SAND2022-10893C

Half-Precision Scalar Support in Kokkos and Kokkos Kernels: An Engineering Study and Experience Report

Evan Harvey, Reed Milewicz, Christian Trott, Luc Berger-Vergiat, Siva Rajamanickam
eharvey@sandia.gov,rmilewi@sandia.gov,crtrott@sandia.gov,lberge@sandia.gov,srajama@sandia.gov
Center for Computing Research
Sandia National Laboratories
1450 Innovation Pkwy SE
Albuquerque, New Mexico 87123

***Abstract*—To keep pace with the demand for innovation through scientific computing, modern scientific software development is increasingly reliant upon a rich and diverse ecosystem of software libraries and toolchains. Research software engineers (RSEs) responsible for that infrastructure perform highly integrative work, acting as a bridge between the hardware, the needs of researchers, and the software layers situated between them; relatively little, however, has been written about the role played by RSEs in that work and what support they need to thrive.**

To that end, we present a two-part report on the development of half-precision floating point support in the Kokkos Ecosystem. Half-precision computation is a promising strategy for increasing performance in numerical computing and is particularly attractive for emerging application areas (e.g., machine learning), but developing practicable, portable, and user-friendly abstractions is a nontrivial task. In the first half of the paper, we conduct an engineering study on the technical implementation of the Kokkos half-precision scalar feature and showcase experimental results; in the second half, we offer an experience report on the challenges and lessons learned during feature development by the first author. We hope our study provides a holistic view on scientific library development and surfaces opportunities for future studies into effective strategies for RSEs engaged in such work.

***Index Terms*—research software engineering, scientific computing, software libraries, floating-point arithmetic**

I. Introduction

The present era of high performance computing (HPC) scientific computing is one of relentless innovation in computing capabilities, with scientific software developers having to adapt to ever-changing circumstances. From continuing advances in field programmable gate arrays (FPGAs) and graphics processing units (GPUs) to emerging technologies such as neuromorphic and quantum computing, we anticipate many more cycles of disruption in HPC hardware and software environments. From a software engineering perspective, finding better ways to manage that complexity and facilitate the evolution of scientific codes as they transition from one computing platform to another is key to the sustainability and productivity of HPC scientific software projects.

The overarching goal of the Kokkos Ecosystem is to provide a programming model, math kernels, and utilities engineered for *performance portability*, that is, maximizing the amount of user code that can be compiled for diverse devices and obtain comparable performance as a variant of the code that is written specifically for that device [1]. The primary components of the Kokkos Ecosystem are Kokkos Core [2], [3], Kokkos Kernels, and Kokkos Tools. Kokkos Core is a programming model consisting of thread-safe data structures, parallel programming constructs, and parallel algorithms which aim to abstract hardware architecture nuances for application developers[4]; Kokkos Kernels is a collection of math kernels providing BLAS, Sparse BLAS and Graph kernels which aims to provide performance portable, shared memory, parallel linear algebra kernels[5]; lastly, Kokkos Tools provides profiling and debugging utilities for Kokkos users.

As part of a Kokkos Kernels development effort to provide a batched-GEMM host-level interface for machine learning use-cases, experimental half-precision scalar support was added to Kokkos Core. Machine learning is a growth area for scientific computing, and our team seeks to stay ahead of the curve by providing useful features to the community. In this paper, we present two views on the development of half-precision support for Kokkos: (1) an engineering research study on the design and implementation of the feature and (2) an experience report, from a research software engineering (RSE) perspective, on the challenges and lessons learned in developing such features for the scientific software community.

Accordingly, the first half of the paper (Section II) presents technical details on the implementation of the half-precision capability and showcases experimental results. The second half of the paper (Section III) seeks to critically analyze the effort that went into that feature, as a way of illustrating the work of RSEs in library development. Finally, Section IV summarizes the key results of the paper and outlines directions for future

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

work.

II. Engineering Study

The engineering study is structured as follows. Subsection II-A describes the recent history of hardware support for reduced-precision computations, Subsection II-B presents related work on the use of half-precision floating point in scientific computing and machine learning, Subsection II-C details the implementation, and finally Subsection II-D offers examples of how the choice to use half-precision support in Kokkos affects performance.

A. Background

Host and device processors generally support two different floating point formats for half-precision computations: `binary16` and `bfloat16`. `binary16` is the IEEE 754 half-precision binary floating-point format with one signed bit, five exponent bits and 10 fraction bits. `bfloat16` was invented by Google (see [6]) and is based on the IEEE 754 single-precision floating-point format with one signed bit, eight exponent bits, and seven fraction bits. There are trade offs to selecting a format for your application. `binary16` has higher precision (3 more fraction bits) than `bfloat16`; while `bfloat16` has roughly the same range as float32. An illustration of the differences between these formats is shown in Figure 1.

Both `binary16` and `bfloat16` can reduce any algorithm's memory footprint, bandwidth congestion, and computational intensity. For algorithms targeting accelerators, simply loading half-precision input on the host and copying it to the accelerator can provide a speedup at initialization. In some cases, half-precision enables very large input sets to fit in accelerator memory. Half-precision types provide lower latency and higher throughput arithmetic operations. On Nvidia's V100 GPU, `binary16` support provides user's with additional parallelism via tensor cores, thereby making available up to 120 TFLOPS via CUDA-defined WMMA tile sizes [7]. Hardware and toolchain support for `binary16` and `bfloat16` varies across vendors. Some toolchain vendors support one or more 16-bit floating point types as a storage only type such that assembly instructions for load and store operations will be emitted by the compiler while any C++ arithmetic operator will generate a compiler error. Other vendors support one or more 16-bit floating point types and will generate arithmetic instructions. Vendors such as Nvidia and AMD provide compiler intrinsics for casting to and from the selected half-precision type. Some GPUs support `binary16` while others such as NVIDIA's A100 supports both `binary16` and `bfloat16`. Some CPUs support `binary16` as well. One particular problem of the GPU support for half-precision types is that arithmetic operations are only defined in device code. Furthermore, the GPU `binary16` and CPU `binary16` are not interoperable at the language level. Table I summarizes several vendor's support for half-precision.

