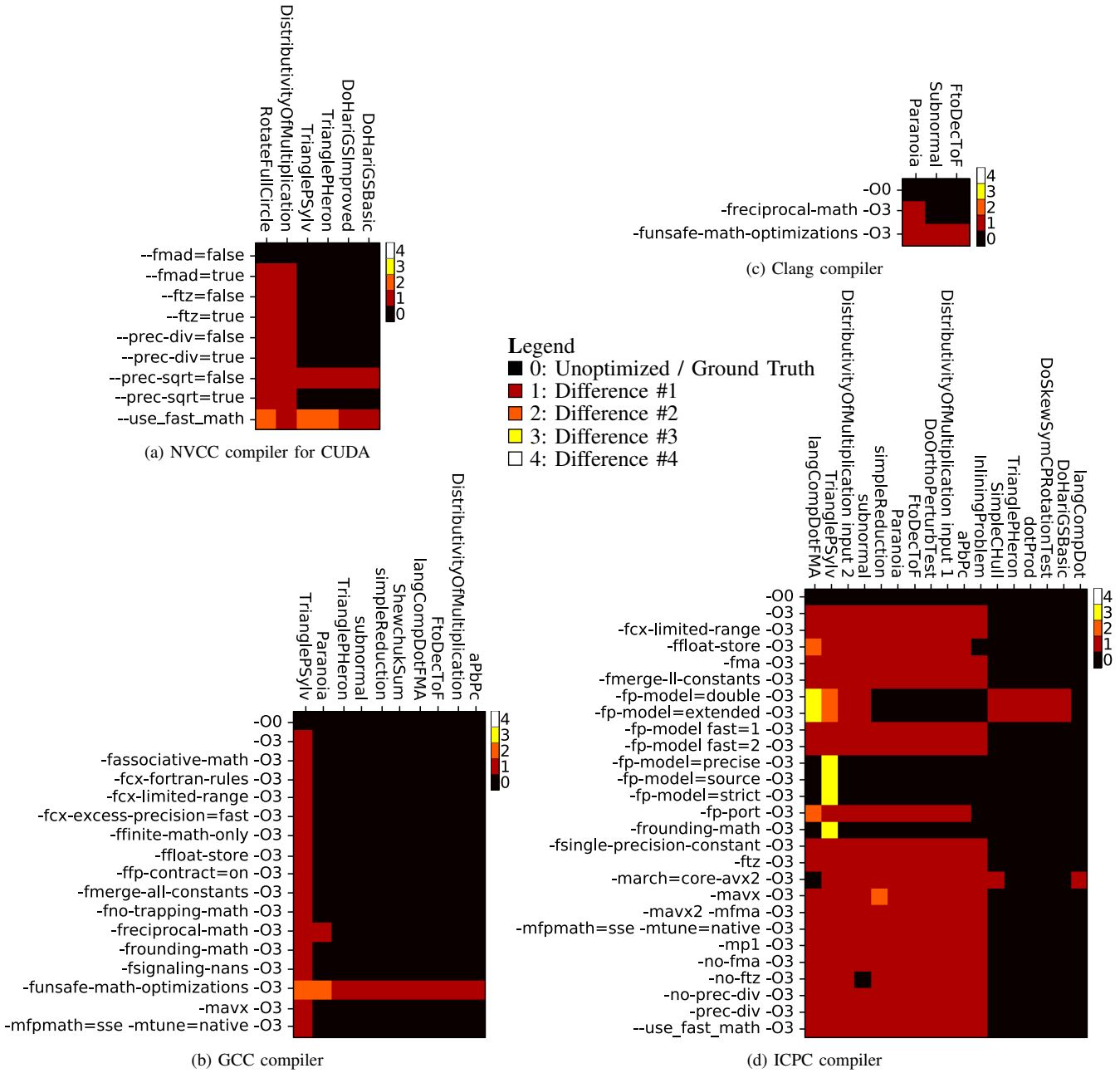
This test coupled with the FLiT tool allow us to gain valuable insights into compilers and their numerous flags. Certainly optimizations are desired for improved performance, yet often we unknowingly sacrifice reproducibility or necessary guarantees.

