Guarding Numerics Amidst Rising Heterogeneity

Ganesh Gopalakrishnan*, Ignacio Laguna†, Ang Li‡,
Pavel Panchekha§, Cindy Rubio-González¶ and Zachary Tatlock‖

Univ of Utah (ganesh@cs.utah.edu) *, LLNL (ilaguna@llnl.gov) †, PNNL (ang.li@pnnl.gov) ‡,
Univ of Utah (pavpan@cs.utah.edu) §, UC Davis (crubio@ucdavis.edu) ¶, UW (ztatlock@cs.washington.edu) ‖

***Abstract*—**
New heterogeneous computing platforms—especially GPUs and other accelerators—are being introduced at a brisk pace, motivated by the goals of exploiting parallelism and reducing data movement. Unfortunately, their sheer variety as well as the optimization options supported by them have been observed to alter the computed numerical results to the extent that reproducible results are no longer possible to obtain without extra effort. Our main contribution in this paper is to document the scope and magnitude of this problem which we classify under the heading of numerics. We propose a taxonomy to classify specific problems to be addressed by the community, a few immediately actionable topics as the next steps, and also forums within which to continue discussions.

I. Introduction

The quest to exploit parallelism coupled with combating data movement costs has necessitated the exploration of newer computer architectures by the HPC community. At the same time, these architectures are charged with the ability to handle problems of ever-increasing scale and resolution while maintaining energy-efficiency and productivity. These problem solving methods are fundamentally important for modern society, given that they play a crucial role in thousands of important applications ranging from climate simulation [1] to medical research [2], [3]. Unfortunately, the development and testing of software for such applications now faces many serious productivity crises brought about by the increased use of heterogeneous computing elements.

In practical terms, extreme heterogeneity implies the use of a variety of CPUs, GPUs, and other accelerators,[1] the adoption of number representations and precision regimes different from the IEEE-754 standard for floating-point arithmetic, the use of compilers that perform aggressive optimizations, etc. Such changes have both fundamental and pervasive impact throughout the software stack, altering the computed results in subtle ways. Testing and debugging these numerical correctness-compromising aspects ("numerics" for short) is this paper's focus of discussion.

In upcoming HPC systems, the overall numerical integrity can be affected by a variety of causes, not all of which are fully understood or have been encountered in existing HPC systems. This makes it important that the testing methods we propose extrapolate trends that have been observed at the edge of today's designs, and also remain nimble to adjust to more such challenges. For example, problem-domain-specific number systems are increasingly being employed (e.g., [4]), and often compiler optimizations can go so far as altering the basic science predictions [5]. It is important to develop testing methods that address such aspects both within legacy codes that are being ported to newer hardware and compilers, and within brand-new codes that combine both HPC and machine learning (ML) that have inherently different types of precision and result-accuracy demands.

A central problem we will face at these upcoming scales and levels of heterogeneity is that error tolerances are often unclear even to experts. While the IEEE standard for floating-point arithmetic clearly defines how much absolute (and relative) rounding error is introduced by each basic operation, insisting on meeting such rounding error targets can often be both impractical and inefficient. Stating absolute error targets is impractical because programmers do not know what error bounds to specify for complex pieces of code. Furthermore, all methods of rounding error estimation are overapproximate, and do not easily scale to large applications. Already, alternate criteria are under investigation. For example, in high-end HPC, lossy compression algorithms (approaches that precision-tune entire data planes) are bound to play an increasing role [6], [7] and already measures such as signal-to-noise ratio (PSNR, L2-norm error, etc. [8]) are under consideration as acceptable result tolerances. Things could be even more extreme in ML where the number of bits allocated are determined through training [9]. Future testing frameworks in this area must accommodate multiple acceptance criteria and help automate their application.

Unfortunately, with multiple testing methods, we no longer have one notion of when software correctly ports across architectures, compilers, and precision regimes. While result disagreement between platforms is no longer a surprising issue in HPC [5], the manner in which variability arises within GPU-based applications is much more nuanced, poorly understood as well as rarely discussed. New GPUs are released once every three years by multiple vendors. Despite adherence to the IEEE standard, GPU hardware includes units such as *special function units* and *tensor cores* that are fast, but also produce result deviations from standards (elaborated in later sections). GPUs are inherently poorer at reporting as well as handling exceptions: there are no prevailing strong guidelines for this. Meanwhile, the advance of GPUs themselves are motivated

[1]We use "GPU" to cover all types of accelerators, although the main reference is to General Purpose GPUs made by Nvidia, AMD, Intel, etc., that are slated for introduction within upcoming large-scale DOE machines.

