Testing GPU Numerics: Finding Numerical Differences Between NVIDIA and AMD GPUs

Anwar Hossain Zahid
Department of Computer Science
Iowa State University
Ames, IA
ahzahid@iastate.edu

Ignacio Laguna
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA
ilaguna@llnl.gov

Wei Le
Department of Computer Science
Iowa State University
Ames, IA
weile@iastate.edu

***Abstract*—As scientific codes are ported between GPU platforms, continuous testing is required to ensure numerical robustness and identify potential numerical differences between platforms. Compiler-induced numerical differences can occur when a program is compiled and run on different GPUs and compilers, and the numerical outcomes are different on the same input. We present a study of compiler-induced numerical differences between NVIDIA and AMD GPUs, two widely used GPUs in HPC cluster. Our approach uses a random program generator (Varity) to generate thousands of short numerical tests in CUDA and HIP, and their inputs; then, we use differential testing to check if the program produced a numerical inconsistency when run on NVIDIA and AMD GPUs, using the same compiler optimization level. We also use the AMD's `HIPIFY` tool to convert CUDA tests into HIP tests and test if there are numerical inconsistencies induced by `HIPIFY`. In our study, we generated more than 600,000 tests and found subtle numerical differences occurring between the two classes of GPUs. We found that some of the differences come from (1) math library calls, (2) differences in floating-point precision (FP64 versus FP32), and (3) converting code to HIP with `HIPIFY`.**

***Index Terms*—differential testing, HIP, CUDA, random program generation**

I. Introduction

Testing scientific software in different platforms is crucial for developers to ensure the consistency, accuracy, and reliability of the numerical results. As scientific software is ported to heterogeneous platform involving different classes of GPUs, it is essential to test the programming environment in such platforms to understand when numerical differences or reproducibility issues emerge. Previous studies have shown that scientific applications can output numerical results that differ from each other when run in the different heterogeneous systems [1], depending on several factors such the compilers used to emit code, the compiler optimizations used, and how the GPU implements floating-point arithmetic.

While NVIDIA GPUs are the most used in HPC clusters, AMD GPUs have been the choice to build recent supercomputers. Exascale computing platforms from the US Department of Energy (DOE), such as Frontier and El Capitan use AMD GPUs (MI250 and MI300 GPUs, respectively). AMD GPUs are also widely used in several areas of scientific research, including machine learning, climate research and genomics. As scientists port codes from NVIDIA GPUs to AMD GPUs, or vice versa, it is crucial to understand the numerical differences and numerical reproducibility challenges that developers can observe in these platforms.

Previous work have studied numerical portability between various GPUs, including AMD GPUs. For example, authors in [2] target math function results relative to earlier versions of GPUs. Li et al. [3] create tests to compare numerical differences and capabilities in NVIDIA and AMD GPUs with a focus on matrix accelerators and tensor cores. While these studies reveal interesting differences between these classes of GPUs, they have several drawbacks: (1) tests are manually generated, which limits the number of floating-point arithmetic expressionsthat are tested; (2) they do not consider compiler optimizations between the GPU compilers—studies have shown that compiler optimizations can be the source of significant differences between GPU platforms [1], [4], [5].

Random program generation has been used before in the `Varity` [6] framework to test numerical differences that are induced by compilers between NVIDIA GPUs and CPUs. The approach generates thousands of random floating-point programs and their numerical inputs, compiles them with multiple optimization flags, runs them on NVIDIA GPUs and CPUs, and identify cases that induce different numerical results. The work in [6], however, did not include tests for HIP, the programming interface for AMD GPUs.

In this paper, we use the approach of random program generation and differential testing to expose *compiler-induced numerical inconsistencies* between NVIDIA and AMD GPUs. These inconsistencies arise when a given test program P and its input I, compiled with the same optimization level, and run on two different platforms, produces different numerical results. For example, consider a test program P with inputs $I = \{0.1, 0.2, 0.3\}$. Suppose we compile P with the HIP compiler `hipcc`, which produces the binary P_{HIP}. Compiling the CUDA version of P with `nvcc` (the NVIDIA compiler) produces the binary P_{NVCC}. Examples of compiler-induced inconsistencies occur when both P_{HIP} and P_{NVCC} are run with the same input I, but they produce different outputs, e.g., 3.1415 versus 3.999, or 3.1415 versus a not-a-number (`NaN`).

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-868447).

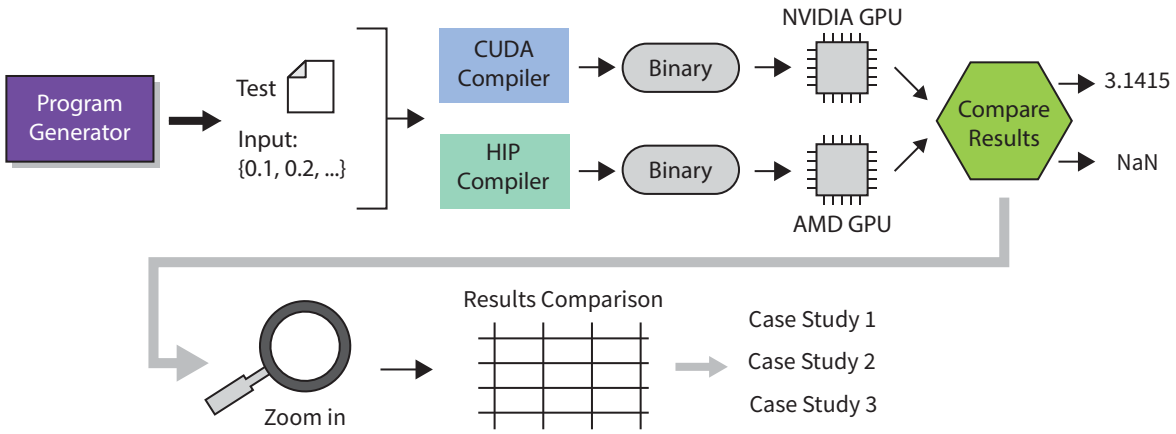

Fig. 1. Overview of testing approach via random program generation for both GPUs (NVIDIA and AMD).