IV. VISUALIZING FLIT RESULTS

The Figures in 4 present our visualizations for NVCC, Clang, GCC and ICPC.

Figure 4 is limited to the unoptimized compilation (`-O0` with no flags) and all flags with the `-O3` optimization level. Those flags that showed no variability were removed from the plot. Also all tests that showed no variability were removed from the plot for conciseness. The results from the unoptimized compilation is considered the "reference answer" rather than the ground truth answer because, in general, it may not be the most accurate answer. For example, FMA provides a more accurate computation for $a \cdot b + c$ than the unoptimized computation.

Each column of a heat map is independent. For example, in Figure 4(d), with the `TriangePSylv` test, we see that column ranges from 0 to 3, meaning for that test code, and a static input value, the output from the algorithm ranged from four distinct values. The only variable here was the compilation flags given to the ICPC compiler. The different colors do not mean anything more than being different outputs from other colors. We do not encode here a "distance" from



(a) NVCC compiler for CUDA

(b) GCC compiler

(c) Clang compiler

(d) ICPC compiler

Fig. 4: Heat maps of compiler-induced variability. The horizontal axis of each heat map is which test was executed. The vertical axis is the compilation flags provided. In each case, the unoptimized version (e.g. "`-O0`") returns the reference answer shown in black. Anything not black gave a different answer than the unoptimized compilation. Note: each column is independently scaled since they are independent tests. This demonstrates how many different answers are achieved varying only compiler flags for fixed code and fixed algorithm inputs. Litmus tests and compilations with no variability are excluded from these heat maps. To keep the heat maps of reasonable size, only the unoptimized compilation and `-O3` are shown.

the true answer, which is shown on the figure as black. One may wonder how different these answers are. This difference information is stored in the database and can be mined by the user if desired. For these small litmus tests, the answers are very similar, which is not surprising considering their size. However, that misses the point we are presenting – that static code can be compelled to generate up to four different values simply by compiling it in different ways. It will be informative to investigate how many different answers a complex piece of real-world code is capable of generating.

Note that in Figure 4, the compilers are separated in their analysis. This is for two reasons: (1) some compiler flags do not apply to all compilers and (2) compiler flags and optimization levels do not have uniform meaning across compilers. We can compare compilers in isolation to gain larger trends. The Clang compiler in Figure 4(c) is the least aggressive with unsafe floating-point optimizations. The ICPC compiler in Figure 4(d) is the most aggressive and is aggressive in different

ways allowing for the four different behaviors by some of the tests. The GCC compiler in Figure 4(b) is primarily aggressive with the `-funsafe-math-optimizations`. That by itself is not as surprising as discovering that the other flags for GCC are bitwise reproducible for all but two litmus tests.

Only tests that showed variability are shown in Figure 4. Some litmus tests were expected to show variability but did not. These tests provide examples of code that, for these compilers and this architecture, is reproducible under all compilations. These tests are not analyzed in this paper. Users can download and experiment with which kinds of operations have reproducibility problems and which kinds do not.

There is another visualization that can be useful on an individual test basis. The FLiT tool is primarily an aid to help the developer navigate the tradeoffs between reproducibility and performance. In Figure 5(a), we took a single example litmus test, `TrianglePSylv`, and show the speedup compared with the unoptimized version of `gcc` (`-O0` with no other flags). There is a clear indication here of which compilations get the same answer as the unoptimized compilation (as seen with the blue dots). Using this chart, one can determine very easily the most performant compilation for this particular test that maintains reproducibility, `clang++ -O1 -fexcess-precision=fast`, clocked at a speedup of 2.66. To see that in comparison, the fastest non-reproducible compilation was `g++ -O3 -funsafe-math-optimizations` at a speedup of 4.69. This means that for this particular kernel, if we sacrifice reproducibility, we can get a speedup of 1.77 from the fastest reproducible compilation. This kind of analysis is invaluable to developers weighing the tradeoff between reproducibility and performance.

