Design and Evaluation of GPU-FPX: A Low-Overhead tool for Floating-Point Exception Detection in NVIDIA GPUs

Xinyi Li
Kahlert School of Computing
University of Utah
USA
xin_yi.li@utah.edu

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

Bo Fang
Pacific Northwest National Laboratory
USA
bo.fang@pnnl.gov

Katarzyna Swirydowicz
Pacific Northwest National Laboratory
USA
kasia.swirydowicz@pnnl.gov

Ang Li
Pacific Northwest National Laboratory
USA
ang.li@pnnl.gov

Ganesh Gopalakrishnan
Kahlert School of Computing
University of Utah
USA
ganesh@cs.utah.edu

ABSTRACT

Floating-point exceptions occurring during numerical computations can be a serious threat to the validity of the computed results if they are not caught and diagnosed. Unfortunately, on NVIDIA GPUs—today's most widely used type and which do not have hardware exception traps—this task must be carried out in software. Given the prevalence of closed-source kernels, efficient binary-level exception tracking is essential. It is also important to know how exceptions flow through the code, whether they alter the code's behavior and additionally whether these exceptions can be detected at the program outputs or are killed inside program flow-paths. In this paper, we introduce GPU-FPX, a tool that has low overhead, allows for deep understanding of the origin and flow of exceptions, and also how exceptions are modified by code optimizations. We measure GPU-FPX's performance over 151 widely used GPU programs, detecting 26 serious exceptions that were previously not reported. Our results show that GPU-FPX is 16× faster with respect to the geometric-mean runtime in relation to the only comparable prior tool, while also helping debug a larger class of codes more effectively. GPU-FPX and its benchmarks have been released.

CCS CONCEPTS

• **Software and its engineering** → **Software organization and properties**; **Software safety**; *Software maintenance tools*; • **Computer systems organization** → **Single instruction, multiple data**.

KEYWORDS

Floating-point exceptions; GPUs; High-performance computing; Machine Learning; Binary Instrumentation; Numerical Programs

ACM Reference Format:
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 *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3588195.3592991

HPDC '23, June 16–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0155-9/23/06.
https://doi.org/10.1145/3588195.3592991

1 INTRODUCTION

Motivation

With the growing scale and variety of today's high-performance computing (HPC) and machine learning (ML) systems, programmers must be ever-vigilant of numerical errors. One of the challenges encountered in the contemporary high-performance computing (HPC) community with heterogeneous systems is handling exceptions on GPUs, as discussed in [11]. This paper is on efficiently detecting *floating-point exceptions* [15] that can render the computed results unreliable and/or add to debugging effort.

Exceptions arise due to a number of reasons, and either impede one's ability to use a piece of numerical software ("outputs are NaNs") or produce normal-looking results that are unreliable. There are five types of floating-point exceptions as IEEE 754 [12] defined: underflow, overflow, invalid operation, division by zero, and inexact. Among these, invalid operation and division by zero are caused by improper mathematical behavior, such as taking the square root of a negative number. Meanwhile, underflow, overflow, and inexact are caused by the limited representation range of floating-point numbers. Efficient detection of exceptions has become harder with growing problem-scale and platform variety.

A related concept to exceptions is that of *exceptional values* (NaN, INF and subnormals), also catalogued under IEEE exceptions, and used to model things such as unknown or unrepresentable data. Demmel et al. [9] discuss how exceptional values can cause unreliable behavior of the widely used LAPACK suite: even though the input "A" matrix contains NaNs, the LAPACK SGESV() can output a NaN-free solution, which can fool the user into thinking that the computation is reliable. Such examples highlight the need for incisive floating-point exception detection and *diagnosis* tools.

Given the growing use of GPUs in high-end HPC and ML programs, as well as rising heterogeneity [11], *GPU-specialized* exception detection tools are much needed. Unfortunately, building such tools is challenging, as NVIDIA GPUs—the most widely used

of GPU types—do not have hardware-level exception trap mechanisms[1] [1]. As a result, even when exceptional values are present in the program output, it can be difficult to correct the underlying issue, as even the location(s) of the underlying exception(s) are not known. Additionally, many widely used GPU-supported functions are available only as assembly code in a language called SASS that carries no official documentation (source-codes are not available). A search for "NaN" on GitHub in CUDA/Python retrieves thousands of user-reports, often with respect to closed-source GPU libraries.

Practical exception-detection tools must be sufficiently efficient to analyze large software libraries and applications. They must be capable of analyzing closed-source GPU programs, and even in such usage, they must produce adequate amounts of debugging information. Such debugging information is vital for problem-diagnosis, coming up with *repairs*, and even to conclude whether the detected exceptions can be ignored by end-users.[2] Tools with such features can play a crucial role at many stages of GPU software development. For instance, when compiler optimizations such as FAST_MATH are applied, the exception-behavior of the code can change in confusing ways. Another valuable insight to obtain is whether the exceptions detected deep within kernels fail to appear at their output, and/or whether they alter internal control-flows in unintended ways (an example to be given momentarily). Well-designed GPU exception detection tools must, in such cases, help the user understand that the output of the kernel is not dependable.

In this paper, we introduce a new tool called GPU-FPX that is the first to support all the aforesaid features. We present a comprehensive study of exceptions in GPU-based computations, including the impact of compiler optimizations, and actual examples illustrating the flow, disappearance, and overall impact of exceptions.

Achieving high performance in GPU-FPX while retaining essential debugging information call for innovative solutions that minimize unnecessary tracing and transmission of data. In this paper, we evaluate three approaches for this: 1) keeping a table in the GPU global memory for creating deduplicated exception records, 2) transmitting diagnostic data from the GPU to the CPU only when exceptional values arise, and 3) employing selective instrumentation of the kernel ("sampling") to minimize JIT-compilation overheads during repeated kernel execution. Furthermore, we added an *analyzer* to report instruction-level exceptions for users who wish to dive deeper into the code and identify the root causes of these exceptions.

Limitation of state-of-art approaches

Support for floating-point exception checking in GPUs was first demonstrated in the FPChecker [17] tool. This work studied a handful of GPU kernels, relying on LLVM-level instrumentation of GPU kernels compiled using Clang. Unfortunately, the majority of high-performance GPU programs are compiled using NVCC which is a closed compiler where the opportunity for external users to instrument its internal LLVM code does not exist. Moreover, many widely used libraries are available *only* in binary—at the SASS level (sources or LLVM are unavailable).

BinFPE [19] is the next exception checking tool of interest, performing SASS-level GPU exception analysis. This tool has several limitations compared to GPU-FPX.

First, BinFPE is orders of magnitude slower than GPU-FPX. Next, BinFPE was studied only on a limited number of programs. Also, there was no study of how exceptions change with compiler optimizations. There was no attempt at characterizing the severity of the detected exceptions. Also, they did not consider how exception flow through instructions or are affected by control-flows.

To explain how exceptions may skew control-flows and render a program's output unreliable, consider the statement *if* $a < b$ *then* P *else* Q. Here, if a or b are NaN, the predicate evaluates to $false$. This can result in selecting the Q code-path, which may not be what the programmer intended. Moreover, potential exceptions within P are never triggered. Such control-flow altering exceptions cannot be detected using BinFPE. In fact, all the instructions in the right-hand side column of Table 1 (this column includes FSEL that governs control flows) are missed by BinFPE.

In summary, GPU-FPX is a much more reliable and faster GPU exception debugging and diagnosis tool. We demonstrate its effectiveness on 151 important ML and HPC programs. Notably, GPU-FPX successfully terminates on benchmarks on which BinFPE hangs. Additionally, our study of the CuMF-Movielens example (in §4) shows that, when applying all previously mentioned optimizations (including sampling), GPU-FPX executes in 5 minutes, compared to BinFPE's 6-hour runtime, without missing any exceptions.

GPU-FPX also aids in understanding the root causes of exceptions and devising mitigation strategies[3]. Its ability to analyze exception flows is vital, particularly given that the latest IEEE 754 standard [12, 14] mandates NaN propagation for functions like MAX and MIN. However, NVIDIA adheres to the 2008 IEEE standard [15, 27], which does not require NaN propagation. Consequently, using GPU-FPX safeguards users beyond NVIDIA's requirements.

