FloatGuard: Efficient Whole-Program Detection of Floating-Point Exceptions in AMD GPUs

Dolores Miao
University of California, Davis
Davis, USA
wjmiao@ucdavis.edu

Ignacio Laguna
Lawrence Livermore National Laboratory
Livermore, USA
ilaguna@llnl.gov

Cindy Rubio-González
University of California, Davis
Davis, USA
crubio@ucdavis.edu

Abstract

Porting scientific applications across different GPU architectures introduces floating-point arithmetic variations that can affect reproducibility, making efficient detection and mitigation of exceptions like NaNs and infinities crucial. While NVIDIA has dominated the GPU market, AMD GPUs are increasingly used in HPC systems as well, yet existing floating-point exception detection frameworks focus on NVIDIA, leaving a gap for AMD GPUs. We present FloatGuard, the first framework for efficiently detecting floating-point exceptions in HIP programs running on AMD GPUs. FloatGuard leverages AMD GPU hardware registers to detect floating-point exceptions, overcoming the limitations of AMD's built-in trapping mechanisms through a novel algorithm that combines assembly- and source-level instrumentation with debugger-guided execution. We evaluate FloatGuard on 565 HIP programs, detecting floating-point exceptions in 507 cases with a slowdown ratio that increases at most linearly with the number of exceptions discovered. Furthermore, we analyze the impact of compiler optimizations on exceptions trapped, and compare FloatGuard with the state-of-the-art tool for detecting floating-point exceptions in CUDA programs, which further reveals key differences between AMD and NVIDIA's floating-point exception behaviors.

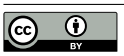
CCS Concepts

• **Software and its engineering → Software verification and validation**; **Software testing and debugging**; **Compilers**; **Software reliability**; • **Computing methodologies → Parallel programming languages**.

Keywords

floating-point exception detection, high-performance computing, heterogeneous computing, compiler optimization, numerical reproducibility, program portability

ACM Reference Format:
Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. 2025. FloatGuard: Efficient Whole-Program Detection of Floating-Point Exceptions in AMD GPUs. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3731545.3731586



This work is licensed under a Creative Commons Attribution 4.0 International License.
HPDC '25, Notre Dame, IN, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3731586

1 Introduction

Modern scientific computing requires applications to be ported to different classes of GPUs, which allows researchers to leverage higher computational throughput and energy efficiency. While IEEE 754 defines the Standard for Floating-Point Arithmetic, GPU architectures may implement floating-point arithmetic differently even when they adhere to most of the standard. As a result, variations in precision, compiler optimizations, or handling of special cases, such as floating-point exceptions (e.g., NaNs and infinities) can lead to discrepancies in the results. Scientific applications rely on reproducibility across all architectures (including GPUs). If exceptions, such as NaNs, are not detected and handled, they can lead to inconsistent results across different runs of the same computation, undermining the credibility of the results. Efficient tools to detect and mitigate floating-point exceptions are therefore crucial when porting code between GPU architectures.

While NVIDIA has long dominated the GPU market, AMD has made significant strides in recent years and has started to be used in many HPC clusters. For example, exascale computing platforms from the US Department of Energy (DOE), such as El Capitan and Frontier—the two fastest supercomputers[1]—use AMD GPUs (MI300 and MI250 GPUs, respectively). Previous works on detecting floating-point exceptions have focused on NVIDIA GPUs, using various methods such as compiler instrumentation [21, 24] or binary instrumentation [23, 25], however to the best of our knowledge no work has presented a framework to detect the occurrence of such exceptions in AMD GPUs.

Previous studies indicate that significant numerical differences may arise when code is compiled and executed on NVIDIA GPUs and later run on AMD GPUs, or the reverse [33]—these differences include the occurrence of exceptions in GPU *A* and normal results in GPU *B*, even when using the same program inputs. A study of floating-point exceptions detected in a comprehensive set of CUDA programs is presented in [25]; however, currently, no tools or frameworks are available to enable a comparable study for AMD (or HIP) programs. Developing such a tool is far non-trivial and requires addressing several challenges: (1) reverse engineering the low-level mechanisms that the AMD architecture provides to enable exception traps (if they exist), (2) enable such mechanisms efficiently in all GPU kernels, and (3) provide the ability to detect *all* exceptions while ensuring correct execution. Our work in this paper addresses all of these challenges.

Main Contributions. In this paper, we present FloatGuard, the first tool to detect floating-point exceptions in HIP programs

[1] https://top500.org/lists/top500/2024/11/

running in AMD GPUs. FloatGuard relies on the AMD GPUs hardware registers (e.g., trap status and mode registers) to enable floating-point exception detection in HIP and efficiently gather and analyze all the exceptions. While the mechanism provided by AMD to enable such traps is useful, it is not easy to use to detect exceptions in *all* kernels and to ensure correct execution when such modes are enabled. If a developer enables the provided trap manually (e.g., with the ROCgdb debugger), it must be done for *every* kernel thread, which is unfeasible, and it must be enabled at the beginning of the kernel, which would only trap *one* exception (the first one that occurs). We present an algorithm (implemented in FloatGuard) that addresses such challenges using code instrumentation at two levels of the program— assembly-level and source-level—and a debugger-guided execution.

FloatGuard consists of two modules implemented in Python: (1) a **compiler wrapper** which processes HIP program source files, compiles them into assembly, and injects code during link time that enables floating-point exception capture—it also accepts a list of already-captured floating-point exception code locations to prevent them from being triggered in subsequent iterations; and (2) a **testing framework**, which runs the instrumented HIP program with a custom ROCgdb debugger that detects and records floating-point exceptions as they occur. Both modules are run iteratively until *all* floating-point exceptions have been reported.

We evaluate FloatGuard on 565 HIP programs: 65 programs from various GPU benchmarks such as Rodinia [7], and 500 synthetic GPU programs generated by Varity [22]. FloatGuard detects floating-point exceptions in 89.7% of the programs (507 programs), ranging from 1 to 823 exceptions. We also show that FloatGuard's slowdown ratio increases at most linearly with the number of exceptions detected. We study the impact of certain compiler optimization flags on the number and types of floating-point exceptions triggered in the same programs, and find that using aggressive floating-point optimization options reduces the amount of exceptions reported in the aforementioned programs by 37.3%.

Lastly, we compare FloatGuard against the state-of-the-art tool for CUDA platforms, GPU-FPX [25], on the same set of programs, and propose possible causes for differences in floating-point exceptions reported between the two platforms.

In summary, the contributions of this paper are as follows:

- The first approach for detecting all floating-point exceptions in HIP programs running on AMD GPUs, which utilizes registers provided by the GPU hardware, assembly-level and source-level code instrumentation, and a debugger-guided execution (Sections 2 and 3).
- An implementation of our approach in the tool FloatGuard, which requires a minimal number of modifications to the program build systems (Section 3).
- An evaluation of FloatGuard on 565 HIP programs, including both benchmark programs and synthetic GPU programs, where FloatGuard detects floating-point exceptions in 507 out of 565 programs, ranging from 1 to 823 exceptions, with a slowdown ratio that increases at most linearly with the number of exceptions found in a program (Section 4.2).
- An empirical analysis of the impact of compiler optimizations on floating-point exceptions showing a decrease of exceptions when using aggressive compiler optimizations (Section 4.3).
- A comprehensive comparison of floating-point exceptions triggered in both AMD HIP and NVIDIA CUDA versions of the same program—using both FloatGuard and the state-of-the-art CUDA tool GPU-FPX—reveals the difference in numerical characteristics between the two platforms (Section 4.4).

2 Background

2.1 AMD GPU Toolchain

AMD provides a software stack called ROCm [5] to enable developers to create, compile and run programs on AMD GPUs. Apart from kernel drivers and low level libraries, ROCm provides a programming API and kernel language called HIP. A HIP program has a structure similar to that of CUDA programs on NVIDIA GPUs, where host code runs on CPUs and may call AMD GPU kernels to run on the GPU device. These kernels can create numerous GPU threads that run in parallel.