Summary of Contributions. We present a test-guided study about the compiler-induced numerical differences between NVIDIA and AMD GPUs. We use a random program generation approach to generate **thousands** of floating-point arithmetic tests that expose the numerical differences developers can experience when running numerical codes in these GPUs. When such numerical differences are observed, the framework provides to the user a *small test* with the inputs that induce such inconsistency. Such small tests have several benefits for testing HPC systems: (a) the tests are easier to analyze than a large scientific applications—they are composed of a single kernel and a single set of numerical inputs; (b) the tests can be provided to vendors for further investigation—they are self-contained; (c) they can be re-used to test newer systems, potentially during acceptance testing.

Our study uses the `Varity` [6] framework to generate such tests. While `Varity` originally generates CUDA programs, it did not generate HIP tests. We have extended it to generate HIP tests and tested it with the recent ROCm compilers. We perform a large-scale testing campaign and run more than 600,000 tests in these two classes of GPUs. We present several case studies showing how these numerical differences can emerge when using different compiler optimizations.

In addition to generating HIP tests directly through our *extended Varity framework* (hereafter referred to as `Varity`), we use AMD's `HIPIFY`[1] tool to convert CUDA source code into HIP. `HIPIFY` facilitates the porting process between NVIDIA and AMD platforms by automatically translating CUDA applications into HIP. We ran 123,750 tests on `HIPIFY`-converted programs to compare the numerical results with those produced by the HIP tests generated by `Varity`, aiming to identify any discrepancies introduced by the `HIPIFY` conversion.

In summary, our contributions are the following:

- We present an approach to test the numerical differences induced by compiler optimizations between two widely used GPUs in HPC clusters, NVIDIA and AMD GPUs. Our approach extends the `Varity` framework and adds support for testing HIP programs.
- We evaluate the approach in two production clusters at the Lawrence Livermore National Laboratory, one with NVIDIA V100 GPUs (Lassen), and the other one with AMD MI250 GPUs (Tioga). We present a detailed evaluation and results for different levels of optimization on these platforms.
- We use the AMD's `HIPIFY` tool to convert CUDA source code into HIP, identifying and analyzing discrepancies introduced by the automatic code translation.
- We present several case studies that reveal numerical inconsistencies on these GPUs, highlighting discrepancies caused by math functions (e.g., `fmod` and `ceil`), the effects of different optimization levels, and variations in handling special values such as `NaN` and `Infinity`.

[1]https://rocm.docs.amd.com/projects/HIPIFY/en/latest/

II. Background and Overview

In this section, we present background information needed to understand our approach and an overview of the approach.

A. Compiler-Induced Numerical Differences

Compilers used in heterogeneous HPC platforms (e.g., `clang`, `nvcc`, `hipcc`) provide several levels of optimization flags from `-O0` to `-O3`. With higher optimization levels, program performance can be improved, at the cost of potentially generating non-compliant IEEE 754 floating-point code. There are optimization flags that explicitly violate the IEEE 754 standard, but can offer significant speedups. We say that a *compiler-induced numerical difference* (or inconsistency) occurs when a program tested on two different compilers (e.g., a compiler for GPU 1 and another compiler for GPU 2) produced different numerical results, even when the same optimization level is used, e.g., `-O0`.

Table I (taken from [7]) shows an example of a numerical inconsistency between an NVIDIA GPU an a CPU in the BT NAS program—the compiler used in the NVIDIA GPU is `nvcc` and for the CPU `clang`. The Table shows the program runtime and the maximum relative error for each compiler and

TABLE I
INCONSISTENCIES IN BT.S.

Compiler Options	Runtime	Error
nvcc -O0	0.104s	6.98176E-13
nvcc -O3 -use_fast_math	0.052s	9.73738E-13
clang -O0	0.349s	8.32928E-13
clang -O3 -ffast-math	0.059s	3.50905E-12

TABLE II
IEEE 754 STANDARD EXCEPTIONS.

	Event	Description
IEEE 754	`Inexact`	Result is produced after rounding
	`Underflow`	Result could not be represented as normal
	`Overflow`	Result did not fit and it is an infinity
	`DivideByZero`	Divide-by-zero operation
	`Invalid`	Operation operand is not a number (NaN)

optimization flag combination. Using `-O3 -use_fast_math` with nvcc yields 100% speedup compared to `-O0`, but at the cost of the error being 39% larger. The performance and error with Clang is generally worse, with the largest error in `clang -O3 -ffast-math` being 402% larger than `nvcc -O0`.

In real-world scientific applications, such compiler-induced numerical inconsistencies are not uncommon. They can happen when migrating software to other hardware/software platforms, switching applications to a new compiler, or just using more aggressive optimization flags for compilation. These inconsistencies may cause major software failures that take tremendous amount of effort to identify and resolve (see [1]).

B. IEEE 754 Exceptions and Optimizations

A floating-point number has the form

$$x = \pm m \times \beta^e \tag{1}$$

where the sign, the mantissa m, the exponent e can be stored in memory or a register. We assume $\beta = 2$, since this is the most used format for representing floating-point numbers. The IEEE 754 Standard defines five classes of *exceptions*, that can result from arithmetic operations. Table II shows these five events.

The IEEE 754 Standard defines five classes of *exceptions*, that can result from arithmetic operations. Table II shows these five events. When one of these events occurs, the floating-point unit can set a status register specifying which event occurred. Existing frameworks for CPU analysis, such as [8] read these registers to detect the occurrence of such events. Additionally, with the help of the compiler and system routines, they raise a floating-point exception signal (e.g., `SIGFPE`) when these events occur in the CPU. Unlike CPUs, NVIDIA GPUs have no mechanism to detect floating-point exceptions, set a status register or raise a signal when an exception occurs.

*1) **Exceptional Quantities**:* Except for the `Inexact` exception in Table II, the rest of the events will result in either a `NaN` (not a number), `INF` (infinity, positive or negative), or a subnormal number (i.e., a number smaller than a normal floating-point number but that is not zero). More specifically, `Overflow` and `DivideByZero` result in `INF`, and `Invalid` result in `NaN`. `Underflow` can result in zero or a subnormal number—in our case, we are interested in underflows that result in subnormal numbers. The `Inexact` event results in a rounding operation; however, this occurs frequently in numerical programs and it is usually of no interest to programmers. In summary, our goal is to identify inputs that produce any of these cases: `NaN`, `INF+`, `INF-`, or subnormal number quantities (positive or negative).