Half-precision scalars for both `binary16` and `bfloat16` were released in Kokkos version 3.6.00. Kokkos provides two processor-agnostic, reduced-precision types: `half_t` and `bhalf_t` which encapsulate `binary16` and `bfloat16`, respectively. These types are intended to be used in Kokkos applications and are continuously unit-tested in Kokkos Core as well as in the batched-GEMM Kokkos Kernel. These Kokkos types have already been used to run experiments within the GMRES example in Kokkos Kernels[16] and to enable initial tensor code support in the SPMV Kokkos Kernel. Any C++ developer may try using these experimental types by including `Kokkos_Half.hpp` in their C++ code.

B. Related Work

Computational science and machine learning researchers interested in utilizing half-precision to optimize and scale their algorithms, could benefit from the addition of half-precision support.

a) Computational Science: Utilizing mixed precision arithmetic to achieve better performance has been a topic of interest for several computational science applications. Two recent reports from the Department of Energy's Exascale Computing project summarizes several works related to the use of mixed precision within the codes in this project [17]. Several methods developed in this domain focus on using mixed-precision without much loss of accuracy. Loe et al. [18] and Lindquist et al. [19] looked into using single precision to accelerate their mixed-precision Generalized Minimum Residual solver using an approach called the GMRES-IR where iterative refinement is used to recover the loss of accuracy. Mixed-precision approaches have also been utilized within Block-Low-Rank methods to store singular vectors corresponding to small singular values in lower precision to reduce the memory footprint and floatng point operations needed for the factorizations [20]. Gratton et al. [21] take a different approach to show how the precision of the inner-products in the Generalized Minimum Residual solver can be reduced as the iterations proceed, without affecting the accuracy of the solver.

b) Machine Learning: In recent years, there have been substantial gains in artificial intelligence (AI) and machine learning (ML) performance by increasing the size of models and their datasets. Case in point, in realm of AI/ML for NLP, Brown et al. demonstrated with GPT-3 that scaling up the number of parameters in language models can lead to dramatic improvements in task-agnostic performance[22]; GPT-1 had 115 million parameters, GPT-2 1.5 billion, and GPT-3 175 billion. As models grow in size, however, they require more compute and memory resources to train, becoming prohibitively expensive and introducing new challenges (*e.g,* in deployment).

Mixed precision strategies, including the use of half-precision formats, have been explored as a way of simultaneously reducing training time and model size while minimizing loss of model accuracy. Micikevicius et al. used `binary16` to scale their neural networks for improved accuracy[23]. Kalamkar et al. demonstrated that `bfloat16` could be substituted for 32-bit floating point to match state-of-the-art results

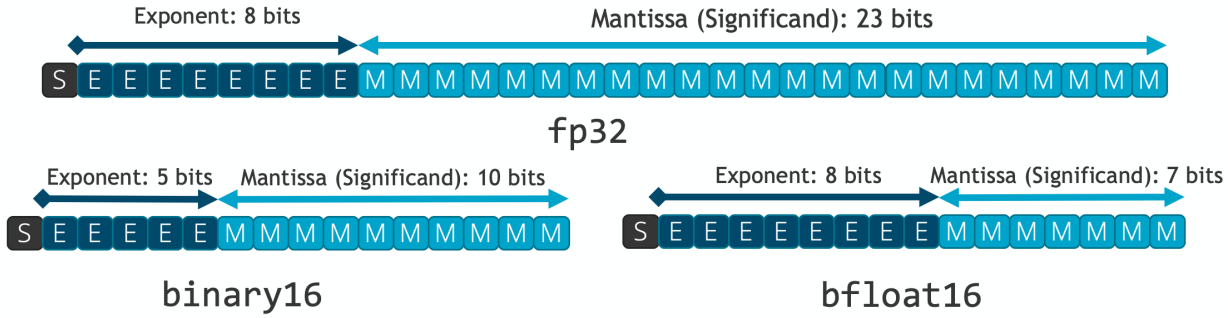

Fig. 1: An illustration comparing `binary16` and `bfloat16` floating-point representations to the IEEE standard 32-bit implementation.

Architecture	Toolchains	`binary16`	`bfloat16`	Storage Only	Casting Intrinsics	Type Names	Further Reading
Intel 64	llvm 10+, intel 19+, gcc 7+	Yes	No	Depends	Yes	`_Float16`, `__fp16`	[8][9][10]
ARM V8 ThunderX	llvm 9+, arm 20+	Yes	No	Depends	No, toolchain casts	`_Float16`, `__fp16`	[10][11]
Volta70	Cuda 10+	Yes	Toolchain-only	No	Yes	`__half`	[12][13]
Amphere	Cuda 11+	Yes	Yes	No	Yes	`__half`, `__nv_bfloat16`	[14][15]

TABLE I: A summary of toolchain support for half-precision by different vendors.

across several domains within the same number of training iterations and with no changes to hyper-parameters[24]. Meanwhile, newer GPUs such as the Nvidia V100 provide native `binary16` support in tensor cores which has been used to accelerate neural network training[7].

c) Half-Precision Support in Python, Matlab, and Julia: Outside of the Kokkos Ecosystem, half precision abstractions exist in NumPy, MatLab and Julia. NumPy v1.6.0 introduced a binary16 type via `numpy.half` and `numpy.float16` [25]. MatLab and Julia also support a `binary16` type via the `half` [26] and `Float16` [27] typenames. None of these ecosystems provide support for bfloat16. An initial literature review indicates that researchers are utilizing third-party half precision abstractions rather than the native NumPy, MatLab or Julia half precision support.

C. Implementation

In this section, we describe the design decisions and implementation details for half-precision support within the Kokkos Ecosystem.

1) Design of half-precision types: We introduce `half_t` and `bhalf_t` as portable half-precision types in Kokkos. They're implemented as instantiations of a single templated class called `floating_point_wrapper` which encapsulates a single private data member that is aliased to the underlying toolchain type[1]. The Kokkos `floating_point_wrapper` class uses operator overloads as well as implicit and explicit conversion constructors in an effort to behave as closely as possible to any other C++ built-in floating point type. To address the missing half-precision arithmetic intrinsics for the GPU toolchains, `floating_point_wrapper` has distinct host and device operator overloads for each arithmetic function. The host overload casts the internal data member to and from `float` for arithmetic operations. The device overload uses the available arithmetic intrinsic operations. In order to enable portability to platforms where no underlying hardware or toolchain support for half-precision exists, we provide a fall-back implementation where `half_t` and `bhalf_t` are aliased to `float`.