An AMD HIP program is compiled into host and device binaries using a compiler interface—HIPCC—which calls an underlying Clang compiler with an AMD GPU backend. In terms of debugging, ROCm offers ROCm Debugger [4] (ROCgdb), the AMD source-level debugger for Linux based on GDB [14], which provides support for debugging AMD GPU kernels. AMD ROCm supports a range of AMD GPUs with GCN, RDNA and CDNA architectures. FloatGuard is tested to work with both RDNA2/CDNA2 GPU architectures [1]. The evaluation of FloatGuard in Section 4 is performed on an RDNA2 GPU, but our approach is also applicable to other existing or future RDNA/CDNA architectures as long as they keep the existing mechanism to access the same set of registers related to floating-point exceptions.

2.2 Floating-Point Exceptions

IEEE floating-point types are represented by three fixed-point components: the sign bit, s, where 0 indicates a positive number and 1 indicates a negative number; the exponent e, which determines the exponential range of the floating-point value; and the mantissa m, which specifies the position of the value within the range defined by e. The interpretation of m differs depending on whether e is zero or non-zero; when e is zero, the floating-point number is within the *subnormal* range, which starts from zero, so that the floating-point type can represent numbers that are even closer to zero. Also, the IEEE 754 standard defines special values, such as infinity (Inf) and not-a-number (NaN), when e has all of its bits set to 1. These values correspond to situations where the limit of representing numbers is reached. The double precision, 64-bit, floating-point number representation, using s, m, e, can be written as follows:

$$f_{64}(s,m,e) = \begin{cases} \text{Inf} & e = 2047,\ m = 0 \\ \text{NaN} & e = 2047,\ m > 0 \\ (1-2s)(1+m)2^{e-1023} & 0 < e < 2047 \\ (1-2s)m2^{e-1023} & e = 0 \end{cases} \quad (1)$$

Given the limits of the IEEE floating-point types in representing the infinite number of real numbers, IEEE 754 defines certain situations, where the limits of these types are reached, as *floating-point*

Table 1: Exception types in AMD GPUs. An asterisk in the exception type denotes that it is an AMD-specific exception.

Exception Type	EXCP bit	Description
invalid operation	0	Result of an operation is not-a-number (NaN), such as $0/0$, $\infty/0$, or $\infty \mod 0$.
input denormal*	1	At least an operand is a subnormal number.
divide by zero	2	Division of a non-zero floating-point number by floating-point zero.
overflow	3	Absolute value of an operation result exceeds maximum value in floating-point type.
underflow	4	An operation result is a subnormal number.
inexact	5	Operation result cannot be precisely represented and rounding is involved.
integer divide by zero*	6	Division of a non-zero integer number by integer zero.

Table 2: Bit distribution of the 32-bit mode register and when its value is 0x5F2F0.

Others (19-31)	Exception bits (12-18)	11	10	IEEE	DX10_CLAMP	denormal mode (4-7)	round mode (0-3)
0000000000000	1011111	0	0	1	0	1111	0000

exceptions. Furthermore, hardware and software manufacturers may define additional types of floating-point exceptions as well.

The AMD RDNA2/CDNA2 GPU architecture defines seven types of floating-point exceptions, listed in Table 1. In addition to the 5 types of floating-point exceptions defined by IEEE 754, AMD GPUs also define two additional exception types: input denormal, and integer divide by zero, which are also useful in analyzing abnormal numerical program behavior. FloatGuard focuses on detecting all floating-point exceptions listed in Table 1 except for the inexact exception, which is a rounding error that occurs in virtually all numerical programs.

The RDNA2/CDNA2 GPU architecture provides a 32-bit mode register to check and control whether floating-point exceptions are trapped or not in a GPU kernel program. The layout of the mode register is presented in Table 2, adapted from [1]. Bits 12-18 in the mode register sets which exception types are enabled. Additionally, there is also a trap status register, with its `EXCP` bits 0-6 (corresponding to bits 12-18 in the mode register) set when a floating-point exception of a respective type is triggered during kernel execution. If the GPU kernel is set to trap these exceptions during runtime, program execution stops when an exception occurs, and a floating-point exception signal (SIGFPE) is sent; if the GPU kernel is set to not trap these exceptions, execution continues when an exception occurs, and the corresponding `EXCP` bit in the trap status register is set. By default, bits 12-18 in the mode register are all set to 0 (as in disabled) when each kernel starts execution; to enable them so that floating-point exceptions can be trapped, these mode register bits must be set to 1. In addition, to achieve IEEE 754 compliance, bits 8 (`DX10_CLAMP` bit: DX10-style treatment of NaNs) and 9 (`IEEE` bit) should be set to disabled and enabled, respectively.

To observe the runtime floating-point exception characteristics of AMD GPUs, a preliminary test is conducted using ROCgdb on a sample program listed in Figure 1. The program triggers four floating-point exceptions: a divide by zero floating-point exception in function `k_fp` in line 4, two divide by zero exceptions in function `k_mixed` in lines 9 and 10, and an invalid exception from the addition in line 10. Breakpoints are set at the beginning and end of each kernel. At the start of each kernel, the values of the mode and trap status registers are recorded. If floating-point exceptions are enabled for the kernel, the mode register is updated accordingly.

```
1  #include "hip/hip_runtime.h"
2
3  __global__ void k_fp(int *mem, float a, float b) {
4    int fret = a / b;
5    *mem = fret;
6  }
7
8  __global__ void k_mixed (int *mem, int a, int b) {
9    int fret = a / b;
10   fret = fret + (float)a / (float)b;
11   *mem = fret;
12 }
13
14 int main (int argc, char* argv[]) {
15   int *mem;
16   hipMalloc(&mem,4);
17   k_fp<<<1,1>>>(mem,1,0);
18   hipDeviceSynchronize();
19   k_mixed<<<1,1>>>(mem,1,0);
20   hipDeviceSynchronize();
21   return 0;
22 }
```

Figure 1: A sample HIP program for testing AMD GPU kernel floating-point exception behavior, where the lines of code triggering exceptions are highlighted.

The value of the trap status register is then observed either at the end of the kernel or when an exception occurs.

Three preliminary observations can be drawn from the above example. (1) By default, exception trapping is disabled in AMD GPUs; theoretically, developers can check the trap status register at the end of each kernel's execution to know which exception occurred, however, it is not possible for developers to know which line of code caused the exception, and it is impractical for developers to check the trap status of each GPU thread. (2) The memory location where the program counter stops after a floating-point exception is trapped does not always correspond to the actual instruction that causes the exception, possibly due to delayed exception reporting on GPUs. (3) Once a floating-point exception is triggered in a kernel, the state of the program is not recoverable. This means that even if the SIGFPE signal is cleared and continue program execution, the

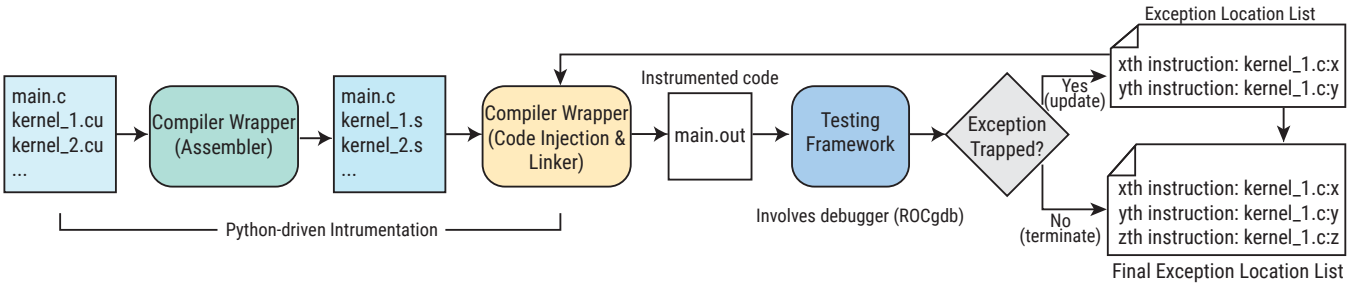


Figure 2: The workflow of FloatGuard.

program state after this point does not exactly correspond to when floating-point exceptions are disabled. These observations about AMD HIP program execution on AMD GPUs influence the design and implementation of FloatGuard.

