# VARITY: Quantifying Floating-Point Variations in HPC Systems Through Randomized Testing

Ignacio Laguna
Lawrence Livermore National Laboratory
ilaguna@llnl.gov

***Abstract*—Floating-point arithmetic can be confusing and it is sometimes misunderstood by programmers. While numerical reproducibility is desirable in HPC, it is often unachievable due to the different ways compilers treat floating-point arithmetic and generate code around it. This reproducibility problem is exacerbated in heterogeneous HPC systems where code can be executed on different floating-point hardware, e.g., a host and a device architecture, producing in some situations different numerical results. We present VARITY, a tool to quantify floating-point variations in heterogeneous HPC systems. Our approach generates random test programs for multiple architectures (host and device) using the compilers that are available in the system. Using differential testing, it compares floating-point results and identifies unexpected variations in the program results. The results can guide programmers in choosing the compilers that produce the most similar results in a system, which is useful when numerical reproducibility is critical. By running 50,000 experiments with VARITY on a system with IBM POWER9 CPUs, NVIDIA V100 GPUs, and four compilers (gcc, clang, xl, and nvcc), we identify and document several programs that produce significantly different results for a given input when different compilers or architectures are used, even when a similar optimization level is used everywhere.**

***Index Terms*—floating-point, differential testing, random testing, numerical reproducibility**

## I. INTRODUCTION

We present VARITY, a framework to quantify floating-point numerical variations in supercomputing systems that provide multiple compilers and are composed of different floating-point hardware.

Floating-point arithmetic can be confusing, and it is sometimes poorly understood by programmers [1]. As heterogeneous systems become more pervasive, programmers face significant challenges when it comes to reasoning about floating-point behavior and ensuring floating-point reproducibility, portability, and numerical consistency. Not only can programs run in at least two different architectures in heterogeneous systems—a host architecture (e.g., x86) and an accelerator architecture (e.g., a GPU)—but also programs may be compiled with different compilers (a compiler that generates code for the host and a different compiler that generates code for the device). Since the floating-point arithmetic semantics in languages such as C are underdefined in the language specification, compilers have freedom in generating very different floating-point code between them for the same input program, sometimes generating code that is not strictly compliant with the IEEE 754-2008 Standard [2][1].

Given that heterogeneous HPC systems usually provide several compilers to their users, numerical reproducibility in these systems is a challenge. For example, the top two HPC systems, according to the top500 list[2], Summit and Sierra, offer users at least five different compilers to choose from: `gcc` [3], `clang` [4], IBM `XL` [5], `PGI` [6], and `nvcc` [7]. A given code can be compiled with compiler $x$ and run on the host, and compiled with compiler $y$ and run on a device. When users want to compare the numerical results that the compiled code produced on different architectures, a challenging question is what combination of compilers ($x$, $y$) would produce the most similar results in a given HPC system.

The above reproducibility problem is not unique of heterogeneous systems—a similar reproducibility problem also arises even if a program is executed only in a host architecture. Consider a program compiled with different compilers $x$ and $z$, or a program that is compiled with compiler $x$ and a library (used in the program) that is compiled with compiler $z$. Since $x$ and $z$ can be chosen from several options in modern HPC systems, and different compilers may optimize floating-point code in very different ways, even when the same optimization level is used everywhere (e.g., `-O2`), comparing the numerical results between the resulting codes can be challenging.

VARITY helps users of heterogeneous HPC systems to make an informed decision on choosing compilers when numerical reproducibility is critical, increasing the confidence that groups of compilers produce similar floating-point results in a system. VARITY uses randomized differential testing [8], a technique that uses random tests to find differences between two similar software components, to compare the numerical results of pairs of compilers and to identify pairs that produce similar and dissimilar numerical results for a given optimization level.

We specify the programs that VARITY can produce by a grammar that is a subset of the `C` language grammar. The grammar specifies the valid randomized tests that we can generate. Productions in the grammar have a certain probability of generation. After a test program is generated, it is compiled with multiple compilers in the same system using the same

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-793958).

[1]The most current version of the standard, IEEE 754-2019, was published in July 2019; however, it incorporates mainly clarifications and errata with respect to the IEEE 754-2008 version. In the rest of the paper, we will refer to IEEE 754-2008 version.

[2]https://www.top500.org/

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00070

optimization level. The program is then executed with random inputs. VARITY compares the results of the executions and reports to users the differences. Using the results, users of HPC systems can make a more informed decision on what compilers to use in their applications when floating-point numerical consistency and reproducibility are essential.

Using VARITY, we identify several interesting cases where two compilers in a system generate programs that produce very different numerical results for the same input, even when the lowest level of (or no) optimization is used. Sometimes the differences that we find can be as different as zero and 1e+300, or as `NaN` (not a number) and 0.1. The random programs and inputs that VARITY generates can serve as examples to programmers of *dangerous* patterns on which differences can occur in realistic programs if one is not careful.

In summary, our contributions are the following:

- We present the design of a randomized differential testing framework to quantify floating-point numerical differences in heterogeneous systems;
- We implement our design in the VARITY framework and use it to analyze floating-point variability on four compilers (`gcc`, `clang`, `XL`, and `nvcc`) and two architectures (IBM POWER9 and NVIDIA V100), using the Nas Parallel Benchmarks [9] to configure the tool. We present several examples of programs and inputs found by VARITY, which produce very significant floating-point variations;
- We present insightful findings on the evaluation of VARITY in the tested system (IBM POWER9 and NVIDIA V100). We find that: (a) when comparing results on the host architecture (i.e., IBM POWER9), `gcc` and `clang` tend to produce the most similar numerical results considering all the optimization levels we tested; for unoptimized code, however, `clang` and `XL` produce the most similar results. (b) when comparing results between device runs and host runs, `nvcc` executables (running on GPUs) versus `XL` executables (running on the host) tend to produce the most similar and consistent numerical results and are the most reproducible combination; (c) while `-O3` in *strict* mode (i.e., strict with respect to the IEEE 754-2008 standard) reduces the rate of inconsistent results in comparison to `-O3` *non-strict*, we do observe numerical variations with `-O3` *strict* in all combinations of compilers; in general, the most consistent numerical results across compilers are found when optimizations are disabled (`-O0`) and fused multiply-add (FMA) is disabled.

## II. BACKGROUND

In this section, we present background information that will be useful in understanding our approach. First, we present a real-world case of a floating-point reproducibility issue that motivates our work. Next, we discuss the general possible situations on which floating-point variations can occur in a given system. Then, we present examples of test programs that could produce such variations—VARITY found these examples after running thousands of random tests in an HPC system. Finally, we give a high-level overview of our approach.

```
116  const double gradv10 = 0.5*(q_gradv[ijN(1,0,2)]+
     q_gradv[ijN(0,1,2)]);
117  q_gradv[ijN(1,0,2)] = gradv10;
...
123  // linalg/densemat.cpp: Eigensystem2S()
124  if (gradv10 == 0) {
125  minEig = (gradv00 < gradv11) ? gradv00 : gradv11
     ;
126  } else {
127    const double zeta  = (gradv11-gradv00) / (2.0*
     gradv10);
...
```

Fig. 1. Function with floating-point numerical inconsistency in the Laghos case.

### A. Compiler Numerical Inconsistency: A Real-World Case

Recently during the process of getting several applications ready for the new LLNL Sierra system, the Laghos (LAGrangian High-Order Solver) application [10]—a shock hydrocode miniapp developed at LLNL—experienced floating-point reproducibility and numerical consistency issues. The new Sierra system uses POWER9 host architecture and NVIDIA V100 GPUs, with IBM `xlc` as the main compiler provided by the system vendor. The application was adapted to study the impact of using the RAJA performance portability layer [11] via C++ lambda functions, which involved significant changes to minimize host-device data motion.

During the porting process, it was observed that the energy computed in the Laghos application was numerically inconsistent when the code was compiled using the `xlc` compiler with `-O3` in contrast to other compilers available in the system, such as `gcc`. Table I shows the computed energy $|e|$ for several compilers and optimization labels (see the last row).