In the other graph, Figure 5(b), we see a similar speedup curve, but a very different reproducibility profile. In this particular case with the `TrianglePHeron` algorithm, the fastest compilation is also reproducible at a speedup of 5.86. This example is particularly interesting because (1) the fastest reproducible compilation is the faster than the fastest non-reproducible compilation (by a small margin), and (2) the fastest reproducible compilation was `clang++ -O3 -funsafe-math-optimizations` and the fastest non-reproducible compilation was `g++ -O2 -funsafe-math-optimizations`. Despite having the same compilation flags, `clang++` maintained reproducibility and `g++` did not. Looking back at Figure 4, one might conclude that the `clang` compiler is not as aggressive as `gcc` in optimizations, yet it can achieve similar performance, at least for the `TrianglePHeron` example, but not for the `TrianglePSylv` example.

V. 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 [14]–[16]. This paper and its ideas were greatly inspired by the excellent empirical study called *Deterministic cross-platform floating-point arithmetics* by Seiler [17], a study done in 2008 but does not seem to have been continued.

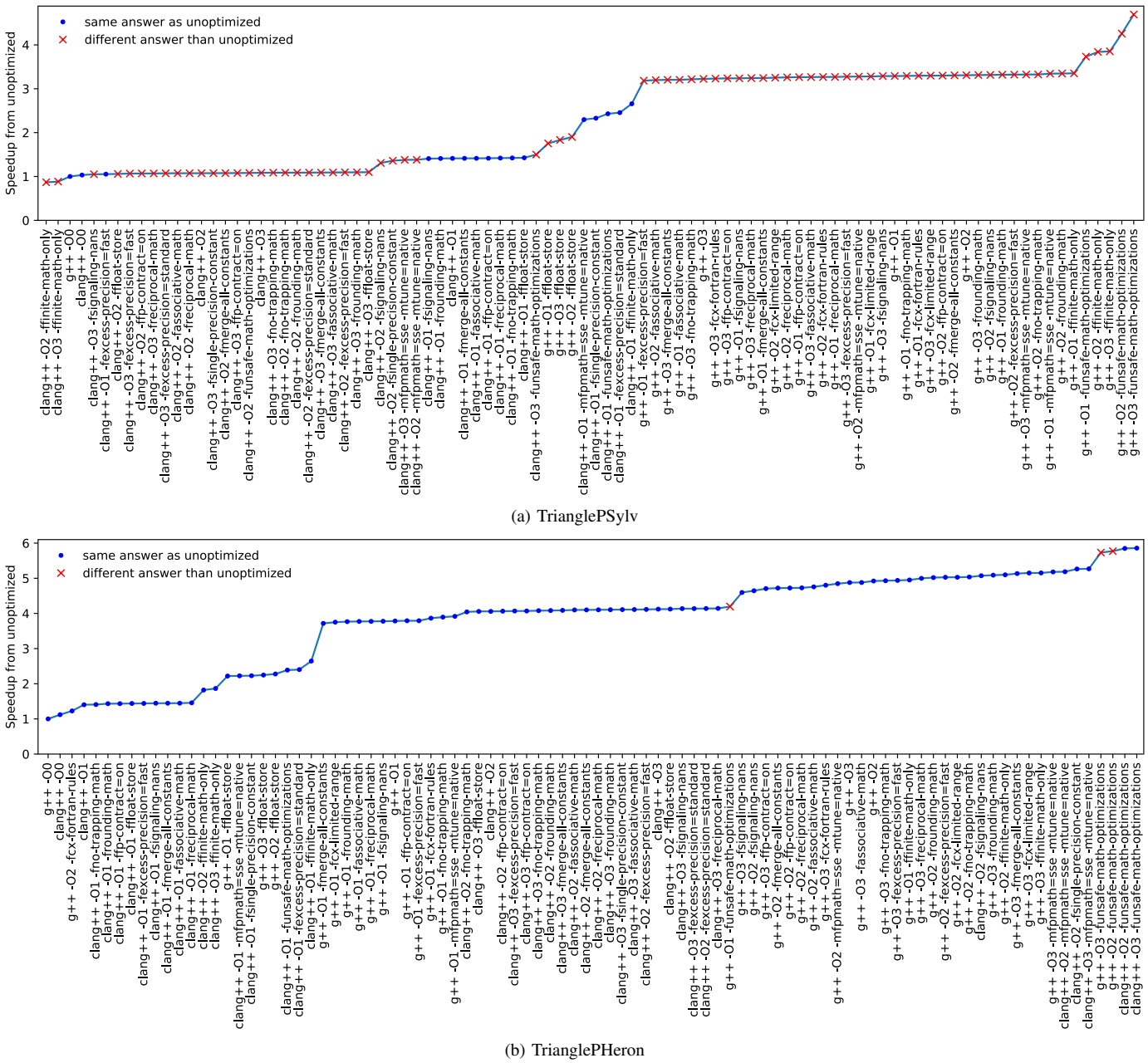
The possibility of cross-platform portability leading to wrong scientific conclusions being drawn is raised in [1]. They also release a tool KGEN [18] that extracts *computation kernels* from a program in order to study them in isolation from the larger system. A complementary approach that the CESM team took was using *Ensemble-based consistency* [19]. 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. We anticipate there to be eventually community-specific workloads as well as result-consistency assessment methods that can be derived from the FLiT framework and its initial workload.