2.3 Overview of FloatGuard

The workflow of FloatGuard is illustrated in Figure 2. FloatGuard consists of two main components: a compiler wrapper and a testing framework.

The **compiler wrapper** acts as an in-place replacement of the HIPCC compiler interface. When an HIP source file is compiled using the compiler wrapper, it replaces output object files with text-based assembly files instead; then, during link time, the compiler wrapper injects input assembly files with instructions for enabling or disabling floating-point exceptions at injection points in a list maintained by both modules in FloatGuard, then call the actual link command to generate the instrumented binary. The instrumented program is then sent to the **testing framework**, which runs the instrumented program, gathers information on exceptions encountered, then updates the list of injection points as mentioned above, to ensure that FloatGuard can report all floating-point exceptions observed in an AMD HIP program at runtime.

A typical usage scenario of FloatGuard is as follows: a software developer would like to know whether their HIP program that runs on an AMD GPU suffers from any floating-point exceptions. The developer then needs to modify their build script by simply replacing the original compiler with our compiler wrapper, and enabling debug information in the build by using the `-g` flag in the compiler command line. The developer can optionally select the types of exceptions to be captured, as some types of exceptions, such as input denormal, may be of less interest because of their lower impact towards numerical correctness of the program. Finally, the developer can invoke FloatGuard, which will test the program to output the source code locations that trigger floating-point exceptions during runtime. The developer can then inspect the source code and devise solutions to mitigate the exceptions, such as reordering operations or increasing floating-point precision.

The next section presents the design and implementation of these modules and how they interact to detect floating-point exceptions.

3 Design of FloatGuard

This section describes the design and implementation of FloatGuard for detecting floating-point exceptions in AMD GPU kernels.

3.1 Floating-Point Exception Detection Control

As mentioned in Section 2.2, setting the individual bits in the mode register in each kernel thread allows a floating-point exception to be trapped or not during program execution. A method to write to the mode register is via debugging commands using the ROCgdb debugger. However, this method requires using breakpoints to pause the program at the beginning of every GPU kernel thread execution. For most AMD GPU kernels, thousands of GPU threads run in parallel. To set the mode register for each GPU threads, the ROCgdb debugger must stop the program thousands of times just for one kernel, which introduces significant overhead (according to our testing), and makes the tool impractical to use.

Assembly Injection. An improved solution is to inject assembly instructions that write to the mode register during program compilation. Below are two sets of instructions that set the mode register. They are interchangeable, but FloatGuard uses them both for the two kinds of code injections described in Section 3.2. The first set of instructions moves an immediate value, first into a scalar general purpose register `s31`, which has no specific reservations in kernel code, then into the mode register.

```
s_mov_b32 s31, 0x5F2F0
s_setreg_b32 hwreg(HW_REG_MODE), s31
```

On the other hand, the second set of instructions directly assigns values to the mode register. Since only 16-bit values are allowed, FloatGuard uses two instructions for the value assignment.

```
s_setreg_imm32_b32 hwreg(HW_REG_MODE, 0, 16), 0xF2F0
s_setreg_imm32_b32 hwreg(HW_REG_MODE, 16, 16), 0x5
```

FloatGuard also uses a similar instruction to set the EXCP bits (0-6) in the trap status register to all zero. This instruction is injected whenever floating-point exceptions are enabled during program execution, so that only the floating-point exception type that is trapped is recorded.

```
s_setreg_imm32_b32 hwreg(HW_REG_TRAPSTS, 0, 7), 0
```

A naive method for detecting floating-point exceptions with these instructions is to simply enable floating-point exceptions at the *beginning* of all AMD GPU kernels. However, as mentioned in Section 2.2, it is not possible to recover an AMD HIP program from a trapped exception. Thus, if FloatGuard simply enables exceptions at the beginning of each kernel call, the program would stop at the point where the first floating-point exception is triggered, irrevocably, and no further exception would be triggered. To work

around this issue, FloatGuard injects code in multiple locations of the program. This is further explained in the next section.

3.2 Compiler Wrapper and Code Injection

To ensure that floating-point exceptions are enabled or disabled in the original program, injecting exception control instructions (discussed in Section 3.1) is necessary. The compiler wrapper module performs such injection alongside with program compilation. It injects code in the two kinds of locations listed below:

- **Entry points:** at the beginning of each AMD GPU kernel, instructions to enable floating-point exceptions are injected. This ensures that exception capture is always enabled when an AMD GPU kernel starts execution.
- **Previously-triggered exception locations:** the compiler wrapper may also accept a list of assembly code locations at which previously triggered exceptions have been observed. The compiler wrapper injects exception disabling instructions before, and exception enabling instructions after these locations, so that no exceptions are triggered when the instructions within these locations are executed. This is to prevent previously-triggered exceptions from being triggered again. Additionally, the trap status register is cleared when floating-point exceptions are re-enabled, because the trap status register is still set when an instruction that causes a floating-point exception at the location is executed. Clearing the trap status register ensures that the next floating-point exception trapped will return the correct exception type.

Two widely-used methods for code injection in floating-point research are Clang plugins and LLVM passes. Calls to a function containing the inline instructions from Section 3.1 can be injected either into the source code, or into its LLVM intermediate representation (IR). There are advantages and disadvantages of these two methods. For example, Clang plugins apply source-to-source transformations and thus are compiler agnostic; meanwhile LLVM passes can be adapted to other language front-ends that HIP supports, such as HIP FORTRAN. However, during the implementation of FloatGuard, we found that neither Clang plugins nor LLVM passes can solve the problem of instruction reordering: optimization passes in the AMD GPU back-end of the default HIPCC compiler reorders assembly code *after* code injection is performed. In particular, compiler optimizations may alter code regions bracketed by function calls enabling and disabling floating-point exceptions, and undermine their effectiveness. Unfortunately, using software barriers does not appear to work as intended either, as their presence may also interfere with optimization passes and lead to numerical behaviors different from those observed without them.

Assembly Files Injections. To solve the above issue, FloatGuard instead implements code injection by directly inserting assembly instructions into assembly files, after all compilation and optimization steps are performed, but before the linker combines all assembly files into a single binary executable. This method combines the advantages of both previously mentioned methods: it does not require source code to inject code into the program, and thus it works regardless of what kind of high-level language features exist in the program source code; it also preserves source location information when exceptions occur, as long as such debug information is built into the assembly files from compiler options `-g` in HIPCC; it is compiler and language agnostic, as it uses Python scripts to read, process, and modify AMD kernel assembly code, without any other software dependency; it is transparent to the end user, as it does not require them to drastically modify the build process of their program, and all the resulting instrumented code exists in a human-readable assembly form; and lastly it can be easily adapted to a different AMD GPU architecture as long as the equivalent of mode register control instructions are still available.

Algorithm 1: The testing algorithm in FloatGuard.

```
1  Function TestAlgo(Program):
2      ExceptionLocations = ∅;
3      while ProgramFinished == false do
4          InstrumentedProgram = Program;
5          foreach Location ∈ ExceptionLocations do
6              InstrumentedProgram =InjectException
                 (InstrumentedProgram, Location);
7          RunProgramUntilStop (InstrumentedProgram);
8          if InstrumentedProgram has finished running then
9              ProgramFinished = true;
10         else
               // signalled exception
11             ExceptionLocations =
                 ExceptionLocations∪GetExceptionInfo ();
12             OutputExceptionInfo ();
```

Instructions controlling the mode register are injected as follows. First, an assembly file is read, and the compiler wrapper scans for the listing of AMD GPU kernel functions, which is marked by `__CLANG_OFFLOAD_BUNDLE` markers; then the compiler wrapper scans for the beginning of a kernel function, which is marked by a mangled function name beginning with `_Z`. The instructions are then injected into the body of these kernel functions.