C++ operator overloads and conversion constructors provide many but not all the abstractions that a compiler writer can provide for a built-in type. The Kokkos `floating_point_wrapper` class does not support implicit conversion from the underlying toolchain type to another built-in type. This implicit casting to built-in types enables expressions like `double OP float`, where `float` is implicitly upcast to `double`. This cannot be abstracted in C++ for all operations on `half_t` and `bhalf_t`. If we were to allow implicit conversion from `floating_point_wrapper` to `float`, this would result in a compile-time ambiguity for all C++ expressions involving `half_t` or `bhalf_t` with another C++ built-in type. This compile time ambiguity arises since the compiler considers all permutations of the expression `float OP half_t` which are: `float OP half_t` and `float OP float`. Even though `float OP half_t` simply upcasts the right hand side and returns the floating point format, we have no way as user's of a C++ compiler, to assign a weight to resolve these two conflicting permutations. User's can be slightly more verbose when using `half_t` and `bhalf_t` by writing `float = static_cast<float>(half_t)` which then utilizes the explicit conversion constructors in the `floating_point_wrapper` class and also works with

[1] *e.g.,* __half for CUDA11+

any built-in right hand size type, `T`.

2) Backend plugin mechanism: The Kokkos half-precision implementation uses forward declarations to hide the toolchain-specific intrinsic function names from the half-precision operators and constructors. By providing a set of common, type-agnostic wrappers in `floating_point_wrapper`, every Kokkos Core backend need only define their internal casting wrappers and type alias to their underlying toolchain type in order to enable half-precision scalars in a sustainable and reproducible manner across Kokkos backends. Backends may choose to provide a explicit specialization of `floating_point_wrapper`, however, that has not been required yet. Table II summarizes Kokkos version 3.6.00 backend support for half-precision scalars.

Kokkos Backend	`half_t`	`bhalf_t`
CUDA	Yes	Yes
SYCL	Yes	No
HIP	Yes	No

TABLE II: Kokkos backend support for half-precision scalars.

3) Testing: The Kokkos half-precision support is both unit and integration tested. To facilitate checking results, we define constants including `FP16_EPSILON` and `BF16_EPSILON`. Within Kokkos Core, we launch unit testing on both the host and device within the default test execution space. This unit testing is broken into two files: `TestHalfConversion.hpp` and `TestHalfOperators.hpp` where we aim for full coverage of both the casting wrappers and `floating_point_wrapper` class. Within Kokkos Kernels, we launch integration testing of half-precision on all targeted backends for every batched-GEMM kernel that we support. In this way, the Kokkos Kernels batched-GEMM unit tests also serves as an integration test for the Kokkos half-precision support. Both Kokkos Core and Kokkos Kernels use continuous integration tests and nightly tests on the develop branch to guard against regressions.

D. Performance Evaluation

To demonstrate the usability and performance impact of half-precision support in Kokkos, we modified the conjugate gradient solver (CGSOLVE) benchmark code used in [3]. The problem run in CGSolve is a simple heat conduction problem with a cubic grid of cells. CGSolve performs three math kernels: sparse matrix vector multiply (SPMV), vector addition (AXPBY), and an inner product (DOT). There are different possibilities for using half-precision and mixed precision. The four critical possibilities for distinct choices of what precision to use are: (i) value type of the matrix, (ii) value type of vectors, (iii) accumulation precision for DOT, and (iv) accumulation precision for the inner reduction over a matrix row in SPMV. We modified the code so that each of those types can be chose independently via a compile time choice.

It is worth noting that the CGSolve is numerically unstable, when using anything but full double-precision, if run for many iterations. Since we are mostly interested in a performance and functionality analysis, and are not doing an algorithmic investigation, we simply chose to restrict runs to 10 iterations.

Four configurations were investigated: all double (dddd), all float (ffff), matrix and vector value types using `half_t` but reductions done in float (hhff), and everything but the dot product using `half_t` (hhfh). It is worthwhile noting, that for the dot product, only the accumulation is done with single-precision in the last case, the actual memory loads and the multiplication of two values from the two vectors is done in the vector scalar precision (i.e. half-precision).

We ran problem sizes from 200^3 to 100^3. In figure 2, the x-axis is the cubic number of cells (N3) and the y-axis is the average runtime.

Generally, we observe the expected trends: full double-precision runs are the slowest, half-precision runs the fastest, and single-precision falls in between. We also see expected linear scaling with the problem size. However, we do not observe a 4x and 2x theoretical improvement over double-precision, respectively. This isn't too surprising. While the total memory movement requirements go down significantly, SPMV (which is by far the most expensive operation) requires the load of integral-typed indices from memory, which do not change in size. Thus, assuming perfect caching of the vectors, the expected memory movement from global memory is only reduced by a factor of two when using half-precision instead of double-precision. Furthermore, the SPMV data access on the right hand side is somewhat random, making access latencies and metrics such as number of possible simultaneous memory operations important. Last but not least, to gain the full benefit of half-precision types on GPUs one has to utilize packed half-precision operations.

All in all we see about a 20% reduction in kernel runtime from using single-precision instead of double-precision, and another 10-20% from using half-precision instead of single-precision. It is worthwhile to note, that there was no performance benefits from doing the inner reductions in half-precision, which occurs within a single warp on the GPU. The reduction performance is largely determined by the shuffle operation and warp synchronization latencies.

As an example of a kernel where half-precision types potentially offer larger performance improvements, we benchmarked a batched matrix-matrix multiplication kernel, the original motivation for this work. This kernel's flops to bytes ratio depends on the matrix size. In the limit of matrix size being 1x1, the bytes to flops ratio is the same as for vector add (AXPBY). At larger matrix sizes, the kernel becomes more and more compute intensive. However, the matrix size regime our users are interested in never reaches a size where the code is compute bound.