Key insights and contributions

Our key insight is that addressing the issue of floating-point exceptions in GPU codes requires a comprehensive approach that includes considerations of efficiency, handling closed-source programs, understanding how exceptions propagate, studying optimization flags, and exploring techniques for repair. This is especially important given that GPUs do not inherently generate hardware traps. Issuing PRINTFS in GPU codes is a poor substitute to either hardware exception traps or an efficient exception analyzer such as GPU-FPX. GPU-FPX provides, for the first time, scalable and *comprehensive* support for debugging floating-point exceptions in GPU codes.

Another key insight and discovery is that one must include the capability to track the flow of exceptions *even within individual instructions*. As we discuss later, this is a key link[4]in the chain of techniques that help provide insights into the appearance, propagation, and disappearance of exceptions.

Our next insights are that many of the widely used programs within the 151 we studied actually contain 26 serious exceptions. Notably, these exceptions are generated *even by the test inputs*

[1] In contrast, AMD GPUs enable hardware-based exception trapping [4], potentially simplifying exception-checking tool development.

[2] An example of when exceptions do not matter occurs when an INF is detected but later vanishes due to division by INF, which is a standard mathematical behavior.

[3] Languages such as Julia provide such operator variants that help flow exceptions; our analysis supports capabilities with similar benefits at the analysis level.

[4] For example in FADD R1 R2 R3, if R3=INF, R1=INF, and R2 does not have an exceptional value, then we can conclude that INF flowed from R3 to R1.

Table 1: Opcodes in SASS instructions: that GPU-FPX supports.

Computation Opcodes		Control Flow Opcodes	
Instructions	Description	Instructions	Description
FADD	FP32 Add	FSEL	Floating Point Select
FADD32I	FP32 Add	FSET	FP32 Compare And Set
FFMA32I	FP32 Fused Multiply and Add	FSETP	FP32 Compare And Set Predicate
FFMA	FP32 Fused Multiply and Add	FMNMX	FP32 Minimum/Maximum
FMUL	FP32 Multiply	DSETP	FP64 Compare And Set Predicate
FMUL32I	FP32 Multiply		
MUFU	FP32 Multi Function Operation		
DADD	FP64 Add		
DFMA	FP64 Fused Mutiply Add		
DMUL	FP64 Multiply		

provided with the programs. We show how GPU-FPX can help resolve these exceptions through suitable repairs. A study of this magnitude requires an efficient tool which GPU-FPX is.

Limitations of the proposed approach

Even with the availability of a tool such as GPU-FPX, users will not often be able to make significant progress with vendor-provided binary-only kernels. We recently experienced these difficulties when debugging a NaN issue in cuSPARSE (closed source) library using GPU-FPX. Through trial-and-error, we had to resort to boosting the diagonal values of the matrix being used using a facility provided within cuSPARSE. Our guessed strategy was to eliminate a zero pivot that we suspected to be the underlying cause for the NaN. This helps eliminate the observed NaN (detailed in §5), but with no further assurances.

These difficulties notwithstanding, GPU-FPX still is the only tool available today that provides this modest level of insight. A far more useful future use of GPU-FPX would be one in which the developers of closed-source libraries such as cuSparse used it to test their libraries, as well as *help document* the exact conditions under which they might produce exceptions. For programs with sources, GPU-FPX provides much more helpful reports including details of kernel launches from C++-Lambdas, line-number information associated with exceptions, etc.

Key Results in this Paper

- We present GPU-FPX, an efficient tool that detects exceptions three orders of magnitude faster than the best bespoke tool.
- GPU-FPX reports exceptions in detail including how they are introduced, propagate, and vanish.
- GPU-FPX helps programmers realize whether compiler optimization flags change exception behaviors, including when FP64 instructions are converted to FP32 under optimization.
- GPU-FPX includes efficient sampling-based methods to mitigate JIT-ting overheads.
- We employed GPU-FPX to study 151 ML and HPC programs, showing how GPU-FPX helps understand the root-cause of exceptions well-enough to come up with mitigation methods.
- The programs in our study are available at the public repository at https://github.com/LLNL/GPU-FPX.

2 BACKGROUND

2.1 IEEE Floating-Point Exceptions Basics

A binary floating-point number $x = \pm s \times 2^e$ consists of the significant s, and the exponent e, with three special values:

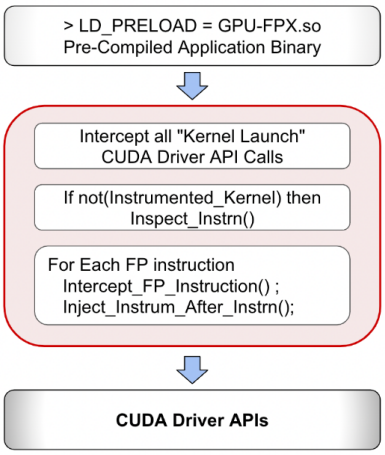

Figure 1: Application Interception, Inspection, and Instrumentation

(1) If the exponent is FF(FFFF in FP64) Hex, then:
   (a) if the mantissa is 0, it represents the INF exception.
   (b) on the other hand, if the mantissa is is non-zero, then it represents the NaN exception.
(2) Finally, if the exponent is 00 and the mantissa is non-zero, then it represents the subnormal exception.

These three special values may be used when the following conditions are incurred: overflow, underflow, or division by zero (INF), invalid operation (NaN), an subnormal number generation (see IEEE 754 [12]).

2.2 NVBit and SASS Basics

NVIDIA provides the only binary instrumentation framework called NVBit [26] for observing and controlling the behavior of GPU programs at the binary SASS assembly program level; GPU-FPX builds on NVBit. Basically, NVBit provides functions to instrument SASS instructions by intercepting the CUDA driver APIs. As illustrated in Figure 1, an NVBit tool inspects and instruments a CUDA program by loading the NVBit tool shared library prior to all other libraries. To help understand these binary-instrumentation activities for floating-point exception detection, we provide a concise summary of SASS floating-point instructions. Given that NVIDIA does not provide the details, we relied on past work [13, 16, 19] and our own findings (through reverse-engineering) on top of that. The opcodes of instructions handled by GPU-FPX include the FP64 (double precision) and FP32 (single precision) operations in Table 1.

Instruction-Format: The compute capabilities 7.x–8.x use the instruction format `(Op) (DestReg), (Param1), (Param2) ...` where: `Op` is the opcode defined in Table 1; `DestReg` is the destination register that stores the result of the instruction; `Param` are the sources for the instruction. Hence, GPU-FPX can handle `Param` with REGISTER, IMM_DOUBLE, and GENERIC types [26].

Double precision quantities: SASS registers are FP32 registers natively. FP64 results are stored in two adjacent FP32 registers, i.e., given a FP64 instruction `INST` R_d, R_p, R_q, where R_d is the destination register, the instruction uses R_d and R_{d+1} to store the

result of the FP64 operation. Similarly, the instruction uses R_p and R_{p+1} to read the first parameter and R_q and R_{q+1} to read the second parameter. For example, consider instruction `DMUL R0, R2, R4` which is a multiply. Here, `R0` and `R1` will end up containing the result of the FP64 multiplication. To check for exceptional values in the result of FP64 operations, we combine these registers into an FP64 value that is passed into our analysis functions.

Division operation: Division is carried out in software by first computing the reciprocal (use `MUFU.RCP(64H)`) and then employing an iterative algorithm. There are `FCHK` checks employed in such codes for guarding against division-by-zero, etc. As it turns out, the division algorithm gets expanded differently on Turing and Ampere GPUs, while also generating a different number of exceptions.[5]

2.3 Overview of BinFPE

BinFPE is an NVBit binary instrumentation tool designed to detect floating-point exceptions. By adhering to NVBit's design principles, it instruments each floating-point arithmetic instruction, recording the destination registers. These values are subsequently sent back to the host (CPU) for analysis to identify any existing exceptions.

BinFPE has the folllowing drawbacks which `GPU-FPX` attempts to overcome. First, it transmits data far in excess of what is required to diagnose exceptions, which can bogs down the GPU-to-CPU communication channel. Also, the BinFPE-provided debugging information does not allow one to determine the severity of the detected exceptions, as it merely reports exceptions based on the value carried in the destination register of a SASS instruction and not the detailed flow of exceptions caused by control-flow instructions.