An additional issue arises from the delayed exception reporting problem described in Section 2.2. To address this, FloatGuard associates each trapped exception with a *set* of instructions rather than one instruction. We have empirically found that delayed exception reporting often is off by at most 5 instructions. Thus, FloatGuard associates a reported exception with the instruction at which execution halted and its four preceding instructions. These are treated as a set of instructions for which floating-point exception should be disabled in subsequent testing. Consequently, the compiler wrapper injects instructions to disable exceptions before, and to re-enable them after this set of instructions.

3.3 Testing Framework

FloatGuard runs the instrumented program and aims to capture all floating-point exceptions in AMD GPU kernels. This testing component is built with Python spawning a ROCgdb instance with the `pexpect` library, which attaches to the instrumented program and controls whether floating-point exceptions are enabled/disabled before every AMD GPU kernel's execution. When a floating-point exception is trapped, the testing framework outputs the crash site information such as source file and lines, function name, adjacent assembly instructions, and exception types.

The testing algorithm of FloatGuard is illustrated in Algorithm 1. It is designed and fine-tuned to adapt to the characteristics

of AMD GPU kernel behavior with floating-point exceptions described in Section 2.2. The detailed steps are as follows:

▷Step 1: Initially, FloatGuard compiles the original program using the compiler wrapper, inserting instructions at the beginning of each AMD GPU kernel to enable exceptions for all code inside all HIP kernels, then attaches ROCgdb to the instrumented program. Also, the list of injection sites that trigger floating-point exceptions are set to empty. Furthermore, the linker command used during the build process is saved separately so that future injection of assembly instructions can be performed without running the whole compilation process again, improving the efficiency of the tool.

▷Step 2: If no exception is found and the program exits normally, FloatGuard finishes running and reports that no floating-point exception is found; otherwise, when a floating-point exception occurs and ROCgdb stops, it parses and outputs exception information from ROCgdb output. Information that is saved includes source code location, the number of instructions from the beginning of the function to the instruction that the program stops at, and the exception type. If the source code location is from an AMD HIP library, FloatGuard goes up the call stack until it finds a source code location that is from the program source code rather than from an existing library, and saves this location as well.

▷Step 3: FloatGuard also records the location of the exception-occurring kernel in the execution trace (kernel name and the number of instructions from the beginning of the function to the exception site) to a list of injection sites.

▷Step 4: FloatGuard then runs the linker command saved in step 1; this time the compiler wrapper reads the list of injection sites saved in Step 3, and injects code around these sites, so that floating-point exceptions from instructions around the injection sites are not trapped the next time.

▷Step 5: The program is then restarted with ROCgdb, and return to Step 2. As more sites that trigger floating-point exceptions are found and recorded, all code that triggers such exceptions will eventually be included in the list of injection sites; at this point, the program finishes without triggering any other exceptions.

Consider again the sample program from Section 2.2. Exceptions are enabled at the beginning of both kernels in the program. When the program is run with ROCgdb attached, an exception occurs in `k_fp` line 4, whose function location is recorded in the list of injection sites. The linker command is executed again, and the compiler wrapper injects code around the location previously recorded so that floating-point exceptions are disabled before reaching the location, and enabled after it. The program is re-run with ROCgdb attached, rinse and repeat, until the program no longer triggers any exception before exiting normally. The list of injection sites then corresponds to the list of floating-point exceptions detected.

Slowdown. FloatGuard undoubtedly introduces slowdown, given the overhead introduced by the various components, and the need to run the program multiple times to detect all floating-point exceptions. The slowdown introduced by FloatGuard consists of the following parts: spinning up ROCgdb and running the program every time it needs to recover from a previous floating-point exception; collecting exception information when it is trapped; and running the linker every time a new exception location is added. The overhead introduced by the instructions that set the mode and trap status registers is negligible. The slowdown ratio can be calculated using the following equation:

$$\text{Slowdown} = \frac{\sum_{i=1}^{N} \mathrm{P_i} + \mathrm{OH} \cdot (\mathrm{N}+1) + \mathrm{P}}{\mathrm{P}} \tag{2}$$

where P is the program running time; P_i is a strictly increasing sequence of the time of the ith run of the program, from beginning of program execution to when the exception occurs; OH is a relative constant cost of the overhead for each re-run of the program, including time running the linker, spinning up ROCgdb, and collecting exception information after it occurs, all of which are constants depending on the program; and N is the number of floating-point exceptions triggered. When P is small (less than 1s in programs tested in Section 4.2), OH is relatively large compared to P, and the slowdown ratio increases in an approximately linear manner with the number of floating-point exceptions. Meanwhile, when P is large, OH is relatively small, and the relationship between slowdown ratio and number of exception detected depends on the ratio between P_i and P, which is discussed in Section 4.2.

Table 3: Benchmark programs tested with FloatGuard to detect floating-point exceptions.

Benchmark Set	Benchmarks
Rodinia	backprop, cfd, gaussian, heartwall, hotspot, hotspot3D, lavaMD, lud, myocyte, nn, nw, particlefilter, streamcluster
PolyBench-ACC	correlation, covariance, 2mm, 3mm, atax, bicg, doitgen, gemm, gemver, gesummv, mvt, syr2k, syrk, gramschmidt, lu, adi, convolution-2d, convolution-3d, fdtd-2d, jacobi-1d-imper, jacobi-2d-imper
Parboil	bfs, cutcp, histo, lbm, mri-gridding, mri-q, sad, sgemm, spmv, stencil, tpacf
SHOC	bfs, fft, gemm, md, md5hash, reduction, scan, sort, spmv, stencil2d, triad, qtclustering, s3d
GPGPU-Sim	cp, libor, lps, mum, rayTracing, wp
HPCG	HPCG
Varity	500 randomly-generated programs containing numerical inconsistencies

4 Experimental Evaluation

This evaluation answers the following research questions:

- **RQ1** How many floating-point exceptions are detected by FloatGuard in AMD HIP programs, and how much slowdown does FloatGuard introduce?
- **RQ2** How do compiler optimization options affect the number of floating-point exceptions detected by FloatGuard?
- **RQ3** How do floating-point exceptions differ when the same GPU programs are executed on AMD HIP and NVIDIA CUDA?

4.1 Experimental Setup

Benchmark Selection. FloatGuard is evaluated on 565 programs, which are categorized in Table 3. The first 65 programs are from five heterogeneous benchmark suites: 13 programs are from Rodinia [7], where only programs with floating-point operations are selected; 21 programs are from PolyBench-ACC [16]; 6 programs are from benchmarks distributed with the GPGPU-Sim simulator [20]; 11 programs are from the Parboil benchmark suite [30]; 13 programs are from the Scalable HeterOgeneous Computing (SHOC) benchmark suite [11]; And lastly, we also include the HIP version of the HPCG benchmark [2] based on the original HPCG implementation [13]. These benchmarks cover a wide range of uses for heterogeneous computing, such as linear algebra for machine learning, data mining, medical imaging, and fluid dynamics.

We also include 500 programs generated using Varity [22], a floating-point program random generator. The programs are publicly available in Varity's GitHub repository[2] and have been previously identified as exhibiting numerical inconsistencies when compiled with different optimization settings—specifically, comparing `-O3 -use_fast_math` against `-O0` using the CUDA NVCC compiler. These inconsistencies often arise from compiler optimizations that affect the precision, associativity, or evaluation order of floating-point operations. Such optimizations can also influence whether a floating-point exception is triggered. By including these synthetic programs, we aim to capture edge cases where aggressive optimization interacts with exception conditions in a nontrivial fashion, and may not be well represented in real-world HPC workloads but are critical for evaluating the robustness and coverage of floating-point exception detection tools.

All programs in this evaluation, with the exception of ROC HPCG, are only available as CUDA programs; thus we convert them for AMD HIP using the HIPIFY tool [3] and then fix minor issues manually, so that the end results of the program are either comparable with the original CUDA versions, or they pass in-built verification routines, when available.