978-1-6654-2061-7/21/$31.00 ©2021 IEEE
DOI 10.1109/Correctness54621.2021.00007

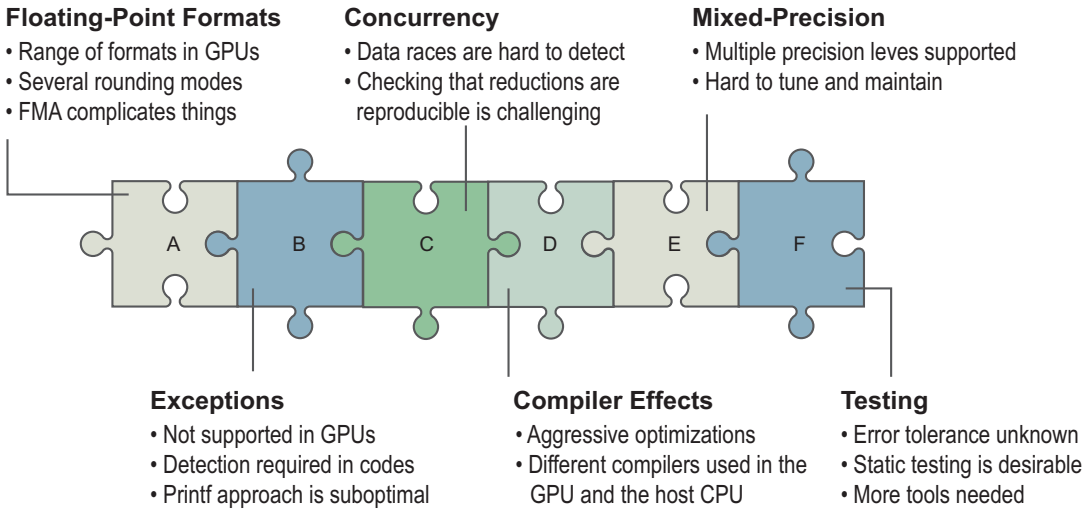

Fig. 1. Overview of the challenges faced in numerical applications for GPUs.

by their central role in Machine Learning where different (and often softer) numerical and exception-related criteria are applicable; HPC users are often forced to use these "ML GPUs" as those are likely the best performers.

The upshot is that the HPC community is ill-prepared to handle these lurking dangers of embracing heterogeneity. Performance portability is clearly a central concern, and much more attention is being devoted to it. While the correctness community has documented many of these issues surrounding GPUs [10], such studies are more geared toward handling "traditional software defects"—for instance related to the semantics of concurrency and synchronization. Our focus in this paper is on numerical correctness (or "numerics" for short) in the heterogeneous era, addressing GPU-based systems. We organize our discussion by providing a taxonomy of the various challenges confronting the numerics community due to rising heterogeneity (§II). In §III, we emphasize the need for a community portal to keep abreast of the evolution of numerics across GPUs, and invite the community to join one such forum [11]. Figure 1 is a conceptual view listing the headings under which the problems we discuss can be classified, the challenges faced by the community as well as solutions needed in these areas.

II. Challenges, Solutions

We provide the specifics of GPUs under the headings of number formats and precision (§II-A), exceptions (§II-B), variations caused by dynamic (concurrency/schedule) aspects (§II-C), compiler-induced variability (§II-D), and mixed precision (§II-E). This sets the stage for discussing testing tools for heterogeneous codes in §II-F.

A. Floating-Point Formats

GPUs provide a range of number formats, and the choice of format is essential because these formats provide dramatic cost/accuracy tradeoffs. There are, of course, the classic single and double precision floating-point operations; for NVIDIA GPUs, their performance is shown in Figure 2. IEEE-FP32 has been supported by NVIDIA GPUs ever since its first generation, while IEEE-FP64 was enabled from compute-capability 1.3 (CC-1.3).

But there are also more exotic formats; all the floating-point formats supported by NVIDIA GPUs are listed in Table I. Starting from the Jetson TX-1 embedded-system board (CC-5.3), IEEE-FP16 was featured, and later introduced into the next flagship Tesla-P100 GPUs. From P100 to V100 to A100, the FP16 throughput (excluding tensor cores) increases from 21.2 to 31.4 to 78 TFLOP/S. The Int8 vector multiply operations were added in Tesla-P40 (CC-6.1) and from P40 to V100, the Int8 throughput in CUDA cores increased from 48 to 64 TOP/S.