There have been publications that describe the general lack of portability across architectures and platforms such as from Intel [20] and Microsoft [8] and also pertaining to specific programming languages such as Java [21]. Intel reports on compiler reproducibility [20] 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.

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 [22] work addresses ill-conditioned situations by offering multiple higher precisions. The MPFR library [23] is another effort offering higher precision when necessary.

Future Work: We plan to convert FLiT into a library that the applications can call into before they invoke a specific kernel hidden within the application. This way, the kernels themselves can be present in the context of the full application, and enjoy the inputs that the full application provides. FLiT can run the



(a) TrianglePSylv

(b) TrianglePHeron

Fig. 5: Speedup of two different algorithms for calculating the area of morphing triangles (using 32-bit floats) as compared to the unoptimized version using GCC. This plot also shows which ones are safe (with a blue dot) and which ones get a different answer from the unoptimized version (with a red x). Only `gcc` and `clang` were considered for this speedup comparison.

specific kernel embedded within the application under different optimizations, dynamically linking different binaries generated for it. There are two advantages to this approach: (1) avoiding the headache of pulling out a kernel and designing its input generators; (2) the ability to design application-specific result acceptance criteria, thus being able to move closer toward the goal of making a recommendation as to which flags are safe and which to avoid.

The testing method described in §III-C 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.

In ongoing work, we are investing significant effort in collecting and classifying kernels from HPC researchers so as to extend the existing workload of FLiT. This may also help 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.

This work avoided combining compiler flags in an effort to reduce the search space and to isolate the effect of individual flags. How compiler flags interact and influence one another is the subject of future work.

Finally, we have demonstrated how compilers may effect small coding examples called litmus tests. This allowed us to isolate specific types of variability invoked by the compiler optimizations. However, we are interested to see how the FLiT framework could be used to seek reproducibility of large real-world code bases. This investigation is ongoing.

Acknowledgments: This work was supported in part by NSF awards ACI 1535032, CCF 1421726 and 1704715.