*2) **Subnormal Numbers**:* Note that while subnormal numbers can represent specific real number quantities, they are often dangerous for several reasons. First, they indicate that computational results are becoming too small to be represented in the current precision. Second, if they propagate to the denominator of a division, the result can produce `INF`. For example, $\frac{1}{x}$, where x =1e-309 (a FP64 subnormal number), produces `INF`[2]. Third, when subnormal numbers are combined with compiler optimizations, they can cause reproducibility issues [4]. Therefore, it is crucial to mitigate them and understanding when they occur as well as `NaN` and `INF`.

C. Approach Overview

Figure 1 shows an overview of the approach. The program generator first generates the source code of tests and inputs to the tests. A file test is generated with extension `.cu` for CUDA and `.hip` for HIP. Then the tests are compiled with the corresponding compilers (`nvcc` and `hipcc`) using the same optimization label, which produces two binaries. The binaries are executed on the NVIDIA and AMD GPUs using the same input. We then compare the numerical results of the tests and find anomalies, i.e., differences. After all tests are executed, meta-data is saved in a JSON file with all the results. We then analyze them and identify several case studies (see Section IV-D).

III. APPROACH

In this section, we present the details of our approach. We first give a high-level overview of `Varity` [6], the approach we use to generate tests. Next, we explain out implementation of HIP test generation in the framework.

A. Varity's Framework

Here, we present a high-level description of the `Varity` framework. More details can be found in the paper [6]. `Varity` generates random programs that expose a wide range of floating-point arithmetic operations, and other structures encountered in scientific codes, such as, `for` loops, and `if` conditions. `Varity` generates also random floating-point inputs for the programs. `Varity` was originally designed to generate C/C++ tests and CUDA tests. In this work, we extend it to support the generation of HIP tests.

[2]This behavior may depend on the platform, compiler, and optimizations applied to the code.

TABLE III
CHARACTERISTICS OF THE RANDOM PROGRAMS

Floating-Point Types	Variables using single and double floating-point precision (i.e., `float` and `double`).
Arithmetic Expressions	Arithmetic expressions can use any operator in {`+`, `-`, `*`, `/`}, parenthesis "( )", and functions from the `C math` library. The grammar also allows boolean expressions.
Loops	`for` loops with multiple levels of nesting. 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.
Conditions	`if` conditions, which can be true or false based on a boolean expression.
Variables	Programs can contain temporal floating-point variables. Variables can store arrays or single values.

```
__global__
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) {
  if (comp == -1.3857E-36 + var_2) {
    double tmp_1 = +1.3305E12 / var_3;
    comp += -1.7744E-2 * tmp_1;
    comp += cos(var_4 - +1.4014E2 * (var_5 + var_6 *
        var_7));
    for (int i=0; i < var_1; ++i) {
      comp -= sqrt(var_8 + -1.7976E3);
    }
  }
  printf("%.17g\n", comp);
}
```

Fig. 2. Example of a simple test random program in FP64 precision.

*1) **Grammar to Define Possible Tests**:* A grammar is defined to specify the set of possible tests (or programs) that the tool can generate. Due to space limitations, we do not show the grammar int his paper, but it can be found in [6]. `Varity`'s grammar considers 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 characteristics of the programs that `Varity` generates are shown in Table III.

B. Program Output

All 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, the generators generates 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. Figure 2 shows a sample random test.

C. FP32 and FP64 Support

Our approach supports the generation of test programs in FP32 and FP64 precision. This is crucial because some compilers optimization may behave differently depending on the precision of arithmetic operations. There is a configuration option that controls the types for variables and functions used in the generation phase. When using FP64, all types are `double`. When using FP32, all types are `float`; this case includes using the FP32 math functions where function names and constants end with `f`, e.g., `cosf()`, instead of `cos()`, and $1.23F$ instead of 1.23.

D. HIP Extensions

To support HIP, we take advantage of the fact that HIP is considered to be a subset of CUDA; thus, several of the CUDA constructs can be reused in HIP—for example, a GPU kernel is declared as `__global__` in both APIs. There are, however, a few differences in the API that we considered in our `Varity` implementation, for example, the kernel launch API in CUDA uses `<<< >>>`, while HIP uses `hipLaunchKernel`.

Compiler Matching. Generating random tests for a given platform involves deciding which compiler to use for CUDA or HIP. Compiler matching is done automatically depending on the program extensions—a random test file ends with `.cu` is automatically compiled with `nvcc`, while HIP files are compiled with `hipcc`, compiler driver utility from AMD ROCm.

Fast Math Optimizations. The `-ffast-math` flag is a compiler option that enables a suite of optimizations designed to improve the performance of floating-point arithmetic by making several assumptions about the mathematical operations, such as no NaNs or Infinities, relaxed precision, and others. This flag includes optimizations like `-fno-math-errno`, `-funsafe-math-optimizations`, `-fno-trapping-math`, `-fassociative-math`, `-freciprocal-math`, `-fno-signed-zeros`, `-fno-rounding-math`, and `-ffinite-math-only`. While these optimizations work effectively in CUDA, they create issues in HIP when dealing with NaNs and Infinities, particularly due to the `-ffinite-math-only` flag. As `Varity` generates tests that may produce NaNs or infinities, the `-ffast-math` flag in HIPCC leads to errors. Therefore, the ROCm developers recommend using the `-DHIP_FAST_MATH`[3] flag instead, which provides similar performance benefits without causing the same issues. In summary, while CUDA handles the `-ffast-math` flag without problems, we are bound to use `-DHIP_FAST_MATH` in HIP to avoid issues with special floating-point values.

E. Between-Platform Comparisons

GPUs from different vendors are usually installed on different HPC clusters, a cluster with NVIDIDA GPUs typically

[3]https://github.com/ROCm/HIP/issues/28

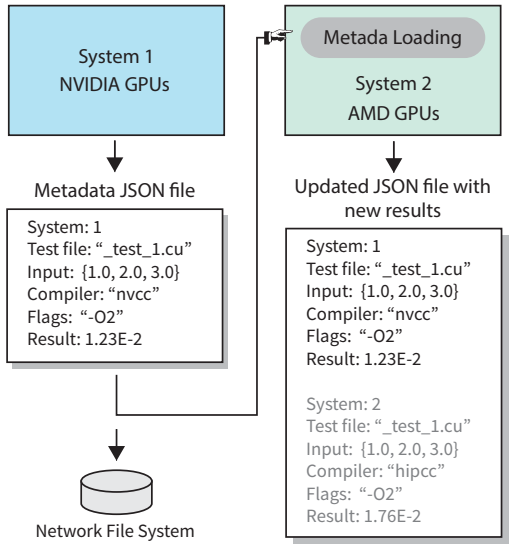

Fig. 3. Process to perform between-platform comparisons.

does not have AMD GPUs and vice versa. Comparing numerical results between these GPUs require running the same random tests in both clusters. Our testing approach supports this by storing metadata of the tests generated in cluster C_1, which is then used in cluster C_2 to run exactly *the same* tests and inputs.