Moreover, our implementation of batched-GEMM uses two different algorithms for different size regimes. For the smallest matrices, each output matrix element in C of a given matrix-matrix multiply is performed by a single thread. In an inter-

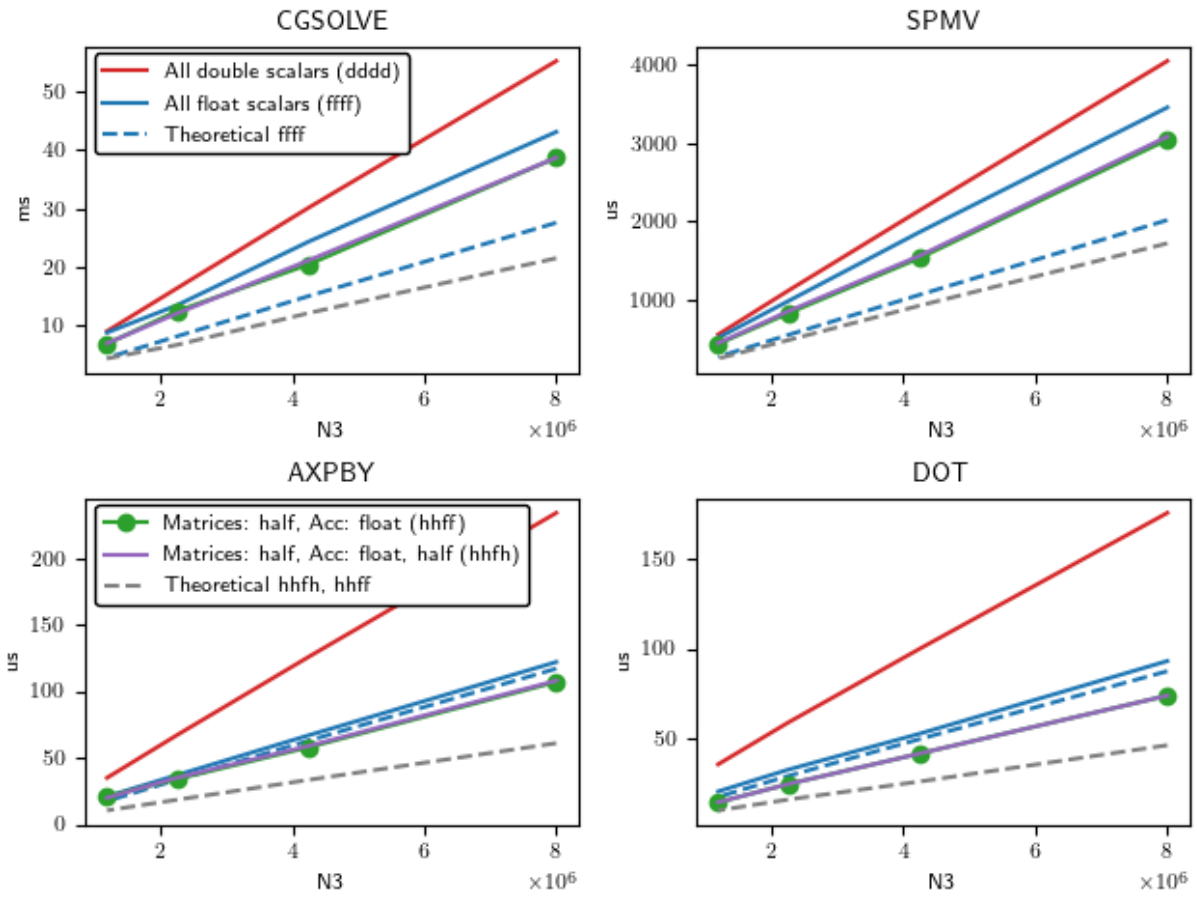

Fig. 2: Average runtime of CGSolve, SPMV, AXPBY, and DOT with 10 timed iterations on Nvidia V100. The legends apply to all four subplots and refer to the precision choices explained in II.D, (i) - (iv); "Matrices:" refer to matrix and vector value types and "Acc:" refer to accumulation value types.

mediate size regime we use a double buffering scheme with a team of threads per matrix, before switching back to the original scheme. As with the CGSolve performance evaluation above, the only code change between `float`, `half_t` and `bhalf_t` is the chosen template argument for the scalar type.

The observed performance demonstrates that for small matrices, the kernel is purely latency limited; `float`, `half_t` and `bhalf_t` have the same throughput. Then there is a middle regime, where we see `half_t` performing better than `float` on V100, but not on A100. We suspect that on V100 we run out of L1 cache for `float`, while `half_t` still fits. The increased L1 cache size on A100 makes `float` fit too.

At the larger sizes, the increased concurrency kicks in, and the kernel is finally able to saturate more of the available bandwidth. At that point, the smaller size of `half_t` and `bhalf_t` leads to performance gains in a similar range to what was observed in the bandwidth limited CGSolve.

In Figures 3 and 4, we perform 10 untimed warmup runs followed by 20 timed runs on square matrix sizes, all with a batch size of 1024.

To reproduce the performance results in this section, we have provided the table below with supplementary information the footnote below. [2]

TABLE III: Dependency information for reproducing results from our computational experiments

Type of Dependency	Libraries/Versions Used
Kokkos Software Versions	kokkos-3.6.00, kokkos-kernels@us-rse-escience-2022, code-examples@us-rse-escience-2022.
V100 Third Party Libraries (TPLs)	`cmake-3.19.1`, `gcc-8.3.0`, `cuda-11.2.142`.
A100 TPLs	`cmake-3.22.0`, `gcc-7.5.0`, `cuda-11.2.142`.

III. Experience Report

In the previous sections, we described the implementation details of half-precision support within the Kokkos Ecosystem

[2]For further guidance on how to reproduce the results from our paper, consult the instructions found at the following links:
CGSolve reproducer instructions.
bathced-GEMM reproducer instructions.

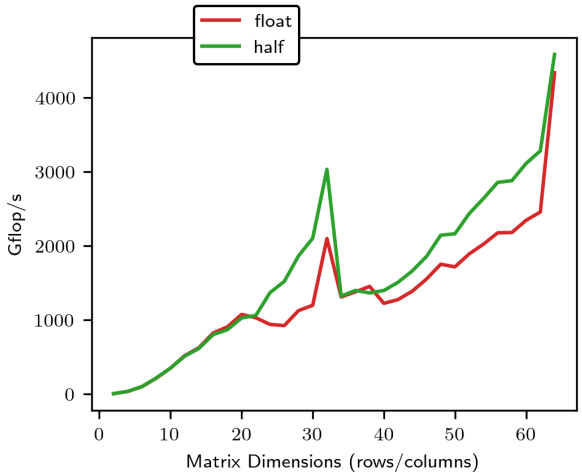

Fig. 3: Runtime of batched-GEMM with 20 timed iterations on Nvidia V100.