TABLE I
LAGHOS UNDER DIFFERENT COMPILATION SETTINGS

| Compiler | Optimization | $\|e\|$ |
|---|---|---|
| `clang` | -O1 | 129941.1064990107 |
| `clang` | -O2 | 129941.1064990107 |
| `clang` | -O3 | 129941.1064990107 |
| `gcc` | -O1 | 129941.1064990107 |
| `gcc` | -O2 | 129941.1064990107 |
| `gcc` | -O3 | 129941.1064990107 |
| `xlc` | -O1 | 129941.1064990107 |
| `xlc` | -O2 | 129941.1064990107 |
| `xlc` | -O3 | 144174.9336610391 |

It took several weeks of effort to debug the issue and isolate the lines of code in the application that, combined with this compiler and optimization level, caused the numerical inconsistency. Using several search techniques to compile files and functions with different levels of optimization, such as delta debugging [12], the issue was isolated to a single function (`rUpdateQuadratureData2D`). In particular, it was found that a conditional statement in that function compares a floating-point number to zero (see line 124 in Figure 1), a dangerous code pattern[3]; when the conditional statement was changed to

[3]A better way to compare two floating-point numbers is to compare their difference to a small value.

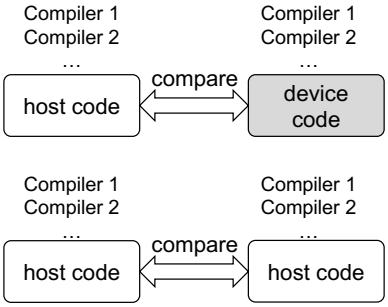


Fig. 2. Generally speaking, there are two main classes of comparisons that could yield different numerical results: (a) the same code is run in the host and in the device; in this case, we assume that different compilers are used (one that compiles the host code and another that compiles the device code); (b) the same code runs in the host, and different compilers are used. In both cases, we assume that the same level of optimization is used for all objects.

a comparison of the floating-point number to a small value instead of zero, the issue was alleviated and the computed energy at level `-O3` was consistent with the energy at other optimization levels and other compilers. Figure 1 shows the conditional statement in line 124, which was changed to "`if (gradv10 <= FLT_EPSILON)`".

As we develop tools to automatically isolate issues such as the Laghos case and recommendations to programmers about the compilers and optimization levels to use when numerical reproducibility is critical, it is crucial to be able to generate test cases such as the Laghos case that uncover such inconsistencies in advance, i.e., before they affect production runs. Unfortunately, there is a lack of tools in the literature to generate such test cases systematically. VARITY aims at addressing this problem by providing a framework to automatically generate many similar cases (similar to the Laghos case but as smaller stand-alone tests) and to use them to systematically identify these hidden variations between several compilers and optimization levels.

### B. Source of Numerical Variations

Numerical reproducibility based on enforcing IEEE 754-2008 arithmetic for floating-point instructions is challenging in modern HPC systems: Multiple compilers could be used to generate floating-point code, and code can be executed on different floating-point hardware (e.g., host and device hardware). Since roundoff errors make floating-point arithmetic non-associative, different compilers and hardware units could produce result variations in the implementations of the same algorithm. In the rest of the paper, we are interested in studying two general classes of floating-point variations: *host-to-device* variations and *host-to-host* variations.

**Host-to-Device Variations.** An interesting source of numerical variations is when the same code is run on different architectures, e.g., a host and device architecture, even when the same optimization level is used to compile all objects. In this case, the device code could be compiled with a compiler that is different from the compiler used to compile host code. For example, in a system with `x86` processors and NVIDIA GPUs, the `gcc` compiler can be used to generate program code to be run in the `x86` processor while the NVIDIA `nvcc` compiler can be used to generate program code to be run in the GPU. In this case, two sources of variations come to play: different architectures and different compilers.

```
1 __global__ /* __global__ is used for device run */
2 void compute(double comp, int var_1,double var_2) {
3  for (int i=0; i < var_1; ++i) {
4  comp += (+1.4792E306 / var_2);
5  }
6  printf("%.17g\n", comp);
7 }
```

```
Input: +1.5290E13 5 +1.2482E305

Output:
xlc -O3:  15290000000059.254
nvcc -O3: 15290000000059.258
```

Fig. 3. Example 1: host-to-device small numerical variation using `-O3` between `xlc` and `nvcc` compilers

**Host-to-Host Variations.** Another source of numerical variations occurs even when the same architecture is used to run a floating-point program. For example, in a host architecture, such as IBM POWER9, one could compile code with different compilers, e.g., `gcc`, `clang`, or `XL`. Each execution may produce different results, even when the same compilation level is used to compile everything; the results depend heavily on how the compiler generates optimized floating-point code.

Figure 2 shows at a high-level the two sources of floating-point differences that we study in this paper. In this study, we assume that all objects of a program are compiled with the same optimization level (whatever the level is). Since it is not common practice to compile some objects of a program with one level (say `-O1`) and other objects with another level (say `-O3`), we do not consider this case.

### C. Examples of Variations

Here, we present four examples of floating-point programs that we discover using VARITY, on which numerical variations are observed. For these examples we use a system with IBM POWER9 processors for the host architecture and NVIDIA V100 GPUs for the device architecture.

**Example 1.** The first example is shown in Figure 3. Here, VARITY generates a program composed of a for loop. The framework generates a GPU version and a CPU version (which does not use `__global__`) of a kernel named `compute`. The kernel is called from a `main()` function, which passes the parameters to the kernel by value.

This example is compiled with the `xlc`[4] compiler and run on the host, and with the `nvcc` compiler and run on the GPU (using a single thread). In both cases, we use the `-O3` optimization level. The result for the provided input (shown in the Figure) is 15290000000059.254 for `xlc` and 15290000000059.258 for `nvcc`. The results here differ only by the last digit. This kind of difference, while small, is not

[4]`xlc` is the `C` compiler from the IBM XL compilers family

```
__global__ /* __global__ is used for device run */
void compute(double comp, int var_1,double var_2,
double var_3,double var_4,double var_5,double var_6,
double var_7,double var_8,double var_9,double var_10,
double var_11,double var_12,double var_13,
double var_14) {
  double tmp_1 = +1.7948E-306;
  comp = tmp_1 + +1.2280E305 - var_2 +
    ceil((+1.0525E-307 - var_3 / var_4 / var_5));
  for (int i=0; i < var_1; ++i) {
    comp += (var_6 * (var_7 - var_8 - var_9));
  }
  if (comp > var_10 * var_11) {
    comp = (-1.7924E-320 - (+0.0 / (var_12/var_13)));
    comp += (var_14 * (+0.0 - -1.4541E-306));
  }
  printf("%.17g\n", comp);
}
```

```
Input: -0.0 5 -0.0 -1.3121E-306 +1.9332E-313
+1.0351E-306 +1.1275E172 -1.7335E113
+1.2916E306 +1.9142E-319 +1.1877E-306
+1.2973E-101 +1.0607E-181 -1.9621E-306 -1.5913E118-03

Output:
clang -O3:  NaN
nvcc -O3:   -2.3139093300000002e-188
```

Fig. 4. Example 2: host-to-device large numerical variation using `-O3` between `clang` and `nvcc` compilers

```
void compute(double comp, double var_1,double var_2,
    double var_3,double var_4) {
  comp += (-1.8541E-307 + var_1);
  comp = exp(+1.8107E-208 - (-0.0 + tanh(var_2 /
      (var_3 * +1.5958E-307 + var_4))));
  printf("%.17g\n", comp);
}
```

```
Input: +1.5117E-321 -1.6300E-216 +1.6978E254
-1.3913E-312 +0.0

Output:
xlc -O0:  2.7182818284590451
gcc -O0:  0.36787944117144233
```

Fig. 5. Example 3: host-to-host large numerical variation using `-O0` between `xlc` and `gcc` compilers

uncommon since we are using a high optimization level and by default `xlc` has the potential to slightly alter the semantics of the program with `-O3`.

**Example 2.** A more interesting case is shown in Figure 4. In this example, VARITY finds a more complex program, on which the results between device and host executions vary drastically. In this case, `-O3` is also used; however, the program execution on the host—compiled with `clang`—produces `NaN` whereas the program execution on the device produces $-2.3139093300000002\text{e-}188$. Again, since we are using a high optimization level, numerical differences are not unexpected; however, in this case the difference is significant—one answer is the result of an exception (a non-normal floating-point), while the other is a normal floating-point number.