Figure 3 shows the process of between-platform testing. We first run in system C_1. After all experiments are run, `Varity` saves a JSON metadata file containing details about the tests, inputs, compilers used and results. Then, this metadata along with the tests are transferred to system C_2. In system C_2, we load the metadata, locate the tests we used in C_1, recompile them with the appropriate compiler, and run the experiments again using different GPUs. We then saves a new metadata file that contains all the results. We use this new JSON file to analyze the findings and search for cases with numerical inconsistencies.

F. Enabling HIPIFY

`HIPIFY` is a set of tools that developers can use to automatically translate CUDA source code into HIP source code. Since application developers may use `HIPIFY` to translate CUDA applications into HIP applications, it is important to understand if numerical differences occur in this approach. We implement two approaches to generate HIP tests. The first approach simply creates random tests using the random test generator we use for generating CUDA tests. The second approach first generates a CUDA test and then uses `HIPIFY` to translate the randomly generated CUDA source code into HIP source code.

IV. Evaluation

In this section, we evaluate our approach on two high-performance computing systems, each employing NVIDIA and AMD GPUs. We begin by summarizing the numerical inconsistencies identified in our experiments, followed by an in-depth analysis of three case studies that provide insights into the sources of these discrepancies.

Our evaluation addresses the following research questions:

Q1 How effective is our approach in detecting compiler-induced numerical inconsistencies between NVIDIA and AMD GPUs?

Q2 What specific classes of inconsistencies are revealed through our testing?

Q3 Can we determine the root causes of these inconsistencies?

A. Systems and Software

1) System 1: Lassen with NVIDIA GPUs: We conducted our experiments on the Lassen system at Lawrence Livermore National Laboratory (LLNL). Lassen is equipped with NVIDIA V100 GPUs and is a smaller-scale version of the classified Sierra system, offering a peak performance of 23 petaflops compared to Sierra's 125 petaflops. Notably, Lassen was ranked #10 on the June 2019 Top500 list. The system comprises IBM Power9 CPUs with 44 cores per node, totaling 34,848 cores, alongside 3,168 NVIDIA V100 GPUs. We utilized CUDA version 12.2.2, the latest available on Lassen, and ran experiments using Python 3.9.12. The built-in Python libraries were sufficient for our needs, eliminating the requirement for a separate Python environment. The operating system used was Red Hat Enterprise Linux (RHEL).

2) System 2: Tioga with AMD GPUs: For experiments involving AMD hardware, we used the Tioga system at LLNL, which features 128 AMD MI-250X GPUs. Tioga is powered by AMD Trento CPUs, with 64 cores per node, amounting to a total of 2,048 cores. We used ROCm version 6.1.2, AMD's open-source platform for GPU programming, to run our experiments. The system operates on TOSS 4, a customized version of the Linux operating system. As with Lassen, all the tests on Tioga were conducted using Python 3.9.12, and the default Python libraries were adequate for our experiment.

B. Experiments Configuration

We executed a total of **652,600** experimental instances, each consisting of a program and input combination. We tested these instances under two settings: double precision (FP64) and single precision (FP32). We also used the HIPIFY tool to convert FP64 tests from CUDA to HIP.

Due to resource constraints, we divided the tests into multiple batches, executed each batch separately, and then compiled the results into a comprehensive dataset, as described in Subsection III-E. We evaluated each generated code across **five optimization levels**: `O0`, `O0`, `O2`, `O3`, and `O3` with the `-ffast-math` flag. `O0` represents no optimization, while `O1` through `O3` denote progressively more aggressive optimization levels. The `-ffast-math` flag further increases optimization aggressiveness by prioritizing speed over numerical precision, potentially sacrificing accuracy for performance gains.

We identified four possible outcomes from any test: `NaN` (Not a Number), `Inf` (Infinity), `Zero`, and `Number`[4]. We

[4]The term "Number" refers to a non-zero real-valued floating-point number.

TABLE IV
SUMMARY OF EXPERIMENTAL RESULTS

Metric	FP64	FP64 with hipify	FP32
Total Programs	3,540	3,540	2,840
Total Runs per Option per Compiler	24,750	24,750	15,760
Total Runs per Option	49,500	49,500	31,520
Total Runs	**247,500**	**247,500**	**157,600**
Runs on NVCC	123,750	123,750	78,800
Runs on HIPCC	123,750	123,750	78,800
Total Discrepancies	**2,426**	**2,716**	**14,188**
Total Discrepancies (% of Total Runs)	*0.98%*	*1.10%*	*9.00%*

categorized the observed discrepancies into **seven distinct types** based on the results from the `nvcc` and `hipcc` compilers: `NaN vs. Inf`, `NaN vs. Zero`, `NaN vs. Number`, `Inf vs. Zero`, `Inf vs. Number`, `Zero vs. Number`, and `Number vs. Number`.

We make this distinction between `Zero` and `non-zero floating point numbers` to emphasize how computations involving near-subnormal or near-infinite values can lead to zero results due to rounding or precision errors. This distinction is important because extreme values often result in zero. However, we excluded inconsistencies such as `-NaN vs. +NaN`, `-Inf vs. +Inf`, and `-Zero vs. +Zero`, as these do not represent true numerical differences.

C. Experimental Results

Table IV summarizes the experimental results, detailing key metrics for FP64, HIPIFY-converted FP64, and FP32 tests across various optimization levels. We conducted a total of `652,600` runs: `247,500` each for FP64 and HIPIFY-converted FP64 tests, and `157,600` for FP32 tests. We observe discrepancies in **0.98%** of the FP64 runs, **1.10%** of HIPIFY-converted FP64 runs, and **9.00%** of FP32 runs, highlighting differences in numerical consistency across precision settings and compiler environments.