3 DETAILS OF GPU-FPX'S DESIGN

Figure 2 presents an overview of `GPU-FPX` organized in terms of two components: *detector* and *analyzer*. The faster *detector* helps pinpoint exception-generating locations across all kernels, while the (relatively) slower *analyzer* helps compile exception flow information, and also helping ascertain the importance of these exceptions. In all the performance comparisons against BinFPE reported in this paper, we measure and report the time taken by the *detector*, as its functionality matches that of BinFPE (BinFPE has no analysis components).[6] Both the *detector* and the *analyzer* rely on `LD_PRELOAD` to load shared object files before any other libraries. They follow the architecture illustrated in Figure 1, intercepting the targeted kernel and instrumenting the executing floating-point instructions. We now detail *detector* in §3.1, and *analyzer* in §3.2.

3.1 Implementation of a Scalable *Detector*

The *detector*'s design resembles that of BinFPE, as it checks the destination register's value for floating-point operations. However, in contrast to BinFPE, `GPU-FPX`'s checking process *takes place on the GPU device rather than the host.* Additionally, `GPU-FPX`'s injected code utilizes a table in global memory to record unique exceptions, aiming to minimize data transfers between the device and the host. By referring to this table, the *detector* can decide whether to send back information, as depicted in the *detector* section of Figure 2.

[6]Turning on the analysis components of binary instrumentation, by itself, only adds modest overheads, and typically done only on exception-generating kernels.

To further enhance performance, users can selectively instrument temporally repeating kernels, thereby decreasing the associated just-in-time (JIT) compilation overheads (a cost paid per invocation while using NVBit).

The basic slowdown of an application caused by NVBit is discussed by its authors in [26]. In this section, we explain how we minimize the additional overheads introduced by our approach.

3.1.1 On-the-fly Parallel Exception Checking: `GPU-FPX`'s *detector* conducts on-the-fly exception checking in the GPU injected code. For every floating-point instruction, we inject code that examines the values in the destination register, following the exceptional value definitions outlined in §2.1.

Various instruction types may necessitate unique checking methodologies. For instance, in division by zero exceptions, it is essential to verify if the opcode is `MUFU.RCP(64H)` and the destination register holds a NaN or INF value. In the case of FP64 type instructions, we concatenate two consecutive registers (that together carry the result) to perform the check. As a result, we implement four specialized injection functions and choose the appropriate one based on Algorithm 1. Notice that in Line 12 - 16, when the opcode contains `64H`, the register stores the high 32-bits of the FP64 value.

Algorithm 1 Specialized Injection Functions

```
Require: FP SASS instruction
 1: procedure Injection algorithm(Op,RdestNum)
 2:     if Op contains MUFU.RCP then
 3:         if Op contains 64H then
 4:             inject check_64_div0(RdestNum-1,RdestNum)
 5:         else
 6:             inject check_32_div0(RdestNum)
 7:         end if
 8:     else
 9:         if Op has FP32 Prefix then
10:             inject check_32_nan_inf_sub(RdestNum)
11:         else if Op has FP64 Prefix then
12:             if Op contains 64H then
13:                 inject check_64_nan_inf_sub(RdestNum-1,RdestNum)
14:             else
15:                 inject check_64_nan_inf_sub(RdestNum,RdestNum+1)
16:             end if
17:         else skip instrumentation
18:         end if
19:     end if
20: end procedure
```

3.1.2 Early Exception Notification, Efficient Transfers: To reduce the overhead associated with transmitting exceptional information to the host, we need to transfer only the minimal data necessary for generating the report. For example, if an FP32 NaN exception has already been reported at a specific location, sending the same information again is redundant.

To achieve this, we allocate a hash table *GT* in the global memory when launching the GPU context. This table keeps track of the location, exception type, and floating-point format information. After completing the check for each instruction, `GPU-FPX` examines whether the same check result for the same location and floating-point type has been encountered previously. If not, the exception information is sent back to the host using a channel API.

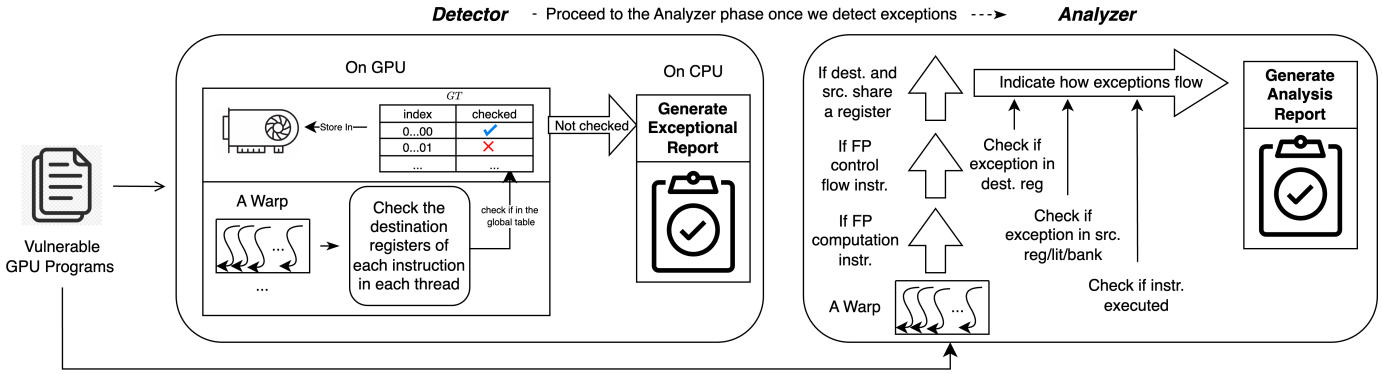

Figure 2: Overview of GPU-FPX. Utilizing the faster *detector* for initial screening of susceptible programs and applying the *analyzer* to those with detected exceptions for a more efficient workflow.

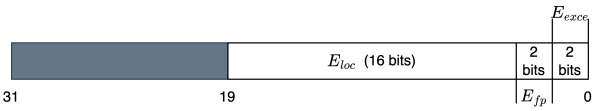

Figure 3: Exception-Record Format (see Algorithm 2)

GT Design: GT maintains a record of exceptional information occurrences. Each **key** in the table corresponds to an exception-record (Figure 3) – a triplet of $\langle E_{exce}, E_{loc}, E_{fp} \rangle$. Here,

- E_{exce} employs two bits to encode crucial exceptions such as NaN, INF, SUB (subnormal), and DIV0 (division by zero);
- E_{loc} consists of 16 bits, allowing for storage of different 2^{16} locations;
- E_{fp} accommodates up to four FP formats (presently FP32 and FP64, with future plans to include FP16 and more).

Each **value** indicates whether the associated combination (represented as the key) has occurred or not. Given that the smallest GPU memory access size is 32 bits, we utilize a 32-bit integer for value storage. We opt for a hash table due to its $O(1)$ access time. The 16-bit location index is chosen to maintain the table size at 4MB.

Sending back information based on the GT table: After performing on-the-fly checking within the injected codes, we determine whether the checked information should be sent back to the CPU, as depicted in the *detector* part of Figure 2. We detail our injection function in Algorithm 2. Upon checking the exceptions in Line 2, all threads broadcast the checking result `e_type` to the leading thread in the warp, as shown in Line 6. Then, in the leading thread, we encode $E_{exce}, E_{loc}, E_{fp}$ from each thread as displayed in Figure 3, Line 10 and only push the information to the host if the same combination hasn't occurred before. Such new exceptions alert users about exceptions before (hour-long) GPU runs finish. A complete record of all exceptions is available in *GT* for detailed analysis after the GPU program terminates.

3.1.3 Selective Instrumentation: Reference [26] highlights that a significant portion of `NVBit`'s overhead comes from JIT-compilation which is incurred *each time a kernel is launched at runtime*. To combat this overhead, we have implemented selective instrumentation techniques to reduce the number of kernels requiring JIT-ting.