**Example 3.** Thus far, the previous examples considered cases where `-O3` was used; what about cases where we indicate to the compiler that no optimizations should be used,

```
void compute(double comp, int var_1,double var_2,
double var_3,double var_4,double var_5,double* var_6,
double var_7,double var_8,double var_9,double var_10,
double var_11,double var_12,double var_13,
double var_14) {
  comp += +1.2470E-319 - +0.0 / (var_2 / var_3);
  comp += (var_4 * exp((var_5 * +1.0514E-312 -
      +1.5551E-322)));
  for (int i=0; i < var_1; ++i) {
    comp = (var_7 + (var_8 / -1.2858E-321));
    var_6[i] = -1.8695E-311;
    comp = var_6[i] - (var_9 - var_10);
  }
  if (comp > var_11 * var_12 - (-0.0 + cosh(+1.2278
      E306))) {
    comp = (-0.0 * (-1.0653E306 / (var_13 + (+1.6553
        E305 * var_14))));
  }
  printf("%.17g\n", comp);
}
```

```
Input: +1.6770E305 5 +1.5646E305 +1.0217E305
+1.8449E-4 -1.6826E-306 -0.0 -1.9900E-83
+1.3745E306 +0.0 +1.2869E124 -1.4736E305
-1.2075E128 +1.6557E-306 -1.9159E306

Output:
xlc -O0:    -0
clang -O0:  1.2869e+124
```

Fig. 6. Example 4: host-to-host large numerical variation using `-O0` between `xlc` and `clang` compilers

e.g., when using `-O0`? Figure 5 shows an example found by VARITY where, even with `-O0`, we observe significant floating-point variations. In this example, VARITY compiles the test with different compilers, `xlc` and `gcc`, but runs both tests in the host. The result for `xlc` is $2.7182818284590451$ and for `gcc` is $0.36787944117144233$, which is also a significant difference.

**Example 4.** Our final example is shown in Figure 6. Here, like in the previous example, the program is compiled with different compilers: `xlc` and `clang`, using `-O0` and run on the host. The results are zero (negative) and $1.2869\text{e+}124$, also very different results. In Section IV, we will present more cases of numerical variations and will give more details on the source of variations of the above examples.

### *D. Approach Overview*

Figure 7 presents a high-level overview of our approach. We start by defining the structure of test programs using a grammar—the grammar defines valid random programs generated by VARITY. Generated programs are then handled by a driver, which handles three steps: (1) compiling the programs with the compilers that are available in the HPC system (this generates executables), (2) generating valid inputs for the generated executables, and (3) running the executables (with corresponding inputs) in the host and device architectures.

**Differential Testing.** We use randomized differential testing [8] to detect floating-point variations in the executed test programs. The intuition behind this method is that if two executions of the same compiled program produce distinct floating-point results, there must be a difference in either how the compiler generates floating-point code or in how the

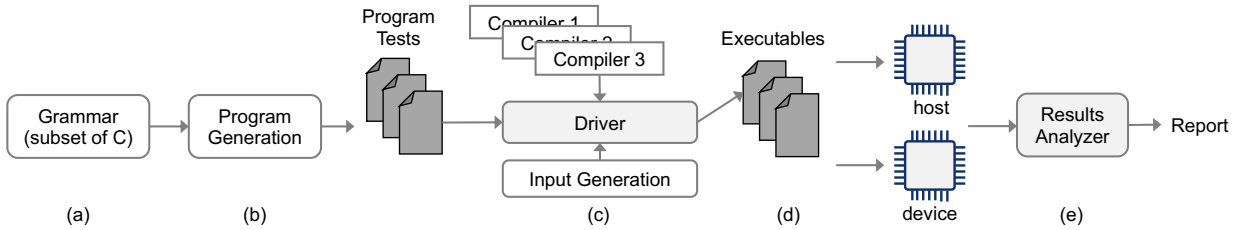


Fig. 7. Overview of our approach for finding floating-point variations across architectures and compilers.

architecture executed the code. In any case, users are interested in understanding when these differences occur to better chose the compilers for a given application in a system.

Finally, a results analyzer engine parses the results and presents a report that summarizes the floating-point differences found in the test runs. In the next section, we will describe each of the components of this approach in more detail.

## III. Design

We describe each component of our approach in detail.

### A. Programs Structure

Our approach generates random programs that make use of floating-point arithmetic. Our goal is to find as many programs as possible that expose differences in how compilers treat and generate floating-point code. While ideally, one would want to explore the entire space of all possible programs, this is not practical since the space is too large. Therefore, we restrict our search to a subset of programs: programs written in the `C` language with a well-defined structure. To formally define this structure, we use a grammar.

**Grammar.** To design the grammar, we consider the most important aspects of HPC programs and use the characteristics of programs that could (most likely) affect how floating-point code is generated and executed. The grammar allows us to generate programs with the following characteristics:

1) **Different Floating-Point Types:** we can generate variables using single and double floating-point precision (i.e., `float` and `double`).
2) **Arithmetic Expressions:** arithmetic expressions can use any operator in {`+`, `-`, `*`, `/`}, can use parenthesis "()", and can use functions from the `C math` library. The grammar also allows boolean expressions.
3) **Loops:** loops constitute the main building block of HPC programs; the grammar allows the generation of `for` loops with multiple levels of nesting. More formally, we can generate loop sets $L_1 > L_2 > L_3 > \ldots > L_N$, where $L_1$ encloses $L_2$, $L_2$ encloses $L_3$, and so on up to $L_N$, where $N$ is defined by the user.
4) **Conditions:** the grammar supports `if` conditions, which can be true or false based on a boolean expression.
5) **Variables:** programs can contain temporal floating-point variables. Variables can store arrays or single values.

A description of the grammar is shown in Figure 8. This grammar specifies the programs that are allowed in the

```
<function> ::= "void" "compute"
"(" <param-list> ")" "{" <block> "}"

<param-list> ::= <param-declaration>
| <param-list> "," <param-declaration>

<param-declaration> ::= "int" <id>
| <fp-type> <id>
| <fp-type> "*" <id>

<assignment> ::= "comp" <assign-op> <expression> ";"
| <fp-type> <id> <assign-op>
<expression> ";"

<expression> ::= <term>
| "(" <expression> ")"
| <expression> <op> <expression>

<term> = <identifier> | <fp-numeral>

<block> ::= {<assignment>}+
| <if-block> <block>
| <for-loop-block> <block>

<if-block> ::= "if" "(" <bool-expression> ")"
"{" <block> "}"

<for-loop-block> ::= "for" "(" <loop-header> ")"
"{" <block> "}"

<bool-expression> ::= <id> <bool-op> <expression>

<loop-header> ::= "int" <id> ";" <id> "<" <int-numeral>
    ";" "++" <id>
```

Fig. 8. Grammar specification for the random test programs. `<fp-type>` supports {`float`, `double`}, `<assignment-op>` supports {`=`, `+=`, `-=`, `*=`, `/=`}, `<op>` supports {`+`, `-`, `*`, `/`}, and `<bool-op>` supports {`<`, `>`, `==`, `!=`, `>=`, `<=`}.

generation step in VARITY (step (b) in Figure 7). The key floating-point operations are enclosed in a kernel function named `compute`. The kernel function does not return anything; instead, it computes a floating-point value and stores it in the `comp` variable. The value of `comp` is printed in standard output.

Note that, in addition to the `comp` kernel function, we generate a `main()` function and code to allocate and initialize arrays (if arrays are used in the test program). For simplicity, we do not present this in the grammar. The `main()` function reads the program inputs and copies them to the `comp` kernel function parameters before calling the kernel function.

**CUDA Code.** When generating CUDA code (for GPUs), the `comp` kernel function is defined as `__global__` and the kernel is always launched from `main()` using one block and one grid with dimensionality one.