Answer to Q1: Our approach is effective at identifying compiler-induced numerical inconsistencies between NVIDIA and AMD GPUs. As shown in Table IV, we observed discrepancies in 0.98% of the 247,500 runs conducted under FP64 settings. In tests using HIPIFY-converted FP64 settings, discrepancies increased to 1.10%, indicating a significant difference. For FP32 tests, discrepancies were found in 9% of the 157,600 runs. **These results demonstrate the effectiveness of our approach in detecting numerical inconsistencies.**

1) FP64 Tests: Table V categorizes discrepancies for FP64 tests by optimization level. While `O3` with `-ffast-math` (`O3_FM`) exhibited the highest discrepancies at `519`, the `O0` setting also had a significant count at `440`, suggesting that both aggressive and no optimizations can lead to variations. The Number vs. Number discrepancies were the most frequent across all optimization settings, indicating persistent challenges in achieving numerical consistency. Table VI presents adjacency matrices for various optimization levels.

2) HIPIFY-Converted FP64 Tests: This experiment allowed us to test whether the HIPIFY tool introduces additional inconsistencies during the conversion from CUDA to HIP. We found that `HIPIFY introduces discrepancies in floating-point computations` between NVIDIA and AMD architectures. Table VII categorizes the discrepancies observed for HIPIFY-converted tests across different optimization levels. The results reveal that `O3` with `-ffast-math` (`O3_FM`) exhibits the highest discrepancy count at `575`, while `O0` records `494` discrepancies. Discrepancies in the Number vs. Number category remain predominant across all optimization levels, like the previous `FP64` tests. Table VIII presents adjacency matrices for various optimization levels.

3) FP32 Tests: Single-precision floating-point computations are prevalent in machine learning applications running on 32-bit systems. Table IX categorizes the discrepancies observed per optimization flag for FP32 tests, revealing a stark contrast in numerical consistency compared to FP64 tests. The results show that the `O3` with `-ffast-math` (`O3_FM`) option yielded the highest discrepancy count at `13,877`, compared to only `45` discrepancies in the `O0` setting. This significant increase highlights the aggressive nature of `O3_FM` optimizations, which prioritize speed and may sacrifice numerical precision. Notably, the Number vs. Number category accounted for the majority of discrepancies, indicating persistent challenges in maintaining consistency. Table X provides a comprehensive analysis of discrepancies across various optimization levels.

Answer to Q2: We identified seven classes of numerical inconsistencies. These discrepancies arise from differences in floating-point computations between NVIDIA and AMD GPUs. In our FP64 tests, we did not observe NaN vs. Zero and NaN vs. Number discrepancies, indicating that a larger search space may be necessary to detect them, if they exist. In FP32 tests, discrepancies such as NaN vs. Inf, Nan vs. Num, and Num vs. Zero predominantly occurred under the `-O3` flag with `-ffast-math` enabled. **Overall, we observed all classes of inconsistencies across various tests**.

D. Case Studies

We focus on three interesting cases in this section.

1) Case Study 1: Inconsistency in Real-Valued Results: During our experiments, `Varity` generated the code shown in Figure 4 and it comprises three parts. The first part shows a code snippet. Upon inspection, the **compute** method performs several floating-point operations, including division and the use of the `fmod` function for remainders. A conditional statement determines whether the code enters a loop, where we identified variations in the results generated within the loop.

In the second part, we see the failure-inducing input. For the same input, the `nvcc` compiler produced an output of `8.6551990944767196e-306`, while the `hipcc` compiler produced `9.3404611450291972e-306`. To determine the cause of this discrepancy, we analyzed the intermediate results and the assembly code generated by the compilers.

TABLE V
DISCREPANCIES PER OPTIMIZATION OPTION FOR FP64 TESTS

Opt Flags	Disc. Count	NaN, Inf	NaN, Zero	NaN, Num	Inf, Zero	Inf, Num	Num, Zero	Num, Num
O0	**440**	7	0	0	24	46	10	**353**
O1	**489**	22	0	0	24	46	10	**387**
O2	**489**	22	0	0	24	46	10	**387**
O3	**489**	22	0	0	24	46	10	**387**
O3_FM	**519**	**32**	**0**	**0**	**24**	**52**	**10**	**401**
Total	**2,426**	**105**	**0**	**0**	**120**	**236**	**50**	**1,915**

0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%

TABLE VI
ADJACENCY MATRICES FOR DIFFERENT OPTIMIZATION LEVELS FOR FP64 TESTS

Opt Flags	NVCC\HIPCC	(±) NaN	(±) Inf	(±) Zero	Num
O0	**(±) NaN**	—	7, 0	0, 0	0, 0
	(±) Inf	—	—	8, 16	4, 42
	(±) Zero	—	—	—	0, 10
	Num	—	—	—	**353, 353**
O1	**(±) NaN**	—	19, 3	0, 0	0, 0
	(±) Inf	—	—	8, 16	4, 42
	(±) Zero	—	—	—	0, 10
	Num	—	—	—	**387, 387**
O2	**(±) NaN**	—	19, 3	0, 0	0, 0
	(±) Inf	—	—	8, 16	4, 42
	(±) Zero	—	—	—	0, 10
	Num	—	—	—	**387, 387**
O3	**(±) NaN**	—	19, 3	0, 0	0, 0
	(±) Inf	—	—	8, 16	4, 42
	(±) Zero	—	—	—	0, 10
	Num	—	—	—	**387, 387**
O3_FM	**(±) NaN**	—	**29, 3**	**0, 0**	**0, 0**
	(±) Inf	—	—	**8, 16**	**4, 48**
	(±) Zero	—	—	—	**0, 10**
	Num	—	—	—	**401, 401**

We found that until satisfying the condition and entering the loop, there were no issues with this input. During the first iteration, the expression `(-1.7538E305 * (var_8 / (+0.0 / var_9 - +1.3065E-306)))` was computed, resulting in the same value on both devices: `1.5917195493481116e+289`. The discrepancy arose when this result was passed to the `fmod` function. The `nvcc` compiler computed `fmod(1.5917195493481116e+289, 1.5793E-307)` as `1.4424471839615771e-307`, whereas the `hipcc` compiler produced `7.1923082856620736e-309`. This small numerical difference then propagated through subsequent iterations, magnified with each loop iteration, and by the end of the loop, these differences compounded, resulting in significantly different final outputs on the two devices, as shown in the third part of the Figure.

