Array Programming on GPUs: Challenges and Opportunities

Xinyi Li
xin_yi.li@utah.edu
University of Utah, Kahlert School of Computing
Salt Lake City, USA

Mark Baranowski
baranows@cs.utah.edu
University of Utah, Kahlert School of Computing
Salt Lake City, USA

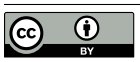
Harvey Dam
harvey.dam@utah.edu
University of Utah, Kahlert School of Computing
Salt Lake City, USA

Ganesh Gopalakrishnan
ganesh@cs.utah.edu
University of Utah, Kahlert School of Computing
Salt Lake City, USA

Abstract

Today, the lion's share of machine learning and high-performance computing workloads is executed on GPUs, including high-stakes applications such as self-driving cars and fusion reactor simulations. Unfortunately, GPU computations are carried out on largely undocumented hardware units that cannot trap or report floating-point exceptions. Worsening the situation is an ongoing and accelerating shift toward lower-precision arithmetic, driven by performance demands—yet this shift only exacerbates the frequency and severity of floating-point exceptions. Increasingly, matrix multiplications are offloaded to specialized hardware such as Tensor Cores. However, because these units do not adhere to a unified arithmetic standard, their computed results can deviate to unacceptable levels.

This experience report aims to consolidate our previously published work and relate it to array programming in two key ways: (1) by providing tools to diagnose bugs that may arise during array computations, and (2) by addressing broader correctness challenges inherent to array-based programming. This report highlights GPU-FPX, a debugging tool extended to analyze computations involving Tensor Cores. It addresses key correctness challenges, such as the potential for different Tensor Core implementations to produce inconsistent results for the same input. These discrepancies can be systematically uncovered using a targeted testing approach known as FTTN. We conclude with a discussion on how formal methods, particularly those based on SMT solvers, can play a critical role in identifying and bridging gaps in manufacturer-provided hardware specifications—and, in the long term, in proving desired correctness properties.



ARRAY '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1927-1/25/06
https://doi.org/10.1145/3736112.3736144

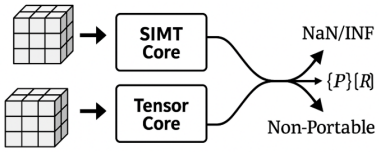
CCS Concepts: • **General and reference** → **Verification**; *Reliability*; • **Software and its engineering** → **Software notations and tools**; • **Mathematics of computing** → *Mathematical software.*

Keywords: GPU Program Debugging, Tensor Cores, Numerical Exceptions, Reproducibility, Binary Instrumentation, Formal Methods

ACM Reference Format:
Xinyi Li, Mark Baranowski, Harvey Dam, and Ganesh Gopalakrishnan. 2025. Array Programming on GPUs: Challenges and Opportunities. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '25), June 17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3736112.3736144

1 Introduction

Now that GPUs have taken a toll on everyone's purse, one must ask: how much longer can the programming community afford to ignore the threats they pose to program correctness? The following cartoon captures the gist of this article:



GPUs support array programming not only through the familiar single-instruction multiple-thread (SIMT) cores, but also increasingly via Tensor Cores, which perform matrix multiply-and-accumulate (MMA) instructions of the form $\mathbf{D} = \mathbf{A} \cdot \mathbf{B} + \mathbf{C}$ where all the matrices involved are "small" (e.g., 4×4). Tensor Cores serve as dedicated inner accelerators within GPUs, and their growing silicon footprint reflects their increasing importance in modern GPU architectures. The results computed by a GPU may be rendered unreliable or unverifiable in three key ways: (1) they may contain, or be affected by, floating-point exceptions such as NaNs and

infinities; (2) the behavior of the hardware (denoted by the pre/post-condition pairs {P} and {R}) may not be specified by GPU manufacturers; and/or (3) the results may change when the same code is ported across different GPU architectures.

The Correctness Topics we Cover: GPUs are highly parallel computing systems in support of machine learning, high-performance computing and a plethora of other types of code. It is safe to say that at present a majority of GPU codes deal with parallel *floating-point* computations. This means that both parallelism bugs and floating-point bugs must be eliminated. *Floating-point bugs are the sole focus here.* The floating-point bug type that has traditionally been discussed the most is that of rounding. Here, we only focus on *how rounding changes* across GPU platforms. Clearly, floating-point exceptions do arise, and must be addressed—either deemed to be innocuous or suitably eliminated—*a prominent focus in this paper.* Given that manufacturers often do not provide all pertinent details [11, 13, 24], the community is often forced to discover missing behaviors through testing—*again our focus.*

Floating-point exceptions are often safe to ignore, but often are as egregious as null-pointer dereference errors that are taken much more seriously. Unfortunately, the programming community has not researched this topic sufficiently—especially in the light of rising heterogeneity and reducing floating-point precision—situations that easily lead to "infinity" (`INF`) exceptions and subsequently (upon further computation) to `NaN` ("not a number") exceptions. Many case studies involving exceptions are presented in this paper. The elimination of harmful exceptions is an involved topic that we have barely scratched. Given the increasing prominence of Tensor Cores, we must also detect exceptions generated by them. We have recently extended GPU-FPX to include this capability and these are the new results reported in this paper. We now set the tone for the rest of this paper through three debugging stories.

Story 1: Difficulty of Locating and Eliminating Floating-Point Exceptions: In a GitHub issue report, the developer of the "SRU" codebase (Simple Recurrent Units for Highly Parallelizable Recurrence [22]) describes their inability to eliminate persistent NaNs that impeded their work [17, 38]. The SRU is designed to retain the modeling power of recurrent networks while being much faster and more parallelizable. Unable to debug the issue, the user posted a GitHub issue, which remains open. We developed a solution and have posted a fix—essentially pointing out that the root cause of the bug was "hiding in plain sight" (detailed in our Github issue fix reported at [38]): in essence, the user allocates a Tensor and falsely assumes that it has been initialized with zeros.

Our Contribution: We employed our unique tool called GPU-FPX [23] which is based on *binary instrumentation* to debug this problem. While programming languages researchers often champion the virtues of abstraction and encapsulation, GPUs can quickly undermine these ideals. GPU library components are frequently closed-source (available only in binary form) and can generate unhandled floating-point exceptions from their innards. In our experience, binary instrumentation offers the necessary "X-ray vision" for uncovering subtle numerical issues. Analyzing GPU array programs at the source-code level (e.g., CUDA C) or even at intermediate-code levels (e.g., PTX or LLVM) often fails to expose these problems. This is largely because compiler optimizations can significantly transform the generated code—sometimes even silently altering the intended numerical precision.

As examples, computations originally written in 64-bit floating-point (FP64) may be downgraded to 32-bit (FP32) precision [23]. Also some operations such as division are not supported in hardware. Consider a source-level expression like a/b. During compilation, this is typically transformed into $a \cdot (1/b)$, where the multiplier is realized by `MUFU.RCP`, a fast reciprocal approximation in hardware. The rest of the computation is typically realized using fused multiply-add (`FFMA`) instructions. Whether this transformation behaves correctly depends on the GPU's support for subnormal numbers. If b is a subnormal value, $(1/b)$ may either be a very large, representable number; or, if b is flushed to zero, then the result can be the INF (infinity) value.

