Journal Title
XX(X):1–12

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Reproducibility and variable precision computing

David H. Bailey[1]

Abstract

Recently some scientific computing users have discovered that they can replace 64-bit with 32-bit operations for carefully selected portions of the computation, and still retain acceptable accuracy in the final results. In addition, developers of some emerging applications such as machine learning have discovered that they can achieve acceptable results with only 16-bit precision in certain portions of the code. At the other end of the precision spectrum, some users have explored using 128-bit arithmetic in some particularly demanding applications, while others have done computations using much higher precision — hundreds or even thousands of digits. Such work has underscored the need to develop new mathematical and software frameworks to support a dynamically variable level of precision, and, more generally, to rethink what "reproducibility" means in a variable precision environment. This article summarizes some of the work being done in this arena, and lists research problems that need to be solved.

Introduction

The vast majority of floating-point computations in research and engineering employ either IEEE single (32-bit) or IEEE double (64-bit), mostly one or the other exclusively within a single program. However, recent developments have demonstrated the need for a broader range of precision levels, and also for a level of precision that varies dynamically within a single application. There are certainly performance advantages to variable precision, including faster processing, better cache utilization, lower run-time memory usage, lower offline data storage and lower energy costs. But effective usage of variable precision also requires appropriate software tools, a more sophisticated mathematical framework, as well as some fundamental rethinking of what reproducibility means in floating-point computation.

The issue of numerical reproducibility was addressed at a 2014 workshop held at the Institute for Computational and Experimental Research in Mathematics (ICERM), in Providence, Rhode Island, USA. As the authors of the summary report, Bailey et al. (2014), noted,

[1]Lawrence Berkeley National Laboratory (retired), Berkeley, CA 94720, and University of California, Davis, Department of Computer Science, Davis, CA 95616; dhbailey@lbl.gov

> Numerical reproducibility has emerged as a particularly important issue, since the scale of computations has greatly increased in recent years, particularly with computations performed on many thousands of processors and involving similarly large datasets. Large computations often greatly magnify the level of numeric error, so that numerical difficulties that were once of little import now are large enough to alter the course of the computation or to draw into question the overall validity of the results.

As a single example that has been reported to the present author, the ATLAS experiment at the Large Hadron Collider (which was employed in the 2012 discovery of the Higgs boson) relies on the ability to track charged particles with high accuracy (10 microns over a 10m length) and high reliability (over 99% of roughly 1000 charged particles per collision correctly identified). The software used for these detections involves roughly five million lines of C++ and Python code, developed over a 15-year period by some 2000 physicists and engineers. Recently, in an attempt to speed up these calculations, researchers found that merely changing the underlying math library (which should only change at most the final bit of the library function results) caused some collisions to be missed and others misidentified. This suggests that their code has significant numerical sensitivities, and the results might not be reliable in certain cases.

From these and other considerations, it is clear that increased attention must be given to the question of whether sufficient numeric precision is employed to produce reliable results. On the other hand, there is no need to use more numeric precision than is truly required, since as noted above there are multiple performance advantages to using lower precision where this is sufficient. Thus the optimal approach is to employ a varying level of numeric precision, with higher precision in numerically sensitive portions, then switching to lower precision where this is adequate.

At the low end of precision, many graphics, artificial intelligence and machine learning applications in recent years have successfully utilized some form of 16-bit floating-point — usually either the IEEE 16-bit "half" precision standard (five exponent bits and ten mantissa bits) or else the emerging "bfloat16" format (eight exponent bits and seven mantissa bits). At the high end of precision, some high-performance computing users are utilizing 128-bit arithmetic, in an effort to control increased numerical roundoff error, while others are utilizing even more extreme precision (hundreds or even thousands of digits) in some specialized computations in mathematics and physics. Some of the most common formats and precision levels are shown in Table 1.

Mixed half-single precision

As mentioned above, some users of machine learning applications, for instance, have found that a mixed half-single precision approach (where "half" is either IEEE half, ARM half or bfloat16) to be adequate for their applications, although some form of scaling is often required. For example, in NVIDIA (2019), researchers found that a training application (the bigLSTM English language model), implemented using a combination of IEEE 32-bit arithmetic and the bfloat16 arithmetic, achieved nearly the same training loss reduction curve, compared with a 100% IEEE 32-bit implementation. However, it was necessary to scale the 16-bit results in portions of the computation, due to the limited exponent range of the 16-bit formats.