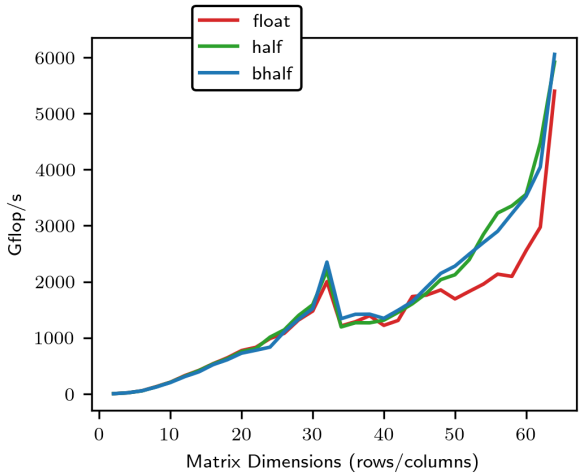

Fig. 4: Runtime of batched-GEMM with 20 timed iterations on Nvidia A100.

and provided preliminary results to demonstrate the trade-offs offered to end-users. Now we intend to shift gears and turn our attention to the RSE perspective on the contribution. The term of research software engineer has grown in popularity to encompass a diverse range of software professionals in the scientific domain [28], [29], [30], but relatively little has been written about how RSEs carry out their highly interdisciplinary work and what their needs are. In the case of middleware library development like Kokkos, RSEs must carefully consider the needs of researchers, the capabilities of the baremetal hardware, and the engineering of the abstractions between them. A better understanding of how RSEs navigate these rich and diverse problem spaces may help inform training, tool development, and organizational support for RSEs and their work.

To that end, this paper presents an self-reflective experience report on the development of half-precision support in Kokkos. The first author, Evan Harvey, is a research software engineer (RSE) in the Department of Software Engineering and Research at Sandia National Laboratories[31][32], who was principally responsible for implementing the feature. He describes challenges and lessons learned, while Reed Milewicz, the second author on this work and a software engineering researcher has supplemented these experiences by drawing connections and comparisons to the available literature.

The experience report is structured as follows: Subsection III-A catalogs challenges faced by the first author (an RSE) alongside commentaries from the second author (a software engineering researcher) on those challenges, and Subsection III-B captures lessons learned that can be carried forward to future development activities.

A. Development Challenges

a) Managing Evolving Requirements: A formal requirements elicitation was performed for the batched-GEMM work with a researcher in our center who needed the feature. Of note, Reed (the second author), had previously conducted a rapid literature review for me on requirements elicitation in domain-specific contexts (see [33]); he created a guidebook on requirements gathering techniques and when to apply them, and this proved very helpful.

After prototyping a half-precision type alias in Kokkos Kernels without operator overloading and demonstrating that batched-GEMM would run on Intel, AMD, and CUDA, it was clear that a simple type alias would not be portable nor usable for Kokkos users. At this point, we saw value in expanding half-precision support beyond the batched-GEMM use-case and decided to use both casting wrappers and operator overloads rather than a type alias. While another requirements gathering effort was not performed for half-precision support, there was one guiding requirement: to implement a `half_t` type that behaves as closely as possible to `float`. As we implemented the expanded half-precision support, we found that C++ cast operators could be supported via explicit conversion constructors, getting us closer to mimicking the behavior of a C++ built-in type.

Commentary: Smith et al. has argued that there has long been a prevailing belief that requirements gathering is impractical or infeasible for most scientific software development, when in truth they are "no more challenging than for any other domain – requirements are difficult for everyone" [34]. Indeed, requirements engineering is especially critical when developing scientific software libraries for the community. For RSEs developing such software, knowing how to effectively interact with researchers to ascertain those requirements is an important competency, but such soft skills are not covered by the training most RSEs have received [35].

The communication barriers faced by RSEs are well-attested in the literature. According to Jay et al., "many common software engineering challenges, such as requirements gathering, communication difficulties[etc. ...] are particularly challenging— and often qualitatively different—within science, due

to the nature of the research process, and the environment in which the work is conducted"[36]. RSEs and the researchers they work with come from different backgrounds and perspectives, which Chue Hong et al. note " highlights a need for varied skills and good communication [...] Researchers are more likely to be working with others who have different technical expertise, use different technical terminology, and may be communicating in a tertiary language"[37].

Here, Evan calls attention to benefiting from a rapid review, which is a time-boxed literature review meant to deliver actionable guidance to practitioners in the field [38]. A review of the literature on requirements elicitation surfaced recommendations for how to apply such techniques in domain-specific contexts, particularly with respect to interviews, surveys, scenarios, brainstorming exercises, and ethnographic techniques. Training on these kinds of requirements gathering techniques may help to reduce friction between RSEs and the scientific software teams they work with.

Challenge Observation 1: Soft skills, such as when working with researchers to gather software requirements, are important to RSEs and their work, but few have specific training in these skills. Targeted training on soft skills like requirements gathering techniques may help RSEs in their work with scientific software teams.

b) Balancing User Expectations and Hardware Realities: Many of the challenges we faced when implementing `half_t` arose in 3 categories: implicit casts, volatile overloads, and parallel reduction support. We knew that researchers were already using mixed precision expressions with `float` and `double` and, in an effort to provide a strong abstraction, we added friend operators to support mixed precision on binary arithmetic operations involving `half_t` and `float` or `double`. We also added implicit conversion constructors in order to downcast integral types to `half_t` and allow toolchains to reuse the existing `half_t` operator overloads. At this point, we had still preserved `half_t` as a trivially copyable type. However, the Kokkos parallel constructs such as `parallel_reduce` required both byte alignment and support for volatile stores and loads. During this phase, the Kokkos CUDA backend did not yet support reductions on types less than 4-bytes in size.