**Alternatives to the Grammar.** We use the above grammar to specify the programs that can be generated in our approach; the goal is to be able to generate random programs that reflect

the critical computations of scientific programs, i.e., loops, conditionals, and arithmetic expressions. An alternative to this approach could be to generate programs that are somewhat *similar* to existing programs (e.g., a set of benchmarks), perhaps using slight perturbations to the existing programs. This approach, however, may only discover differences that are already present in the existing seed programs. We use the former approach (i.e., entirely random programs) since we are interested in the discovery of new (unknown) variations. We plan to explore in future work the latter method.

### B. Program Generation

**Randomness.** We use randomness in the generation of test programs. We use the same approach that is used in previous work [13] to construct a random program, i.e., uniform distributions are used to choose elements of the program. To limit the search space, we do not randomize the generation of all the aspects of the program—for example, the name of the kernel function is always `compute`. However, aspects that are relevant to floating-point arithmetic are chosen randomly. The following features are chosen randomly: (1) type of arithmetic operations, (2) type of boolean operations, (3) size of arithmetic expressions, (4) size of boolean expressions, (5) size of blocks (i.e., number of statements), and (6) number of nesting levels of blocks.

We impose limits on the above parameters since exploring infinite sets of them would be infeasible. The following parameters are used to limit the generation of program features:

- `MAX_EXPRESSION_SIZE`: defines the maximum number of terms in an expression (arithmetic or boolean).
- `MAX_NESTING_LEVELS`: defines the maximum number of nesting levels of blocks (`if` condition and `for` loop blocks).
- `MAX_LINES_IN_BLOCK`: a block can have several lines containing, e.g., temporal variable definitions or assignments. This defines the maximum number of lines in a block.
- `ARRAY_SIZE`: maximum number of elements in arrays.
- `MAX_SAME_LEVEL_BLOCKS`: in addition to assignments (or other expressions), a block can have other blocks. This defines the maximum number of blocks at the same nesting level inside a block.
- `MATH_FUNC_ALLOWED`: defines whether or not to use functions from `math.h` in arithmetic expressions.
- `INPUT_SAMPLES_PER_RUN`: defines the number of distinct sample inputs used per program test.

**Probability Distributions.** Whenever we have a choice in selecting the features mentioned above (except for math functions), the probability distribution of selecting an element of the feature set is uniform. More formally, if we have $K$ possible options for a given program feature, the probability of selecting a feature $f_i$ in $\{f_1, f_2, \ldots, f_K\}$ is $1/K$. For example, suppose we are constructing an expression $var_1 \oplus var_2$, where $\oplus$ can be an arithmetic operator in the set OP $= \{$`+`, `-`, `*`, `/`$\}$, the probability $P(\oplus = x | x \in \text{OP}) = 1/K$, where $K = 4$ (since there are 4 elements in OP).

For the random selection of math functions, we use a smaller (fixed) probability. According to the above description, since selecting a math function in an arithmetic operation is a true/false decision, the probability of choosing a math function would be 1/2, which may be too high—math functions are used in scientific programs; however, they are not necessarily ubiquitous in all arithmetic expressions of programs. To choose math function in expressions, we use a default value of $0.10$, but users are allowed to use a different probability.

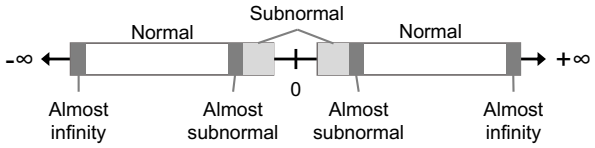


Fig. 9. Illustration of the classes of floating-point numbers that the input generation module generates.

**Program Tests.** The output of the program generation phase (step (b) in Figure 7) is a set of unique C and CUDA programs, each containing a unique kernel function. To avoid generating repeated programs, we maintain a cache of generated programs in a given experimental run and discard programs that have been previously generated.

### C. Input Generation

Floating-point inputs are generated via an input generation module. This module can generate five kinds of floating-point numbers: *normal* numbers, *subnormal* numbers, *almost infinity* numbers, *almost subnormal* numbers, and zero (positive and negative). The normal, subnormal, and zero numbers correspond to those defined in the IEEE 754-2008 Standard. Almost infinity and almost subnormal numbers are extreme cases, which are not defined in the Standard. We define an almost infinity number as a number, which is close to infinity (`+INF` or `-INF`), but that it still a normal number. We define an almost subnormal number as a number that is close to being a subnormal number, but that it is still a normal number.

We do not generate special numbers, such as `NaN` or $\pm$`INF`, because usually applications do not use these numbers as input. We, however, check for **results** that fall in these categories later in the analysis phase (see Section III-E).

### D. Driver

The driver module performs two tasks: it uses the compilers that are available in the system to generate executables out of the test programs, and it uses the input generation module to generate valid inputs for the executables (Figure 7 (c)).

**Code Generation and Compiler Options.** The set of compilers can be specified by the user or they can be detected automatically. The user can also specify compilation options. By default, the framework uses the following levels of optimization, which are often available in all compilers: `-O0`, `-O1`, `-O2`, and `-O3`.

**Code Execution.** After executables are generated, the driver executes them in the host and device architectures that are available in the system (see step (d) in Figure 7). The results of the executions are gathered for each combination (*executable*, *input*, *compiler*, *optimization*) and stored in the JSON format.

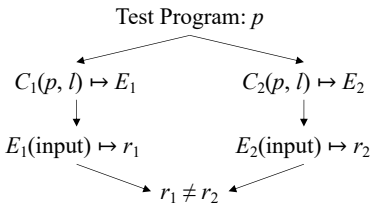


Fig. 10. Differential testing approach of checking the results $r_1$ and $r_2$ that came from executing the $E_1$ and $E_2$ executables with a given input. $E_1$ and $E_2$ were produced by compiling program $p$ with the same optimization level $l$ with compilers $C_1$ and $C_2$, respectively.

```
for (i = ist; i <= iend; i++) {
  for (j = jst; j <= jend; j++) {
    for (m = 0; m < 5; m++) {
      v[i][j][k][m] = v[i][j][k][m]
        - omega * (ldz[i][j][m][0] * v[i][j][k-1][0]
        + ldz[i][j][m][1] * v[i][j][k-1][1]
        + ldz[i][j][m][2] * v[i][j][k-1][2]
        + ldz[i][j][m][3] * v[i][j][k-1][3]
        + ldz[i][j][m][4] * v[i][j][k-1][4]  );
    } } }
```

Fig. 11. Example of a large expression (11 arithmetic operations) in LU

### E. Results Analyzer

The results analyzer module analyzes the JSON files generated on the executions phase. The analysis consists of analyzing floating-point results from pairs of different configurations and checking when floating-point variations occur. The analysis core idea is to use differential testing to compute differences of the kind $\{r_1, r_2\}$, such that $r_1 \neq r_2$, where $r_1$ is the numerical result that came from executing the executable generated by compiler 1, and $r_2$ is the numerical result that came from executing the executable generated by compiler 2.

**Differential Testing Formal Definition.** Suppose a program $p$ is compiled by compiler $C_i$ using optimization level $l$. The output of this compilation is the executable $E_i$; thus, we write $C_i(p, l) \mapsto E_i$, i.e., the executable $E_i$ is produced when compiler $C_i$ compiles $p$ with level $l$. When the executable $E_i$ is run with a given input, we say that this produces $r_i$, thus, we write $E_i(\text{input}) \mapsto r_i$. We perform this process for all pairs of compilers $(C_i, C_j)$, and check whether there is a difference of the kind $\{r_1, r_2\}$, such that $r_i \neq r_j$, where $i \neq j$.

Figure 10 shows an example of a check using two compilers $C_1$ and $C_2$. In a full **testing campaign** for a given system with two compilers, we quantify the number of times the following condition is true:

$$C_1(p, l)(\text{input}) \neq C_2(p, l)(\text{input}),$$

for all $p \in$ *Programs*, $input \in$ *Inputs*, and $l \in$ *Levels*, where *Programs* is the set of randomly generated programs, *Inputs* is the set of randomly generated inputs, and *Levels* is the set of optimization levels.