In particular, the NVIDIA strategy employed for this and several other machine learning applications is given by NVIDIA (2019) as follows: Maintain a master copy of the training system weights in IEEE 32-bit precision, and select a scaling factor S. Then for each iteration:

Formal name	Nickname	Number of bits				Approx. Digits
		Sign	Exponent	Mantissa	Hidden	
IEEE 16-bit	IEEE half	1	5	10	1	3
(none)	ARM half	1	5	10	1	3
(none)	bfloat16	1	8	7	1	2
IEEE 32-bit	IEEE single	1	7	24	1	7
IEEE 64-bit	IEEE double	1	11	52	1	15
IEEE 80-bit	IEEE extended	1	15	64	0	19
IEEE 128-bit	IEEE quad	1	15	112	1	34
(none)	double double	1	11	104	2	31
(none)	quad double	1	11	208	4	62
(none)	double quad	1	15	224	2	68
(none)	multiple	1	varies	varies	varies	varies

Table 1. Commonly used floating-point formats and precision levels

- Convert the array of 32-bit weights to a bfloat16 array.
- Perform forward propagation with bfloat16 weights and activations.
- Multiply the resulting loss with a scaling factor S.
- Perform backward propagation with bfloat16 weights, activations and gradients.
- Multiply the weight gradient by $1/S$.
- Update the 32-bit weights using the bfloat16 data.

A somewhat more sophisticated approach, also described in NVIDIA (2019), automatically and dynamically selects the scaling factor S.

Mixed single-double precision

Many exascale system users and others running very large numerical simulations and analyses are reconsidering their traditionally generous usage of numerical precision, since as mentioned above there are clear performance advantages to employing reduced precision where possible. In particular, researchers have found that if 32-bit calculations can be substituted for 64-bit operations in selected parts of the application, then significant speedups may be achieved.

This general strategy has led to new mixed-precision approaches for common linear algebra operations, as given by Baboulin et al. (2009); Buttari et al. (2008). For example, consider the process of pivoted Gaussian elimination. The usual scheme is to factor a pivoted coefficient matrix $PA = LU$, where L is lower triangular and U is upper triangular, and then solving $Ly = Pb$ and $Ux = y$, all done using double precision (IEEE 64-bit) arithmetic. In a mixed single-double precision approach, the factorization $PA = LU$ and the solution of the triangular systems $Ly = Pb$ and $Ux = y$ are computed in single precision (IEEE 32-bit). This is followed by a calculation of the residual, in double precision, and then the solutions are also done using double precision. Note in particular that the factorization of A, which is the only operation that has computational complexity $O(n^3)$, is handled exclusively in single precision.

A similar approach may be taken for sparse systems. The authors Baboulin et al. (2009), for instance, were able to obtain speedups of up to 1.858 on an AMD Opteron 246, up to 1.859 on an IBM PowerPC 970, and up to 1.799 on an Intel Xeon 5100, by employing a mixed-precision approach for direct sparse solution of a suite of test problems.

More broadly, the potential for mixed single-double computation has renewed interest in iterative refinement, where initial iterations are performed using half- or single-precision, as described by Haidar et al. (2018). Researchers are also exploring the use of floating point compression, not only for I/O, but also for storing solution state variables during run time, as in Lindstrom (2014); Diffenderfer et al. (2018).

Mixed double-quad precision

Exascale computing has also exposed the need for even greater precision than IEEE 64-bit double in some cases, because the greatly enlarged problem sizes typically used in these calculations often result in correspondingly magnified numerical sensitivities, so that one can no longer be certain that results are numerically reliable. One remedy is to use IEEE 128-bit quad precision in selected portions of the computation. Unfortunately, as of this date the major microprocessor and accelerator vendors have not yet implemented the 128-bit IEEE standard in hardware (one exception is the IBM Power9 processor). It is, however, now available via software in some compilers, notably the gfortran compiler (by declaring quad-precision variables to be of type `real(16)`). Even though software implementations are typically quite slow (up to 50X slower, depending on the system and operation mix), the IEEE quad format is now being used in a growing number of research applications. As a single example, researchers at Stanford have had remarkable success in using quad precision in computational biology applications that involve heavy-duty multiscale linear programming, as in Ma and Saunders (2017).