Algorithm 2 Injected codes for each instruction

Require: `reg_val`: destination register value
Require: `GT`: Table *GT*
Require: `locfp`: The encoded E_{loc} and E_{fp} for this instruction

```
1:  procedure EXCEPTION-RECORD-XFER(reg_val, GT, locfp)
2:      e ← CHECKEXCE(reg_val)                    ▷ Perform checking
3:      for each thread T in the warp do
4:          exn_type[T] = e
5:          loc_fp[T] = locfp
6:          Broadcast e_type to the leading thread in the same warp
7:      end for
8:      if in the leader thread in the warp then
9:          for each exn_type[T] > 0 do
10:             index ← ENCODE_ID(loc_fp[T], exn_type[T])
11:             if GT[index] is not empty then
12:                 Push index into GPU-CPU Channel
13:             end if
14:         end for
15:     end if
16: end procedure
```

One approach we use is the "white-list" method, where code developers can select a fixed set of important kernels to be included in the instrumentation. Additionally, we observe (especially in neural-network GPU codes) that many kernels are repeatedly invoked. To exploit this fact, in GPU-FPX, we allow users to instrument a kernel once in k of its invocations (**FREQ-REDN-FACTOR** in the Algorithm 3). We observe that this is a good tradeoff between efficiency and collection efficacy (measurement in Figure 6).

In Algorithm 3, Line 3 to Line 9 examine whether the user has specified a "white-list". If such a list exists and the current kernel is included, the `instr` variable is set to `true`. If no list is provided, the current kernel will be instrumented by default. Line 10 to Line 13 determine if undersampling is performed.

3.2 Implementation of *Analyzer*

We now detail *analyzer*'s flow-tracking abilities.

3.2.1 Handling Control-Flow Opcodes, Source Operands: We now describe how these unique features of GPU-FPX that support accurate exception root-causing are implemented.

Algorithm 3 Whenever CURRENT_KERNEL is invoked, decide whether to instrument it (in white-list and once every k calls).

Require: current_kernel: Kernel currently intercepted by NVBit
Require: k: FREQ-REDN-FACTOR
1: **procedure** SELECTIVE_INSTR(current_kernel)
2: instr ← false ▷ instr variable indicate instrument or not.
3: **if** user_specified_kernels **then**
4: **if** current_kernel in enable_kernels **then**
5: instr ← true
6: **end if**
7: **else**
8: instr ← true
9: **end if**
10: **if** $k! = 0$ **then**
11: **if** num[current_kernel]% k != 0 **then**
12: instr ← false
13: **end if**
14: **end if**
15: num[current_kernel]++
16: NVBIT_ENABLE_INSTRUMENTED(instr)
17: **end procedure**

Control Flow Opcodes: As the instrumentation process progresses, control flow opcode must be recorded in order to trace exception flow. To achieve this, we utilize a vector map that associates the SASS string of the opcode with an integer, referred to as `opcode_id`. This integer is then passed along and incorporated into the injected code appropriately to permit exception-flow tracing.

Source Operands: Source operands come in various types, including REG, IMM_DOUBLE, GENERIC, and CBANK, each of which requires a different handling strategy. For IMM_DOUBLE and GENERIC types, their values are known at compile-time. For instance, the instruction "FADD RZ RZ +INF" has an IMM_DOUBLE type "+INF" as one of its sources, while the instruction "MUFU.RSQ RZ -QNAN" has a GENERIC type "-QNAN" as one of its sources. In these cases, we obtain the relevant information during the JIT-compilation process and pass it to the instrumented codes. On the other hand, the values in REG and CBANK types can only be determined during runtime. Thus, we record the register id, cbank id, and offset, and pass them as variadic arguments to the injection function. The injected codes then read the corresponding values at runtime. Listing 1 presents the information to be passed to the injection function, while the handling strategies for different operand types are detailed in Listing 2.

Shared registers in dest. and src. When designing the exceptional value analyzer, we must consider the scenario where a register is used as both a source and destination operand in an instruction. For example, in the instruction "FADD R6, R1, R6", register "R6" serves both as a source and destination. In such cases, if we were to conduct our analysis after the instruction has been executed, we may not accurately identify exceptional values in the source register, as its value would have been overwritten by the computation result. To account for this, we perform an *additional check* prior to the execution of the instruction. Specifically, we compare the first register number in the register list, which always corresponds to the destination register, with the remaining register numbers to

```

/* Map the sass string to an integer opcode_id*/
int opcode_id = opcode_to_id_map(instr->getSASS());

/* Store the list of register numbers */
std::vector<int> reg_num_list;

/* Store the list of cbank ids and offsets */
std::vector<int> cbank_list;

/* A variable to indicate if an exception is known
    during compile-time */
int compile_e_type = NO_EXCEPT

/* A variable to record the number of values to be
    obtained at runtime */
int num_run_vals = 0;

for (int i = 0; i < instr->getNumOperands(); i++) {
    /* Handle different types of operands*/
}
/*Pass the num_run_vals, compile_e_type, reg_num_list
     and cbank_list into the injection function. */
```

Listing 1: Information passed to injection function.

```
/* get the operand "i" */
    InstrType::operand_t *op = instr->getOperand(i);

/*Push the register number into the register list*/
    if(op->type == InstrType::OperandType::REG) {
        reg_num_list.push_back(op->u.reg.num);
        num_run_vals++;
    }

/*Push the cbank id and offset into the cbank list*/
    else if(op->type==InstrType::OperandType::CBANK)
        {
        cbank_list.push_back(op->u.cbank.id);
        cbank_list.push_back(op->u.cbank.imm_offset);
        num_run_vals++;
    }

/*Check if the IMM_DOUBLE is an exceptional value*/
    else if (op->type==InstrType::OperandType::IMM_
        DOUBLE) {
        double imm_value = op->u.imm_double.value;
        if(isnan(imm_value)){
            compile_e_type = NAN_EXCEPT;
        }
        else if(isinf(imm_value)){
            compile_e_type = INF_EXCEPT;
        }
    }

/*Check if the GENERIC is an exceptional value*/
    else if(op->type==InstrType::OperandType::GENERIC
        ) {
        std::string gen_value = op->u.generic.array;
        if(gen_value.contains("NAN")){
           compile_e_type = NAN_EXCEPT;
        } else if(gen_value.contains("INF")){
            compile_e_type = INF_EXCEPT;
        }
    }
    else {continue.}
}
```

Listing 2: Details of handling different types of operands

Table 2: Instruction state categorization based on the instruction information gathered by the Analyzer. EV is a concrete exceptional value (i.e., NaN, INF, SUB)

Instr. Info.				
Share Reg.	Ctrl. Flow Ops	Dest. Except.	Srcs. Except.	State
✓				Comparison
✗	✓			Appearance
✗	✗	Except=EV	No EV	Propagation
✗	✗	No Except	With EV Except	Disappearance

determine whether the destination and source operands share the same register.

3.2.2 Generating Analysis Reports: Upon gathering the necessary information during both compilation and runtime, it becomes possible to track the flow of exceptional values through the instructions. We categorize the state of instructions into five types, as shown in Table 2. It is important to note that, for each case, GPU-FPX prints the state, location, SASS code, and value types in registers, allowing for comprehensive analysis and understanding.

4 EVALUATION OF THE GPU-FPX DETECTOR

We now present a comprehensive evaluation of GPU-FPX *detector*, using a diverse set of benchmark programs. The benchmarks consist of a total of 151 programs, including a mix of HPC and machine learning applications, from various benchmark suites (Table 3).[7]

Table 3: Our evaluation target programs and their containing benchmark suites (use of the red font indicates that exceptions were detected in them).

Suite	Programs
gpu-rodinia	b+tree, backprop, bfs, cfd, dwt2d, gaussian, heartwall, hotspot, hotspot3D, huffman, hybridsort, kmeans, lavaMD, leukocyte, lud, myocyte, nn, nw, srad, srad_v1
shoc	BFS, FFT, GEMM, Stencil2D, MD, Reduction, Scan, Sort, Spmv, Triad, MD5Hash, S3D, QTC
paraboil	histo, mri-q,sad, stencil, mri-gridding, tpacf, spmv, bfs, cutcp, sgemm
GPGPU_SIM	wp, cp, lps, mum, rayTracing, libor
Exascale Proxy Applications	Laghos, Remhos, XSBench, Sw4lite, Kripke, LULESH
polybenchGpu	2DCONV, 2MM, 3DCONV, 3MM, ADI, ATAX, BICG, CORR, COVAR, FDTD-2D, GEMM, GEMVER, GESUMMV, GRAMSCHM, JACOBI1D, JACOBI2D, LU, MVT, SYR2K, SYRK
NVIDIA HPC-Benchmarks	HPCG
cuda-samples	71 programs
ML open issues	CuMF-Movielens, SRU-Example, cuML-HousePrice

We ran our experiments on two different machines: Machine 1: AMD Ryzen 5 3600 6-Core processor with NVIDIA GeForce RTX 2070 SUPER GPU, Machine 2: Intel(R) Core(TM) i9-10900K CPU (10 cores) @ 3.79 GHz with GeForce RTX 3060 GPU

Our key findings are these: (1) the detection of previously unknown exceptions across widely used GPU programs, as shown in Table 4; (2) a 3-orders of magnitude speedup compared to the best available tool, as shown in Table 5; (3) the effectiveness of our sampling strategy in mitigating JIT-ting overheads without missing relevant exceptions, as shown in Table 5; (4) the effect of compiler optimizations on exceptions, as shown in Table 6, and (5) analysis of the impact of exceptional values on programs in Table 7. The rest of this section provides more details on these findings, including tables and figures that summarize our results.