## IV. Evaluation

In this section, we evaluate VARITY in a heterogeneous HPC system and summarize the observed results. We designed the evaluation to answer the following research questions:

**Q1** What are the most common class and the least common class of floating-point variations?

**Q2** Are there host-to-device floating-point variations? If so, how frequent are they and when do they occur?

**Q3** Are there host-to-host floating-point variations? If so, how frequent are they and when do they occur?

### A. Floating-Point Number Categories

Test programs can produce various classes of numerical results. Our analysis considers five categories of numerical results. The first category is normal and subnormal numbers, which are the most common expected results in any scientific application. We group normal and subnormal numbers in the *Real* category. The second category is *Zero*; here, we group both positive zero (+0) and negative zero (−0). The remaining three categories are positive infinity, negative infinity, and not-a-number, as defined in the IEEE 754-2008 standard, i.e., *+Inf*, *-Inf*, and *NaN*, respectively.

### B. Configuring VARITY with Existing HPC Programs

We analyze the structure of the C version of the NAS Parallel Benchmarks (BT, CG, EP, FT, IS, LU, MG, and SP) version 3.0 [9] and configure the parameters of VARITY following the code structure of these benchmarks. We analyze the main computation loops and computation functions of these benchmarks. Based on our analysis, we configure the most critical parameters of VARITY following our findings: `MAX_EXPRESSION_SIZE`, `MAX_NESTING_LEVELS`, `MAX_SAME_LEVEL_BLOCKS`; these parameters are described in Section III-B. We describe the analysis and configuration process as follows.

We find that the maximum level of nesting (`MAX_NESTING_LEVELS`) in most of the computation loops of the benchmarks are in the range of 3–4 (we found only a few exceptions in the CG benchmark with 6 nesting levels, but it was for only a few loops); thus we use 4 for `MAX_NESTING_LEVELS` in the evaluation. Similarly, we observe that for most of the benchmarks, the `MAX_SAME_LEVEL_BLOCKS` metric is on average 3, which we use to set this parameter in VARITY.

We observe a large variability for the size of expressions (`MAX_EXPRESSION_SIZE`) in the benchmarks. On one hand, for some benchmarks the max size of expressions is in the range of 3–6 (e.g., FT, EP, IS), while on the other hand, for other benchmarks, expressions can be quite large—we found expressions in BT of up 29 and in LU of up to 50. Figure 11 shows a nested loop example from LU with an expression of size 12 (it has 12 terms with 11 arithmetic operations). As a result, we perform two groups of experiments: one with `MAX_EXPRESSION_SIZE`=6 (to mimic small expressions) and one with `MAX_EXPRESSION_SIZE`=50 (to mimic large expressions). We start with `MAX_EXPRESSION_SIZE`=6 and report our findings in the next sections; later in Section IV-I we describe what we found when we set `MAX_EXPRESSION_SIZE`=50.

**Parameters Related to Inputs.** For parameters that are independent of the code structure of the benchmarks, such as those related to inputs, we use a small value to set their sizes: `INPUT_SAMPLES_PER_RUN = 25`, `ARRAY_SIZE = 10`. VARITY chooses random floating-point numbers to populate the inputs with the previous constraints for the number of inputs; Section III-C gives more details on the input generation algorithm. We set `MATH_FUNC_ALLOWED = True` since many scientific codes use functions from `math.h`.

### C. System and Setting

We perform all the experiments in the Lassen system at the Lawrence Livermore National Laboratory. Lassen is the top 10 system in the top500 list[5] and it has IBM POWER9 22C 3.1GHz nodes (44 cores/node) as host architecture, with Dual-rail Mellanox EDR Infiniband, and NVIDIA Tesla V100 GPUs as device architecture (4 GPUs/node). The system runs Red Hat Enterprise Linux 7.6 (Maipo).

**Compilers.** We use four of the available compilers in the system: `gcc` version 7.2, `clang` version 8.0, `xlc` version 16.01, and `nvcc` version 9.2. `gcc`, `clang`, and `xlc` are always used to generate code for the host only, while `nvcc` is always used to generate code for the device only. By default, all host compilations are linked to the math library (`libm.so.6`) that comes from the GNU C Library (glibc), except for the device compilations, which use the CUDA math library.

**Runs and Types.** We perform in total 50,000 unique *runs*—a run is defined as a combination of a test program and an input. 25,000 runs are executed for double-precision floating-point programs, or FP64, and 25,000 runs are executed for single-precision floating-point programs, or FP32.

### D. Fused MultiplyAdd (FMA)

A fused multiply-add (FMA) is a floating-point multiply-add operation $x = a + (b \times c)$ performed in one step, with a single rounding. In 2008, the IEEE 754 standard was revised to include FMA operations. Because the FMA uses only a single rounding step (as opposed to two), the result is computed more accurately. While not all floating-point hardware units provide FMA, both the IBM POWER9 host architecture and the NVIDIA V100 architecture provide FMA.

In our investigation of several cases, such as the examples in Section II-C, we found that one of the reasons floating-point variations are observed is the use of FMA. In particular, we found that, in some compilers, such as the IBM `xlc` compiler, FMA operations are generated whenever possible. This is the case even when the user indicates to the compiler that code should not be optimized, i.e., the user uses `-O0`. For such compilers, FMA is not considered a significant optimization, so it is enabled by default with `-O0`. For example, in Examples 3 and 4 (Section II-C), when we disable FMA, we observe that the difference in the results disappears.

To completely disable FMA, one has to indicate to the compiler to disable it specifically. For such a case, in the experiments, we denote `-O0_nofma` as one level lower to `-O0`, i.e., where `-O0` is used and in addition we disable FMA.

[5]https://www.top500.org/

### E. Fast Math and Strict Compiler Options

When code is highly optimized (e.g., with `-O3`), floating-point optimizations can provide significant performance improvements, but at the same time, they can generate code that is not compliant with IEEE 754-2008. The philosophy of open-source compilers, such as `gcc` and `clang`, is by default to be compliant, or *strict*, with respect to IEEE 754-2008, unless the user indicates the opposite; the `-ffast-math` option can be used in `gcc` and `clang` to produce programs that are not compliant with IEEE. `nvcc` provides a similar option, `--use_fast_math`.

On the other hand, the philosophy of commercial compilers, such as `xlc`, is different: in the `xlc` compiler, `-O3` by default performs aggressive optimizations that alter the semantics of the program and are not compliant with IEEE, i.e., by default with `-O3` it is *non-strict* with respect to IEEE. One can use the `-qstrict` option in `xlc` to generate code that is compliant to IEEE. In the experiments, we use both *strict* and *nonstrict* modes—Table II shows the flags that we use for each mode.

### F. Experiment 1: Classes of Numerical Variations

In this experiment, we analyze the number of times we observe numerical floating-point differences in different classes. More specifically, we compute the percentage of times we observe a difference of the kind $\{r_i, r_j\}$, such that $r_i \neq r_j$, as it is defined in Section III-E. Here, $r_i$ and $r_j$ correspond to any of the numerical categories that we describe in Section IV-A, i.e., $r_i, r_j \in \{Real, Zero, +Inf, -Inf, NaN\}$.

Tables III and IV show the percentage of times that these differences occur. Note that the results are scaled up by $\times 10^2$, i.e., the percentages are smaller than what is presented. We observe that when `O0_nofma` is used for FP64 programs, there are almost no differences of any kind, except for some cases where we observe {*Real*, *Real*} differences, i.e., both results were *Real*, but they differ by at least a digit, and a few of the kind {*Real*, *NaN*} and {*Zero*, *NaN*}. For FP32 programs, we do observe differences of other classes even when `O0_nofma` is used, but overall a small percentage.

**Answer to Q1:** The most common class of numerical variations are {*Real*, *NaN*}, i.e., cases when the program coming from one compiler produced a numerical result, but the program coming from a different compiler produced a *NaN*, for the same input and optimization level. We also note that variations of the kind {*Real*, *Real*} are prevalent. The least common class of numerical variations are {*Zero*, *-Inf*}. We observe that the largest degree of variation is observed with `O3_nonstrict`, as expected. While `O3_strict` reduces the rate of differences, we still observe variations with this mode.