For reasons not yet fully understood, many working in machine learning or low-precision numerical software seem plagued by NaNs and other exceptions—with little external support or guidance in sight. If the truth of this claim seems doubtful, we invite you to search for "pytorch nan loss autocast" (or a similar phrase) on the internet and browse the thousands of results—each a testament to how widespread and unresolved the issue remains. It is high time these users received meaningful support from the programming community—through robust debugging tools today, and proactive defect-prevention measures in the future.

GPU-FPX: how it helps, and is being improved: We developed GPU-FPX based on substantial reverse-engineering of SASS (short for streaming assembly) instruction semantics—an effort made necessary by the absence of manufacturer-provided specifications. At present, GPU-FPX offers programmers crucial insights that can often help rescue stalled machine learning (ML) training runs. It can also assist in diagnosing and repairing numerical algorithm loops that may misbehave—such as skipping conditional statements whose predicates are affected by NaNs [7]. Previously [23], we had analyzed over 150 benchmarks and applications using GPU-FPX, uncovering INF and NaN exceptions in 36 of them (some were deemed serious upon later analysis). Interestingly, these exceptions arose while running the applications on their own distributed datasets—without even straying beyond the intended input ranges. (Exceptions that

are easily triggered by deviating from these inputs are of no interest.)

In this experience report, we present additional perspectives. First, we describe how GPU-FPX was used to debug two lossy data compressors. Second, we detail new extensions to GPU-FPX that enable support for GPU Tensor Cores [13, 24]. Supporting Tensor Cores is essential for analyzing exceptions in highly optimized GPU programs, as their adoption is on the ascent.

Open Challenges: Much room for improvement beyond GPU-FPX remains. Open questions include how to move more analyses higher up the programming abstraction stack and whether certain programs can ever be proven exception—free, assuming that one can create reliable formal models for GPU instructions. This is because even for decidable programs, proving exception-freedom entails the daunting (and impractical) complexity of proving that all program variables remain within specific value-ranges.

Story 2: We designed an experiment in which a user performs the calculation $\mathbf{D} = \mathbf{A} \cdot \mathbf{B} + \mathbf{C}$ where **A** and **B** are 8K-sized FP16 floating-point matrices, and **C** and **D** are of the same size, but have FP32 precision. We filled the right-hand side matrices with specific entries that were picked to reveal computational platform differences (e.g., rounding modes, precision used in summation, etc.) that one can reasonably expect. The variety of **D** matrices we obtain is startling: a matrix filled with all zeros on NVIDIA A100, V100, AMD MI250, and a conventional CPU realizing the IEEE round-to-nearest specification; a matrix filled entirely with 255.875 on the AMD MI100; and one filled with 191.875 on the NVIDIA H100. We summarize the contents of these matrices later in this report. Such large differences are not caused by the non-associativity of floating-point addition—a very natural conclusion one might make—but actually due to differences in terms of how multi-term addition is implemented within different Tensor Cores, as we explain later.

One Current Solution to Specification Discovery: We have released a tool called Feature-Targeted Testing of Numerics (FTTN) [24, 25] to discover specific differences in GPU arithmetic realizations. FTTN works by executing targeted tests and analyzing their outputs to infer the specific hardware features a GPU must possess to produce the observed results. Similar test-driven methodologies are gaining traction more broadly; for example, the framework presented by Xie et al. [41] reveals addition ordering differences through carefully crafted tests.

A fundamentally more principled approach to testing can be developed using SMT solvers. Preliminary work in this direction is reported in [39], where Tensor Core-based matrix multiplication schemes are formally analyzed. We believe that beginning with an FTTN-like approach and generalizing the initial tests using SMT would be a fruitful direction.

Open Challenges: The fact that GPUs and their accelerators are the only economically viable choice for future HPC is forcefully argued by Reed et al. [36]. Thus, it is important that manufacturers be incentivized to help GPU programmers in crafting a sufficiently comprehensive set of tests that match silicon-level behaviors. While one might be able to compensate for the lack of conformance of GPUs with standard arithmetic practices, one should be mindful not to stray into "irresponsibly reckless" territory (a phrase borrowed from the work reported in [26] where the authors use it to connote an excessive approximation of a matrix).

Story 3: Even with the best of coding practices (e.g., backed by good documentation and clear parameterization, as, say, in [28]) there are a large number of error-prone details that must be handled carefully in order to program Tensor Cores. To minimize this tedium, one might consider the use of AI to generate or augment Tensor Core code. We recently conducted an experiment in which we prompted ChatGPT to extend a Tensor Core based matrix multiplication routine to compute the maximum element of the result matrix. We instructed ChatGPT to use CUDA warp-level primitives that are known to be efficient. While the generated code appeared entirely correct—and was even accompanied by clear and elegant documentation, such code cannot be trusted at face value (e.g., considering that AI agents are known to hallucinate).

Open Challenges: In order to move toward formally verified AI-generated code in this space, one must not only have formal specifications of GPU instructions, but also develop *automated* formal verification techniques that scale.

Roadmap: We begin by outlining relevant background in §2, followed by three case studies that illustrate the severity of floating-point exceptions on GPUs (§3). In §4, we describe our ongoing work to extend GPU-FPX with support for exception tracking across Tensor Core instructions. Section §5 presents our approach to specification discovery via Feature-Targeted Testing of Numerics (FTTN). We then highlight two representative challenges in §6 that emphasize the need for rigorous reasoning frameworks. Finally, in §7, based on our experience, we draw conclusions pertaining to safeguarding array programming.

2 Background

We assume the reader is familiar with GPUs and floating-point arithmetic at a high level. This background section aims to place several relevant concepts in context and introduce the tools that will be discussed throughout the paper.

NVIDIA GPUs remain the dominant commercial platform for high-performance computing and machine learning. However, recent efforts have extended floating-point exception checking capabilities to AMD GPUs as well. FloatGuard [29], a newly developed tool for AMD hardware, introduces several innovative techniques. It leverages the partial

exception information exposed by AMD GPUs—an ability not currently offered by NVIDIA—and extends it to reconstruct a more complete picture of floating-point exception behavior. Awareness of AMD GPUs is growing, and their LLVM-based tooling has made certain tasks more accessible. We hope that future work on safe array programming, currently focused on NVIDIA GPUs, will be replicated and extended to benefit the AMD ecosystem.

On most CPUs, floating-point exceptions can generate software traps. Hauser [15] was one of the early researchers who characterized exceptions and their use to support speculative execution strategies. FPSpy [9] is a CPU-based exception detection tool that uses software traps to record exceptions. Unfortunately, techniques that work in "CPU-land" do not translate well to GPUs, where exceptions do not trigger traps and must instead be detected by decoding the exponent of the generated floating-point results.

For NVIDIA GPUs, high-level code is compiled down to PTX (Parallel Thread Execution), which is then lowered to SASS (Streaming Assembler), the underlying assembly instruction format. SASS is only sporadically documented, and NVIDIA provides no formal guarantees regarding its correctness or long-term stability. In GPU-FPX, we use NVBit [40], a binary instrumentation framework made publicly available by NVIDIA. Much of our tooling builds on and extends this infrastructure.