Benchmark Parameters. Some benchmarks, such as the Varity synthetic programs, and the programs from GPGPU-Sim and SHOC, have no selection of input parameters and datasets; these benchmarks are run once with the default parameters. While others, such as the `cfd` program in Rodinia, have multiple input datasets; some other benchmarks, such as all PolyBench-ACC programs, can be run with various input data sizes. In our evaluation we do the following: for Rodinia, all inputs listed in the `run` script in each program are included; for PolyBench-ACC, all problem sizes from mini, small, default, large to extra large sizes for each benchmark are included; for Parboil, all input datasets are included; and for HPCG, we use the parameter `128 128 64`.

Baselines. As of the writing of this paper, there is no other work that detects floating-point exceptions in AMD HIP programs. There are works detecting floating-point exceptions in other architectures such as x64 and CUDA, which are described in Section 5, and we include a cross-platform comparison for the same programs between AMD and NVIDIA GPUs in Section 4.4.

Runtime environment. We run FloatGuard on a workstation PC, running with a 6-core AMD(R) Ryzen 5 7500F processor, 32 GiB RAM, and an AMD RX 6650 XT GPU with 8GiB of VRAM. The workstation PC runs Ubuntu 22.04 OS, with AMD ROCm 6.1.0 installed. FloatGuard is run on Python 3.10.12 with pexpect 4.9.0 for automatically controlling ROCgdb process input and output, with no other external dependencies.

[2]https://github.com/LLNL/Varity/tree/master/cuda_examples

4.2 RQ1: Exception Detection Effectiveness

In the 65 benchmark programs evaluated, FloatGuard detects floating-point exceptions in 9 of them, spanning across 5 benchmark suites. The number of each type of floating-point exceptions detected by FloatGuard in each of these 9 programs are listed in Table 4. Note that the number of total exceptions may be smaller than the sum of all types of exceptions; this phenomenon occurs in `myocyte`, `s3d` and `wp`, in which input denormal exceptions occur. This is due to the fact that one floating-point operation may trigger both input denormal and other exceptions, for example, dividing a subnormal number with zero.

Also, amongst these programs, there is only one—`stencil` in Parboil—which FloatGuard only detects floating-point exceptions related to subnormal numbers. Meanwhile, all other 8 benchmarks have exceptions related to special values—Inf and NaN—which are considerably more serious floating-point exceptions and have a higher impact on the precision error of these programs.

In the 500 synthetic GPU programs, FloatGuard detects floating-point exceptions in 498. If only exceptions related to special values are counted, 291 programs trigger them. On average, 6.056 floating-point exceptions are detected per program, or 2.088 when considering only those related to special values.

As for the remaining 2 programs which do not have floating-point exceptions detected: the first one, `case_152`, has its set of input parameters all in normal range (which means none of them is subnormal), and all subnormal and division by zero operations are constant, which are likely pre-calculated during compile time. Meanwhile, the other program, `case_473`, has a `compute` function that looks like the following excerpt:

```
1  for(int i=0;i<var_1;++i){
2    comp=var_2*1.5271E-42f*(var_3/floorf(-1.5532E35f
         -1.8528E35f-var_4*-1.4807E36f));
3    comp=var_5+1.1129E35f+var_6/powf(expf(1.4350E-36f),
         var_7/-1.7503E11f*var_8/1.5272E-19f);
4  }
```

The expression in line 2, from its variable to constant operands, all contain floating-point exceptions; but the whole expression is removed from the program binary due to dead code, and only the expression in line 3 is kept. The operands in this expression, from `var_5` to `var_8`, are all normal numbers, and the whole expression is indeed not expected to produce any floating-point exceptions. We confirm our hypothesis for the two programs by running FloatGuard again but with the programs compiled with `-O0` instead, which does not perform either compile-time constant calculation or dead code removal. FloatGuard successfully detects floating-point exceptions in this case.

In terms of slowdown ratio, first for the 65 benchmark programs: the original runtime for these 65 programs ranges from 0.17s which includes the spin-up time for AMD HIP libraries, to 196s in the case of the `correlation` program in PolyBench-ACC. The slowdown

Table 4: A list of benchmarked programs that have floating-point exceptions detected by FloatGuard. The exception type abbreviations NAN, IN_SUB, DIV0, INF, OUT_SUB, and INT_DIV0 correspond to exception type invalid operation, input denormal, divide by zero, overflow, underflow, and integer divided by zero, respectively.

Benchmark Set	Benchmark	Total Exceptions	NAN	IN_SUB	DIV0	INF	OUT_SUB	INT_DIV0
Rodinia	cfd	12	6	0	6	0	0	0
Rodinia	myocyte	50	26	3	0	15	9	0
PolyBench-ACC	correlation	1	0	0	1	0	0	0
PolyBench-ACC	gramschmidt	2	1	0	1	0	0	0
PolyBench-ACC	lu	1	1	0	0	0	0	0
PolyBench-ACC	adi	4	4	0	0	0	0	0
SHOC	s3d	823	6	681	0	7	264	0
Parboil	stencil	2	0	1	0	0	1	0
GPGPU-Sim	wp	68	2	50	0	5	18	0

ratio introduced by FloatGuard ranges from 1.0× for some benchmarks where no floating-point exceptions are detected, to 6888× in the case of `s3d` from the SHOC benchmark, in which the original program runtime is 0.2s, and 823 floating-point exceptions are detected. As for the 500 synthetic GPU programs, the runtimes for these programs are all between 0.17s and 0.18s, and the slowdown ratio introduced by FloatGuard ranges from 2.53× for `case_152` where no floating-point exception is detected, to 118× for `case_365` where 30 floating-point exceptions are detected.

The slowdown ratios for 564 out of 565 programs[3] are shown in Figure 3. We use linear regression to find the line of best fit, and find that for most programs that run for a short time (less than 1s), their slowdown ratios are approximately linearly related to the number of exceptions detected, which corroborates the assertions made in Section 3.3. On the other hand, for programs running for a longer period of time, the slowdown ratio is sublinearly related to the number of exceptions, since the exceptions occur early during program execution, therefore $\sum_N P_i$ in Equation (2) is negligible compared to P. The sole outlier is `s3d` which has a higher slowdown ratio due to its longer linker time (1s) compared to shorter running time (0.2s).

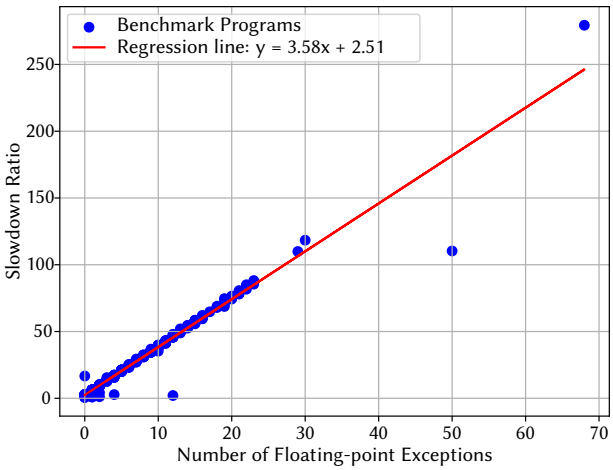


Figure 3: Slowdown ratio compared to the number of floating-point exceptions in the programs tested.

4.3 RQ2: Effect of Aggressive Floating-Point Optimization Flags on Exceptions

The impact of compilers and their optimization flags on the correctness of heterogeneous numerical programs has been well documented [15], and the underlying causes of these effects have been studied in [17, 26, 29]. Optimization flags related to floating-point operations can greatly affect the occurrence of floating-point exceptions in a numerical program. One of the main optimization flags related to floating-point operations in the HIPCC compiler is `-ffast-math`, which does the following:

(1) Floating-point operations are treated as infinite-precision real numbers in terms of algebraic rules such as associative and distributive properties. This may negatively impact floating-point round-off errors, which could cause catastrophic cancellation or loss of significance for some floating-point operation reassociations.
(2) No NaN or infinite values will be operands or results of floating-point operations. However, this does not mean that floating-point hardware does not generate floating-point exceptions about NaN or infinite values internally. These exceptions are still recorded on AMD GPUs.
(3) Fast approximation math functions, which are less accurate, are applied instead of the IEEE 754 standard versions. Fast constant reciprocal instructions are also applied.