Next, to support CUDA reductions, we decided to first add support for volatile loads and stores and then make a second pass to reduce the `half_t` storage requirement back to 2-bytes. Performing volatile loads and stores was not possible for toolchain intrinsic types. Note that the volatile keyword tells the C++ compiler that an entity outside of this thread of execution may modify the given memory address. Some toolchains supported only volatile stores, others did not support any volatile operations on the half-precision type and this was further complicated for Kokkos codes with accelerator backends such as CUDA that run the same process on both the host and device. We also wanted to keep the operator overloads as simple as possible in an effort to provide the toolchain a path for emitting vector instructions. To support researchers by providing a toolchain and hardware agnostic type, we decided to use a widely supported integral data type that toolchains know how to emit volatile load and store instructions for. In this way, we added overloads for volatile operators which explicitly reinterpret the underlying `float16` or `bfloat16` bits as a 2-byte integral type for the purpose of getting the toolchain to emit a volatile load or store operation. We then had to add a copy constructor to support copying from a `volatile half_t`. Without this copy constructor, it is not practical to use `half_t` since researcher's would have to perform this reinterpretation themselves during any assignment operation with a `volatile half_t` on the right hand side. Additionally, by adding this copy constructor, `half_t` is no longer trivially copyable and therefore, no longer considered a true C++ scalar type. Once we had supported common volatile operations, we added CUDA backend reduction support for 2-byte scalars and reduced the alignment and storage requirements of `half_t` back from 4-bytes to 2-bytes.

Commentary: Library development in the middle of scientific software stacks, such as for performance portability, I/O, or parallel communication, involves a delicate negotiation between what users want to do versus what toolchains and hardware want to support. At each layer of the HPC software stack, infrastructure providers must strategically choose which implementation details to expose and which ones to hide[39], and for library developers, navigating those layers can entail complex, integrative knowledge-work[40].

As seen above, successfully implementing the half-precision support feature in Kokkos required intimate knowledge of the hardware vendor interfaces (CUDA), language standards and compilers (C++), and researchers' software development preferences. Among other things, this raises interesting questions about how we think about RSEs and their skillsets. In general, the boundaries of software development practice tend to be very dynamic and open-ended, and RSEs are no exception; as Sims has commented, "As an emerging professional identity, the RSE role is still imprecisely defined (often intentionally so) [...] definitional issues around professional identity and boundaries are still under active discussion"[41]. This includes questions about what work RSEs do in practice and what competencies are they expected to have. Insofar as this is relevant to hiring, training, and career growth for RSEs, more studies are needed to ascertain what kinds of work RSEs find themselves doing and how they can be better supported in that work.

Challenge Observation 2: RSEs do a wide range of boundary-spanning, interdisciplinary work, which com-

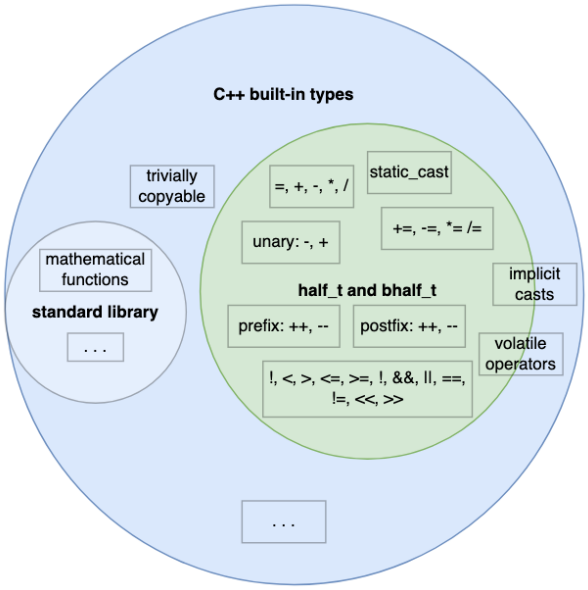

Fig. 5: A Venn diagram illustrating type support for half-precision in C++. Green indicates the subset that Kokkos aims to support.

plicates how we define what an RSE role entails and what skillsets are needed for RSEs to thrive. More studies are needed to ascertain what kinds of work RSEs find themselves doing and how they can be better supported in that work.

c) Anticipating Cognitive Load: Kokkos users leverage the half-precision support by using the `half_t` or `bhalf_t` type in place of their built-in floating point scalar types. The CGSolve performance evaluation demonstrates that, even for moderately complicated code including reductions at various levels of the parallelism hierarchy, this replacement is straightforward and trivially works. However, existing algorithms may need to be modified to perform explicit upcasts for assignments and extend library overloads to include `half_t` and `bhalf_t` parameter types. For example, Kokkos already provides portable support for C++ mathematical functions in the standard library. But since we have not yet extended that support to the half-precision types, users would need to upcast explicitly to float in order to use them. We plan to provide, at a minimum, fallback implementations which do that upcast internally in the next release. User codes that use overloaded functions specific to a scalar type, may also need to add explicit support for `half_t` and `bhalf_t`. While Kokkos half-precision support abstracts many of the fine-grained details that would otherwise result in costly duplicate workarounds in every researcher's code, we cannot abstract every detail.

In Figure 5 we can see that, without toolchain support, Kokkos users do not have full support for implicit casting nor standard library routines. Without half-precision overloads for standard library routines, there is high potential for rapid growth of duplicate overload implementations resulting in technical debt. Without full support for implicit casts, Kokkos user's will need to explicitly cast assignment expressions with `half_t` or `bhalf_t` on the right hand side. Furthermore, Kokkos version 3.6.00 codes that leverage accelerators such as GPUs should be aware that half-precision arithmetic on the host is always upcast to single-precision. For algorithms that are sensitive to floating point rounding, these codes will produce slightly different results depending on what steps are offloaded to the device. Without full host and device toolchain support for both `binary16` and `bfloat16`, researchers cannot ignore these subtle device-dependent details like they can with `float`, `double` and `long double`. As a result, accelerating codes via the adoption of half-precision types increases researcher's cognitive load.