To illustrate the subtleties involved in floating-point exception handling, consider the following C macro that implements the MAX function [5]:

```
#define MAX(x, y) ((x) >= (y) ? (x) : (y)).
```

At first glance, this macro appears correct—it simply returns the greater of two values. However, it contains a subtle flaw. Suppose x is set to NaN (e.g., by a previous statement such as x = $\sqrt{-42}$). Since any comparison involving NaN returns false, the macro will return y, potentially a valid value such as 3.14, effectively hiding the NaN from the user. Now consider another user who calls MAX(y, x) instead; the result will be x, the NaN—revealing the non-commutative nature of this macro when NaNs are involved. Demmel [7] documents many similar examples and even cites real-world accidents resulting from such behaviors. NVIDIA's advise on NaNs [32], requiring users to detect them in software (specifically, by explicitly checking the exponent field of a result to be "all 1's").[1] In this sense, the exceptional value for INF (infinity) is similar to NaNs but with the mantissa being all 0's. Likewise, the creation of subnormals (SUB) is an exception, with the exceptional value consisting of an exponent field of all zeros and the mantissa field being non-zero.[2]

[1] In this context, the distinction between exceptions and exceptional values is handy: the former connotes an event that could have been trapped on a CPU, while the latter connotes the encoding of the result of the exception in the floating-point result register (e.g., how we described the NaN encoding, above).

[2] A handy website for learning about floating-point (FP32) encodings is [18].

Finally, the importance of formal methods in GPU programming—especially for scientific computing and machine learning—has been highlighted in two U.S. Department of Energy reports [5, 12], the latter produced in collaboration with the NSF. Our work in [39] represents an early step in this direction.

3 Exception Root-Causing and Mitigation

We now describe three case studies carried out using GPU-FPX.

3.1 Debugging an Issue in the SRU Unit

Salient excerpts from the SRU example, as listed in the GitHub page [38], are reproduced below:

```
import torch
from sru import SRU, SRUCell

# input has length 20, batch size 32 and dimension 128
x = torch.FloatTensor(20, 32, 128).cuda()

input_size, hidden_size = 128, 128

rnn = SRU(input_size, hidden_size,
    num_layers = 2,          # number of stacking RNN layers
    dropout = 0.0,           # dropout applied between RNN layers
    bidirectional = False,   # bidirectional RNN
    layer_norm = False,      # apply layer normalization on the
                             # output of each layer
    highway_bias = -2,       # initial bias of highway gate (<= 0)
)
rnn.cuda()

output_states, c_states = rnn(x)      # forward pass
```

We executed the SRU code using the following commands, first without GPU-FPX instrumentation, and then with GPU-FPX's exception detector enabled. Enabling observation incurs a noticeable increase in runtime (a geometric-mean slowdown of 30× is reported in [23]).

```
time python run_sru.py
(no exceptions seen as a printout)
real    0m3.141s, user    0m1.255s, sys     0m3.229s

time LD_PRELOAD=./detector.so python run_sru.py
--- NVBit (NVidia Binary Instrumentation Tool v1.7.2) Loaded ---
...
Running #GPU-FPX:

kernel
[void at::
native::vectorized_elementwise_kernel] ...

Running #GPU-FPX: kernel [ampere_sgemm_32x128_nn] ...
#GPU-FPX
LOC-EXCEP INFO: Warning: in kernel [ampere_sgemm_32x128_nn], ..
(SUB) found @ /unknown_path in [ampere_sgemm_32x128_nn]:0 [FP32]

#GPU-FPX
LOC-EXCEP INFO: in kernel [ampere_sgemm_32x128_nn], NaN found
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 in ...

[void (anonymous namespace)::sru_cuda_forward_kernel_simple]:0 [FP32]

...
```

```
------------ GPU-FPX Report -----------

--- FP16 Operations ---
Total NaN found:              0
Total INF found:              0
Total underflow (subnormal):  0
Total Division by 0:          0
--- FP32 Operations ---
Total NaN found:              2
Total INF found:              1
Total underflow (subnormal):  2
Total Division by 0:          1
--- FP64 Operations ---
Total NaN found:              0
Total INF found:              0
Total underflow (subnormal):  0
Total Division by 0:          0
--- Other Stats ---
Kernels:      4
The total number of exceptions are: 128

real    0m24.033s, user    0m25.253s, sys     0m4.426s
```

The key lines to observe include the invocation of GPU-FPX via the `LD_PRELOAD` mechanism, which loads its dynamic library and activates the detector component, as well as the output lines where a `SUB` exception is flagged, followed by the appearance of a `NaN`. While the detector executes quickly and flags the presence of exceptions, it provides limited insight into their origin and propagation. To gain deeper understanding, the user is advised to run the GPU-FPX analyzer component:[3]

```
LD_PRELOAD=./analyzer.so python run_sru.py

#GPU-FPX: Instrument all kernels.
Running #GPU-FPX: kernel [...::native::vectorized_elementwise_kernel]
Running #GPU-FPX: kernel [ampere_sgemm_32x128_nn] ...
#GPU-FPX-ANA SHARED REGISTER:
Before executing the instruction  @ /unknown_path in
[ampere_sgemm_32x128_nn]:0
Instruction: FFMA R15, R35.reuse, R42.reuse, R15 ;
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  @ ...
...
```

The key observation is that the NaN value was incoming into the code proper (see the arrow above; some guesswork is needed in surmising this). We then suspected that the issue stemmed from input, which is the following line:

```
x = torch.FloatTensor(20, 32, 128).cuda()
```

At this point, we realized that `torch.FloatTensor(...)` does not fill the tensor with any defined patterns (it carries forward whatever was in that memory region). Replacing that line with this line that initializes the memory (in this case by random sampling from a normal distribution with mean 0 and standard deviation 1), the exception disappears (we do not show the exception-free run, to conserve space):

```
x = torch.randn(20, 32, 128).cuda()
```

[3]The output from the `time` command is omitted here, as the analyzer run is aborted early. Note that the analyzer is significantly slower, as it performs both intra- and inter-instruction exception flow analysis. Strategies for reducing runtime, including sampling, are discussed in [23].

In conclusion, while GPU-FPX is effective at flagging floating-point exceptions, it remains the developer's responsibility to determine whether these exceptions impact the correctness of their computation. As discussed in detail in [23], many such instances arise in practice. In one notable case involving a linear solver, GPU-FPX helped localize the exception specific GPU kernels. Through collaboration with a numerical expert, we identified a practical workaround: boost the diagonal entries of the matrix A (involved in solving the linear system $Ax = b$), which mitigated the instability and allowed the linear solver to finish. In the realm of HPC debugging, such pragmatic approaches often prove sufficient, particularly when the experimental scaffolding is intended as a single-use or temporary measure.