Modern GPUs contain tensor cores, which can execute highly restricted operations with a complex mixture of precision choices while also yielding higher throughput. Tensor cores were first featured in the Volta architecture (CC-7.0) with only FP16 matrix-multiply-add (MMA) operations. Then, the Turing architecture (CC-7.5) enabled Int-8, Int-4, and Int-1 binary [12], [13] precision in tensor cores, and the latest Ampere A100/A40 GPUs further support TF32, BF16 and FP64 formats for tensor cores, offering extra precision choices. Brain floating-point 16 (BF16), proposed by Google Brain, tackles the small dynamic range issue of FP16 in AI utilization (BF16 is supported in Tensorflow and PyTorch). Tensor floating-point 32 (TF32), firstly proposed in the latest Ampere GPUs, serves as a reduced alternative to FP32; it essentially uses 19 bits. The main advantage of TF32 is that it has the same exponent bits as FP32, so that TF32-enabled Ampere tensor cores can directly operate on FP32 inputs by rounding from a 23-bit mantissa to a 10-bit mantissa, which

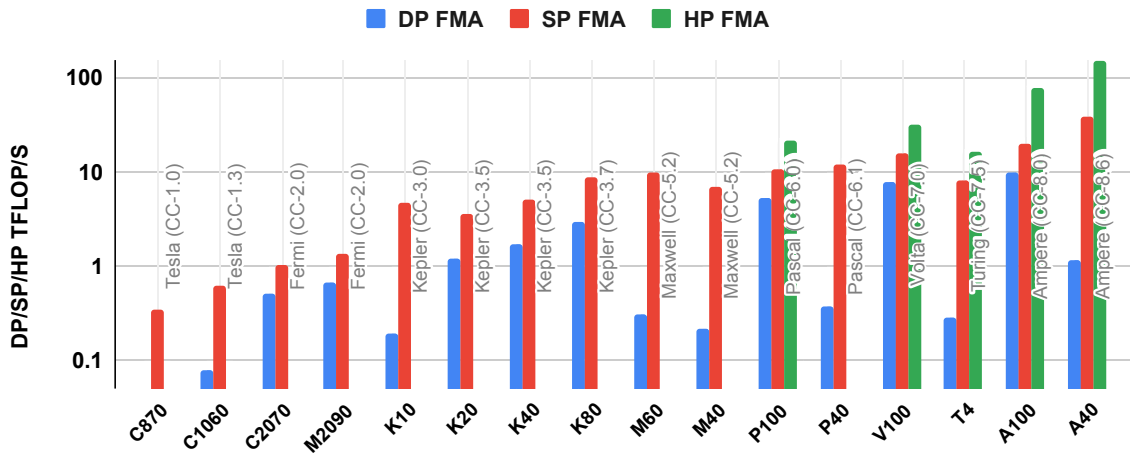

Fig. 2. Double/single/half-precision (i.e., DP/SP/HP) performance scaling for the NVIDIA HPC-focused Tesla GPUs from the first CUDA-enabled Tesla architecture (CC-1.0) until the latest Ampere (CC-8.0/8.6) architecture. FMA refers to fused-multiply-add operations. The unit is trillion floating-point operations per second (teraflops). The vertical axis is in log.

can then be multiplied, accumulated in FP32 and output in FP32 for keeping high-precision. This approach of "internally" leveraging TF32 in tensor cores does not require any change on FP32-based user-code. Not only have these capabilities grown, they have also scaled in throughput. From Tesla V100 to A100 GPUs, the FP16 throughput of tensor cores grew from 125 to 312 TFLOP/s, while for FP64, TF32 and BF16 the theoretical compute performance is 19.5, 156 and 312 TFLOP/S, respectively.

The point of this litany of precisions and features is to highlight the diversity available and the importance of capitalizing on it to achieve maximum performance. Unfortunately, not only do GPUs support all of these precisions, they also support a diversity of rounding modes. Modern GPUs support all 4 standard IEEE rounding modes (round to nearest, to zero, to ∞, and to $-\infty$). Rounding modes can be changed by calling specific intrinsics (like `__fdiv_rz` to divide while rounding towards zero). Furthermore, the *-ftz=true* compiler flag can flush denormal numbers to zero, which speeds up FP32 computation in GPUs with CC-2.0 or later.[2] Here again, there are a variety of choices. The FMA units in NVIDIA GPUs can handle denormal numbers internally through hardware so that operating on denormal numbers does not introduce extra delay for multiply-add. However, for transcendental functions such as square root, detecting and handling denormal numbers incur considerable overhead. The aforementioned flag enforces denormal numbers being flushed to zeros, avoiding such overhead at the cost of potential degraded accuracy and underflow. Again, intelligent selection of rounding modes can speed up a program, but can also make it less accurate.