Others have used “double-double” precision, namely a software extension that uses two IEEE 64-bit floats to jointly represent a floating-point datum, with the same dynamic range as IEEE 64-bit, but with twice as many mantissa bits (104), corresponding to roughly 31-digit accuracy. These implementations tend to run faster than a full-scale software implementation of IEEE 128-bit. Some implementations include high-level interfaces for C++ and Fortran code, which greatly simplify the process of converting an application to use double-double arithmetic — see Hida et al. (2001).

It is worth mentioning that the authors Baboulin et al. (2009), mentioned above, also extended their approach for linear algebra computations to a mixed double-quad precision regime, both for classical dense matrix solutions and also for sparse and iterative refinement operations. Here they were able to demonstrate quite remarkable speedups — up to 94.8X speedup on an iterative refinement problem of size 1000.

Is higher-than-64-bit arithmetic ever really necessary?

Before continuing, it should be noted that some have questioned whether quadruple precision (either IEEE 128-bit floating-point or double-double) should ever be required in real-world scientific computation — surely any scientific calculation where 128-bit floating-point arithmetic is employed could be performed more economically in 64-bit, using more advanced, numerically stable algorithms. However, as the present author and two colleagues documented at length in an earlier study, Bailey et al. (2012), there are numerous real-world applications that do in fact require more than 64-bit arithmetic to produce numerically meaningful results.

As a single example, suppose that one wishes to recover the integer polynomial that produces the result sequence $(1, 32771, 262217, 885493, 2101313, 4111751, 7124761)$ for integer arguments $(0, 1, \ldots, 6)$ — certainly a very simple, plausible numerical application. For such a problem, many scientists and engineers will employ a least-squares scheme, since this is a very familiar tool in scientific data analysis, and efficient library software is readily available. In this approach, one constructs the $(n+1) \times (n+1)$ linear system

$$\begin{bmatrix} n+1 & \sum_{k=1}^{n} x_k & \cdots & \sum_{k=1}^{n} x_k^n \\ \sum_{k=1}^{n} x_k & \sum_{k=1}^{n} x_k^2 & \cdots & \sum_{k=1}^{n} x_k^{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} x_k^n & \sum_{k=1}^{n} x_k^{n+1} & \cdots & \sum_{k=1}^{n} x_k^{2n} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{n} y_k \\ \sum_{k=1}^{n} x_k y_k \\ \vdots \\ \sum_{k=1}^{n} x_k^n y_k \end{bmatrix}, \qquad (1)$$

where (x_k) are the integer arguments and (y_k) are the sequence values, then solves for $(a_1, a_2, \cdots, a_n)$ using, for example, the LAPACK software (see Anderson et al. (1999)), rounding the final vector of results to the nearest integer.

In the specific problem mentioned above, a double-precision floating-point implementation of the least-squares scheme succeeds in finding the correct polynomial coefficients, which, after rounding to the nearest integer, are $(1, 0, 0, 32769, 0, 0, 1)$, or, in other words, $f(x) = 1 + (2^{15} + 1)x^3 + x^6$. Unfortunately, this scheme, using double precision, fails to find the correct polynomial for a somewhat more difficult problem, namely to find the degree-8 polynomial that generates the 9-long sequence $(1, 1048579, 16777489, 84941299, 268501249, 655751251, 1360635409, 2523398179, 4311748609)$, for integer arguments $(0, 1, \cdots, 8)$.

Numerical analysts may point out that this type of problem is more effectively solved using a Lagrange interpolating polynomial. A Lagrange scheme, implemented with 64-bit IEEE arithmetic, correctly deduces that the 9-long data sequence above is produced by the integer polynomial $1 + (2^{20} + 1)x^4 + x^8$. However, this scheme fails when given the more challenging 13-long input data vector $(1, 134217731, 8589938753, 97845255883, 549772595201, 2097396156251, 6264239146561, 15804422886323, 35253091827713, 71611233653971, 135217729000001, 240913322581691, 409688091758593)$, which is generated by the integer polynomial $1 + (2^{27} + 1)x^6 + x^{12}$. The state-of-the-art algorithm in this area, as far as the present author is aware, is a technique given in Demmel and Koev (2005). This scheme successfully solves the degree-6 and degree-8 problems mentioned above, but, like the Lagrange polynomial scheme, fails for the degree-12 problem.