3.2 Root-causing NaNs in a simple GPU-based Lossy Data Compressor

A toy version of a data compressor, adapted from `cuszp`[16] (see our artifact[2] for details), initially produced no floating-point exceptions when run on a data array initialized in a simplistic manner. However, since this toy compressor did not declare any input constraints, we constructed an input array spanning the entire FP32 range, which led to the occurrence of floating-point exceptions. Fortunately, the CUDA binary for this compressor was compiled with source-level line information, allowing us to easily trace the offending instruction to line 120:

```
/home/ganesh/repos/BreakingCompressors/main1.cu:120
 120     const float recipPrecision = 0.5f / eb;

Instruction: MUFU.RCP R4, c[0x0][0x188] ;
We have 2 registers in total.
Register 0 is VAL.
Register 1 is INF.

#GPU-FPX-ANA APPEAR :
NaN appears at the destination  @
/home/ganesh/repos/BreakingCompressors/main1.cu:120
Instruction: FFMA R3, R4, -R0, 1 ;
We have 3 registers in total.
Register 0 is NaN.
Register 1 is VAL.
Register 2 is INF.
...
```

In this line, the variable `recipPrecision` obtained the exceptional value `INF` as the result of executing the `MUFU.RCP` instruction, subsequently causing the following `FFMA` instruction to output a `NaN`. This scenario underscores the importance of clearly specifying preconditions for floating-point computations in HPC code. Although, in this particular case, the exception might be directly traced to user-written code, the widespread reliance on extensive third-party libraries poses significant and often intricate debugging challenges, as we discuss next.

3.3 Vulnerabilities in a Pytorch-based Compressor

In this case study, we examine a lossy data compressor named PyBlaz[1, 35], developed in our prior research using the widely adopted PyTorch[34] framework. We illustrate how

the compressor (or "codec") is instantiated, how the input PyTorch tensor x is created, and how compression and subsequent decompression are performed on x. Finally, we investigate whether any NaN values appear in the final output and whether such values were generated at any point during the process:

```
# script3.py
import torch
torch.manual_seed(42)
from pyblaz.compression import PyBlaz
import time
# Create a compressor
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
codec = PyBlaz(
    block_shape=(128, 128),
    dtype=torch.float32,
    device=device,
    compute_mode="tf32",
    compile=True
)
# Create a tensor with extreme values
print("Create a tensor with TF32   values; compress/decompress it")
tf32_big = (2 - 2 ** -10) * (1 << 64)
x = torch.randn(2048, 2048, device=device) * tf32_big
start_time = time.time()

compressed_x = codec.compress(x)
decompressed_x = codec.decompress(compressed_x)

print(f"dec output partial =", decompressed_x[0:16])
# does x have a nan or inf?
if (x.isnan().sum() > 0):
    print("x has a nan!")
if (x.isinf().sum() > 0):
    print("x has an inf!")
end_time = time.time()
print(f"Elapsed time: {end_time - start_time:.6f} seconds")
```

We observed that although the final output contains neither NaN nor INF values (the "has a nan/inf" prints did not happen), such exceptional values do arise during intermediate computations. This raises concerns about the integrity and reliability of the final results—particularly in light of our earlier discussion of the MAX macro. The output from this experiment is shown below:

```
python script3.py
(...no exceptions printed...)
real    0m7.405s, user    0m2.815s,   sys 0m5.737s

(...now run with instrumentation...)

time LD_PRELOAD =./detector.so python script3.py
#GPU-FPX: Instrument all kernels.
Running #GPU-FPX: kernel [...::vectorized_elementwise_kernel]
--- Create a tensor... ---

Running #GPU-FPX: kernel
[void at::native::(anonymous namespace)

::distribution_elementwise_grid_stri\
de_kernel] ...
Running #GPU-FPX: kernel [void at::native::(anonymous namespace)
::CatArrayBatchedCopy] ...

HMMA.1688.F32.TF32 R4, R132.reuse, R2, R4 ; : MMA being used!
...(many more)...
HMMA.1688.F32.TF32 R120, R192.reuse, R170, R120 ; : MMA being used!
HMMA.1688.F32.TF32 R124, R192.reuse, R168, R124 ; : MMA being used!
HMMA.1688.F32.TF32 R128, R192, R150, R128 ; : MMA being used!
Running #GPU-FPX: kernel [void cutlass::Kernel] ...
#GPU-FPX LOC-EXCEP INFO: in kernel
[void cutlass::Kernel], NaN found @ /unknown_path in [void cutlas\
s::Kernel]:0 [FP32]
Running #GPU-FPX: kernel [void at_cuda_detail

::cub::DeviceReduceKernel]
Running #GPU-FPX: ... [DeviceReduceSingleTileKernel]
Running #GPU-FPX: kernel [DeviceCompactInitKernel]
Running #GPU-FPX: kernel [DeviceSelectSweepKernel]
Running #GPU-FPX: ... [write_indices]
Running #GPU-FPX: kernel [index_elementwise_kernel]
Running #GPU-FPX: kernel [reduce_kernel]
Running #GPU-FPX: kernel [elementwise_kernel]
Running #GPU-FPX: kernel [unrolled_elementwise_kernel]
Running #GPU-FPX: kernel
[CatArrayBatchedCopy_aligned16_cont\
ig]
------------ GPU-FPX Report -----------

--- FP16 Operations ---
Total NaN found:          0
Total INF found:          0
Total underflow (subnormal):  0
Total Division by 0:          0
--- FP32 Operations ---
Total NaN found:          1
Total INF found:          0
Total underflow (subnormal):  0
Total Division by 0:          0
--- FP64 Operations ---
Total NaN found:          0
Total INF found:          0
Total underflow (subnormal):  0
Total Division by 0:          0
--- Other Stats ---
Kernels:      14
The total number of exceptions are: 32

real    0m45.069s, user    1m3.745s, sys 0m8.656s
```

This run provides visibility into the underlying PyTorch codebase and reveals several informative details—for instance, the use of MMA (matrix multiply-accumulate) instructions. Such behavior only emerged when the tensor x was sufficiently large. Further investigation using the analyzer yields additional insights (with the output manually trimmed, as in previous examples):

```
Running #GPU-FPX: kernel
[void at::native::(anonymous namespace)::CatArrayBatchedCopy] ...
Running #GPU-FPX: kernel
[void cutlass::Kernel] ...
#GPU-FPX-ANA SHARED REGISTER:
Before executing  ..  @ /unknown_path in [void cutlass::Kernel]:0

Instruction: FSEL R160, R160, -QNAN , !P0 ; We have 2 registers in total.
Register 0 is NaN.
Register 1 is NaN.
#GPU-FPX-ANA SHARED REGISTER: After ...  @ /unknown_path in
[void cutlass::Kernel]:0

Instruction: FSEL R160, R160, -QNAN , !P0 ; We have 2 registers in total.
Register 0 is NaN.
Register 1 is NaN.
#GPU-FPX-ANA SHARED REGISTER: Before ...    @
/unknown_path in [void cutlass::Kernel]:0

Instruction: FMNMX R13, R13, R160.reuse, !PT ; ..3 registers in total.
Register 0 is VAL.
Register 1 is VAL. Register 2 is NaN.
```

Key Takeaways: The fact that the final output contains no exceptional values—despite their occurrence during intermediate computations—is troubling. Without access to tools like GPU-FPX, users would remain unaware of these silent anomalies. This is particularly alarming as machine learning libraries such as PyTorch may find their way into safety-critical operations. Yet, as in this example, we often cannot

diagnose why such exceptions are triggered or whether they ultimately affect correctness—underscoring the limitations of current tooling in the realm of array programming.