4.1 Exception Results, Table 4.

This table lists the exceptions detected using the *detector* in GPU-FPX. The results show the exceptions generated by FP64 and FP32 floating-point instructions when running the benchmark programs on the data sets *that came with the programs*. The presence of the more serious exceptions, namely NaN, INF, or DIV0, are denoted with red fonts in our tables.

Out of the 151 benchmark programs, we found exceptions in 26 programs where the exceptions may be meaningful[8], with nine of them involving NaN, INF, or DIV0. These programs are widely used and include implementations of basic physical and biological processes that may be used in real-world simulations. We further analyze these programs in Section §5.

For some programs, we found exceptions under both FP64 and FP32 modes, despite the original code using FP64 exclusively. This is because of the binding of some of the operations by the compiler onto GPU special function units (SFUs [5, 22]) that provide higher performance, but also higher rounding error.

4.2 Performance Across Tool Evolution, Figure 4

The performance metric we used in the evaluation is **slowdown** – the ratio of the program's running time with our tool and the program's original runtime (i.e., without exception checking). Figure 4 illustrates the slowdown distribution of the 151 programs tested using BinFPE and the two phases of GPU-FPX's evolution. In the first Phase, GPU-FPX implements parallel checking and sends back exceptions (w/o GT), while in the second phase it adds a global table, returning only deduplicated information (w/ GT).

Figure 4 demonstrates that our techniques end up *shifting* more programs from the higher slow-down ranges down to lower slow-down ranges—showing that far less programs take longer. Although the addition of the global table does not appear to yield significant improvements, it resolves the hanging issues in previous cases—showing that deduplication has the effect of avoiding communication-related congestion.

In summary, when using GPU-FPX's detector, over 60% of the programs experience a slowdown of less than 10x, compared to only 40% of the programs with BinFPE. To better visualize the final overall improvement over BinFPE, we present Figure 5.[9] The X-axis represents the (log2) slowdown caused by GPU-FPX, and the Y-axis shows the corresponding (log2) slowdowns caused by BinFPE. Each

[7]We studied, but do not include the programs from CUDA-Samples in this table, given that there are 71 programs in this suite.

[8]For some programs involving Monte Carlo or compression algorithm, exceptional values may be meaningless, so we didn't show them in this table.

[8]Notice that GPU binary instrumentation tools suffer from significantly more slowdown than other instrumentation slowdowns—even CPU binary instrumentation tools—due to JIT-ting and other aspects [26]. Even so, designers are known to use a tool that has 100x slowdown if that is the only tool that can do a certain task—e.g., the popular Valgrind has around this level of slowdown.

[9]The three outlier examples (simpleAWBarrier, reductionMultiBlockCG, and conjugateGradientMultiBlockCG) are ones where GPU-FPX is significantly slower. These are examples where there are very few floating-point operations for which the allocation of the global-table causes a net slow-down without any role to play.

Table 4: Exceptions detected by GPU-FPX. Red fonts indicate NAN, INF, and DIV0 exceptions.

		FP64				FP32			
Benchmark Suites	Program	NAN	INF	SUB	DIV0	NAN	INF	SUB	DIV0
polybenchGpu	GRAMSCHM	0	0	0	0	**7**	**1**	0	**1**
	LU	0	0	0	0	**3**	0	0	**1**
gpu-rodinia	cfd	0	0	0	0	0	0	13	0
	myocyte	**57**	**63**	2	**3**	**92**	**76**	8	0
SHOC	S3D	0	0	0	0	0	**7**	129	0
Parboil	stencil	0	0	0	0	0	0	2	0
GPGPU	wp	0	0	0	0	0	0	47	0
	rayTracing	0	0	0	0	0	0	10	0
cuda-samples	interval	**1**	**1**	0	0	0	0	0	0
	conjugateGradientPrecond	0	0	0	0	0	0	7	0
	cuSolverDn_LinearSolver	0	0	2	0	0	0	0	0
	cuSolverRf	0	0	1	0	0	0	0	0
	cuSolverSp_LinearSolver	0	0	1	0	0	0	0	0
	cuSolverSp_LowlevelCholesky	0	0	1	0	0	0	0	0
	cuSolverSp_LowlevelQR	0	0	1	0	0	0	0	0
	BlackScholes	0	0	0	0	0	0	1	0
	FDTD3d	0	0	0	0	0	0	1	0
	binomialOptions	0	0	0	0	0	0	1	0
ECP	Laghos	**1**	**1**	1	0	**1**	0	0	0
	Remhos	0	0	1	0	0	0	0	0
	Sw4lite (64)	**1**	**1**	1	0	0	0	0	0
	Sw4lite (32)	0	**1**	0	0	**1**	0	5	0
HPC-Benchmarks	HPCG	**1**	0	0	**1**	0	0	0	0
ML open issues	CuMF-Movielens	0	0	0	0	**29**	0	0	**2**
	SRU-Example	0	0	0	0	**3**	**1**	**2**	**1**
	cuML-HousePrice	**1**	**1**	0	0	**1**	0	0	0

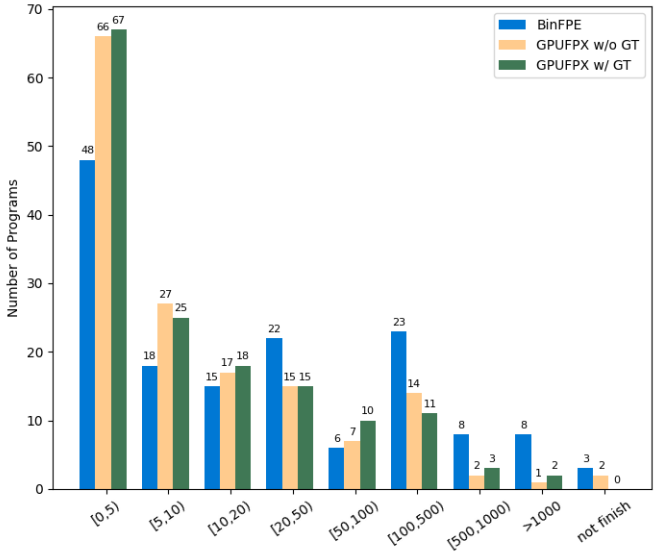

Figure 4: Slowdown distribution: BinFPE vs. GPU-FPX without global table implementation vs. GPU-FPX with global table implementation.

dot represents a program and indicates how one outperforms the other tool. Specifically, if one dot is above the main diagonal (blue), that means the GPU-FPX outperforms BinFPE and vice versa. This plot reveals that even without sampling, GPU-FPX is significantly faster than BinFPE due to the performance-enhancing approaches we have taken (§3.1). In particular, there are 49 programs where GPU-FPX is two orders of magnitude faster, and four programs which are three orders of magnitude faster.

4.3 Invocation Undersampling Study, Figure 6

It is efficient (§3.1) to reduce the frequency of instrumentation when the same kernel is repeatedly invoked. This can be achieved by setting the parameter FREQ-REDN-FACTOR to a higher value, such as 16 (causing Algorithm 3 to generate instrumented codes for kernels only once every 16 calls). As an example of the improvements we've achieved, consider the "CuMF-Movielens" program. By setting the FREQ-REDN-FACTOR to 256, we were able to evaluate this program in just 5 minutes, compared to 70 minutes without using our sampling technique. For reference, using BinFPE would have taken a staggering 6 hours to evaluate the same program. It is worth noting that this improvement comes without the loss of any previously detected exceptions.