TABLE II

FLAGS USED IN COMPILERS FOR *strict* AND *non-strict* OPTIMIZATION WITH RESPECT TO IEEE 754-2008.

| Optimization Type | gcc | clang | nvcc | xlc |
|---|---|---|---|---|
| O3_nonstrict | `-O3 -ffast-math` | `-O3 -ffast-math` | `-O3 --use_fast_math` | `-O3` |
| O3_strict | `-O3` | `-O3` | `-O3` | `-O3 -qstrict` |

TABLE III

PERCENTAGE (%) OF TIMES ($\times 10^2$) THAT THERE IS A DIFFERENCE OF THE KIND $r_1, r_2$, WHERE $r_1 \neq r_2$, BETWEEN TWO COMPILERS, FOR ALL TESTS, INPUTS, AND COMPILERS, FOR FP64 PROGRAMS.

| | Real, Real | Real, Zero | Real, NaN | Real, +Inf | Real, -Inf | Zero, NaN | Zero, +Inf | Zero, -Inf | NaN, +Inf | NaN, -Inf | +Inf, -Inf |
|---|---|---|---|---|---|---|---|---|---|---|---|
| O0_nofma | 1.131% | – | 0.004% | – | – | 0.001% | – | – | 0.001% | – | – |
| O0 | 1.768% | 0.010% | 0.116% | 0.017% | 0.013% | 0.032% | 0.023% | 0.004% | 0.356% | 0.351% | 0.080% |
| O1 | 1.768% | 0.010% | 0.116% | 0.017% | 0.013% | 0.032% | 0.023% | 0.004% | 0.355% | 0.351% | 0.079% |
| O2 | 1.768% | 0.010% | 0.116% | 0.017% | 0.013% | 0.032% | 0.023% | 0.004% | 0.355% | 0.351% | 0.079% |
| O3_nonstrict | 35.082% | 11.102% | 90.365% | 9.278% | 8.355% | 23.909% | 1.763% | 1.544% | 44.219% | 41.733% | 2.913% |
| O3_strict | 1.747% | 0.018% | 0.178% | 0.016% | 0.016% | 0.071% | 0.024% | 0.007% | 0.389% | 0.425% | 0.095% |
| All | 43.262% | 11.150% | 90.895% | 9.345% | 8.410% | 24.077% | 1.855% | 1.564% | 45.676% | 43.211% | 3.246% |

TABLE IV

PERCENTAGE (%) OF TIMES ($\times 10^2$) THAT THERE IS A DIFFERENCE OF THE KIND $r_1, r_2$, WHERE $r_1 \neq r_2$, BETWEEN TWO COMPILERS, FOR ALL TESTS, INPUTS, AND COMPILERS, FOR FP32 PROGRAMS.

| | Real, Real | Real, Zero | Real, NaN | Real, +Inf | Real, -Inf | Zero, NaN | Zero, +Inf | Zero, -Inf | NaN, +Inf | NaN, -Inf | +Inf, -Inf |
|---|---|---|---|---|---|---|---|---|---|---|---|
| O0_nofma | 2.554% | 0.002% | 0.002% | 0.001% | 0.004% | 0.002% | – | – | 0.001% | 0.001% | 0.003% |
| O0 | 3.332% | 0.024% | 0.144% | 0.017% | 0.022% | 0.046% | 0.016% | 0.006% | 0.392% | 0.346% | 0.108% |
| O1 | 3.335% | 0.026% | 0.149% | 0.019% | 0.022% | 0.048% | 0.016% | 0.008% | 0.399% | 0.351% | 0.144% |
| O2 | 3.335% | 0.026% | 0.149% | 0.019% | 0.022% | 0.048% | 0.016% | 0.008% | 0.399% | 0.351% | 0.144% |
| O3_nonstrict | 42.878% | 67.959% | 93.268% | 9.495% | 8.928% | 31.170% | 2.618% | 2.425% | 43.773% | 41.703% | 2.616% |
| O3_strict | 21.561% | 62.456% | 32.780% | 4.738% | 4.682% | 9.470% | 1.193% | 1.043% | 14.640% | 14.447% | 0.726% |
| All | 76.993% | 130.495% | 126.492% | 14.289% | 13.680% | 40.785% | 3.858% | 3.490% | 59.606% | 57.200% | 3.741% |

TABLE V

FOR DIFFERENCES OF THE TYPE {REAL, REAL}, THE PERCENTAGE (%) OF TIMES ($\times 10^2$) THAT THERE IS A DIFFERENCE OF THAT TYPE BETWEEN TWO COMPILERS. THE SECOND ROW FOR EACH CASE SHOWS $[a, b]$, WHERE $a$ AND $b$ ARE THE MINIMUM AND MAXIMUM NUMBER OF DIGITS OF ACCURACY OBSERVED, RESPECTIVELY, FOR FP64.

| | clang, gcc | clang, nvcc | clang, xlc | gcc, nvcc | gcc, xlc | nvcc, xlc |
|---|---|---|---|---|---|---|
| O0_nofma | 0.021% | 0.356% | – | 0.377% | 0.021% | 0.356% |
| | [16, 16] | [0, 16] | – | [0, 16] | [16, 16] | [0, 16] |
| O0 | 0.021% | 0.502% | 0.158% | 0.524% | 0.180% | 0.383% |
| | [16, 16] | [0, 16] | [0, 16] | [0, 16] | [0, 16] | [0, 16] |
| O1 | 0.021% | 0.502% | 0.158% | 0.524% | 0.179% | 0.383% |
| | [16, 16] | [0, 16] | [0, 16] | [0, 16] | [0, 16] | [0, 16] |
| O2 | 0.021% | 0.502% | 0.158% | 0.524% | 0.179% | 0.383% |
| | [16, 16] | [0, 16] | [0, 16] | [0, 16] | [0, 16] | [0, 16] |
| O3_nonstrict | 5.188% | 6.508% | 6.062% | 4.833% | 5.561% | 6.931% |
| | [0, 16] | [0, 16] | [0, 16] | [0, 16] | [0, 16] | [0, 16] |
| O3_strict | 0.021% | 0.502% | 0.139% | 0.524% | 0.160% | 0.400% |
| | [16, 16] | [0, 16] | [0, 16] | [0, 16] | [0, 16] | [0, 16] |
| All | 5.294% | 8.873% | 6.675% | 7.305% | 6.280% | 8.835% |

TABLE VI

FOR DIFFERENCES OF THE TYPE {REAL, REAL}, THE PERCENTAGE (%) OF TIMES ($\times 10^2$) THAT THERE IS A DIFFERENCE OF THAT TYPE BETWEEN TWO COMPILERS. THE SECOND ROW FOR EACH CASE SHOWS $[a, b]$, WHERE $a$ AND $b$ ARE THE MINIMUM AND MAXIMUM NUMBER OF DIGITS OF ACCURACY OBSERVED, RESPECTIVELY, FOR FP32.

| | clang, gcc | clang, nvcc | clang, xlc | gcc, nvcc | gcc, xlc | nvcc, xlc |
|---|---|---|---|---|---|---|
| O0_nofma | 0.084% | 0.800% | – | 0.787% | 0.084% | 0.800% |
| | [7, 8] | [0, 8] | – | [0, 8] | [7, 8] | [0, 8] |
| O0 | 0.084% | 0.987% | 0.178% | 0.975% | 0.262% | 0.844% |
| | [7, 8] | [0, 8] | [0, 8] | [0, 8] | [0, 8] | [0, 8] |
| O1 | 0.003% | 0.975% | 0.264% | 0.978% | 0.267% | 0.847% |
| | [7, 7] | [0, 8] | [0, 8] | [0, 8] | [0, 8] | [0, 8] |
| O2 | 0.003% | 0.975% | 0.264% | 0.978% | 0.267% | 0.847% |
| | [7, 7] | [0, 8] | [0, 8] | [0, 8] | [0, 8] | [0, 8] |
| O3_nonstrict | 5.665% | 8.592% | 5.811% | 9.011% | 5.066% | 8.733% |
| | [0, 8] | [0, 8] | [0, 8] | [0, 8] | [0, 8] | [0, 8] |
| O3_strict | 0.003% | 7.052% | 0.232% | 7.052% | 0.235% | 6.986% |
| | [7, 7] | [0, 8] | [0, 8] | [0, 8] | [0, 8] | [0, 8] |
| All | 5.842% | 19.382% | 6.750% | 19.781% | 6.182% | 19.057% |

### G. Experiment 2: Percentage of Differences for Compilers

In this experiment, we analyze only the cases when we see differences of the kind {*Real*, *Real*}. Here, we analyze all possible combinations of compilers to see which combinations produce the largest and smallest number of differences. For each case, we also compute the minimum and the maximum number of digits of accuracy observed. We always print 17 digits for floating-point results. If all the 17 digits match, there is no difference; the minimum and the maximum number of digits of accuracy are therefore 0 and 16, respectively. Tables V and VI show the results of the experiment.