There is a clear need for more tools like GPU-FPX—tools that are not only faster but also offer stronger coverage and guarantees. As shown in [23], GPU-FPX incurs a geometric-mean slowdown of 30× across more than 150 applications (with further performance details in §4.3). While coarse-sampling strategies, such as monitoring every k^{th} kernel invocation, can reduce overhead [23], they run the risk of missing critical floating-point exceptions.

4 GPU-FPX Bug-Fixes, Extensions, Performance

While tools like GPU-FPX may represent only a modest step forward individually, their development, validation, and refinement require substantial non-trivial effort. We encourage contributions from the broader community to ensure that the critically important domain of GPU array programming continues to advance and benefit all practitioners. In what follows, we describe several recent bug fixes, followed by a major new feature: support for MMA (matrix multiply-accumulate) instructions. We then assess the expected performance impact of GPU-FPX when applied to tensor core–based workloads. Finally, we discuss the tool's modular design and its implications for future extensibility.

4.1 Tool Fixes and Improvements

We extended and refined the existing GPU-FPX tool to improve reliability and adaptability. Key updates include these:

- We introduced a new error recording format that summarizes exceptions across threads (for easier downstream processing) using an `XOR` reduction instruction.
- We fixed a premature early-exit bug that could have prevented logging all exceptions—for example when two threads encounter an exception error.
- We added new detectors which check for exceptional values generated by Tensor Core operations.
- We significantly updated our exception reporting infrastructure to support new floating-point types such as half precision.
- We upgraded GPU-FPX to use a newer version of NVBit, and in the process addressed a deadlock issue encountered with newer GPU drivers (following the NVBit team's suggestions).

All these additions are documented in the GPU-FPX website [14].

4.2 Adding Support for MMA Instructions

The following is an example to show what is needed to add support for a hypothetical MMA instruction, `HMMA.442.F16 R0 R4 R6`. Figure 1 illustrates the data distribution across registers for this operation, which computes the product of

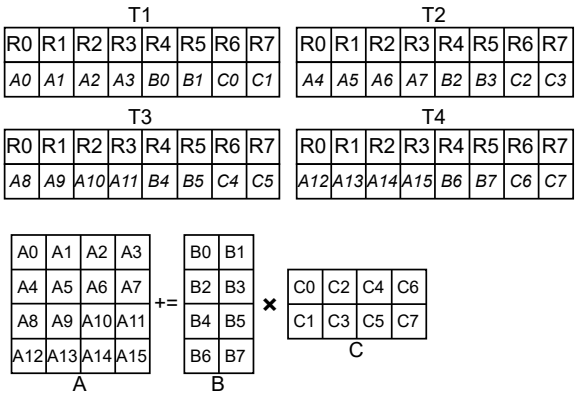


Figure 1. Example distribution of data in registers for an MMA operation $\mathbf{A} += \mathbf{B} \cdot \mathbf{C}$.

a 4×2 matrix **B** and a 2×4 matrix **C**, and accumulates the result into a 4×4 matrix **A** (these dimensions are written as "442"' as part of the instruction). Matrices **A** and **B** are stored in row-major order, while **C** is stored in column-major order. In this example:

- Each of four threads (`T1`–`T4`) holds part of the matrices in their registers.
- Thread `T1`, for instance, holds the first four elements of **A** in `R0`–`R3`, the first row of **B** in `R4`-`R5`, and the first column of **C** in `R6`–`R7`.
- Similarly, each thread holds its parts of **A**, **B** and **C** in its registers, respectively, `R0`, `R4` and `R6`.

If we want to examine the operands for this hypothetical instruction for possible exceptional values, we need to know how the data is distributed in the registers. In this example, we see that we need to treat the first operand as representing a vector of four registers and the second and third operands as representing vectors of two registers. We observe that this is how this distribution must be done, since the matrix **A** has 16 elements distributed across four threads, so each thread must hold four elements.

This hypothetical example is instructive for determining the data layout for other—more general—instructions, such as `HMMA.1684.F32.TF32`. This instruction computes:

$$\mathbf{D} = \mathbf{A} \cdot \mathbf{B} + \mathbf{C}$$

where **A** is a 16×4 matrix, **B** is a 4×8 matrix, and **C** and **D** are 16×8 matrices. The matrices **A** and **B** are in the TF32 format which is a reduced precision format which has fewer mantissa bits compared to a standard 32-bit floating point number.

In this example, we will focus on detecting exceptional values, that is, NaN or infinity, appearing in the destination matrix **D**. Unlike the prior example, each thread is in a group of 32 threads, called a warp. Since there are 128 elements in the result, each thread must have four elements of the

result in its registers. Therefore, if the destination register of the instruction were `R0`, then the full result would appear in registers `R0-R3`.

The instrumentation of these instructions is done as follows:

- We use NVBit to send the contents of the four register values from above to an error checking function.
- The error checking function performs a check for exceptional values such, as infinity and NaN.
- These errors are reduced through `xor` so that one thread can update a global log of errors.
- Notify the user of an error if this is the first time it has been detected for an instruction.

A similar process applies to instructions like `HMMA.16168.F16.F16`, where operands are in half-precision. Since each 32-bit register holds two 16-bit values, register access must consider this packing format.

4.3 Performance Analysis

Previously, GPU-FPX has been measured to produce slowdowns of up to 500 times [23] on stock programs. However, there wasn't a measurement of the performance overhead incurred by the instrumenttion itself. We measured the impact on the highly optimized matrix-multiplication kernel [33], which uses tensor cores. The generated assembly code is arranged in such a way that the tensor operations are interleaved with loads from the GPU shared-memory. This interleaving reduces the apparent latency from loading from memory, meaning the code is closer to its pure compute capacity. In our experiment, we multiplied a 5120×4096 matrix and a 4096×4096 matrix using this kernel. Without instrumentation, this multiplication takes 4.9ms on average on a RTX 3080 Ti. With full instrumentation, we saw an average time of 2.99s, representing a 610× slowdown. We also ran this with GPU-FPX's option which allows the tool to periodically enable instrumentation. With periodic sampling, we observed an average of 310ms per multiplication across 1,000 runs with a sample interval of 50. This results in a slowdown of 63×. The slowdown here represents the cost of full instrumentation, and the cost of having the binary instrumentation tool present.

While there is some risk of missing an exceptional value when sampling, NaN and infinity values tend to propagate, so they should still be detected later when a kernel is instrumented. Matrix multiplication has a lower chance of having missed exceptions, since each entry of the result depends on whole rows and columns. This is particularly amplified when matrix multiplication is being done iteratively on the same matrix, such as in solvers.

4.4 Development for Modular Extensibility

We are working to add support for new formats and instructions to GPU-FPX. This includes support for half-precision instructions which are vectors by default as they are used in 32-bit registers. For example, an add instruction can add both half values, or it can select the first or second only. This is being done as part of a wider effort to make the tool more generic to facilitate easier addition of support for new instructions. This will also allow the tool to be adapted more easily to new uses, for example, a new tool which dynamically tracks memory accesses for later analysis.

5 Testing-Based Specification Discovery

Given the often incomplete nature of GPU datasheets, it is natural to explore uncovering undocumented behaviors through targeted testing. Such tests are easy to share and can serve not only to reveal hidden quirks, but also to assess the degree to which different GPUs exhibit consistent behavior.