If developers use optimization flags such as `-ffast-math`, not only may their program have higher precision errors, they may also encounter different floating-point exceptions to be triggered, due to changed results in operations and these results possibly affecting program execution paths. Thus, studying how these optimization flags affect floating-point exceptions is important. To do so, we run the same programs with the optimization flags: `-ffast-math` and `-fdenormal-fp-math=preserve-sign`. The inclusion of the second flag is to investigate how exception behaviors related to subnormal numbers change; simply enabling `-ffast-math` does not enable this flag.

[3]Except `s3d` which has over 800 exceptions detected and is not shown here to preserve the clarity of the chart.

Table 5: How the number of floating-point exceptions detected changes when `-ffast-math -fdenormal-fp-math=preserve-sign` is enabled in the compiler. The exception type abbreviations are the same as in Table 4.

Benchmark	Total Exceptions	Total Exceptions (excl. SUB)	NAN	IN_SUB	DIV0	INF	OUT_SUB	INT_DIV0
cfd	12 → 14	12 → 14	6 → 8	0	6	0	0	0
myocyte	50 → 28	41 → 23	26 → 24	3 → 0	0	15 → 2	9 → 2	0
correlation	1	1	0	0	1	0	0	0
gramschmidt	2	2	1 → 0	0	1 → 2	0	0	0
lu	1	1	1	0	0	0	0	0
adi	4 → 6	4 → 6	4 → 2	0	0 → 4	0	0	0
s3d	823 → 739	13 → 5	6 → 0	681 → 608	0	7 → 5	264 → 309	0
stencil	2 → 3	0	0	1	0	0	1 → 2	0
wp	68 → 87	4	2 → 1	50 → 78	0	5 → 3	18 → 16	0

In total, when aggressive floating-point optimization flags are enabled, the average number of floating-point exceptions for all tested programs drops by 37.3%. First we look at the 9 benchmark programs in which FloatGuard detects floating-point exceptions, when aggressive floating-point optimization flags are enabled, the number of exceptions detected changes, as shown in Table 5. The results are mixed: 2 programs have the number of exceptions decreased, 4 programs increased, and the remaining 3 have their numbers unchanged. If we only count the number of exceptions related to special values, then 2 programs have the number of exceptions decreased, 1 program increased, and the remaining 6 have their numbers unchanged. On one hand, there are programs like `myocyte` where all types of floating-point exceptions have their numbers decreased. On the other hand, programs like `wp` have their floating-point exceptions numbers change up or down, indicating that aggressive optimization flags in the HIPCC compiler change rather than reduce the characteristics of exceptions in a program.

Next we look at the 500 synthetic GPU programs. We observe that for the 472 programs[4] that successfully execute when optimized, the average number of floating-point exceptions fall from 6.023 to 3.19 when aggressive floating-point optimization flags are enabled. FloatGuard reports 297 (62.9%) programs with fewer floating-point exceptions, 45 (9.5%) programs with more exceptions, and 130 (27.5%) with an unchanged number of exceptions. An example of fewer reported exceptions when aggressive floating-point optimization flags are enabled is `case_68`, where the single floating-point exception detected is an input denormal exception, which is not triggered when aggressive floating-point optimization flags are enabled. On the contrary, an example of more floating-point exceptions detected when aggressive floating-point optimization flags are enabled is `case_387`. When aggressive floating-point optimization flags are disabled, the first floating-point exception likely comes from the first line of code in `compute`, where we verify that this `if` statement always returns `false` regardless of compiler options:

```
if(comp==cosf(var_1-1.8906E35f-(1.1216E14f/var_2))) {
```

Variable `var_2` is assigned a subnormal number from program input, triggering an overflow exception with the result of the division. When the aggressive optimization flags are enabled, `var_2` is flushed to zero, therefore triggering a divide by zero exception instead. There is also a new exception detected when aggressive floating-point optimization flags are enabled; it is of the input denormal type, and it is likely that different ways to compile the equal operator in this line of code is the cause of this new exception.

4.4 RQ3: Comparing AMD HIP and NVIDIA CUDA Exception Behavior

Floating-point operations are inherently sensitive to variations in how they are compiled and executed. This has been shown in previous studies described in Section 5. Many heterogeneous GPU programs, including most programs used in our evaluation, are written for specific hardware and software environments, most commonly to run on NVIDIA GPUs and CUDA, and then ported to the AMD HIP platform. Studying the differences in floating-point exceptions between NVIDIA CUDA and AMD HIP versions of the same program can reveal how each platform handles IEEE 754 compliance, compiler optimizations, and exception propagation. It helps identify numerical divergences that are less obvious, guides platform-specific debugging, and informs trade-offs between program performance and correctness. Such understanding subsequently ensures portability and reliability of scientific applications across GPU vendors. Thus, in this section, we investigate differences with respect to floating-point exceptions when running a same program on both AMD and NVIDIA platforms.

To study the differences of program behavior between AMD and NVIDIA GPUs, we use the same set of 565 programs described in Section 4.1, running on both HIP and CUDA platforms. We use the CUDA version of HPCG [27], while using the same codebase for other programs, and only differing in HIP/CUDA function calls. To detect floating-point exceptions in CUDA programs, we use the the state-of-the-art tool GPU-FPX [25]. We use the latest GitHub revision as of writing[5] and build it using CUDA 11.8 and NVBit 1.7.2. We run the detector module of GPU-FPX to detect floating-point exceptions on CUDA. Our experiments are run on an 18-core Intel(R) i9-10980XE processor workstation with 256 GiB RAM, an NVIDIA RTX 4090 GPU, and with Ubuntu 22.04 OS installed.

Since both tools use different hardware, software, algorithms, and possibly different metrics to track floating-point exceptions, a direct comparison of slowdown ratio or the amount of exceptions

[4]The remaining 28 programs cause runtime exceptions unrelated to floating point when compiled and run with aggressive floating-point optimization flags, and are thus excluded from discussion in this subsection.

[5]https://github.com/LLNL/GPU-FPX/tree/ab39d7f286fed890c394af014175721b619b8d28

triggered is not practical. Instead, we simply run both tools separately to detect floating-point exceptions on both environments, on the same set of programs, with the same optimization level, no aggressive floating-point optimization flags like `use_fast_math` or `-ffast-math`. We compare them for any differences in both detected exceptions and the end results, to gain an insight on diverging behavior for the same programs across platforms.

First we run both tools with the same 500 synthetic GPU programs. These programs are compiled with `-O3` on both platforms. We find that GPU-FPX only finds 404 out of 500 in the CUDA versions, compared to AMD HIP versions where 498 programs out of 500 are detected with floating-point exceptions. We look into the GPU-FPX source code and find that GPU-FPX does not detect input denormal exceptions. After removing input denormal exceptions from the FloatGuard results, the number of HIP programs with detected floating-point exceptions drops from 498 to 365. We look into a few example programs where only the CUDA or HIP version triggers floating-point exceptions. We analyze their source code, compiled assembly, program control flow during execution, and try out different compiler optimization flags to determine possible cause for these differences.

The first program we analyze is `case_450`. GPU-FPX does not detect any floating-point exceptions in this program, while FloatGuard detects an underflow exception from the first line of the `compute` function:

```
if (comp < (-1.2964E-35f / var_2)) {
```

where `var_2` is assigned a value of -1.9453E27f. The result of the division operation is too small to be represented by a 32-bit floating-point type, regardless of whether subnormal numbers are allowed as operation results, and is rounded to zero. Since GPU-FPX simply checks the results of operations, which is zero in this case, it does not treat this as an exception. Still, reporting such underflow exceptions is helpful to developers, since with a slight change to `var_2`, this division operation can still generate subnormal results.