**Answer to Q2:** Host-to-device variations occur; any column in Tables V and VI where `nvcc` is used correspond to a host-to-device difference. The most common case occurs between `nvcc` and `xlc`, specifically when `O3_nonstrict` is used; the second most common case occurs between `nvcc` and `clang`, also with `O3_nonstrict`. This finding suggests that when numerical results are to be compared between `nvcc` and `xlc` compilations, this combination tends to produce the most frequent floating-point numerical differences, so it should be avoided if possible.

TABLE VII
PERCENTAGE (%) OF TIMES ($\times 10^2$) THAT THERE IS A DIFFERENCE BETWEEN THE NVCC COMPILER AND ANOTHER COMPILER, FOR FP64.

| | clang | gcc | xlc |
|---|---|---|---|
| O0_nofma | 0.356% | 0.377% | 0.356% |
| O0 | 0.502% | 0.524% | 0.383% |
| O1 | 0.502% | 0.524% | 0.383% |
| O2 | 0.502% | 0.524% | 0.383% |
| O3_nonstrict | 6.508% | 4.833% | 6.931% |
| O3_strict | 0.502% | 0.524% | 0.400% |
| All | 8.873% | 7.305% | 8.835% |

TABLE VIII
PERCENTAGE (%) OF TIMES ($\times 10^2$) THAT THERE IS A DIFFERENCE BETWEEN THE NVCC COMPILER AND ANOTHER COMPILER, FOR FP32.

| | clang | gcc | xlc |
|---|---|---|---|
| O0_nofma | 0.800% | 0.787% | 0.800% |
| O0 | 0.987% | 0.975% | 0.844% |
| O1 | 0.975% | 0.978% | 0.847% |
| O2 | 0.975% | 0.978% | 0.847% |
| O3_nostrict | 8.592% | 9.011% | 8.733% |
| O3_strict | 7.052% | 7.052% | 6.986% |
| All | 19.382% | 19.781% | 19.057% |

Tables VII and VIII show a more specific case, i.e., the percentage of times that there is a difference between results from `nvcc` and the other compilers—these include all classes of numerical differences, not only {*Real*, *Real*}. The results confirm our previous finding for host-to-device variations, i.e., for `O3_nonstrict` the largest number of numerical variations occur when results coming from `nvcc` are compared to results coming from `xlc`. However, we observe that, with `O3_strict`, `xlc` tends to produce fewer variations with respect to `nvcc`, in comparison to `gcc` and `clang`.

**Answer to Q3:** Host-to-host variations occur. These variations are most frequent when results from an `xlc` compilation are compared to results from `clang` with `O3_nonstrict`. We observe that for `O3_strict`, the most similar results come from compilations between `gcc` and `clang`, while for `O3_nonstrict`, the most different results come from compilations between `xlc` and `clang`. Surprisingly we did not observe differences for `O0_nofma` between `clang` and `xlc` (both for FP64 and FP32); thus, this operation mode produces code with high numerical consistency.

TABLE IX
PERCENTAGE (%) OF TIMES ($\times 10$) THAT THERE IS A DIFFERENCE BETWEEN ANY OPTIMIZATION LEVEL AND O0_NOFMA, FOR FP64.

| | clang | gcc | nvcc | xlc |
|---|---|---|---|---|
| O0 | 0.263% | 0.000% | 0.700% | 0.811% |
| O1 | 1.117% | 0.195% | 0.700% | 1.047% |
| O2 | 1.147% | 0.195% | 0.700% | 1.047% |
| O3_nostrict | 1.138% | 0.195% | 0.700% | 66.259% |
| O3_strict | 58.142% | 34.341% | 0.786% | 66.259% |
| All | 61.807% | 34.925% | 3.586% | 135.424% |

TABLE X
PERCENTAGE (%) OF TIMES ($\times 10$) THAT THERE IS A DIFFERENCE BETWEEN ANY OPTIMIZATION LEVEL AND O0_NOFMA, FOR FP32.

| | clang | gcc | nvcc | xlc |
|---|---|---|---|---|
| O0 | 0.227% | 0.000% | 0.622% | 0.730% |
| O1 | 1.042% | 0.268% | 0.622% | 0.682% |
| O2 | 1.088% | 0.268% | 0.622% | 0.682% |
| O3_nostrict | 1.152% | 0.265% | 0.622% | 60.275% |
| O3_strict | 59.631% | 37.499% | 58.737% | 60.275% |
| All | 63.140% | 38.300% | 61.226% | 122.643% |

### H. Experiment 3: Comparisons to O0 without FMA

In this experiment, we compare the results of `-O0_nofma` to all other optimization levels. We consider `-O0_nofma` the least intrusive (or most compliant with IEEE) level in terms of floating-point reproducibility, thus, we want to measure how other levels differ from `-O0_nofma` in all compilers. Tables IX and X show the percentage of times that there is a difference in the results when `-O0_nofma` and any other level. Note that here the comparisons are not made between compilers; instead, comparisons are made in the same compiler between different optimization levels.

We observe no variations at all between `-O0` and `-O0_nofma` for `gcc`, with `clang` as the second one with the least variations in this category. The most substantial variations are observed with `xlc`; these variations are particularly significant when `O3_nonstrict` is used and compared against `-O0_nofma`. We also observe variations even when `O3_strict` is used for `xlc`. In general, the compiler with the smallest amount of variations with respect to `-O0_nofma` is `nvcc`; we found that for `nvcc`, the optimization level does not make a significant difference when results are compared to those from `-O0_nofma`.

### I. Experiments with Large Expressions

We perform experiments with `MAX_EXPRESSION_SIZE=50` to mimic cases when expressions are long as observed in various NPB benchmarks. We do not observe a significant difference in these experiments with respect to the results previously presented with `MAX_EXPRESSION_SIZE=6`. We do observe that there is a slight increase in the number of differences involving *NaN*. We hypothesize that this is due to an increased occurrence of division and multiplication operations due to large arithmetic expressions.

## V. Discussion

**Other Floating-Point Formats.** The bfloat16 floating-point format has been proposed to accelerate machine learning workloads; Tensor Processing Units (TPUs) [14], for example, use this format. In contrast to the half-precision format (FP16), bfloat16 represents floating-point numbers using more bits in the exponent than in FP16, which increases the range of possible values: 1e-38 − 1e+38 in bfloat16 versus 5.96e-8 − 65504 in FP16. This has the disadvantage that bfloat16 is less precise than FP16 since fewer bits are used in the mantissa of bfloat16 numbers. As a result, we expect that comparing bfloat16 results to FP16 results could exhibit significant numerical inconsistencies for some codes and inputs. Our framework VARITY can be extended to such formats and could be used to give insights to programmers on the code, compilers, and flags that could intensify or decrease such differences, e.g., when FP16 is used in the host and bfloat16 is used in a device.

Algorithmic C (AC) datatypes are a C++ library that provides arbitrary-length integer and fixed-point types[6]. These datatypes are designed to provide uniform and consistent semantics so that they are numerically predictable, i.e., there are no corner cases introduced by precision limits. While the use of AC datatypes can significantly reduce in general numerical inconsistencies, comparing results between pure floating-point programs and AC datatypes programs could still produce numerical variations. This could be due to the effect of compiler optimizations, or limitations of AC datatypes, e.g., dividers are limited to consuming a maximum of 64 bits, and the library supports a limited set of math functions. As a result, understanding the source of such variations could be challenging. VARITY could help understand this space by identifying relevant tests. Extending VARITY to support AC datatypes, however, requires extensions to the grammar (to support AC operations) and to the comparison routines.