However, there is another, simpler approach to these problems: simply modify the source code of any reasonably effective solution scheme to invoke quad precision arithmetic in selected numerically sensitive portions of the code. For example, when a Fortran least-squares scheme was modified to employ double-double precision (approximately 31-digit accuracy), using the QD software mentioned above in Bailey et al. (2012), it was able to correctly solve all three problems (degrees 6, 8 and 12). Converting the Lagrange polynomial scheme to use double-double arithmetic was even easier, and the resulting program also solved all three problems without incident.

Extreme precision applications

Some have also claimed that there could not be any useful scientific computation employing higher precision, such as several hundred or several thousand digits, but, as before, this claim is mistaken, as documented in Bailey et al. (2012). Some examples that the present author is aware of include:

1. Solutions of ordinary differential equations arising from planetary orbit models (32 digits).
2. Supernova simulations (32–64 digits).
3. Scattering amplitudes of fundamental particles (32 digits).
4. Solutions of certain discrete dynamical systems (32 digits).
5. Coulomb n-body atomic system simulations (32–120 digits).
6. Solving ordinary differential equations using the Taylor algorithm (100–300 digits).
7. Recognizing integrals from the Ising theory of mathematical physics in terms of simple mathematical identities (200–1000 digits).
8. Finding minimal polynomials connected to the Poisson equation of mathematical physics (1000–64,000 digits).

The last two items are typical of numerous recent "experimental mathematics" studies that have utilized extreme-precision computations, mostly in mathematical physics settings. The typical strategy is to first compute values of the entities in question to very high precision, then use an integer relation algorithm, such as the "PSLQ" algorithm given in Ferguson et al. (1999); Bailey and Broadhurst (2000), to recognize these numerical values in terms of simple mathematical formulas (although such relations still must be rigorously proven by traditional methods).

Given a vector $X = (x_0, x_1, \cdots, x_{n-1})$ of high-precision numerical values, an integer relation algorithm such as PSLQ finds integers a_i (not all zero), if they exist, such that $a_0x_0 + a_1x_1 + \cdots + a_{n-1}x_{n-1} = 0$, to within the tolerance of the available numeric precision. It can be shown that calculations using an integer relation algorithm such as PSLQ require very high precision to obtain numerically reproducible results. In general, if an n-long vector of integers (a_i) is bounded in size by 10^d, then all input numerical values (x_i) must be known to at least nd-digit precision, and the integer relation algorithm must be performed using at least nd-digit precision.

However, it is possible to do a PSLQ calculation by performing nearly all iterations using ordinary double precision, updating high-precision arrays only when the accuracy of the double precision arrays is nearly exhausted. A mixed double-multiprecision implementation of this sort typically runs hundreds of times faster than a purely multiprecision implementation. For large problems, a three-level precision strategy is even more efficient, although programming this type of dynamically varying precision application is quite challenging.

We will briefly describe here just one of these extreme precision applications, namely the elucidation of polynomials connected to the Poisson potential function of mathematical physics, as documented in Bailey et al. (2016). This research began in 2012, when Richard Crandall, while investigating techniques to sharpen images for a smartphone camera application, noted that each pixel was given by a form of the 2-D Poisson potential function of mathematical physics:

$$\phi_2(x, y) = \frac{1}{\pi^2} \sum_{m,n \text{ odd}} \frac{\cos(m\pi x)\cos(n\pi y)}{m^2 + n^2}$$