The second program we analyze is `case_350`. GPU-FPX detects an underflow exception while FloatGuard does not detect any exception. The body of the `compute` function is as follows:

```
if (comp<=(-0.0f-var_1-1.9945E-44f/-1.8945E36f)){
  float tmp_1=(1.6933E34f-(-1.3140E20f/+1.2012E-26f));
  float tmp_2=-1.5160E-43f;
  comp=tmp_2/tmp_1*var_2*var_3;
}
```

We investigate the PTX code generated by NVCC and find that under `-O3` optimization, all constant expressions are calculated at compile time, and the `if` statement is compiled as a selection instruction. Since the `comp` variable has a subnormal value before this code snippet, and the `if` condition is false, the subnormal value in turn becomes the result of the selection instruction thus triggering GPU-FPX to report an underflow exception. Meanwhile, in the AMD HIP version, even though it is still compiled as a selection instruction, the hardware registers do not trigger exceptions here. It only triggers an input denormal exception when input denormal exceptions are enabled. Our conclusion is that FloatGuard, utilizing hardware registers, can more correctly identify the types of exceptions than GPU-FPX.

As for the 65 benchmark programs, GPU-FPX detects floating-point exceptions in 11 of them: `rayTracing` and `wp` in GPGPU-Sim benchmarks, `s3d` in SHOC, `stencil` in Parboil, `correlation, gesummv, gramschmidt, lu` in PolyBench-ACC, `cfd` and `myocyte` in Rodinia, and the HPCG benchmark. As comparison, FloatGuard detects floating-point exceptions in 9 programs. Next we focus on the following programs where only one tool detects exceptions in its respective platform.

rayTracing in GPGPU-Sim: FloatGuard does not report any floating-point exceptions; meanwhile GPU-FPX reports 10 floating-point exceptions. After investigation, we find that all of them result from CUDA implementations of `__saturatef` and `pow` functions, frequently used in color manipulations to avoid using branching statements in GPUs. It is likely that the AMD HIP version of these function implementations avoids triggering floating-point exceptions. We also compare the rendered images for both versions of this program and find that they are indeed slightly different.

gesummv in PolyBench-ACC: when the dataset is extra large, there are two floating-point exceptions captured with GPU-FPX in the `gesummv_kernel` function. One is an underflow exception from line 115, which is a multiply-add operation; the other is an overflow exception from line 118, which are two multiply-add operations. We manually add debug code after the two expressions, to check if the results are subnormal or infinite, and we find that the CUDA version shows subnormal or infinite results while the AMD HIP version does not. This is possibly due to aggressive optimizations from CUDA.

adi in PolyBench-ACC: FloatGuard reports 4 floating-point exceptions in 4 different kernels (corresponding to expressions in lines 164, 177, 201, 212 of the original CUDA version). All four kernels involve dividing by an element in matrix B. If we put `isnan` calls for the results for these 4 expressions we can find that NaNs do occur in these kernels when running on AMD GPUs. However, they do not occur in the CUDA version.

HPCG: FloatGuard reports no floating-point exceptions in the HIP version; meanwhile, GPU-FPX reports two exceptions, both in the `reduce_1Block_kernel` which is a closed-source kernel in the cuBLAS library. It is likely that the AMD implementation of the library, hipBLAS, avoids these exceptions.

4.5 Limitations

The evaluation results of FloatGuard, in terms of the floating-point exceptions captured by the tool, are contingent upon hardware and software environments and the inputs of the program. Different AMD GPUs, different compiler versions, or a different set of inputs, may trigger a different set of exceptions in a same program. The code coverage of a program when different inputs are tested can also vary, and may trigger floating-point exceptions on different code paths. FloatGuard users need to be cautious when they interpret exception information from the output of FloatGuard; proposed fixes to these floating-point exceptions may impact the correctness of the program when it is run under different environments and inputs. Lastly, subsequent exceptions captured by FloatGuard may be causally dependent on earlier ones; modifying code to prevent an earlier exception may also suppress others.

Therefore, thoroughly evaluating all potential floating-point exceptions during execution may require analyzing all instructions that can trigger exceptions, along with an input generation strategy to explore those conditions.

In terms of tool compatibility, since FloatGuard modifies assembly code, any inserted code is dependent on the GPU architecture of interest. Users may need to modify the instructions used in case a newer architecture changes register names, bit fields, and/or the instructions used to set registers, according to the instruction set architecture manuals for each GPU architecture.

In terms of reported floating-point exception information, FloatGuard outputs source code locations at which floating-point exceptions are triggered; however, these source code locations are based on the ROCgdb's output, which in turn is based on compiler debugging information. Compiler optimization passes, both from the Clang front-end and the AMD GPU back-end, may cause inaccuracies. Furthermore, the accuracy of the reported floating-point exceptions is dependent on the existing GPU exception triggering mechanism; the binary code locations of floating-point exceptions given by ROCgdb may be delayed, as described in Section 2.2, also resulting in potentially inaccurate information. Lastly, due to ROCgdb's limitations, FloatGuard only provides source code line information. Developers must investigate the source code along with assembly locations around the reported locations to pinpoint the cause of these floating-point exceptions.

Similar to the exceptions triggered by FloatGuard, slowdown ratio of FloatGuard is contingent upon the environment it runs on. Different AMD GPU models, ROCm versions, and input parameters may affect P in Equation (2); meanwhile CPU models and RAM speed may affect OH in the same equation. Both variables affect the slowdown ratio. Furthermore, as described in Section 4.2, the slowdown of using FloatGuard increases as the number of floating-point exceptions increases; and some programs have a massive amount of floating-point exceptions, likely due to the fact that exceptional values, both subnormal values and special values like Inf or NaN, can be passed downstream and continue to trigger floating-point exceptions. In cases like these, FloatGuard introduces greater slowdown with many reported exceptions being secondary. It is recommended that users of FloatGuard either (1) focus on resolving the first exception found in a kernel, then rerun FloatGuard to uncover any remaining exceptions, or (2) filter out floating-point exceptions of less impact, such as input denormal and underflow. Lastly, FloatGuard's overhead is also affected for programs with longer link times.

Overall, the slowdown ratio may be reduced by minimizing the overhead of repeatedly re-linking and re-executing the program upon each floating-point exception by methods such as preserving the program state on the host side across successive kernel invocations. As AMD's ROCm platform continues to evolve, current limitations in floating-point exception handling may be alleviated. For instance, future iterations of ROCm may allow recovery of the program state from the GPU thread where the exception occurred. In such cases, FloatGuard could be modified to eliminate or reduce the need for repeated execution, thereby improving performance. Additionally, the development of tools similar to NVBit [32] for AMD HIP could enable new approaches to floating-point exception detection on AMD GPUs.

5 Related Work

Floating-Point Exception Analysis. Similar to FloatGuard, FPSpy [12] leverages existing condition codes inside status registers in x64 processors to detect floating-point exceptions. It performs binary instrumentation of existing x64 applications on Linux, without modifying the binary or requiring extra work by developers, which differs from our work, even though it is possible to rewrite FloatGuard to directly inject code into binaries given appropriate tools. Ariadne [6] transforms a numerical program to check for exception triggering conditions and uses symbolic execution on the transformed program to find inputs that can trigger floating-point exceptions. More recently, EXCVATE [31] spoofs floating-point exceptions in numerical libraries to detect instances in which the resulting special values are not propagated according to exception-handling policy. For those, EXCVATE then uses an off-the-shelf SMT solver to generate concrete inputs that induce the buggy behavior. However, none of the works above detects floating-point exceptions for GPU programs.

Floating-point exceptions detection tools on GPUs adapt to its heterogeneous, parallel execution characteristics. Various methodologies are developed on CUDA programs running on NVIDIA GPUs due to the fact that there is no exception status hardware presented in those GPUs. FPChecker [24] uses LLVM passes to inject checking functions for each floating-point operation into the LLVM intermediate representation (IR) of the program. These checking functions are then executed alongside the original program during runtime analysis, and detects and records exceptional events. FloatGuard ultimately does not utilize the approach of injecting code into LLVM IR because of extensive backend optimizations done to the program after the post-optimization LLVM IR code is generated, and instead opt for assembly level injection. BinFPE [23] and GPU-FPX [25] both uses the NVBit [32] framework provided by NVIDIA to perform binary instrumentation on floating-point code. Both of them detect floating-point exceptional events during GPU kernel execution, and transmit data from GPU to the host program. GPU-FPX provides superior performance and better analysis of floating-point exceptions with its detector and analyzer modules.