Table 5 provides further analysis of how exception detection within programs containing many exceptions ("severe exceptions") changes with FREQ-REDN-FACTOR equal to 64. The table shows that, out of the 12 programs containing severe exceptions, three experienced a decrease in the number of severe exceptions detected.

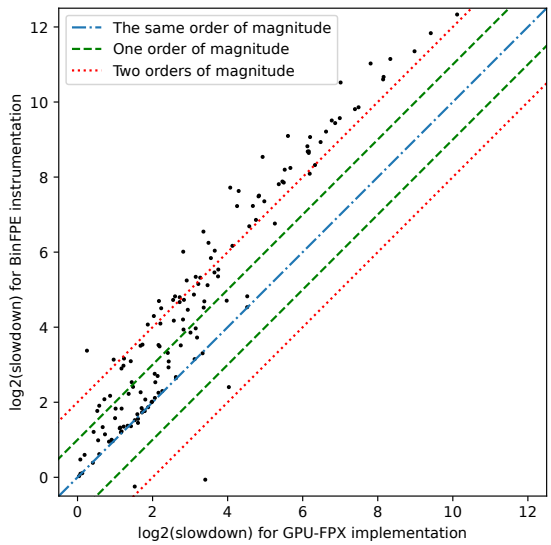

Figure 5: log(slowdown) for BinFPE and GPU-FPX. Except for a few programs, the dots are above the line, meaning that GPU-FPX provides a significant performance improvement over BinFPE (three orders of magnitude more).

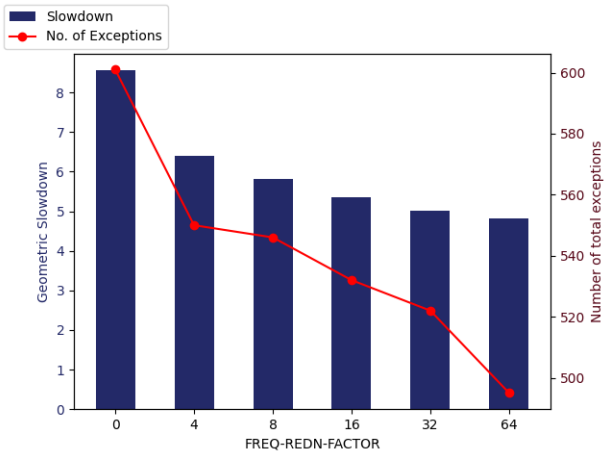

Figure 6: Impact of FREQ-REDN-FACTOR on perf. and exception detection. Blue bars represent geom. mean slowdown; Red line is total num. of exceptions.

Nevertheless, the number of programs with exceptions remains the same, ensuring that all programs can be diagnosed later if necessary.

4.4 Compiler Effect on Exceptions, Table 6

We now study how exceptions are affected by the `--use_fast_math` compiler flag—a first-time study of this type. According to NVIDIA's official documentation [23], the use of `--use_fast_math` performs the following numerical optimizations that affect exceptions:

(1) Flushes all single-precision denormals to zeros.
(2) Uses a faster (coarser) approximation for single-precision floating-point division, reciprocal, and square root.
(3) Enables the contraction of floating-point multiplies and adds/-subtracts into floating-point multiply-add operations.
(4) Some functions[10] are mapped to special function units (SFU).

As the caveat in their documentation *"In addition to reducing the accuracy of the affected functions, it may also cause some differences in special-case handling"*. We set out to better understand this, and evaluate benchmarks, presenting the impacted cases in Table 6.

We observe that in **GESUMMV, cfd, myocyte, S3D, stencil, wp,** and **rayTracing**, all subnormals just vanish, exactly as NVIDIA's documentation (item 1) mentions. In **myocyte**, six division-by-0 exceptions are raised immediately after eight disappearences of subnormal number exceptions under `--use-fast-math` optimization. GPU-FPX also helps identify the locations where such changes happened. The code fragment affected was contained within **myocyte** `kernel_ecc_3`. In particular, without the fast-math flag, we could detect a subnormal at `kernel_ecc_3.cu:776`. This subnormal disappears and a new INF is now raised at `kernel_ecc_3.cu:777` in `fast-math` mode.

While more analysis need to be done by the original authors of these programs, even from our limited studies, we can see the consequences of using the `fast-math` flag. Tools such as GPU-FPX can offer the required insights before programmers can feel confident about their use of the `-use_fast_math` flag, and rule out the danger of introducing cascading exceptions that may prove to be hard to diagnose.

To summarize: (1) GPU-FPX detected previously unknown exceptions in 23 programs from significant GPU benchmarks; (2) GPU-FPX outperforms BinFPE by 12x on average, and is also up to three orders of magnitude faster; (3) GPU-FPX helps explore compiler and architecture impact on exceptions. (4) GPU-FPX enables effective diagnosis and resolution of exceptions through its ability to locate and identify the source of the problem.

5 ANALYZING THE EXCEPTIONS WITH *ANALYZER*

Once exceptions are detected, there comes the question of diagnosing and repairing the exceptions. We use *analyzer* to help diagnose the programs that involve severe exceptions from Table 4. We summarize the results in Table 7. From our analysis, the intervention of experts is needed for myocyte, Laghos Sw4lite, and HPCG.

For other programs, each seems to merit its own treatment. We explore the diagnosis and repair methods for these programs, which also encompasses details about their distributions and source code availability. Additionally, we present two comprehensive case studies in §5.2 and §5.3.

5.1 Diagnosis and Repair Strategies

We now describe the kinds of capabilities that GPU-FPX provides by summarizing several exception-diagnosis studies we have conducted.[11]

[10] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#intrinsic-functions
[11] Space prevents fuller descriptions, but these examples will be included in the Docker image to be supplied with GPU-FPX.

Table 5: Exception detection decrease going from *full* GPU-FPX version to *64-invocations instrumentation*

	FP64				FP32			
Program	NAN	INF	SUB	DIV0	NAN	INF	SUB	DIV0
myocyte	**57 → 54**	**63 → 53**	2 → 0	3	**92 → 87**	53	8 → 1	0
Sw4lite (64)	**1→ 0**	1	1	0	0	0	0	0
Laghos	1	**1→ 0**	1	0	1	0	0	0

Table 6: Details of kernels with respect to optimization flags and exceptions detected

		FP64				FP32			
program	with fastmath	NAN	INF	SUB	DIV0	NAN	INF	SUB	DIV0
GRAMSCHM	✗	0	0	0	0	7	1	0	1
	✓	0	0	0	0	**5**	**0**	0	1
LU	✗	0	0	0	0	3	0	0	1
	✓	0	0	0	0	**1**	0	0	1
cfd	✗	0	0	0	0	0	0	13	0
	✓	0	0	0	0	0	0	**0**	0
myocyte	✗	57	63	2	3	92	76	8	0
	✓	57	63	4	3	**90**	**81**	**0**	**6**
S3D	✗	0	0	0	0	0	7	129	0
	✓	0	0	0	0	0	7	**0**	0
stencil	✗	0	0	0	0	0	0	2	0
	✓	0	0	0	0	0	0	**0**	0
wp	✗	0	0	0	0	0	0	47	0
	✓	0	0	0	0	0	0	**0**	0
rayTracing	✗	0	0	0	0	0	0	10	0
	✓	0	0	0	0	0	0	**0**	0

Table 7: Overview of Exception Diagnoses and Repairs using *Analyzer* for Programs with Severe Exceptions

Program	Diagnose?	Exceptions Matter?	Fixed?
GRAMSCHM	✓	✓	✓
LU	✓	✓	✓
myocyte	✗	N.A.	N.A.
S3D	✓	✗	N.A.
Interval	✓	✗	N.A.
Laghos	✗	N.A.	N.A.
Sw4lite	✗	N.A.	N.A.
HPCG	✗	N.A.	N.A.
CuMF-Movielens	✓	✓	✓
cuML-HousePrice	✓	✓	✓
SRU-Example	✓	✓	✓

GRAMSCHM and LU from polybenchGpu (sources available): Running GPU-FPX reveals an INF exception due to division by 0. This INF value is subject to a later FMA resulting in a NaN that flows to the output. The solution was to remove 0 values in the input. For the LU routine studied also the cause and solution were the same.