s	Minimal polynomial corresponding to $x = y = 1/s$:
5	$1 + 52\alpha - 26\alpha^2 - 12\alpha^3 + \alpha^4$
6	$1 - 28\alpha + 6\alpha^2 - 28\alpha^3 + \alpha^4$
7	$-1 - 196\alpha + 1302\alpha^2 - 14756\alpha^3 + 15673\alpha^4 + 42168\alpha^5 - 111916\alpha^6 + 82264\alpha^7$ $-35231\alpha^8 + 19852\alpha^9 - 2954\alpha^{10} - 308\alpha^{11} + 7\alpha^{12}$
8	$1 - 88\alpha + 92\alpha^2 - 872\alpha^3 + 1990\alpha^4 - 872\alpha^5 + 92\alpha^6 - 88\alpha^7 + \alpha^8$
9	$-1 - 534\alpha + 10923\alpha^2 - 342864\alpha^3 + 2304684\alpha^4 - 7820712\alpha^5 + 13729068\alpha^6$ $-22321584\alpha^7 + 39775986\alpha^8 - 44431044\alpha^9 + 19899882\alpha^{10} + 3546576\alpha^{11}$ $-8458020\alpha^{12} + 4009176\alpha^{13} - 273348\alpha^{14} + 121392\alpha^{15}$ $-11385\alpha^{16} - 342\alpha^{17} + 3\alpha^{18}$
10	$1 - 216\alpha + 860\alpha^2 - 744\alpha^3 + 454\alpha^4 - 744\alpha^5 + 860\alpha^6 - 216\alpha^7 + \alpha^8$

Table 2. Minimal polynomials discovered by high-precision PSLQ calculations.

In the subsequent study Bailey et al. (2013), Crandall and collaborators numerically discovered and then proved the intriguing fact that for rational (x, y),

$$\phi_2(x, y) = \frac{1}{\pi} \log \alpha$$

where α is *algebraic*, i.e., the root of a some integer polynomial of degree m. By computing high-precision numerical values of $\phi_2(x, y)$ for various specific rational x and y, and applying the multipair variant of the PSLQ integer relation algorithm, they were able to produce the explicit minimal polynomials for α in numerous specific cases — see Table 2.

Based on these preliminary results, Jason Kimberley of the University of Newcastle, Australia conjectured that the degree $m(s)$ of the minimal polynomial associated with the case $x = y = 1/s$, where s is a positive integer, is given by the following rule:

> Set $m(2) = 1/2$. Otherwise for primes p congruent to 1 mod 4, set $m(p) = \text{int}^2(p/2)$, where int denotes greatest integer, and for primes p congruent to 3 mod 4, set $m(p) = \text{int}\,(p/2)(\text{int}\,(p/2) + 1)$. Then for any other positive integer s whose prime factorization is $s = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$,
>
> $$m(s) = 4^{r-1} \prod_{i=1}^{r} p_i^{2(e_i-1)}\, m(p_i).$$

Does this mysterious formula hold for larger s? Additional computations were then undertaken, using more advanced software, including a three-precision-level implementation of multipair PSLQ, based on the MPFR high-precision arithmetic library described in Fousse et al. (2007). These runs confirmed Kimberley's formula for all s up to 40, and for all even s up to 52. These calculations were not easy. The case $x = y = 1/35$, for instance, required 18,000-digit arithmetic and 55 CPU-hours run time, and produced a set of integer polynomial coefficients ranging in size from 1 to approximately $1.702713991 \times 10^{85}$. Other runs required up to 64,000-digit precision and over 2000 CPU-hours.

By examining the computed results, connections were found to a sequence of polynomials defined in a 2010 paper by Savin and Quarfoot. These observations ultimately led to a proof of Kimberley's formula, and also a proof of the fact that when s is even, the minimal polynomial is palindromic (i.e., coefficient $a_k = a_{m-k}$, where m is the degree). Full details are given in Bailey et al. (2016).

Software and language issues

A common experience of researchers who have attempted mixed or variable precision computations such as those described above is that most existing software facilities are rather difficult to use, particularly for large, computationally demanding applications. Such facilities typically require laborious (and potentially error-prone) changes to one's source code, and yet provide little or no guidance as to how much precision is required in different parts of the computation.