Commentary: Milewicz and Rodeghero have observed that usability receives relatively little attention in scientific software development[42], and, as scientific software developers plumb the depths of the software stack from application codes down to the baremetal, ill-fitting and less-than-usable abstractions at each layer contribute to excessive cognitive overhead. In this case, the Kokkos team want clean, performance-portable constructs for researchers, but having to interact with the "low-level" mechanics of half-precision floating point exposes users to complex, device-dependent details; there are nuanced trade-offs involved in the design of these abstractions. RSEs, as professionals who span the scientific and software engineering domains, are uniquely well-positioned to address usability as a software quality. As Benthall and Seth have stressed, "software engineering skills are necessary to produce software that is usable beyond the lab or research group that originates it, which is a necessary path towards software sustainability" [43]. However, there is a lack of evidence-based guidance on how to apply usability tools, techniques, and concepts across scientific software stacks, particularly at the level of library APIs, suggesting a need for further research.

Challenge Observation 3: Usability receives relatively little attention in scientific software development. RSEs are well-positioned to address software quality concerns, including usability, but it is unclear how to map SE tools and techniques for usability to the various problem spaces in which RSEs operate (*e.g.,* in API design), indicating a need for further research.

B. Lessons Learned

a) Identify Prospective Users and Engage Frequently on Requirements: I found that despite a best faith effort to perform a thorough batched-GEMM requirements elicitation, neither the researchers nor myself knew what the Kokkos half-precision support requirements were. In fact, it wasn't until the batched-GEMM work had been fully prototyped in Kokkos Kernels that we had a good idea of what half-precision support the Kokkos ecosystem would require. At this point, we were deep into tangential implementation details and had at least one guiding requirement for the Kokkos half-

precision feature: create a half-precision type that behaves as closely as possible to `float`. A lesson learned here is that as soon as you identify a nested feature request, determine the users and stakeholders of the nested feature and perform another requirements elicitation with them. Getting feedback early and often is critical in scientific software development. Solicit feedback at any opportunity, whether that be stand-up meetings, code reviews, or helping someone use the new feature.

Lesson 1: Feature development work for scientific software libraries should be grounded in the needs of real users. Proactively identify prospective stakeholders and engage with them frequently to gather requirements.

b) Choose an Appropriate Development Methodology: For the Kokkos half-precision feature request, it quickly became clear that an feature-driven development approach was a good fit. In this way, I would identify one half-precision operator feature, write the production code, and then add the test code. This methodology provided me with a solid foundation for maximizing code coverage and implementing each half-precision operator, one at a time. Since I knew the half-precision type could be used as a template argument in any user's code, I wanted to ensure that coverage was maximized. For verifying both host and device results on the host, I decided to store actual and expected results in a single rank Kokkos View (i.e. array) and index into that array with a large enumeration. Each enumeration member maps an operator overload to an array element via a human-readable constant. This leads to the second lesson we learned, which is to choose a development methodology that improves your development experience and user confidence.

Lesson 2: Be intentional in the choice of development methodology, and consider both your individual needs as a developer and those of your customers – different tasks may require different approaches.

c) Refactor Early and Often, and Pay Down Technical Debt: Shortly after the `half_t` support had been prototyped, support for `bhalf_t` was requested. Rather than copying and modifying the `half_t` class, we refactored `half_t` into the type-agnostic `floating_point_wrapper` class and augmented the class to work with both `binary16` and `bfloat16` CUDA intrinsics. At the same time, other backends were beginning to copy `half_t` and modify the class to suit their needs. At this point, I refactored `floating_point_wrapper` to also be backend agnostic and requested that other backends use `floating_point_wrapper` rather than copying the `half_t` class. Our lesson learned here is to refactor early and often to avoid costly technical debt.

The exploratory nature of experimental feature development, however, can nevertheless lead to persistent debt. Since half-precision support was not initially identified as a nested feature request in the batched-GEMM feature request, an artifact of the initial half-precision batched-GEMM prototype persists. This artifact is a small `KokkosKernels_Half.hpp` file which defines `binary16` and `bfloat16` constants required for verifying half-precision arithmetic in Kokkos Kernels. Unfortunately, some of these constants are also defined within the half-precision unit tests in Kokkos Core since I never decided where these constants should be defined. A related piece of technical debt is a type alias in the Kokkos Kernels `AritTraits` class called `halfPrecision`. `halfPrecision` was originally added as a convenience for performing mixed precision expressions with the next lowest precision. `halfPrecision` is now confused as an alias for `half_t` or `bhalf_t`. Unfortunately, `halfPrecision` is used in too many codes to be changed.

Lesson 3: It is important to pay down technical debt by refactoring early and often. Of particular note when developing scientific software libraries, latent technical debt can emerge in public interfaces and, once in place, is persistent and hard to remove.

d) Leverage Language Support: C++ provides powerful features that blur the line between compiler development and library / application development. These C++ features are a double edged sword. The features provide an immediate path forward to enable research via high-level, backend-agnostic abstractions such as `half_t` and `bhalf_t`, at the expense of subtle feature divergences from built-in types as depicted in Figure 5. Our lesson here is to learn and leverage advanced C++ features including class templates, compile-time conditionals, forward declarations, typedefs and include guards to enable type-agnostic and architecture-agnostic abstractions like `half_t` and `bhalf_t`.

Lesson 4: As an RSE, know your tools. Case in point, modern programming languages have powerful and flexible features, but they can also be a source of complexity that must be managed. Knowing what language features to use and when is a key part of good software craftsmanship.

IV. Conclusion

While the potential for a quick and easy way to speedup C++ algorithms with half-precision scalars is both sensible and alluring, it is not practical without an additional level of indirection. As discussed above, toolchain and hardware vendors support half-precision types and arithmetic to varying degrees. This variation poses show-stopping portability challenges which the experimental Kokkos half-precision indirection attempts to alleviate. While modern C++ allows us to provide powerful abstractions like `half_t` and `bhalf_t`, without full toolchain and hardware support for `float16` and `bfloat16`, there is no way to fully mimic the behavior of other C++ built-in floating point types. Regardless of the researcher's ecosystem, the current variation of toolchain and hardware support for half-precision is causing wide-spread and growing software sustainability burdens. Future work for

Kokkos half precision support includes: Adding half-precision Kokkos mathematical function overloads and consolidating half-precision constant definitions such as `BF16_EPSILON` to Kokkos Core.

REFERENCES

[1] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The kokkos ecosystem: Comprehensive performance portability for high performance computing," *Computing in Science Engineering*, vol. 23, no. 5, pp. 10–18, 2021.