3D from shoc (sources available): The program has built-in checks for the INF exception (a robust code) and hence the user does not need to repair this code. GPU-FPX helps explain the inner reasons for this INF.

interval from CUDA Samples (sources available): The generated NaNs are handled by the code (no action needed). GPU-FPX helps explain the inner reasons for these NaNs.

HPCG from NVIDIA's HPC Benchmarks (no sources): GPU-FPX located where the NaNs were generated. We could observe that these NaNs were not used in subsequent calculations (yet we believe the code ought to have detected and reported the NaNs).

CuMF-Movielens from Git (Open Issues) with sources: We could locate the NaN to line 213 of file als.cu and discovered a code repair (setting alpha[0] to 0 when rsnew[0] is 0.

CuML-HousePrice from Git (Open Issues) with partial sources: We could locate the NaN exception source and conjecture a repair that requires interaction with the authors.

5.2 Case Study: CUDA GMRES Solver

Our collaborator was employing the Generalized minimal residual method (GMRES) algorithm to solve, using CUDA, a system of linear equations involving an indefinite nonsymmetric Matrix. When they encountered the issue of the residual always being a NaN right from the first iteration, they sought our help.

Using the *detector* of GPU-FPX, we identified a division by zero exception in the `cuSparse` kernel `csrsv2_solve_upper_nontrans_byLevel_kernel` as reported in Listing 3. This NaN seemed to be propagating to their customized kernel from the *detector*.

Upon reporting this finding, our collaborator suggested that this division by zero may have occurred during the LU step, since they were using a nearly singular matrix.

```
#GPU-FPX LOC-EXCEP INFO: in kernel [void csrsv2_solve_upper_nontrans_byLevel_kernel], DIV0 found @ /unknown_path:0 [FP64]

#GPU-FPX LOC-EXCEP INFO: in kernel [void csrsv2_solve_upper_nontrans_byLevel_kernel], NaN found @ /unknown_path:0 [FP64]

#GPU-FPX LOC-EXCEP INFO: in kernel [void csrsv2_solve_upper_nontrans_byLevel_kernel], NaN found @ /unknown_path:0 [FP32]

Running #GPU-FPX: kernel [MassIPTwoVec] ...

#GPU-FPX LOC-EXCEP INFO: in kernel [MassIPTwoVec], NaN found @ /home/xinyi/NaNexample/customKernels.cu:31 [FP64]
...
```

Listing 3: The exception report generated by GPU-FPX captures NaNs propagating to the custom kernel

```


#GPU-FPX-ANA SHARED REGISTER: Before executing the instruction @ /unknown_path in [void cusparse::load_balancing_kernel]:0 Instruction: FSEL R2, R5, R2, !P6 ; We have 3 registers in total. Register 0 is VAL. Register 1 is NaN. Register 2 is VAL.

#GPU-FPX-ANA SHARED REGISTER: After executing the instruction @ /unknown_path in [void cusparse::load_balancing_kernel]:0 Instruction: FSEL R2, R5, R2, !P6 ; We have 3 registers in total. Register 0 is VAL. Register 1 is NaN. Register 2 is VAL.
```

Listing 4: Application of the *analyzer* to the boosted version. This listing displays the last two pieces of exceptional information before the customized kernel is called.

They resolved this issue by *boosting* the matrix diagonal using an API function already provided in cuSparse (boosting elevates values smaller than a threshold to a larger number). This successfully eliminated the NaNs in the output.

Subsequent checking using GPU-FPX reveals that a division by zero *still exists* in `csrsv2_solve_upper_nontrans_byLevel_kernel`. Using the analyzer on both the original and fixed versions, we observed that in the boosted version, the NaN stops propagating at the FSEL instruction (Listing 4), meaning it is not selected. In contrast, in the original version, the NaN is selected and then passed to a DADD operation (Listing 5). We inferred that this might be a guard in this kernel, but our collaborator still expresses concerns about the division by zero. *Since cuSparse is closed source*, further investigation into this issue requires help from its original developers (such a dialog is underway).

5.3 Case Study: SRU-Example

In order to further test the capabilities of GPU-FPX in resolving open Github issues related to NaN values appearing in GPU computations, we found a promising study subject, namely an issue reported in the Simple Recurrent Unit (SRU) project[12], [21]. This

[12]https://github.com/asappresearch/sru

```
#GPU-FPX-ANA SHARED REGISTER: Before executing the instruction @ /unknown_path in [void cusparse::load_balancing_kernel]:0 Instruction: FSEL R20, R31, R20, !P1 ; We have 3 registers in total. Register 0 is NaN. Register 1 is VAL. Register 2 is NaN.

#GPU-FPX-ANA SHARED REGISTER: After executing the instruction @ /unknown_path in [void cusparse::load_balancing_kernel]:0 Instruction: FSEL R22, R22, R30, P0 ; We have 3 registers in total. Register 0 is NaN. Register 1 is NaN. Register 2 is NaN.

#GPU-FPX-ANA SHARED REGISTER: Before executing the instruction @ /unknown_path in [void cusparse::load_balancing_kernel]:0 Instruction: DADD R8, R8, R22 ; We have 3 registers in total. Register 0 is NaN. Register 1 is NaN. Register 2 is VAL.

#GPU-FPX-ANA SHARED REGISTER: After executing the instruction @ /unknown_path in [void cusparse::load_balancing_kernel]:0 Instruction: DADD R8, R8, R22 ; We have 3 registers in total. Register 0 is NaN. Register 1 is NaN. Register 2 is VAL.
```

Listing 5: Application of the *analyzer* to the original version. This listing displays the last four pieces of exceptional information before the call to the customized kernel.

```
#GPU-FPX LOC-EXCEP INFO: in kernel [mpere_sgemm_32x128_nn], NaN found @ /unknown_path in [ampere_sgemm_32x128_nn]:0 [FP32]

Running #GPU-FPX: kernel [void (anonymous namespace)::sru_cuda\_forward\_kernel\_simple] ...
#GPU-FPX LOC-EXCEP INFO: in kernel [void (anonymous namespace)::sru_cuda_forward_kernel_simple], NaN found @ /unknown_path in [void (anonymous namespace)::sru_cuda_forward_kernel_simple]:0 [FP32]
...
```

Listing 6: Application of the *detector* to SRU example code. We listed the first detected NaN in the report.

project is realized using Python/Pytorch, with details unknown to us. We could successfully reproduce the NaN issue reported by a user, with the NaN appearing at the output of an example code. As detailed in Listing 6, NaN values were detected in the `ampere_sgemm_32x128_nn` kernel by the detector of GPU-FPX.

Due to the unavailability of the source code, all we could go by was information on exception flows. The use of GPU-FPX revealed that NaN values were propagated from the source register as shown in Listing 7.

This led us to investigate the input data, and we found that the original input was generated using the `torch.FloatTensor(20, 32, 128).cuda()` function, which creates a tensor with uninitialized data on GPU memory. We resolved the issue by changing the input generator to torch.randn(20,32,128).cuda(), which eliminated the NaN values in the output.

```
#GPU-FPX-ANA SHARED REGISTER: Before executing the instruction @ /unknown_path in [mpere_sgemm_32x128_nn]:0 Instruction: FFMA R1, R88.reuse, R104.reuse, R1 ; We have 4 registers in total. Register 0 is VAL. Register 1 is VAL. Register 2 is NaN. Register 3 is VAL.

#GPU-FPX-ANA SHARED REGISTER: After executing the instruction @ /unknown_path in [mpere_sgemm_32x128_nn]:0 Instruction: FFMA R1, R88.reuse, R104.reuse, R1 ; We have 4 registers in total. Register 0 is NaN. Register 1 is VAL. Register 2 is NaN. Register 3 is NaN.
```

Listing 7: Application of *analyzer* to SRU example code. This clarify how the first NaN appears; it is from the source register.

The key takeaway is that GPU-FPX is the only tool that brings a designer to a point where they may be able to take these repair steps even when sources are unavailable. It is also clear that the original developers would have been significantly enabled by GPU-FPX's availability to cover all the untested and/or unreported cases of exceptions.