Existing languages also have some serious issues in this regard, notably in the conversion of constant data and in subexpessions involving mixed-precision variables and constants. For example, consider this Fortran code:

```
program testprec
real(8) d1, d2, d3
d1 = sqrt (3. + 1./7.)
d2 = sqrt (3.d0 + 1./7.)
d3 = sqrt (3.d0 + 1.d0/7.d0)
write (6, '(f20.15)') d1, d2, d3
stop
end
```

This code, when compiled with the gfortran compiler with default options, produces the output:

```
1.772810459136963
1.772810522656991
1.772810520855837
```

Why are these three values so different? In the first line of code, the constants 3., 1., 7. are interpreted as single precision constants, and so, by established rules of Fortran mixed-mode operations (which rules are adopted in most other languages), the subexpression $3. + 1./7.$ is computed only to single precision accuracy, as is its square root, which is then converted to double precision by zero extension before storing in the left-hand side. Thus the first output value is only accurate to about seven digits. In the second line, the fraction $1./7.$ is computed to only single precision accuracy, again in accord with established rules of mixed-mode operations, then converted to double precision by zero extension, then added to three, and then the square root of this sum is computed using double precision. Thus, as before, the second result is only accurate to about seven digits. Only the third line produces a fully accurate double precision value.

Such problems are even more vexing, and more difficult to find and remedy, in mixed double-multiprecision calculations.

It should be emphasized, though, that in some mixed precision applications, one may want certain subexpressions to be evaluated using only the lower precision level. For example, surely the Fortran code `P2 = 4.5d0 * (P1 - 5.d0*1.25d0)`, where `P1,P2` are multiprecision, should be evaluated strictly as written, since the constants are all exact binary fractions, and no precision is lost when performing the indicated mixed-mode double-multiprecision operations.

So what is the best way for users of mixed-precision applications to precisely control the precision level used in each operation, and to properly deal with issues such as conversion of constants and mixed-mode operations? Should automatic mixed-mode conversions be optionally disabled for users who desire greater control? At present, there is no solution or clear consensus.

Along this line, it is essential that high-precision arithmetic packages and software tools be *thread-safe*, so that they produce correct results when an application involving high-precision operations is implemented in a shared-memory parallel environment such as OpenMP. For a multiprecision arithmetic library, this means that *each multiprecision datum* must include parameters specifying: (a) the full length of storage allocated to this datum, (b) the length of nonzero data associated with this datum, and (c) the current working precision assigned to this datum. Currently the MPFR package described in Fousse et al. (2007), for example, conforms to this requirement, but many others do not, and, as a result, are not thread-safe and often give incorrect answers when used in a shared-memory parallel application.

Software tools

Fortunately, some software tools have been developed to manage precision levels and detect precision failures. One such tool is known as "Precimonious", described in Rubio-Gonzalez et al. (2013). Its overall objective is to develop software facilities to find and ameliorate numerical anomalies in large-scale computations. It includes:

- Facilities to test the level of numerical accuracy required for an application.
- Facilities to delimit the portions of code that are inaccurate.
- Facilities to search the space of possible code modifications.
- Facilities to repair numerical difficulties, including usage of high-precision arithmetic.
- Facilities to navigate through a hierarchy of precision levels (32-bit, 64-bit or higher as needed).

Additional work in this area is described in Rubio-Gonzalez et al. (2013); Nguyen et al. (2016).

Bit-for-bit floating-point replicability between systems

Although the IEEE floating-point standards guarantee bit-for-bit results (provided the same rounding mode is used) for individual operations, different systems may produce different higher-level results, because of non-associativity: $(A \oplus B) \oplus C \neq A \oplus (B \oplus C)$.

Recently some researchers have proposed an addition to the IEEE-754 standard that would permit guaranteed bit-for-bit replicability, not only on a single system but also on a parallel system — see Ahrens et al. (2016), for example. Along this line, ARM (a U.K. computer processor vendor) has implemented a high-precision accumulator, using an adaptable vector unit, that also permits bit-for-bit replicability over a certain class of operations, with some limitations.

Many users are eager to try such facilities, as they promise to greatly simplify the process of porting a large scientific code from one system to another (or even to the same system, but using a different number of processors). But should this standard become the default on all computer systems? There is a danger that such a standard would lock into place computations that are seriously inaccurate and make it more difficult for a user to even be aware that they have a problem with numerical error. Clearly more study and analysis is needed here.