Further examination of the assembly code revealed that the

```
__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) {
  if (comp >= (var_2 * (var_3 + var_4))) {
    for (int i = 0; i < var_1; ++i) {
      var_5[i] = -0.0 / -1.5942E305 - fmod((+1.7085E
          -315 + var_6), -1.9289E305 / (-1.2924E-311
          - +0.0 + var_7 + +1.3278E-316));
      comp += var_5[i] - fmod((-1.7538E305 * (var_8 /
          (+0.0 / var_9 - +1.3065E-306))), +1.5793E
          -307);
      comp += +1.8753E-306 + var_10;
    }
  }
  printf("%.17g\n", comp);
}
```

```
Input: +0.0 5 +1.7612E-322 +1.1649E-307 -0.0 +0.0
    +1.5461E-311 -1.3680E306 +1.1757E-322 +1.7130E
    -319 +1.6782E-321

Output:
nvcc -O0: 8.6551990944767196e-306
hipcc -O0:  9.3404611450291972e-306
```

```
Expression: fmod(1.5917195493481116e+289, 1.5793E
    -307)

Output:
nvcc -O0: 1.4424471839615706780e-307
hipcc -O0: 7.1923082856620736029e-309
```

Fig. 4. Small Numerical Variation with No Optimization (`-O0`)

`fmod` function is implemented differently on the two platforms. For the `hipcc` compiler, the `fmod` function is a clang-style function named `__ocml_fmod_f64`, which is called each time the function is invoked. This implementation is part of the AMD GPU Instruction Set Architecture (ISA). In contrast, the `nvcc` compiler uses a combination of floating-point arithmetic and bitwise manipulation within its SASS (String Assembler) and PTX (Parallel Thread Execution) assembly languages to implement the `fmod` function.

Interestingly, out of ten randomly generated inputs, only this specific input created a discrepancy. Other inputs, such as `"-0.0 5 +0.0 +1.2150E-306`

TABLE VII
DISCREPANCIES PER OPTIMIZATION OPTION FOR HIPIFY CONVERTED FP64

Opt Flags	Disc. Count	NaN, Inf	NaN, Zero	NaN, Num	Inf, Zero	Inf, Num	Num, Zero	Num, Num
O0	**494**	3	☐ 0	☐ 0	12	20	20	**439**
O1	**549**	23	☐ 0	☐ 0	12	20	20	**474**
O2	**549**	23	☐ 0	☐ 0	12	20	20	**474**
O3	**549**	23	☐ 0	☐ 0	12	20	20	**474**
O3_FM	**575**	**36**	☐ **0**	☐ **0**	**12**	**27**	**20**	**480**
Total	**2,716**	**108**	☐ **0**	☐ **0**	**60**	**107**	**100**	**2,341**

0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%

TABLE VIII
ADJACENCY MATRICES FOR DIFFERENT OPTIMIZATION LEVELS FOR HIPIFY CONVERTED FP64

Opt Flags	NVCC\HIPCC	(±) NaN	(±) Inf	(±) Zero	Num
O0	**(±) NaN**	—	0, 3	0, 0	0, 0
	(±) Inf	—	—	8, 4	2, 18
	(±) Zero	—	—	—	0, 20
	Num	—	—	—	**439, 439**
O1	**(±) NaN**	—	17, 6	0, 0	0, 0
	(±) Inf	—	—	8, 4	2, 18
	(±) Zero	—	—	—	0, 20
	Num	—	—	—	**474, 474**
O2	**(±) NaN**	—	17, 6	0, 0	0, 0
	(±) Inf	—	—	8, 4	2, 18
	(±) Zero	—	—	—	0, 20
	Num	—	—	—	**474, 474**
O3	**(±) NaN**	—	17, 6	0, 0	0, 0
	(±) Inf	—	—	8, 4	2, 18
	(±) Zero	—	—	—	0, 20
	Num	—	—	—	**474, 474**
O3_FM	**(±) NaN**	—	**30, 6**	**0, 0**	**0, 0**
	(±) Inf	—	—	**8, 4**	**2, 25**
	(±) Zero	—	—	—	**0, 20**
	Num	—	—	—	**480, 480**

`+1.2318E224 +1.8418E306 +1.6483E-306 -1.2836E214 -1.0053E305 +1.3789E213 +0.0"` or `"-1.2640E305 5 -0.0 -1.3396E-322 +1.7693E-307 +1.3050E305 +1.4789E-316 -1.1350E-165 +1.7826E305 +0.0 -1.0755E-317"`, produced consistent results between the two platforms. This case study highlights the challenges in achieving consistent floating-point computations across two prominent GPU architectures due to differences in *math functions*.

2) Case Study 2: Inconsistencies in Infinity-Valued Results: Figure 5 shows the code & input-output generated by `Varity`, comprising three parts like the previous case study. The first part shows the **compute** method that performs a division operation involving the `ceil` function. The code was tested using both NVIDIA and AMD compilers to compare the results.

```
__global__ /* __global__ is used for device run */
void compute(double comp) {
double tmp_1 = +1.1147E-307;
comp += tmp_1 / ceil(+1.5955E-125);
   printf("%.17g\n", comp);
}
```

```
Input: +1.2374E-306

Output:
nvcc -O0: Inf
hipcc -O0:  1.34887e-306
```

```
Expression: ceil(1.5955E-125)

Output:
nvcc -O0:  0
hipcc -O0: 1
```

Fig. 5. Numerical variation with Infinity Without Optimization (`-O0`)

In the second part, we see a randomly generate discrepancy-inducing input, `+1.2374E-306`. With this input, the `nvcc` compiler produced an output of `Inf`, while the `hipcc` compiler produced `1.34887e-306`. To determine the cause of this variation, we analyzed the intermediate results and the assembly code generated by the compilers.

The first statement of the `compute` method is an assignment statement and is the same for both devices. However, the discrepancy emerged when the `ceil` function was applied to `+1.5955E-125` in the following statement. The `nvcc` compiler computes `ceil(+1.5955E-125)` as 0, while the `hipcc` compiler results in 1. This difference in the `ceil` function's result led to a **division by zero** on the NVIDIA device, causing the `comp` value to become `Inf`. In contrast, the AMD device performed the division without issues, yielding `1.34887e-306`. The third part of Figure 5 shows the difference in outputs for the same function call, `ceil(+1.5955E-125)`, across `nvcc` and `hipcc` compilers.