6 CONCLUDING REMARKS

In this work, we introduce the importance of tracking floating-point exceptions in GPU codes. Some of these exceptions become impossible to ignore, as they stop a programmer from making progress, say by providing a solution array containing NaNs. In other cases, such NaNs do not appear in the program output, (mis-)leading the programmer into believing that the results are reliable. Unfortunately, as discussed through a number of case studies, we demonstrate the fallacy of this assumption. We provide the first (and only) practical tool in this space—GPU-FPX—that can help GPU programmers make progress.

While we have shown many successes in using GPU-FPX on codes that are unfamiliar to us, we have barely scratched the surface of what is needed. GPU and other accelerators, along with special-purpose hardware such as special function units and Tensor Cores will form the bedrock of future high-performance computing. In all these platforms, the danger of exceeding the number representation ranges lurks. GPU-FPX is just the beginning, as clear from the survey of future problems to expect in this area [11]. The many struggles while training neural networks [2, 3] vividly capture some of the future tooling needs.

Other Related Work: Tools for checking exceptions in CPU codes have taken significantly different routes than GPU-FPX. FPSpy [10] focuses on x86 binaries, offering the advantage of relying on OS-level mechanisms. A new version of FPChecker [20] considers CPU OpenMP and MPI codes, again using LLVM instrumentation. The ideas in these efforts do not directly help develop efficient GPU-based exception checkering tools. It also appears that the use of OS-level mechanisms and LLVM instrumentation do not face the same issues that we had to face and overcome (absence of hardware traps, JIT-ting overhead, etc.).

Rigorous floating-point error analysis methods as well as exception checking methods can work hand-in-hand, in that the former can inform where exceptions might arise. Space does not permit a detailed survey; a few key tools in rogorous error analysis are [6–8, 24, 25].

Future Directions: A number of future directions of work are planned to be pursued based on GPU-FPX, as well as to enhance its capabilities.

Expanding the set of inputs on which a GPU program is run is an important future need. Very likely, this requires designer-input, as they alone typically know the meaningful input ranges of their codes. For GPU libraries, these inputs are implied to be large, and often documented. One recent work has exploited available knowledge and sought to expand input ranges through Bayesian Optimization [18]. This effort has been applied to GPU codes, and shows that stress-testing is successful in revealing even more exceptions than previously known.

One important aspects of the work in [18] is that only the output of the GPU function being tested was observed. As we have shown, even when the output does not reveal exceptions, one must "look inside the kernels" using tools such as GPU-FPX. This would indeed be an exciting symbiotic direction to pursue wherein library developers both stress-test their new libraries, identify problems well before a customer faces them, and solidify as well as document their code before release.

Such strengthening of GPU code-reliability is much-needed, as evidenced by the numerous exception-related open-issues that the community is struggling with. To help initiate this progress, we will release GPU-FPX and our benchmarks in an easy-to-use manner, making sure that all these programs easily run on today's GPU platforms.

ACKNOWLEDGMENTS

We thank the shepherd for their valuable feedback. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, ComPort: Rigorous Testing Methods to Safeguard Software Porting, under Award Number 78284. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL01830. The work at Lawrence Livermore National Laboratory was supported under Contract DE-AC52-07NA27344 (LLNL-CONF-833485). It is also based on NSF CISE Awards 2217154, 2124100 and 1956106, and DOE DE-SC0022252.

REFERENCES

[1] 2022. CUDA C++ Programming Guide, v12. https://docs.nvidia.com/cuda/floating-point/index.html. Online; accessed March, 30, 2022.
[2] 2022. NVIDIA Deep Learning Performance. https://docs.nvidia.com/deeplearning/performance/. Online; accessed March, 30, 2022.
[3] Syed Ahmed, Christian Sarofeen, Mike Ruberry, Eddie Yan, Natalia Gimelshein, Michael Carilli, Szymon Migacz, Piotr Bialecki, Paulius Micikevicius, Dusan Stosic, Dong Yang, and Naoya Maruyama. 2022. https://pytorch.org/blog/what-every-user-should-know-about-mixed-precision-training-in-pytorch/.
[4] AMD. 2015. FLOATING-POINT ARITHMETIC IN AMD PROCESSORS. https://community.amd.com/t5/opencl/amd-gpus-ieee-754-compliance/td-p/98382. Accessed: 2023-04-10.
[5] NVIDIA Corporation. 2021. NVIDIA AMPERE GA102 GPU ARCHITECTURE. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf
[6] Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. 2020. Scalable yet Rigorous Floating-Point Error Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 51, 14 pages.

[7] Marc Daumas and Guillaume Melquiond. 2010. Certification of Bounds on Expressions Involving Rounded Operators. *ACM Trans. Math. Software* 37, 1, Article 2 (2010), 20 pages.

[8] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. 2009. Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. In *Formal Methods for Industrial Critical Systems, FMICS 2009*. Lecture Notes in Computer Science, Vol. 5825. Springer Berlin Heidelberg, 53–69. https://doi.org/10.1007/978-3-642-04570-7_6

[9] James Demmel, Jack Dongarra, Mark Gates, Greg Henry, Julien Langou, Xiaoye Li, Piotr Luszczek, Weslley Pereira, Jason Riedy, and Cindy Rubio-González. 2022. Proposed Consistent Exception Handling for the BLAS and LAPACK. *arXiv preprint arXiv:2207.09281* (2022).

[10] Peter Dinda, Alex Bernat, and Conor Hetland. 2020. 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*. 5–16.

[11] Ganesh Gopalakrishnan, Ignacio Laguna, Ang Li, Pavel Panchekha, Cindy Rubio-González, and Zachary Tatlock. 2021. Guarding Numerics Amidst Rising Heterogeneity. In *Correctness 2021: Fifth International Workshop on Software Correctness for HPC Applications*. https://correctness-workshop.github.io/2021/.

[12] IEEE 754 Working Group et al. 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std* (2019), 754–2008.

[13] Ari B. Hayes, Fei Hua, Jin Huang, Yanhao Chen, and Eddy Z. Zhang. 2019. Decoding CUDA Binary. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 229–241. https://doi.org/10.1109/CGO.2019.8661186

[14] David G. Hough. 2019. The IEEE Standard 754: One for the History Books. *Computer* 52, 12 (2019), 109–112. https://doi.org/10.1109/MC.2019.2926614

[15] 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935

[16] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. https://doi.org/10.48550/ARXIV.1903.07486

[17] Ignacio Laguna. 2019. FPChecker: Detecting Floating-Point Exceptions in GPU Applications. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1126–1129. https://doi.org/10.1109/ASE.2019.00118

[18] Ignacio Laguna and Ganesh Gopalakrishnan. 2022. Finding Inputs that Trigger Floating-Point Exceptions in GPUs via Bayesian Optimization. In *Supercomputing*.

[19] Ignacio Laguna, Xinyi Li, and Ganesh Gopalakrishnan. 2022. BinFPE: Accurate Floating-Point Exception Detection for GPU Applications. In *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (San Diego, CA, USA) *(SOAP 2022)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3520313.3534655

[20] Ignacio Laguna, Tanmay Tirpankar, Xinyi Li, and Ganesh Gopalakrishnan. 2022. FPChecker: Floating-Point Exception Detection Tool and Benchmark for Parallel and Distributed HPC. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 39–50. https://doi.org/10.1109/IISWC55918.2022.00014

[21] Tao Lei, Yu Zhang, Sida I. Wang, Hui Dai, and Yoav Artzi. 2018. Simple Recurrent Units for Highly Parallelizable Recurrence. In *Empirical Methods in Natural Language Processing (EMNLP)*.

[22] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. 2016. SFU-Driven Transparent Approximation Acceleration on GPUs. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 15, 14 pages. https://doi.org/10.1145/2925426.2926255

[23] NVIDIA. 2022. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. Online; accessed March, 30, 2022.

[24] Alexey Solovyev. 2017. *TOPLAS FPTaylor Results Table*. Retrieved October 10, 2017 from http://tinyurl.com/TOPLAS-FPTaylor-Results-Table

[25] Laura Titolo, Marco A. Feliú, Mariano Moscato, and César A. Muñoz. 2017. An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. In *Lecture Notes in Computer Science*. Springer International Publishing, 516–537. https://doi.org/10.1007/978-3-319-73721-8_24

[26] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 372–383.

[27] Nathan Whitehead and Alex Fit-florea. 2022. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. https://docs.nvidia.com/cuda/floating-point/index.html