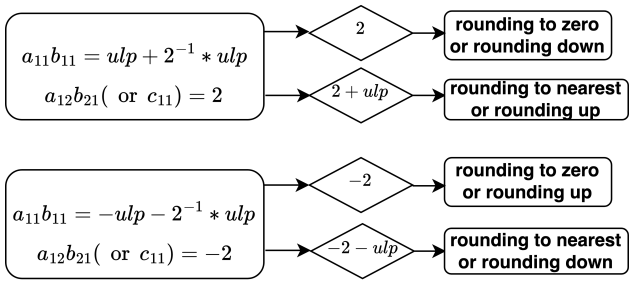


Figure 2. The logic for test T_rnd_dir (test rounding direction) are presented here. By setting the $a_{11}b_{11}$ product to the indicated value (e.g., keep $b_{11} = 1$ and setting a_{11} to this value) and the $a_{12}b_{21}$ product also to the indicated value, the execution is carried out (all other inputs not mentioned are set to 0). Then by examining the d_{11} output, we can decide which case we fall into with respect to the rounding being used.

We now illustrate these ideas in the realm of Tensor Cores—an effort begun in publications such as [3, 11, 30] and subsequently extended by us in [24, 25] to cover both NVIDIA and AMD GPUs. The overall goal of a Tensor Core is to efficiently compute the matrix $\mathbf{D}$ where $\mathbf{D} = \mathbf{A} \cdot \mathbf{B} + \mathbf{C}$, with $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ also being matrices. Since all entries in $\mathbf{D}$ are computed in the same way, it suffices to examine a single entry $\mathbf{D}_{1,1} = \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} + \ldots + \mathbf{A}_{1,n}\mathbf{B}_{n,1} + \mathbf{C}_{1,1}$.

For GPU programmers, understanding the behavior of multi-term additions—a term introduced in works such as [3] and [30]—is essential. Key questions arise: How are the individual product terms summed? Are the additions performed in a fixed order, such as left-to-right? And most importantly, what rounding directions are supported by GPU multi-term adders?

To investigate the last question, we designed a test, `T_rnd_dir`, introduced in [24] and illustrated in Figure 2. The specifics of these tests are not critical here; they follow naturally from the rounding rules defined by IEEE floating-point

arithmetic (see [24] for details). What is more significant is that such tests can be formalized using SMT-based logic, as demonstrated in [39]. This approach not only increases confidence in the correctness of compiled tests, but also paves the way for a shared, trustworthy test suite—readily disseminated across the community to foster deeper understanding and broader verification of GPU behavior.

6 Toward Verified Tensor Core Programming

The topic of multi-term addition has far-reaching implications, with the potential to cause striking variability in results across different GPUs. One such illustrative test is discussed in §6.1. In the current era of automated code generation powered by large language models (LLMs), there is a growing temptation to rely on such tools to accelerate Tensor Core programming. While this holds considerable promise for productivity, it also comes with important caveats. We provide a brief glimpse into both the potential and the pitfalls of this approach in §6.2.

6.1 A Striking Illustration of Non-Portability

The (so-called) trailing matrix update pattern, expressed as $\mathbf{A}_i = \mathbf{A}_i - \mathbf{P}_i \cdot \mathbf{T}_i$, appears frequently in array programming workloads, including in libraries such as cuSolver [6]. More generally, computations of the form $\mathbf{D} = \mathbf{C} - \mathbf{A} \cdot \mathbf{B}$ fall under the Level-3 category of the Basic Linear Algebra Subprograms (BLAS) standard.

To investigate the numerical behavior of this pattern, we design a test case involving matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, each of size $2^{13} \times 2^{13}$ (i.e., 8192×8192). The matrices are populated with carefully chosen values to exercise edge cases in potential multi-term implementations, as detailed in [24].

Specifically, we initialize $\mathbf{C}_{i,j} = 2^{20}$ for all i, j; $\mathbf{A}_{i,0} = 2^{10}$; $\mathbf{A}_{i,j} = 2^{-2}$ for odd j; and $\mathbf{A}_{i,j} = 2^{-3}$ for even j (excluding $j = 0$). For matrix $\mathbf{B}$, we set $\mathbf{B}_{0,j} = 2^{10}$ and all other $\mathbf{B}_{i,j}$ entries to 2^{-3}. All values are represented in FP16 precision. Under real-number semantics, this construction yields $\mathbf{D}_{i,j} \approx 191.99218$ for all i, j

$$\begin{aligned}\mathbf{D}_{i,j} &= -(\mathbf{A}_{i,0} \cdot \mathbf{B}_{0,j} + \sum_{j\%2=1} \mathbf{A}_{i,j} \cdot \mathbf{B}_{i,j} + \sum_{\substack{j\%2=0 \\ j\neq 0}} \mathbf{A}_{i,j} \cdot \mathbf{B}_{i,j}) + \mathbf{C}_{i,j} \\ &= -(2^{10} \cdot 2^{10} - \sum_{2^{12}} 2^{-2} \cdot 2^{-3} - \sum_{2^{12}-1} 2^{-3} \cdot 2^{-3}) + 2^{20} \\ &= 2^7 + 2^6 - 2^{-6} = 191.984375\end{aligned}$$

Unfortunately (as specified under "Story-2" earlier), thanks to the differences in multi-term addition implementation, the answer can be "all zeros" on NVIDIA A100, V100, AMD MI250, and a CPU; a matrix filled with 255.875 on the AMD MI100; and one filled with 191.875 on the NVIDIA H100.

A Sketch of Multi-Term Addition Differences: When summing a series of product terms, several key implementation choices arise: (1) Are product terms grouped—e.g., accumulated using higher-precision registers or shared rounding bits to enhance accuracy? (2) How are IEEE rounding semantics handled? IEEE floating-point arithmetic defines multiple rounding modes, implemented using additional bits: the Guard (G), Round (R), and Sticky (S) bits. On many GPUs, Tensor Cores perform block fused multiply-add (block FMA) operations, in which product terms are grouped—commonly in blocks of size 4, 8, or more—and share GRS bits within each block. In contrast, CPUs "forget" the GRS bits after each operation.

To illustrate how this leads to divergent results, consider why the CPU-computed value of $\mathbf{D}_{i,j}$ may be zero. In left-to-right accumulation, small values are incrementally added to a large leading term (2^{20}), and the result remains effectively 2^{20}. When subtracted from an identical term in $\mathbf{C}_{i,j}$, cancellation yields zero.

However, with wider block FMAs on Tensor Cores, where accumulation may occur in higher precision and GRS bits are preserved across grouped terms, such cancellation may no longer occur exactly. This leads to measurable discrepancies when accumulated across the whole matrix multiplication. Beyond this qualitative explanation, we currently lack a full formal model for explaining the exact results observed on the five GPU models (i.e., why MI100 yielded 255.875, why H100 yielded 191.875, etc.).

The critical conclusion for longer-term research is that such anomalies must, ideally, not be left unexplained. Tensor Cores appear to be of central importance in future of high-performance computing, as argued in [10].