Floating-Point Error Analysis In General. There are other floating-point error detection and analysis tools that work with specific errors in floating-point programs. These conditions found in these tools may correspond to floating-point exceptions that happen in the tested programs, but the tools themselves do not directly identify these exceptions. Select tools that track precision errors are as follows: S3FP [8] finds high precision errors in floating point programs with limited number of variables and predefined value range restrictions; it uses binary guided random testing on a black-box program executable to trigger the conditions that may cause high precision errors. FPGen [18] is similar but uses symbolic execution while injecting error check functions in the LLVM IR to trigger high precision errors, thus it is superior to S3FP in terms of high precision errors. FPSanitizer [9] and PFPSanitizer [10] uses shadow execution to run program snippets in an enhanced precision using posits [19], a replacement format of IEEE floating-points, to achieve the same goal. Herbgrind [28], on the other hand, analyzes program binaries and records the intermediate error of floating-point operations and tracks how these operations influence program

outputs and control flow. It also uses symbolic execution to facilitate numerical analysis of floating-point errors. However, it does not explicitly track floating-point exceptions as defined by IEEE 754.

On the other hand, there are also tools that track compiler-induced numerical inconsistencies, which is characterized by the difference in outputs for the same numerical program when it is compiled with different compilers and/or compiler optimization flags. FLiT [29] uses bisection search to identify the function in a program that is impacted by such inconsistencies; pLiner [17] and Ciel [26] explore the algorithm further and can identify the line of code, or even an expression in the case of Ciel, in a program that causes these inconsistencies. These inconsistencies found are often related to how different floating-point exceptions are handled under different compilers and/or optimization flags, but they themselves do not explicitly identify floating-point exceptions; also, even when the source code locations found in these tools may correspond to a floating-point exception, it does not track *all* floating-point exceptions encountered during program execution.

6 Conclusion

We presented FloatGuard, the first tool to comprehensively capture floating-point exceptions in AMD GPU applications. FloatGuard utilizes AMD GPU hardware registers and introduces an algorithm that tackles key challenges, including the practical difficulty of enabling exception detection for thousands of GPU kernel threads and capturing multiple exceptions beyond the first. This is achieved through a combination of code instrumentation at both the assembly and source levels, along with debugger-guided execution. Our evaluation of FloatGuard revealed floating-point exceptions in 507 out of 535 programs, with a slowdown ratio that linearly or sublinearly increases with the number of floating-point exceptions detected. We also studied the effects of compiler optimization flags on exception behavior, and when the same programs are compiled under both NVIDIA and AMD GPUs. Our source code, containers, and datasets are publicly available at https://github.com/LLNL/FloatGuard.

Acknowledgments

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-2005079), the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award DE-SC0022182, and the National Science Foundation under awards CCF-2119348 and CNS-2346396.

References

[1] 2022. "AMD Instinct MI200" Instruction Set Architecture Reference Guide. [Accessed 16-09-2024].
[2] AMD. 2017. rocHPCG. https://github.com/ROCm/rocHPCG
[3] AMD. 2023. HIPIFY: Convert CUDA to Portable C++ using HIP. https://github.com/ROCm/HIPIFY
[4] AMD. 2024. ROCgdb, the ROCm source-level debugger for Linux. https://rocm.docs.amd.com/projects/ROCgdb/en/latest/ Accessed: 2024-10-03.
[5] AMD. 2025. ROCm: Platform for GPU-Enabled High-Performance Computing. https://rocm.docs.amd.com/ Accessed: 2025-02-05.
[6] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *POPL*. ACM, 549–560.
[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*. IEEE Computer Society, 44–54.
[8] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *PPoPP*. ACM, 43–52.
[9] Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and detecting numerical errors in computation with posits. In *PLDI*. ACM, 731–746.
[10] Sangeeta Chowdhary and Santosh Nagarakatte. 2021. Parallel shadow execution to accelerate the debugging of numerical errors. In *ESEC/SIGSOFT FSE*. ACM, 615–626.
[11] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *GPGPU (ACM International Conference Proceeding Series, Vol. 425)*. ACM, 63–74.
[12] Peter A. Dinda, Alex Bernat, and Conor Hetland. 2020. Spying on the Floating Point Behavior of Existing, Unmodified Scientific Applications. In *HPDC*. ACM, 5–16.
[13] Jack J. Dongarra, Michael A. Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *Int. J. High Perform. Comput. Appl.* 30, 1 (2016), 3–10.
[14] GNU Project. 2024. *GDB: The GNU Project Debugger*. https://www.gnu.org/software/gdb/
[15] Ganesh Gopalakrishnan, Ignacio Laguna, Ang Li, Pavel Panchekha, Cindy Rubio-González, and Zachary Tatlock. 2021. Guarding Numerics Amidst Rising Heterogeneity. In *Correctness@SC*. IEEE, 9–15.
[16] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 innovative parallel computing (InPar)*. Ieee, 1–10.
[17] Hui Guo, Ignacio Laguna, and Cindy Rubio-González. 2020. pLiner: isolating lines of floating-point code for compiler-induced variability. In *SC*. IEEE/ACM, 49.
[18] Hui Guo and Cindy Rubio-González. 2020. Efficient generation of error-inducing floating-point inputs via symbolic execution. In *ICSE*. ACM, 1261–1272.
[19] John L Gustafson and Isaac T Yonemoto. 2017. Beating floating point at its own game: Posit arithmetic. *Supercomputing frontiers and innovations* 4, 2 (2017), 71–86.
[20] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *ISCA*. IEEE, 473–486.
[21] Ignacio Laguna. 2019. Fpchecker: Detecting floating-point exceptions in gpu applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1126–1129.
[22] Ignacio Laguna. 2020. Varity: Quantifying Floating-Point Variations in HPC Systems Through Randomized Testing. In *IPDPS*. IEEE, 622–633.
[23] Ignacio Laguna, Xinyi Li, and Ganesh Gopalakrishnan. 2022. BinFPE: accurate floating-point exception detection for GPU applications. In *SOAP@PLDI*. ACM, 1–8.
[24] Ignacio Laguna, Tanmay Tirpankar, Xinyi Li, and Ganesh Gopalakrishnan. 2022. FPChecker: Floating-Point Exception Detection Tool and Benchmark for Parallel and Distributed HPC. In *IISWC*. IEEE, 39–50.
[25] Xinyi Li, Ignacio Laguna, Bo Fang, Katarzyna Swirydowicz, Ang Li, and Ganesh Gopalakrishnan. 2023. Design and Evaluation of GPU-FPX: A Low-Overhead tool for Floating-Point Exception Detection in NVIDIA GPUs. In *HPDC*. ACM, 59–71.
[26] Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. 2023. Expression Isolation of Compiler-Induced Numerical Inconsistencies in Heterogeneous Code. In *ISC (Lecture Notes in Computer Science, Vol. 13948)*. Springer, 381–401.
[27] NVIDIA Corporation. 2024. NVIDIA HPCG Benchmark. https://github.com/NVIDIA/nvidia-hpcg Accessed: 2024-10-03.
[28] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. 2017. Finding Root Causes of Floating Point Error with Herbgrind. *CoRR* abs/1705.10416 (2017).
[29] Geoffrey Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, and Dong H. Ahn. 2017. FLiT: Cross-platform floating-point result-consistency tester and workload. In *IISWC*. IEEE Computer Society, 229–238.
[30] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127, 7.2 (2012).
[31] Jackson Vanover, James Demmel, Xiaoye S. Li, and Cindy Rubio-González. 2025. EXCVATE: Spoofing Exceptions and Solving Constraints to Test Exception Handling in Numerical Libraries. In *Proceedings of the 32nd IEEE International Symposium on Computer Arithmetic*. IEEE.
[32] Oreste Villa, Mark Stephenson, David W. Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *MICRO*. ACM, 372–383.
[33] Anwar Hossain Zahid, Ignacio Laguna, and Wei Le. 2024. Testing GPU Numerics: Finding Numerical Differences Between NVIDIA and AMD GPUs. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 547–557.