This variety points to the numerical reproducibility challenges waiting to be faced by anyone seeking performance tradeoffs. While the changes in rounding or handling denormals are rigorously specifiable, what becomes unknown is how large-scale codes tend to behave under these variations. Fasi et al. [14] summarize many such unspecified aspects of a GPU and say "*...from a numerical point of view, many essential aspects of tensor cores are not specified. Community effort is needed not only to detect and localize result-variability, but also prevent unwarranted variability that hinders the reliable porting of code.* The user community must also have means to gather and process these facts, even playing the role of a (well-intended) vigilante. We now list other aspects that tend to affect the numerics of the results computed by GPUs (some of these remarks are applicable to CPUs while others largely pertain to GPUs).

[2]Denormal numbers refer to floating points having leading zeros in their mantissas for avoiding underflow, as normalization would result in a very tiny exponent that is not representable.

TABLE I
FLOATING-POINT FORMAT SUPPORTED BY NVIDIA GPUS

Format	Exponent	Mantissa	Sign
Double precision (FP64)	11 bits	52 bits	1 bit
Single precision (FP32)	8 bits	23 bits	1 bit
Half precision (FP16)	5 bits	10 bits	1 bit
Tensor floating-point 32 (TF32)	8 bits	10 bits	1 bit
Brain floating-point 16 (BF16)	8 bits	7 bits	1 bit

a) Fused-Multiply-Add (FMA): The FMA feature is crucial for attaining high performance by combining two operations (+ and *) into one; unfortunately, the use of FMA can bring about another dimension of variability. The magnitude of FMA-induced variability is captured in climate codes [5] where it is noted that the availability of FMA (and its use by a compiler) affected existing codes to such an extent that the statistics of the climatic predictions were drastically affected, and went out of the acceptable realm of results. FMA itself is standardized:[3]in 2008, the IEEE-754 standard was revised

to add fused-multiply-add operation, which increases performance while eliminating one rounding step from otherwise two in FP multiply and FP add, improving accuracy. NVIDIA GPUs natively support FMA as a single instruction. By default, the GPU compiler will automatically seek opportunities to fuse the multiply and add operations sharing the same operand into an FMA instruction. However, this can be turned off through the *–fmad=false* compiler option, which might improve performance for those applications with high register pressure but low chance of jointly using multiply and add. This is because FMA requires three operands being available in registers simultaneously, which may further increase register pressure [15]. This is harmful if little benefit can be accrued through FMA.

b) Special Function Units (SFUs): The availability of SFUs gives another opportunity to affect performance tradeoffs—but under the scrutiny of rigorous correctness checking methods. Our previous work shows that the SFUs of GPUs can accelerate single-precision and double-precision numeric transcendental-function calculations through hardware implemented lookup tables [16], [17]. We published the available capabilities per operation in single and double precision, reporting experimental results pertaining to precision and latency. Also, a recent effort demonstrates that the V100 tensor cores adopted in the OLCF Summit supercomputer essentially implement higher precision than the claimed FP16, at least for FMA instructions [18].

A major problem with these capabilities is that these characteristics vary across GPUs, becoming obsolete every few years. At the same time, GPU-based software is becoming central to societally important areas such as helping fighting SARS-CoV-2 [3]. Such software uses ML methods based on subsystems such as variational auto-encoders that, in turn, have efficient implementations in half-precision (e.g., [19]). While these low precision uses are confined to ML, their use in HPC will bring up the more stringent acceptance criteria that are lacking today. As more such systems are built, their testing methods must keep up; currently we are far from this goal.

B. Exceptions

In traditional CPU-based programming languages and system software, several methods exist to detect floating-point exceptions (e.g., division by zero, overflows and others). For example, in Linux systems, when exceptions occur, one of two things can happen. By default, the exception is simply noted in the floating-point status word, and the program can check the status word to find out which exceptions happened. Alternatively, *traps* for exceptions can be enabled, which allows the program to receive the *SIGFPE* signal. While the default action for this signal is to terminate the program, the effect of the signal can be changed.

On the other hand, the support to detect floating-point exceptions in GPUs is limited and, in some cases, null. For example, NVIDIA GPUs have no mechanism to detect that a floating-point exception occurred according to the CUDA Programming Guide [20]. The guide also states other deviations of the CUDA implementation of floating-point arithmetic with respect to the IEEE-754 standard, including that double-precision floating-point absolute value and negation are not compliant with IEEE-754 with respect to NaNs.

Since mechanisms to detect floating-point exceptions that exist in CPU-based systems are not present in GPU systems, programmers are left with almost no option other than using `printf` statements in the application source code to catch the result of exceptions. This is a less-than-ideal method—printing in standard output from millions of threads can be slow and error prone. Furthermore, some exceptions can be miss-detected and affect control-flow without programmers even noticing them.