Challenge: Given a target GPU's characteristics—such as FMA block width, rounding mode, subnormal support, and more—can one construct fill patterns that produce result differences of a specified magnitude? Solving this challenge would enable execution-based testing of commercial GPUs under a variety of assumed arithmetic models, offering a practical path to reverse-engineering undocumented behavior. Analogous strategies have been explored in other domains—for instance, Collier's work on checking conformance against assumed memory models by analyzing program execution results [4].

What It Might Take to Solve the Challenge: A direct attempt to solve problems at this scale using existing logic-based tools—such as SMT solvers—is unlikely to succeed. Instead, a more viable approach may involve recursively generating test cases by modeling the constituent expressions of matrix operations, simulating their behavior under a given GPU arithmetic model, and extrapolating the results. These are precisely the kinds of foundational challenges that the broader programming languages community is uniquely

well positioned to tackle, given its expertise in semantics, abstraction, and automated reasoning.

6.2 Formal Verification of Array Computations

Suppose one writes code to perform 16×16 matrix multiplication using Tensor Cores, following a standard online reference (e.g., [28]). Now consider using a large language model, such as ChatGPT, to automatically generate code that computes the maximum value in the result matrix, **D**. One such code block was generated entirely by ChatGPT in one of our experiments:

```
// local thread ID within a warp
int warpThreadId = threadIdx.x & (WARP_SIZE - 1);
float localMax   = -65504.0f;  // half-prec. min is roughly -65504

// Step 1: Each thread checks some subset of D
for (int idx = warpThreadId; idx < M * N; idx += WARP_SIZE) {
    float val   = __half2float(D[idx]);
    localMax    = fmaxf(localMax, val);
}
// Step 2: Warp-wide reduce using shuffle intrinsics
unsigned int mask = __activemask(); // active lanes
#pragma unroll
for (int offset = WARP_SIZE / 2; offset > 0; offset >>= 1) {
    float temp = __shfl_down_sync(mask, localMax, offset);
    localMax   = fmaxf(localMax, temp);
}
// Step 3: The lane 0 in the warp stores the final max to MaxVal
if (warpThreadId == 0) {
    *MaxVal = __float2half(localMax);
}
```

A case study based on this code raises several important questions:

- How can we formally verify that this code block correctly computes the maximum value of the **D** array, taking into account GPU-specific behaviors?
- How can we model GPU thread execution and the semantics of low-level operations, including shuffle instructions, masking, and warp-synchronous behavior?

While such questions appear to be well within the reasoning power of modern SMT solvers, the primary challenge seems to be the lack of a rigorous, end-to-end framework for capturing the semantics of GPU programming languages—all the way from high-level CUDA semantics down to low-level assembly instructions.

7 Concluding Remarks

In conclusion, the importance of supporting GPU-based array programming through rigorous—or at least systematic—foundations has been underscored by several prior works [11, 30]. The challenges of reproducibility in this space have been studied by many other groups (e.g., [31]). The importance of floating-point exception checking in the context of GPUs was pioneered in [19] with the help of LLVM instrumentation. The primary tool presented in this work, namely GPU-FPX, improves upon an earlier tool we had built, namely BinFPE [20], that first introduced the idea of SASS-level binary instrumentation. The exception checker presented in [21] focuses on CPU and parallel-program exception checking—another important space to consider. A recent tutorial from our project team available at [37] introduces many other floating-point correctness checking tools besides the ones presented here.

We are now in an era marked by a growing variety of number formats, each bringing its own set of non-standard arithmetic behaviors—such as differing rounding modes and NaN-handling conventions, and distinct tradeoffs in precision and dynamic range (e.g., BF16 vs. FP32). As Reed et al. observe [36], much of this shift has been driven by the demands of machine learning, often imposed on high-performance computing. The assumption that reducing bit width will automatically lead to better outcomes—such as lower memory traffic or improved energy efficiency—is a seductive but ultimately dangerous oversimplification. We might get more (hardware-unhandled) exceptions and loops may terminate more gradually [42]—or never.

Key Takeways.

- Array programming on GPUs merits first-class status in programming languages research. Given the substantial cost and widespread deployment of GPUs, even modest gains in debugging and correctness can yield significant productivity and cost benefits.
- Overlooking correctness in GPU-based array programming carries real societal risks. A growing number of autonomous and embedded systems depend on GPU computations, and even emerging privacy-preserving technologies—such as zero-knowledge computing platforms [27]—are planning GPU integration.
- Ultimately, a unified framework that addresses both parallelism and numerical correctness will be essential—without it, diagnosing and triaging non-reproducible behaviors will remain infeasible.
- Assuming that formally verified correctness is the only viable path forward risks leaving a growing number of practical, day-to-day problems unaddressed. In many other areas of software development, fuzz testing has proven to be an invaluable and pragmatic approach—often yielding results where formal methods remain out of reach. The Verificarlo approach [8] that espouses this approach—and many more of its kind yet to be developed—deserve more attention.

The kinds of GPU programming correctness challenges highlighted here have largely remained underexplored by even seasoned programming languages researchers. This gap is understandable: the semantics of GPU architectures evolve rapidly, making it difficult to establish stable, widely shared conventions and abstractions. However, given the scale and significance of the problems in this domain, broader participation from the PL community would be not only valuable but essential. To encourage those considering engagement with these challenges, we offer the following observations:

- Don't worry if you haven't been following GPUs closely over the past 15 years—a period that roughly coincides

with their rise to prominence. You can still make meaningful contributions by engaging with well-scoped challenges like those we've identified, and developing more such challenges.

- By actively seeking out problems of the kind discussed in this paper, we can foster broader community engagement and, importantly, help bring the next generation of students into the fold of array programming. Our time in this field is limited—we must ensure that younger researchers are equipped and inspired to carry the work forward.

Acknowledgements: Supported in part by NSF CCF NSF 2403379, 2346394, 2217154, and 2446084.

References

[1] Tripti Agarwal, Harvey Dam, Ponnuswamy Sadayappan, Ganesh Gopalakrishnan, Dorra Ben Khalifa, and Matthieu Martel. 2023. What Operations can be Performed Directly on Compressed Arrays, and with What Error? *(SC-W '23)*. 254–262. doi:10.1145/3624062.3625122

[2] artifact-array25 [n. d.]. PLDI Array'25 Workshop Artifact from the University of Utah. https://github.com/ganeshutah/PLDI25-Array-Workshop.

[3] Pierre Blanchard, Nicholas J Higham, Florent Lopez, Théo Mary, and Srikara Pranesh. 2020. Mixed Precision Block Fused Multiply-Add: Error Analysis and Application to GPU Tensor Cores. *SISC* (2020). https://hal.science/hal-02491076

[4] William W. Collier. 1992. *Reasoning About Parallel Architectures*. Prentice-Hall, Inc., Upper Saddle River, NJ, United States.

[5] csc2023 [n. d.]. Correctness in Scientific Computing, a PLDI 2023 Workshop. https://pldi23.sigplan.org/home/csc-2023#program.

[6] cusolver [n. d.]. cuSolver. https://docs.nvidia.com/cuda/cusolver/index.html#cusolverirsrefinement-t.