Further analysis revealed significant differences in how each platform implemented the `ceil` function. NVIDIA's SASS and PTX assembly languages resulted in zero, while AMD's GPU ISA produced the expected result of 1.

TABLE IX
DISCREPANCIES PER OPTIMIZATION OPTION FOR FP32 TESTS

Opt Flags	Disc. Count	NaN, Inf	NaN, Zero	NaN, Num	Inf, Zero	Inf, Num	Num, Zero	Num, Num
O0	**45**	5	□ 0	□ 0	□ 0	□ 0	□ 0	**40**
O1	**86**	24	□ 0	□ 6	□ 0	□ 0	□ 0	**56**
O2	**90**	28	□ 0	□ 6	□ 0	□ 0	□ 0	**56**
O3	**90**	28	□ 0	□ 6	□ 0	□ 0	□ 0	**56**
O3_FM	**13877**	**2328**	**619**	**2134**	**259**	**1056**	**5369**	**2112**
Total	**14188**	**2413**	**619**	**2152**	**259**	**1056**	**5369**	**2320**

0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%

TABLE X
ADJACENCY MATRICES FOR DIFFERENT OPTIMIZATION LEVELS FOR FP32 TESTS

Opt Flags	NVCC\HIPCC	(±) NaN	(±) Inf	(±) Zero	Num
O0	**(±) NaN**	—	0, 5	0, 0	0, 0
	(±) Inf	—	—	0, 0	0, 0
	(±) Zero	—	—	—	0, 0
	Num	—	—	—	**40, 40**
O1	**(±) NaN**	—	19, 5	0, 0	6, 0
	(±) Inf	—	—	0, 0	0, 0
	(±) Zero	—	—	—	0, 0
	Num	—	—	—	**56, 56**
O2	**(±) NaN**	—	23, 5	0, 0	6, 0
	(±) Inf	—	—	0, 0	0, 0
	(±) Zero	—	—	—	0, 0
	Num	—	—	—	**56, 56**
O3	**(±) NaN**	—	23, 5	0, 0	6, 0
	(±) Inf	—	—	0, 0	0, 0
	(±) Zero	—	—	—	0, 0
	Num	—	—	—	**56, 56**
O3_fast_math	**(±) NaN**	—	**101, 2227**	**113, 506**	**139, 1995**
	(±) Inf	—	—	**190, 69**	**197, 859**
	(±) Zero	—	—	—	**161, 5208**
	Num	—	—	—	**2112, 2112**

3) Case Study 3: Inconsistencies in Infinity vs. NaN: In this case study shown in Figure 6, the `compute` method performs several floating-point operations, such as calculating absolute values and using conditional statements. None of these operations cause any discrepancies, making this analysis particularly interesting. Initially, when no optimization flags are applied (`-O0`), both NVCC and HIPCC compilers produce consistent results, yielding `-inf` for the provided input values. This consistency suggests that, without optimizations, both compilers handle floating-point operations similarly.

However, discrepancies arise when any optimization flags are introduced. Notably, the `nvcc` compiler continues to produce `-inf`, while the `hipcc` compiler results in `-nan`. This divergence is not attributed to typical mathematical function behavior or rounding errors but rather to how optimizations impact the computation flow. Through detailed analysis, we found that discrepancies emerge specifically after the optimization modifies intermediary computations within the code.

```
__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 tmp_1 = (-1.8007E-323 - cosh(var_2 / -1.7569
       E192 + (-1.9894E-307 / +1.7323E-313 + var_3)));
  comp += tmp_1 + fabs(+1.5726E-307 - var_4);
  for (int i = 0; i < var_1; ++i) {
    comp += (+1.9903E306 / var_5);
  }
  if (comp >= (-1.4205E305 - (-1.4055E-312 * (var_6 +
       -1.7892E214 / var_7)))) {
    comp += +1.3803E305 * var_8;
  }
  printf("%.17g\n", comp);

}
```

```
Input: -1.5548E-320 5 +1.9121E306 +0.0 -1.1577E124
     -1.8994E-311 +1.3675E306 +1.1296E-318 +1.2915E306

Output:
nvcc -O0: -inf
hipcc -O0:  -inf
```

```
Input: -1.5548E-320 5 +1.9121E306 +0.0 -1.1577E124
     -1.8994E-311 +1.3675E306 +1.1296E-318 +1.2915E306

Output:
nvcc -O1: -inf
hipcc -O1:  -nan
```

Fig. 6. Numerical variation with Infinity & NaN Without Optimization (`-O0`)

The statement `comp += tmp_1 + fabs(+1.5726E-307 - var_4)` within the `compute` method produces `-inf` across all optimization levels and on both NVIDIA and AMD GPUs. This `-inf` value propagates through the subsequent operations, influencing the control flow when it encounters the `if` statement.

As `comp` is compared within the branch, it remains `-inf`. However, once the execution exits the loop, optimizations potentially alter the sequence of calculations, leading to `comp` becoming `-nan`. This suggests that the optimization may adjust how intermediary values are handled, resulting in a propagation of `-nan` instead of `-inf`. The change from `-inf` to `-nan` in `hipcc` under optimization likely results from the

reordering or elimination of intermediate steps, underscoring the complexities involved in maintaining consistency across different GPU architectures and compiler optimizations.

Answer to Q3: We identified the root causes of several inconsistencies. The root causes vary in each case. A key reason is differences in the low-level implementation of mathematical functions such as `fmod` and `ceil`. As demonstrated in Case Study 1, the `fmod` function was identified as a source of inconsistency. Compiler optimizations, particularly aggressive ones like `-ffast-math`, also contribute to discrepancies by affecting computation stability. Additionally, inconsistencies can arise from how intermediary values are handled, due to different optimization strategies across compilers. **These factors collectively lead to variations in outputs on different GPU architectures.**

V. Related Work

The `Varity` framework [6], [9], introduced by Laguna, utilizes random program generation to quantify floating-point variations in HPC systems, focusing on host-to-host and host-to-device testing. Our work extends this framework by incorporating device-to-device testing, offering a more detailed evaluation of numerical discrepancies between NVIDIA and AMD GPUs. Previous studies have examined the trade-offs between performance and numerical accuracy in compiler optimizations. Bentley et al. [5], [10] conducted a multi-level analysis of compiler-induced variability, while Guo et al. [4] developed PLiner to isolate lines of floating-point code contributing to variability. Chowdhary and Nagarakatte [11] introduced parallel shadow execution to accelerate the debugging of numerical errors, highlighting the need for efficient tools in this domain.