Recently, the FPChecker tool [21] was proposed to detect the result of exceptions in CUDA using an LLVM pass to instrument LLVM IR instructions. The tool detects operations that produce NaN, positive infinity, negative infinity, and subnormal numbers (underflows). While the tool has been effective in detecting several exceptions that were unknown to users, it has some limitations: (1) it cannot be used on all CUDA kernels (some limitations exist in clang/LLVM to compile CUDA); (2) it does not catch exceptions from tensor cores. Further work will be required to detect and handle floating-point exceptions as HPC applications are ported to heterogeneous systems that may not provide the same level of support as originally assumed.

C. Concurrency and Schedule-Dependent Numerics

Numerical result deviations can be caused by undetected data races as well as operator reassociations during reduction, as now elaborated.

a) Data Races: Data races are caused by incorrectly synchronized memory accesses [22], causing result variability. It is well known that codes that contain data races can produce unexpected results (even deadlocks [23]), and this can be exacerbated by compiler optimizations [24]. The current state-of-the-art in GPU data race checking is woefully inadequate. Commercial race checkers such as Nvidia's Cuda Memcheck are incomplete in many ways (for instance, they do not handle global memory data races). Despite the many academic race checkers for GPUs (summarized and studied in [10]) and more recent ones that offer compositional checking [25], race checkers for GPUs are impossible to reliably construct given that there is no rigorous specification accompanying GPU synchronization primitives that also tend to vary with GPUs. Another is the non-trivial engineering effort needed to build and maintain front-end tools that can parse realistic GPU codes. The ability to directly check data races on application source codes is necessary for practitioner acceptance of the tools. Currently established flows are largely for Clang/LLVM but typically cover only subsets of GPU programming constructs in use.

[3]There is also an element of confusion pertaining to FMA supported by CPUs that sometimes employ 80-bit internal registers.

b) Reassociations During Reduction: Many GPU codes employ reduction operations such as addition that are non-associative under the floating-point semantics. In some of the reduction algorithms [26], the reduction order can depend on the underlying GPU warp execution order. Some of the more recent algorithms [27] tend to avoid this issue. User discretion is thus required in selecting reduction algorithms for the situation at hand.

D. Compiler Effects

Traditional HPC heterogeneous systems provide several compilers, both commercial and open source. For example, DOE CORAL HPC systems, such as Sierra, provide at least five different compilers. Previous work has found that scientific applications' numerical results can be inconsistent when compiled with different compilers or optimization flags [28], [29]. A given code can be compiled with compiler x and run on the host, and compiled with compiler y and run on a device. When users want to compare the numerical results that the code produces on different architectures, a challenging question is: *what combination of compilers (x, y) would produce the most similar results in an HPC system?* In our previous work [28] we have shown that sometimes numerical results could be as different as zero and 1e+300, or NaN (not a number) and 0.1.

The key reason for such numerical inconsistencies is that most of the aggressive compiler optimizations, e.g., `-fast-math` in gcc and clang, violate IEEE floating-point semantics in exchange for faster code execution. These compiler optimizations make strong assumptions about the code's numerics, including that data is representable within the ranges of normal floating-point numbers, i.e., there are no subnormal numbers, infinity, or other extreme cases. While for some applications, operating on such extreme numerical ranges is acceptable, for others, it is not and it can lead to numerical inconsistency and reproducibility problems.

Our past work has contributed bisection-based methods for locating numerical variability at the file granularity [5], [30]. It has further contributed to source-line-level variability location search [29], [31]. Similar efforts are needed, covering accelerators.

a) Performance Portability Isn't Numerical Portability: The ever-increasingly popular *performance portability* models such as Kokkos [32] and RAJA [33] may help us mitigate the complexity of achieving portable performance in the context of compiling M different GPU codes targeting N different platforms. While Nvidia is the dominant force in the area of GPUs, other vendors will soon be delivering HPC-grade discrete GPUs. Unfortunately, the extent to which performance portability layers change the numerics is not studied. Accelerator code testing methods must be advanced to meet these needs.

E. Mixed Precision

Considerable attention has been devoted to the generation of CPU codes where only parts of the codebase are instantiated at higher precision. We have done some of the earliest work on precision tuning through the Precimonious tool [34], and have stayed active in this area through collaborations [35]–[38]. These approaches can potentially reduce data movement costs and still keep rounding errors under control by allocating precision where the computation demands it.

Unfortunately, mixed-precision GPU codes are very difficult to create and reliably maintain. First of all, altering precision wrecks the GPU memory layout discipline originally assumed in the full precision codes. These layout optimizations that promote coalesced memory accesses and avoid bank conflicts are crucial for GPU codes—whatever be the precision they are deployed at.