[7] 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. doi:10.48550/arXiv.2207.09281

[8] Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. 2016. Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic. In *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, Paolo Montuschi, Michael J. Schulte, Javier Hormigo, Stuart F. Oberman, and Nathalie Revol (Eds.). IEEE Computer Society, 55–62. doi:10.1109/ARITH.2016.31

[9] Peter Dinda, Alex Bernat, and Conor Hetland. 2020. Spying on the Floating Point Behavior of Existing, Unmodified Scientific Applications *(HPDC '20)*. 5–16. doi:10.1145/3369583.3392673

[10] Jack Dongarra, Laura Grigori, and Nicholas Higham. 2020. Numerical algorithms for high-performance computational science. *Philosophical Transactions of the Royal Society A* (2020). http://doi.org/10.1098/rsta.2019.0066.

[11] Massimiliano Fasi, Nicholas Higham, Mantas Mikaitis, and Srikara Pranesh. 2021. Numerical behavior of NVIDIA tensor cores. *PeerJ. Computer science* (2021). doi:10.7717/peerj-cs.330

[12] Ganesh Gopalakrishnan, Paul D. Hovland, Costin Iancu, Sriram Krishnamoorthy, Ignacio Laguna, Richard A. Lethin, Koushik Sen, Stephen F. Siegel, and Armando Solar-Lezama. 2017. Report of the HPC Correctness Summit, January. *arXiv* (2017). doi:10.2172/1470989

[13] Ganesh Gopalakrishnan, Ignacio Laguna, Ang Li, Pavel Panchekha, Cindy Rubio-González, and Zachary Tatlock. 2021. Guarding Numerics Amidst Rising Heterogeneity. In *Correctness*. 9–15. doi:10.1109/Correctness54621.2021.00007

[14] gpu-fpx-website [n. d.]. GPU-FPX: A Low-Overhead tool for Floating-Point Exception Detection in NVIDIA GPUs. https://github.com/LLNL/GPU-FPX.git.

[15] John R. Hauser. 1996. Handling Floating-Point Exceptions in Numeric Programs. *TOPLAS* 18, 2 (March 1996), 139–174. doi:10.1145/227699.227701

[16] Yafan Huang, Sheng Di, Xiaodong Yu, Guanpeng Li, and Franck Cappello. 2023. cuSZp: An Ultra-fast GPU Error-bounded Lossy Compression Framework with Optimized End-to-End Performance *(SC '23)*. doi:10.1145/3581784.3607048

[17] ieee-754-fp-standard [n. d.]. https://standards.ieee.org/ieee/754/6210/.

[18] ieee-fp-calculator [n. d.]. IEEE-754 Floating Point Converter. https://www.h-schmidt.net/FloatConverter/IEEE754.html.

[19] Ignacio Laguna. 2020. FPChecker: detecting floating-point exceptions in GPU applications *(ASE '19)*. IEEE Press, 1126–1129. doi:10.1109/ASE.2019.00118

[20] Ignacio Laguna, Xinyi Li, and Ganesh Gopalakrishnan. 2022. BinFPE: Accurate Floating-Point Exception Detection for GPU Applications. https://pldi22.sigplan.org/home/SOAP-2022#event-overview. In *SOAP '22*.

[21] 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 '22*. 39–50. doi:10.1109/IISWC55918.2022.00014

[22] Tao Lei, Yu Zhang, Sida I. Wang, Hui Dai, and Yoav Artzi. 2018. Simple Recurrent Units for Highly Parallelizable Recurrence. In *EMNLP*. doi:10.18653/v1/D18-1477

[23] Xinyi Li, Ignacio Laguna, Katarzyna Swirydowicz, Bo Fang, 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 2023*. doi:10.1145/3588195.3592991

[24] Xinyi Li, Ang Li, Bo Fang, Katarzyna Swirydowicz, Ignacio Laguna, and Ganesh Gopalakrishnan. 2024. FTTN: Feature-Targeted Testing for Numerical Properties of NVIDIA & AMD Matrix Accelerators. In *CCGrid*. 39–46. doi:10.1109/CCGrid59990.2024.00014

[25] Xinyi Li, Ang Li, Ignacio Laguna, and Ganesh Gopalakrishnan. [n. d.]. https://github.com/LLNL/FTTN.git.

[26] Hatem Ltaief, Marc G. Genton, Damien Gratadour, David E. Keyes, and Matteo Ravasi. 2022. Responsibly Reckless Matrix Algorithms for HPC Scientific Applications. *CISE* 24, 4 (2022), 12–22. doi:10.1109/MCSE.2022.3215477

[27] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. GZKP: A GPU Accelerated Zero-Knowledge Proof System *(ASPLOS 2023)*. 340–353. doi:10.1145/3575693.3575711

[28] Lei Mao. 2020. NVIDIA Tensor Core Programming. https://leimao.github.io/blog/NVIDIA-Tensor-Core-Programming/. Accessed: 2025-04-01.

[29] Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. 2025. FloatGuard: Efficient Whole-Program Detection of Floating-Point Exceptions in AMD GPUs. In *HPDC '25'*.

[30] Mantas Mikaitis. 2023. Monotonicity of Multi-Term Floating-Point Adders. arXiv:2304.01407

[31] Fraser Mince, Dzung Dinh, Jonas Kgomo, Neil Thompson, and Sara Hooker. 2023. The Grand Illusion: The Myth of Software Portability and Implications for ML Progress. arXiv, https://arxiv.org/abs/2309.07181.

[32] no-trap-nvidia [n. d.]. https://docs.nvidia.com/cuda/floating-point/index.html.

[33] Nvidia. [n. d.]. Cutlass TF32 example. https://github.com/NVIDIA/cutlass/blob/main/examples/14_ampere_tf32_tensorop_gemm/ampere_tf32_tensorop_gemm.cu

[34] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary

DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS 32*. 8024–8035. doi:10.48550/arXiv.1912.01703

[35] pyblaz-website [n. d.]. PyBlaz Compressor. https://github.com/damtharvey/pyblaz.git.

[36] Daniel Reed, Dennis Gannon, and Jack Dongarra. 2023. HPC Forecast: Cloudy and Uncertain. *CACM* 66, 2 (January 2023), 82–90. doi:10.1145/3552309

[37] sc24-tutorial [n. d.]. Tools to Diagnose and Repair Floating-Point Errors in Heterogeneous Computing Hardware and Software. https://fpanalysistools.org/SC24/.

[38] sru-nan [n. d.]. SRU NaN Issue. https://github.com/asappresearch/sru/issues/193.

[39] Benjamin Valpey, Xinyi Li, Sreepathi Pai, and Ganesh Gopalakrishnan. 2025. An SMT Formalization of Mixed-Precision Matrix Multiplication. NFM 2025. doi:10.48550/arXiv.2502.15999 Manuscript.

[40] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *MICRO*. 372–383. doi:10.1145/3352460.3358307

[41] Peichen Xie, Yanjie Gao, and Jilong Xue. 2024. FPRev: Revealing the Order of Floating-Point Summation by Numerical Testing. arXiv:2411.00442

[42] Ichitaro Yamazaki, Christian Glusa, Jennifer Loe, Piotr Luszczek, Sivasankaran Rajamanickam, and Jack Dongarra. 2022. High-Performance GMRES Multi-Precision Benchmark: Design, Performance, and Challenges. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 112–122. doi:10.1109/PMBS56514.2022.00015