Recent research has focused on detecting floating-point exceptions and inconsistencies across GPU architectures. Li et al. [12] designed GPU-FPX, a tool for floating-point exception detection, and Innocente and Zimmermann [2] analyzed the accuracy of mathematical functions across different precision levels. Augonnet et al. [13] and Cao [14] explored task scheduling on heterogeneous architectures and the implications of using HIP technology on GPU-like accelerators, respectively, contributing to our understanding of how different architectures handle floating-point computations. In the context of heterogeneous systems, Miao et al. [7] emphasized the importance of understanding compiler-induced inconsistencies. Additionally, Lopez-Novoa et al. [15] surveyed performance modeling techniques for accelerator-based computing, setting the stage for further exploration of discrepancies in numerical results between different GPU platforms. Finally, the adoption of alternative numerical formats like Posits has been explored as a means to enhance accuracy in HPC. Poulos et al. [16] discussed the potential of Posits in improving precision in scientific applications, particularly in the context of exascale and edge computing. These studies underscore the ongoing challenges and opportunities in achieving consistent floating-point computations across different architectures and optimization levels.

VI. Limitations

Our study has limitations. First, since programs are generated randomly, we have no guarantee that these programs reflect the code in real scientific simulations. However, the root cause of several cases arise from simple arithmetic and math operations that are likely be part of several larger codes. Second, the `-ffast-math` option may not be directly comparable between `nvcc` and `hipcc` as they most likely implement optimizations in different ways; thus, programmers must be careful when enabling them on code that runs on both GPUs. Third, while most modern compilers use similar intermediate representations (e.g., LLVM IR), in general optimization levels are not necessarily comparable across compilers, i.e., `-O2` in `nvcc` may not perform the same optimizations than `-O2` in `hipcc`. Nevertheless, it is crucial to understand when numerical inconsistencies occur because programmers often use the same optimization level when moving and running code between different platforms.

VII. Conclusion

Testing for numerical consistency between GPU platforms is crucial to ensure the correctness and robustness of numerical simulations. As new HPC systems include different classes of GPUs, acceptance testing is required to ensure system integrity and to fix issues before production use—therefore testing for numerical consistency between systems is essential. This paper presents a study of compiler-induce numerical inconsistencies between NVIDIA and AMD GPUs, which uses differential testing and random program generation. We extended `Varity` to enable device-to-device testing, uncovering significant numerical discrepancies between NVIDIA and AMD GPUs. These discrepancies stem from various factors, including differences in the implementation of math functions and discrepancies introduced by compiler optimizations, such as `-ffast-math`. Furthermore, based on our analysis of assembly code, we conclude that intermediary value handling differences across compilers seems to contribute to these inconsistencies.

In future work, we will investigate the root causes of mismatches between HIPIFY-converted code and `Varity`-generated code. Additionally, we aim to develop automated debugging tools to efficiently identify and resolve these inconsistencies, minimizing manual analysis, and enhancing numerical consistency across heterogeneous computing platforms.

References

[1] D. H. Ahn, A. H. Baker, M. Bentley, I. Briggs, G. Gopalakrishnan, D. M. Hammerling, I. Laguna, G. L. Lee, D. J. Milroy, and M. Vertenstein, "Keeping science on keel when software moves," *Communications of the ACM*, vol. 64, no. 2, pp. 66–74, 2021.

[2] V. Innocente and P. Zimmermann, "Accuracy of mathematical functions in single, double, double extended, and quadruple precision," 2021.

[3] X. Li, A. Li, B. Fang, K. Swirydowicz, I. Laguna, and G. Gopalakrishnan, "Fttn: Feature-targeted testing for numerical properties of nvidia & amd matrix accelerators," *arXiv preprint arXiv:2403.00232*, 2024.

[4] H. Guo, I. Laguna, and C. Rubio-González, "pliner: isolating lines of floating-point code for compiler-induced variability," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, IEEE, 2020.

[5] 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, 2019.

[6] I. Laguna, "Varity: Quantifying floating-point variations in hpc systems through randomized testing," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 622–633, IEEE, 2020.

[7] D. Miao, I. Laguna, and C. Rubio-González, "Expression isolation of compiler-induced numerical inconsistencies in heterogeneous code," in *International Conference on High Performance Computing*, pp. 381–401, Springer, 2023.

[8] P. Dinda, A. Bernat, and C. Hetland, "Spying on the floating point behavior of existing, unmodified scientific applications," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 5–16, 2020.

[9] I. Laguna, "Varity: Quantifying Floating-Point Variations in HPC Systems Through Randomized Testing." https://github.com/LLNL/Varity, 2020. Accessed: 2024-08-02.

[10] 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 IEEE/ACM International Symposium on Code Generation and Optimization*, 2018.

[11] S. Chowdhary and S. Nagarakatte, "Parallel shadow execution to accelerate the debugging of numerical errors," *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 757–770, 2021.

[12] X. Li, I. Laguna, B. Fang, K. Swirydowicz, A. Li, and G. Gopalakrishnan, "Design and evaluation of gpu-fpx: A low-overhead tool for floating-point exception detection in nvidia gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 374–387, 2023.

[13] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, pp. 187–198, 2010.

[14] K. Cao, "Gpu-hadvppm4hip v1.0: higher model accuracy on china's domestically gpu-like accelerator using heterogeneous compute interface for portability (hip) technology to accelerate the piecewise parabolic method (ppm) in an air quality model (camx v6.10)," 2024.

[15] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso, "A survey of performance modeling and simulation techniques for accelerator-based computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 272–281, 2015.

[16] A. Poulos, S. A. McKee, and J. C. Calhoun, "Posits and the state of numerical representations in the age of exascale and edge computing," *Software: Practice and Experience*, vol. 52, pp. 619–635, 2021.