[2] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[3] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[4] "Kokkos: Core libraries," https://github.com/kokkos/kokkos#readme, accessed May 4th, 2022.

[5] "Kokkos kernels," https://github.com/kokkos/kokkos-kernels#readme, accessed May 4th, 2022.

[6] "The bfloat16 numerical format," https://cloud.google.com/tpu/docs/bfloat16#choosing_bfloat16, accessed May 23rd, 2022.

[7] "Deep learning performance documentation," https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html#opt-tensor-cores, accessed May 4th, 2022.

[8] "6.13 half-precision floating point," https://gcc.gnu.org/onlinedocs/gcc/Half-Precision.html, accessed May 4th, 2022.

[9] "Intel 64 and ia-32 architectures software developer's manual," https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html, accessed May 4th, 2022.

[10] "Half-precision floating point," https://releases.llvm.org/9.0.0/tools/clang/docs/LanguageExtensions.html#half-precision-floating-point, accessed May 4th, 2022.

[11] "Half-precision floating-point data types," https://developer.arm.com/documentation/100067/0612/Other-Compiler-specific-Features/Half-precision-floating-point-data-types, accessed May 4th, 2022.

[12] "Cuda 10 features revealed: Turning, cuda, graphs, and more," https://developer.nvidia.com/blog/cuda-10-features-revealed/, accessed May 4th, 2022.

[13] "Nvidia tesla v100 gpu architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, accessed May 4th, 2022.

[14] "Nvidia a100 tensor core gpu," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf, accessed May 4th, 2022.

[15] "Bfloat16 precision conversion and data movement," https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH___BFLOAT16__MISC.html, accessed May 4th, 2022.

[16] "Gmres kokkos-based solver," https://github.com/kokkos/kokkos-kernels/tree/develop/example/gmres, accessed May 4th, 2022.

[17] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li *et al.*, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 344–369, 2021.

[18] J. A. Loe, C. A. Glusa, I. Yamazaki, E. G. Boman, and S. Rajamanickam, "Experimental evaluation of multiprecision strategies for gmres on gpus," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 469–478.

[19] N. Lindquist, P. Luszczek, and J. Dongarra, "Accelerating restarted gmres with mixed precision arithmetic," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 1027–1037, 2021.

[20] P. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. l'Excellent, and T. Mary, "Mixed precision low rank approximations and their application to block low rank lu factorization," 2021.

[21] S. Gratton, E. Simon, D. Titley-Peloquin, and P. Toint, "Exploiting variable precision in gmres," *arXiv preprint arXiv:1907.10550*, 2019.

[22] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[23] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," in *International Conference on Learning Representations*, 2018.

[24] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[25] "Data types," https://numpy.org/doc/stable/user/basics.types.html, accessed May 4th, 2022.

[26] "half," https://www.mathworks.com/help/fixedpoint/ref/half.html, accessed May 4th, 2022.

[27] "Integer and floating-point numbers," https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/, accessed May 4th, 2022.

[28] R. Baxter, N. C. Hong, D. Gorissen, J. Hetherington, and I. Todorov, "The research software engineer," in *Digital Research Conference, Oxford*, 2012, pp. 1–3.

[29] A. Brett, M. Croucher, R. Haines, S. Hettrick, J. Hetherington, M. Stillwell, and C. Wyatt, "Research software engineers: state of the nation report 2017," 2017.

[30] J. Cohen, D. S. Katz, M. Barker, N. C. Hong, R. Haines, and C. Jay, "The four pillars of research software engineering," *IEEE Software*, vol. 38, no. 1, pp. 97–105, 2020.

[31] "Software engineering & research, 01424," https://cfwebprod.sandia.gov/cfdocs/CompResearch/templates/insert/dept.cfm?org=01424, accessed May 4th, 2022.

[32] R. Milewicz, J. Willenbring, and D. Vigil, "Research, develop, deploy: Building a full spectrum software engineering and research department," *Research Software Engineers in HPC (RSE-HPC-2020)*, 2020.

[33] R. Milewicz, "Towards evidence-based practice in scientific software development," Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2020.

[34] S. Smith, M. Srinivasan, and S. Shankar, "Debunking the myth that upfront requirements are infeasible for scientific computing software," in *2019 IEEE/ACM 14th International Workshop on Software Engineering for Science (SE4Science)*. IEEE, 2019, pp. 33–40.

[35] R. Milewicz and M. Mundt, "An exploration of the mentorship needs of research software engineers," *Research Software Engineers in HPC (RSE-HPC-2021)*, 2021.

[36] C. Jay, R. Haines, D. S. Katz, J. C. Carver, S. Gesing, S. R. Brandt, J. Howison, A. Dubey, J. C. Phillips, H. Wan *et al.*, "The challenges of theory-software translation," *F1000Research*, vol. 9, 2020.

[37] N. P. Chue Hong, J. Cohen, and C. Jay, "Understanding equity, diversity and inclusion challenges within the research software community," in *International Conference on Computational Science*. Springer, 2021, pp. 390–403.

[38] B. Cartaxo, G. Pinto, and S. Soares, "The role of rapid reviews in supporting decision-making in software engineering practice," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, 2018, pp. 24–34.

[39] B. Sims, "Layers of abstraction and the organization of repair in high performance computing," in *2019 Society for Social Studies of Science Annual Meeting*, September 2019.

[40] R. Milewicz and E. Raybourn, "Talk to me: A case study on coordinating expertise in large-scale scientific software projects," in *2018 IEEE 14th International Conference on e-Science (e-Science)*. IEEE, 2018, pp. 9–18.

[41] B. H. Sims, "Research software engineering: Professionalization, roles, and identity." [Online]. Available: https://www.osti.gov/biblio/1845242

[42] R. Milewicz and P. Rodeghero, "Position paper: Towards usability as a first-class quality of hpc scientific software," in *2019 IEEE/ACM 14th International Workshop on Software Engineering for Science (SE4Science)*. IEEE, 2019, pp. 41–42.

[43] S. Benthall, M. Seth, M. Agarwal, C. Calloway, D. Niederhut, and D. Shupe, "Software engineering as research method: Aligning roles in econ-ark," in *Proceedings of the 19th Python in Science Conference*, 2020, pp. 156–161.