As an alternative to mixed precision, rewriting expressions can help improve the floating-point accuracy of the original expression without adjusting memory layout, and may also help eliminate problems such as overflow. A tool such as Herbie [39] can, for instance, conditionally change an expression $\sin x - \tan x$ to the more accurate $-(x^3 + x^5/4)/2$ when $|x| < 0.038$. While rewriting helps preserve the original memory layout, it has two drawbacks: (1) it directly impacts performance, putting in a conditional expression in lieu of the original expression; (2) selecting the expression to rewrite is an unsolved challenge, especially from the point of view of an overall codebase attempting to meet a given precision/performance tradeoff. To some extent, these downsides can be mitigated by incorporating cost estimates in the rewriting process and biasing the tool away from branches. But the two main drawbacks will still stand.

In [40], the authors extend Herbie to a new tool called PHerbie that combines precision tuning with rewriting, producing pareto-optimal versions of the original code. A tool such as PHerbie is a great beginning, but with many steps remaining to be accomplished in making the tool practical for GPU usage in HPC codes. First, PHerbie's cost model for producing candidate (new) expressions is the node-count of the new expression. Given the plethora of implementation options that GPUs support and the dominance of data movement costs over computational costs in many applications, more work is needed to extend PHerbie's cost model and also obtain real runtime measurements after the rewritten codes are installed.

Expression rewriting has one distinct advantage in the GPU context: it can often replace an expression that tends to overflow with another that does not overflow. This can be a huge advantage for GPU codes, given that GPUs cannot detect and report the overflow exception. As an example, we have prototyped the use of the *hypot* function (also suggested by Herbie) to compute the hypotenuse [41] observing the elimination of overflow for many input ranges—showing the advantages of rewriting in the GPU-context.

F. Testing Challenges

HPC software testing consumes a significant amount of an organization's resources. The basic problem begins with HPC not having a strong tradition of software testing. For instance, testers such as QuickCheck [42] have been ported to dozens

of programming languages, and Fuzzers [43] are the go-to tools for testing. For this to happen for HPC, concerted effort is needed, beginning with *clear interface specifications*—and HPC component interfaces are inherently very broad, poorly specified and poorly understood. The general challenges of writing portable GPU codes have been studied [44]; such studies must now be repeated given the passage of time.

III. Conclusions, Community Engagement

In this paper, we summarize how numerical issues are affected by the growing heterogeneity of hardware and software. While we did not intent our discussions to pertain to any specific company, it is clear that we ended up surveying what is dominant in the HPC arena today—namely products from Nvidia Inc. For the sake of completeness, we now discuss a few other GPU types.[4]

We have located two papers about AMD GPUs, both from the same group. In [45], the authors opine: *While there is a general feeling that things are a bit more "open source," the openness of AMD's software does not mean that their scheduling behavior is obvious, especially due to sparse, scattered documentation.* Their paper is an attempt to gather the disparate pieces of documentation into a single coherent source. In [46], they remark: *We argue that an open software stack such as ROCm may be able to provide much-needed flexibility and reproducibility in the context of real-time GPU research, where new algorithmic or analysis techniques should typically remain agnostic to the underlying GPU architecture. In support of this claim, we summarize how closed-source platforms have obstructed prior research using NVIDIA GPUs, and then demonstrate that AMD may be a viable alternative by modifying components of the ROCm software stack to implement spatial partitioning.* A paper on Intel GPUs [47] talks about accelerating encrypted computing.

Community Engagement: The issues discussed thus far clearly bring out the need for better information exchange between GPU vendors and GPU users. Given the volume of information coming out, various community forums must also be formed. In addition to English descriptions of various issues, one must also try and develop pithy examples that highlight the issues being discussed. These organizations can also distribute reliable idioms for use by programmers, thus avoiding needless and error-prone reinvention.

Perhaps one of the best ways to respond to the upcoming challenges in the upcoming era of heterogeneity is to organize a collection of *proxy applications* (similar to ECP Proxy Applications [48]) that allows the community to develop and evaluate correctness checking tools. In [49], under the auspices of a DOE-sponsored project, we have begun collecting proxy applications that emphasize the use of GPUs. We also document tool design efforts that are in progress in our group. We also welcome participation through a forum called FPBench [11] under the auspices of which we hold monthly community meetings, to which we welcome the interested. We are eager to find out what similar organizational activities exist elsewhere and are interested in joining forces.

[4]We deliberately avoid discussing GPUs and accelerators in the mobile and embedded space where one might find significantly more variety.

Acknowledgements: The authors gratefully acknowledge funding from the DOE Award DE-FOA-0002460: X-Stack: Programming Environments for Scientific Computing under which the ComPort effort (this work) is being carried out.