REFERENCES

[1] D. Milroy, A. H. Baker, D. Hammerling, J. M. Dennis, S. A. Mickelson, and E. R. Jessup, "Towards characterizing the variability of statistically consistent community earth system model simulations," in *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, ser. Procedia Computer Science, M. Connolly, Ed., vol. 80. Elsevier, 2016, pp. 1589–1600. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2016.05.489

[2] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "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)*, 2013, pp. 48:1–48:8.

[3] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[4] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991. [Online]. Available: http://doi.acm.org/10.1145/103162.103163

[5] "Intel architecture instruction set extensions programming reference," 2015. [Online]. Available: https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf

[6] S. Boldo, "Deductive formal verification: How to make your floating-point programs behave," Thèse d'habilitation, Université Paris-Sud, Oct. 2014. [Online]. Available: http://www.lri.fr/~sboldo/files/hdr.pdf

[7] "The gram-schmidt process," 2006, http://mathworld.wolfram.com/Gram-SchmidtOrthonormalization.html.

[8] E. Fleegal, "Microsoft visual c++ floating-point optimization," *Microsoft Corp*, 2004, https://msdn.microsoft.com/en-us/library/aa289157(v=vs.71).aspx.

[9] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," *Discrete & Computational Geometry*, vol. 18, no. 3, pp. 305–363, 1997.

[10] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1998.

[11] T. J. Dekker, "A floating-point technique for extending the available precision," *Journal of Numerical Mathematics*, vol. 18, no. 3, pp. 224–242, 1971.

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

[13] N. Whitehead and A. Fit-Florea, "Precision & performance: Floating point and ieee 754 compliance for nvidia gpus," 2012, presented at GTC 2012.

[14] M. Leeser and M. Taufer, "Panel on reproducibility at sc'16," 2016, http://sc16.supercomputing.org/presentation/?id=pan109&sess=sess177.

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

[16] M. Leeser, S. Mukherjee, J. Ramachandran, and T. Wahl, "Make it real: Effective floating-point reasoning via exact arithmetic," in *DATE 2014*, 2014, pp. 1–4.

[17] C. Seiler, 2008, http://christian-seiler.de/projekte/fpmath/.

[18] Y. Kim, J. M. Dennis, C. Kerr, R. R. P. Kumar, A. Simha, A. H. Baker, and S. A. Mickelson, "KGEN: A python tool for automated fortran kernel generation and verification," in *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, ser. Procedia Computer Science, M. Connolly, Ed., vol. 80. Elsevier, 2016, pp. 1450–1460. [Online]. Available: http://dx.doi.org/10.1016/j.procs.2016.05.466

[19] A. Baker, D. Hammerling, M. Levy, H. Xu, J. Dennis, B. Eaton, J. Edwards, C. Hannay, S. Mickelson, R. Neale, D. Nychka, J. Shollenberger, J. Tribbia, M. Vertenstein, and D. Williamson, "A new ensemble-based consistency test for the community earth system model," no. 8, p. 28292840, 2015, doi:10.5194/gmd-8-2829-2015.

[20] M. J. Corden and D. Kreitzer, "Consistency of floating-point results using the intel compiler or why doesnt my application always give the same answer," Technical report, Intel Corporation, Software Solutions Group, Tech. Rep., 2009, https://software.intel.com/sites/default/files/article/164389/fp-consistency-102511.pdf.

[21] W. Kahan, "How java's floating-point hurts everyone everywhere," 2004, https://people.eecs.berkeley.edu/~wkahan/JAVAhurt.pdf.

[22] D. H. Bailey and J. M. Borwein, "High-precision arithmetic: Progress and challenges," 2013, www.davidhbailey.com.

[23] "The gnu mpfr library," 2016, www.mpfr.org.

[24] M. Connolly, Ed., *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, ser. Procedia Computer Science, vol. 80. Elsevier, 2016. [Online]. Available: http://www.sciencedirect.com/science/journal/18770509/80