**FORTRAN Support.** The philosophy of FORTRAN is that the compiler is free to apply to the source code any mathematical identity that is valid over the reals as long as it results in a mathematically equivalent formula. Since the compiler has more freedom when compiling expressions than in C, the performance of a FORTRAN program is likely to be higher than that of the same program written in C; however, this comes at the cost of potentially more floating-point variations across compilers. While VARITY does not support FORTRAN code generation, it can be extended to support it. Most of VARITY's components, such as input generation (see Figure 7(c)–(e)) can be re-used; however, a new grammar would need to be specified to define the space of programs that will be generated and the program generation algorithm must be extended to support FORTRAN loops and conditionals.

[6]https://hlslibs.org/

## VI. Related Work

Random testing [15, 16] has been used as a black-box testing method in which test inputs are generated randomly. Randomized differential testing [8, 17] has been used in previous work to detect bugs in compilers. A notable instance is Csmith [13], which detects compiler bugs in C compilers. The main difference between Csmith and our work is that Csmith detects compiler bugs, whereas our approach finds differences in floating-point results—these differences are not necessarily bugs because of the freedom compilers have in generating floating-point code in languages such as `C`, when optimizations are applied. Also, Csmith is designed to support integer arithmetic (rather than floating-point programs).

Another approach in compiler testing is *program enumeration* [18], which given a template program P generates a set of programs exhibiting all possible variable usage patterns within P to stress-test compilers. Like in Csmith, it focuses on finding compiler bugs and it requires the existence of template programs—we, however, randomly generate previously-unknown programs. Other testing techniques exist to find compiler bugs; an excellent survey of compiler testing techniques is presented in [19].

The FLiT tool [20] uses a set of predefined kernels (43 kernels) to find floating-point variations for different compilation flags in a given compiler. An extension of the tool [21] can identify variations in a real application. FLiT uses a small set of predefined kernels or applications, whereas VARITY can randomly generate a large number of distinct tests (we generate 50,000 distinct tests in our study); FLiT is designed to identify variability within *the same compiler* for multiple compilation flags, whereas our work targets variations *between* different compilers in heterogeneous systems.

Formal methods have been used to verify the compilation of floating-point code. CompCert [22, 23] is a formally-verified compiler which can prove correct compilation of floating-point arithmetic. CompCert comes with a mathematical specification of the semantics of its source language (a large subset of ISO C90) and with a proof that compilation preserves language semantics. Verificalo [24] uses compiler instrumentation to capture the influence of compiler optimizations on the numerical accuracy of programs. Monniaux describes in detail some of the pitfalls in verifying floating-point computations [25]. FPgen [26] is a floating-point test generator, which helps in the verification of processors for floating-point, conforming to the IEEE 754 standard. In contrast to our approach, these approaches focus on a given compiler or floating-point processor.

Herbie [27] helps programmer to automatically discover transformations of mathematical formulas that improve the

floating-point accuracy of the formulas. Herbie intends is to help programmers produce more accurate formulas, while the intention of VARITY is, given a test program, to identify floating-point inconsistencies across a wide range of compilers and floating-point implementations.

Few works study the floating-point differences between GPU devices and host architectures. Whitehead and Fit-Florea [28] give details about floating-point arithmetic in NVIDIA GPUs and how to compare the results from these GPUs to x86 architectures. Hillesland and Lastra [29] develop a benchmark to check the error bounds on floating-point operation results for GPUs. To the best of our knowledge, VARITY is the first tool to compare floating-point results of a device architecture to those of a host architecture for a variety of compilers and compilation options.

## VII. CONCLUSIONS

We present the design and implementation of VARITY, a framework to quantify floating-point variations across compilers and floating-point architectures in heterogeneous HPC systems. We use VARITY to study floating-point variations on four compilers (`gcc`, `clang`, `XL`, and `nvcc`) and two architectures (IBM POWER9 and NVIDIA Tesla V100). Through the generation of 50,000 combinations of test programs and inputs, we have identified several floating-point programs that exhibit significant variations in their results. These variations can occur when different compilers are used to generate programs on the host architecture or when a result comes from a GPU execution versus a result that comes from a host execution. While most of the time, we observe none or small variations of only a few digits, sometimes we observe substantial variations that can be as large as zero versus 1e+300, or as `NaN` (not a number) versus 0.1, even for unoptimized code.

**Future Work.** We plan to extend VARITY to support more floating-point formats, such as half-precision (FP16) and bfloat16; currently, VARITY can generate only single and double precision programs. This work also opens the door for testing and understanding numerical variability in other parallel programming models, such as OpenCL, OpenMP, and performance portability models, such as RAJA [11] and Kokkos [30]. In future work, we plan to use random differential testing to understand numerical differences of OpenMP implementations. Finally, we plan to support floating-point variations that come from nondeterministic executions; this requires modeling the results statistically. VARITY is open source, and it is available at https://github.com/LLNL/Varity.

## REFERENCES


[1] P. Dinda and C. Hetland, "Do Developers Understand IEEE Floating Point?," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 589–598, IEEE, 2018.

[2] W. Kahan, "IEEE standard 754 for binary floating-point arithmetic," *Lecture Notes on the Status of IEEE*, vol. 754, no. 94720-1776, p. 11, 1996.

[3] A. Griffith, *GCC: the complete reference*. McGraw-Hill, Inc., 2002.

[4] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.

[5] IBM, "IBM XL C/C++ Compilers." https://www.ibm.com/us-en/marketplace/ibm-c-and-c-plus-plus-compiler-family, 2019.

[6] B. Lebacki, M. Wolfe, and D. Miles, "The FPGI Fortran and c99 OpenAcc compilers," *Cray User Group*, 2012.

[7] NVIDIA, "Nvidia CUDA Compiler (NVCC)." https://developer.nvidia.com/cuda-llvm-compiler, 2019.

[8] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[9] D. H. Bailey, "Nas parallel benchmarks," *Encyclopedia of Parallel Computing*, pp. 1254–1259, 2011.

[10] CEED, "Laghos (lagrangian high-order solver)." https://computing.llnl.gov/projects/co-design/laghos, 2018.

[11] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: overview and status," tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.

[12] A. Zeller, *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[13] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *Programming Language Design and Implementation*, PLDI 11, (New York, NY, USA), 2011.

[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, IEEE, 2017.

[15] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE transactions on Software Engineering*, no. 4, pp. 438–444, 1984.

[16] R. Hamlet, "Random testing," *Encyclopedia of software Engineering*, 2002.

[17] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 621–631, IEEE, 2007.

[18] Q. Zhang, C. Sun, and Z. Su, "Skeletal Program Enumeration for Rigorous Compiler Testing," in *Programming Language Design and Implementation*, PLDI 2017, 2017.

[19] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 180–190, IEEE, 2016.

[20] G. Sawaya, M. Bentley, I. Briggs, G. Gopalakrishnan, and D. H. Ahn, "FLiT: Cross-platform floating-point result-consistency tester and workload," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 229–238, Oct 2017.

[21] M. Bentley, I. Briggs, G. Gopalakrishnan, D. H. Ahn, I. Laguna, G. L. Lee, and H. E. Jones, "Multi-Level Analysis of Compiler-Induced Variability and Performance Tradeoffs," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 61–72, ACM, 2019.

[22] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond, "A formally-verified C compiler supporting floating-point arithmetic," in *2013 IEEE 21st Symposium on Computer Arithmetic*, pp. 107–115, IEEE, 2013.

[23] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, pp. 107–115, July 2009.

[24] C. Denis, P. D. O. Castro, and E. Petit, "Verificarlo: checking floating point accuracy through Monte Carlo Arithmetic," *arXiv preprint arXiv:1509.01347*, 2015.

[25] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 3, p. 12, 2008.

[26] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, "FPgen–a test generation framework for datapath floating-point verification," in *Eighth IEEE International High-Level Design Validation and Test Workshop*, pp. 17–22, IEEE, 2003.

[27] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically Improving Accuracy for Floating Point Expressions," in *Programming Language Design and Implementation*, PLDI 15, 2015.

[28] N. Whitehead and A. Fit-Florea, "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs," *rn (A+ B)*, vol. 21, no. 1, pp. 18749–19424, 2011.

[29] K. Hillesland and A. Lastra, "GPU floating-point Paranoia," *Proceedings of GP2*, vol. 318, 2004.

[30] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.