References

[1] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. H. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston, "Exascale deep learning for climate analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 2018, pp. 51:1–51:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291724

[2] F. D. Natale, H. Bhatia, T. S. Carpenter, C. Neale, S. K. Schumacher, T. Oppelstrup, L. Stanton, X. Zhang, S. Sundram, T. R. W. Scogland, G. Dharuman, M. P. Surh, Y. Yang, C. Misale, L. Schneidenbach, C. Costa, C. Kim, B. D'Amora, S. Gnanakaran, D. V. Nissley, F. H. Streitz, F. C. Lightstone, P. Bremer, J. N. Glosli, and H. I. Ingólfsson, "A massively parallel infrastructure for adaptive multiscale simulations: modeling RAS initiation pathway for cancer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, M. Taufer, P. Balaji, and A. J. Peña, Eds. ACM, 2019, pp. 57:1–57:16. [Online]. Available: https://doi.org/10.1145/3295500.3356197

[3] L. Casalino, A. Dommer, Z. Gaieb, E. P. Barros, T. Sztain, S.-H. Ahn, A. Trifan, A. Brace, A. Bogetti, H. Ma, H. Lee, M. Turilli, S. Khalid, L. Chong, C. Simmerling, D. J. Hardy, J. D. C. Maia, J. C. Phillips, T. Kurth, A. Stern, L. Huang, J. McCalpin, M. Tatineni, T. Gibbs, J. E. Stone, S. Jha, A. Ramanathan, and R. E. Amaro, "Ai-driven multiscale simulations illuminate mechanisms of sars-cov-2 spike dynamics," *bioRxiv*, 2020. [Online]. Available: https://www.biorxiv.org/content/early/2020/11/20/2020.11.19.390187

[4] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai, and N. Rao, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1740–1750.

[5] D. H. Ahn, A. H. Baker, M. Bentley, I. Briggs, G. Gopalakrishnan, D. M. Hammerling, I. Laguna, G. L. Lee, D. J. Milroy, and M. Vertenstein, "Keeping Science on Keel When Software Moves," *Commun. ACM*, vol. 64, no. 2, p. 66–74, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3382037

[6] A. Fox, J. Diffenderfer, J. Hittinger, G. Sanders, and P. Lindstrom, "Stability analysis of inline ZFP compression for floating-point data in iterative methods," *SIAM J. Sci. Comput.*, vol. 42, no. 5, pp. A2701–A2730, 2020. [Online]. Available: https://doi.org/10.1137/19M126904X

[7] X. Zou, T. Lu, W. Xia, X. Wang, W. Zhang, H. Zhang, S. Di, D. Tao, and F. Cappello, "Performance optimization for relative-error-bounded lossy compression on scientific data," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 7, pp. 1665–1680, 2020. [Online]. Available: https://doi.org/10.1109/TPDS.2020.2972548

[8] K. Zhao, S. Di, X. Liang, S. Li, D. Tao, J. Bessac, Z. Chen, and F. Cappello, "Sdrbench: Scientific data reduction benchmark for lossy compressors," *CoRR*, vol. abs/2101.03201, 2021. [Online]. Available: https://arxiv.org/abs/2101.03201

[9] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, "Compiling kb-sized machine learning models to tiny iot devices," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 79–95. [Online]. Available: https://doi.org/10.1145/3314221.3314597

[10] L. van den Haak, A. Wijs, M. van den Brand, and M. Huisman, "Formal methods for gpgpu programming: Is the demand met?" in *Integrated Formal Methods - 16th International Conference, IFM 2020, Proceedings*, ser. Lecture Notes in Computer Science (including subseries Lecture

Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), B. Dongol and E. Troubitsyna, Eds. Germany: Springer, 2020, pp. 160–177, 16th International Conference on Integrated Formal Methods, IFM 2020 ; Conference date: 16-11-2020 Through 20-11-2020.
[11] http://fpbench.org/.
[12] A. Li and S. Su, "Accelerating binarized neural networks via bit-tensor-cores in turing gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1878–1891, 2020.
[13] B. Feng, Y. Wang, T. Geng, A. Li, and Y. Ding, "Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores," *arXiv preprint arXiv:2106.12169*, 2021.
[14] M. a. d. H. N. Fasi, M. Mikaitis, and P. S, "Numerical behavior of nvidia tensor cores," 2021. [Online]. Available: https://doi.org/10.7717/peerj-cs.330
[15] A. Li, S. L. Song, A. Kumar, E. Z. Zhang, D. Chavarría-Miranda, and H. Corporaal, "Critical points based register-concurrency autotuning for gpus," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1273–1278.
[16] S. F. Oberman and M. Y. Siu, "A high-performance area-efficient multifunction interpolator," in *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*. IEEE, 2005, pp. 272–279.
[17] A. Li, S. L. Song, M. Wijtvliet, A. Kumar, and H. Corporaal, "Sfu-driven transparent approximation acceleration on gpus," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–14.
[18] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, and Y. Ding, "Egemm-tc: accelerating scientific computing on tensor cores with extended precision," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 278–291.
[19] M. Seznec, N. Gac, A. Ferrari, and F. Orieux, "A study on convolution using half-precision floating-point numbers on gpu for radio astronomy deconvolution," in *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2018.
[20] NVIDIA, "CUDA C++ Programming Guide, v11.4," https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2019, online; accessed August, 10, 2021.
[21] I. Laguna, "FPChecker: Detecting floating-point exceptions in GPU applications," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1126–1129.
[22] R. Netzer and B. P. Miller, "What are race conditions? some issues and formalizations," *ACM Letters on Programming Languages and Systems*, 1992.
[23] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-grained synchronizations and dataflow programming on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 109–118.
[24] S. Atzeni, G. Gopalakrishnan, Z. Rakamarić, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "Archer: Effectively spotting data races in large OpenMP applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 53–62.
[25] T. Cogumbreiro, J. Lange, D. L. Z. Rong, and H. Zicarelli, "Checking data-race freedom of gpu kernels, compositionally," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 403–426.
[26] [Online]. Available: https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/
[27] [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
[28] I. Laguna, "Varity: Quantifying Floating-Point Variations in HPC Systems Through Randomized Testing," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020.
[29] H. Guo, I. Laguna, and C. Rubio-González, "pliner: Isolating lines of floating-point code for compiler-induced variability," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, pp. 680–693.
[30] M. Bentley, I. Briggs, G. Gopalakrishnan, D. H. Ahn, I. Laguna, G. L. Lee, and H. E. Jones, "Multi-Level Analysis of Compiler-Induced Variability and Performance Tradeoffs," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC 19. ACM, June 2019, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/3307681.3325960
[31] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error," *SIGPLAN Not.*, vol. 53, no. 4, p. 256–269, Jun. 2018. [Online]. Available: https://doi.org/10.1145/3296979.3192411
[32] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, p. 3202–3216, Dec. 2014. [Online]. Available: https://doi.org/10.1016/j.jpdc.2014.07.003
[33] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.
[34] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning Assistant for Floating-Point Precision," in *Supercomputing (SC)*, 2013, pp. 27:1–27:12, https://github.com/corvette-berkeley/precimonious.
[35] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 333–343. [Online]. Available: https://doi.org/10.1145/3213846.3213862
[36] fpanalysistools.org.
[37] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough, "Floating-point precision tuning using blame analysis," in *ICSE*. ACM, 2016, pp. 1074–1085.
[38] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous Floating-Point Mixed-Precision Tuning," in *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 2017, pp. 300–315. [Online]. Available: https://doi.org/10.1145/3009837.3009846
[39] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically Improving Accuracy for Floating Point Expressions," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015*. ACM, 2015, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737959
[40] B. Saiki, O. Flatt, Z. Tatlock, P. Panchekha, and C. Nandi, "Combining precision tuning and rewriting for faster, more accurate programs," in *ARITH: 28th IEEE Symposium on Computer Arithmetic*, 2021. [Online]. Available: http://arith2021.arithsymposium.org/
[41] [Online]. Available: https://en.wikipedia.org/wiki/Hypot
[42] K. Claessen and J. Hughes, "A lightweight tool for random testing of haskell programs," in *International Conference on Functional Programming*, 2000.
[43] B. Miller, M. Zhang, and E. Heymann, "The relevance of classic fuzz testing: Have we solved this one?" *IEEE Transactions on Software Engineering*, 2020, accepted for publication in 2021; copy posted at http://pages.cs.wisc.edu/~bart/fuzz/.
[44] M. Leeser, D. Yablonski, D. H. Brooks, and L. A. S. King, "The challenges of writing portable, correct and high performance libraries for gpus," *SIGARCH Comput. Archit. News*, vol. 39, no. 4, pp. 2–7, 2011. [Online]. Available: https://doi.org/10.1145/2082156.2082158
[45] N. Otterness and J. H. Anderson, "Exploring amd GPU scheduling details by experimenting with "worst practices"," in *29th International Conference on Real-Time Networks and Systems*, ser. RTNS'2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 24–34. [Online]. Available: https://doi.org/10.1145/3453417.3453432
[46] ——, "AMD GPUs as an alternative to Nvidia for supporting real-time workloads," dOI 10.4230/LIPIcs.ECRTS.2020.12, Article No. 12; pp. 12:1–12:23, Leibniz International Proceedings in Informatics, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany.
[47] Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyashevsky, "Accelerating encrypted computing on intel gpus," 2021.
[48] https://proxyapps.exascaleproject.org/app/.
[49] https://xstack-fp.github.io/.