New ideas for floating-point computation

One common thread of the computational applications described above is that they employ a level of numeric precision that varies dynamically over the course of the computation, performing as much as

possible using relatively low precision (16-bit or 32-bit), and only switching to higher precision (64-bit, 128-bit or higher) when necessary.

Along this line, while many in the application community are experimenting with variable precision using present-day system facilities, some are suggesting that we fundamentally rethink the concept of floating-point computation, replacing the current set of fixed precision floating-point formats with a more flexible system, one that potentially could dynamically monitor and control numerical error. One proposed system is the "Unum" system, proposed by Gustafson and Yonemoto (2017). This and some other alternatives to present-day floating-point formats are summarized by Lindstrom et al. (2018). Although as yet there is no clear consensus on the effectiveness of these proposals, it is clear that such ideas need to be further studied.

Research questions to be explored

Here are some research questions in variable precision, adapted from a proposal for a ICERM workshop proposal in variable precision computing, held in May 2020 (Bailey et al. (2019)):

1. How are scientific disciplines, ranging from pure mathematics to applied science and engineering, currently using variable precision? What new opportunities are on the horizon?
2. What does the explosive growth in artificial intelligence and machine learning mean for hardware, software and tools to support variable precision? What are the fundamental accuracy requirements of such computations?
3. Are there new mathematical approaches to efficiently and accurately compute various mathematical operations and library functions to both very modest precision (2-3 digit accuracy) and very high precision (hundreds or thousands of digits)?
4. What new formats and representation systems to facilitate variable level precision computing are available, along the lines of Gustafson and Yonemoto (2017); Lindstrom et al. (2018)? To what extent do these new proposals overcome strengths and weaknesses in existing systems such as IEEE-754 hardware and MPFR software described in Fousse et al. (2007)?
5. What new support is needed from computer hardware and software vendors for variable precision computing? Could some modest hardware changes be made to support low, variable precision and high precision, rather than relying on all-software implementations?
6. Almost all of published research in numerical mathematics and numerical analysis to date has focused on either 32-bit or 64-bit results. Are there other techniques that may be superior in a variable precision environment?
7. Classical error bounds used in numerical analysis, such as those developed by Wilkinson, typically assume worst-case scenarios and thus are not very useful in a limited precision or variable precision environment. Can more realistic error estimates be derived, as in Higham and Mary (2018)?
8. How can iterative refinement best be performed in a variable precision environment? What new opportunities are there for using iterative refinement with variable precision?
9. What algorithms and software techniques can effectively compress floating-point data, on-the-fly, to save data storage and transfer? Can improved schemes be devised?
10. What software tools are available to automatically or semi-automatically analyze an application code, identify numerically sensitive spots and propose or implement remedies? What other tools are needed?

11. How can we ensure that computations with variable precision levels are reproducible? What does reproducibility even mean when computing, say, with only 16-bit precision?

Summary

In summary, we now are seeing rapidly growing demand for dynamically variable precision levels typically varying within a given application. But variable precision computing presents new challenges, including how to ensure reproducibility, how to manage and control precision, how to determine if the precision level is sufficient, how to determine which portions of a code are numerically sensitive, and the overriding question of whether we need to rethink the fundamental design of floating-point arithmetic. Clearly there is much to be done.

References

Ahrens P, Demmel J and Nguyen HD (2016) Efficient reproducible floating point summation and BLAS. Report EECS-2017-121, U.C. Berkeley EECS.

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A and Sorensen D (1999) *LAPACK Users' Guide*. Third edition. Philadelphia, PA: Society for Industrial and Applied Mathematics. ISBN 0-89871-447-8 (paperback).

Baboulin M, Buttari A, Dongarra J, Kurzak J, Langou J, Langou J, Luszczek P and Tomov S (2009) Accelerating scientific computations with mixed precision algorithms. *Computational Physics Communications* 180: 2526–2533. DOI:http://dx.doi.org/10.1016/j.cpc.2008.11.005.

Bailey DH, Barrio R and Borwein JM (2012) High precision computation: Mathematical physics and dynamics. *Applied Mathematics and Computation* 218: 10106–10121. URL `https://www.osti.gov/servlets/purl/983781`.

Bailey DH, Borwein JM, Crandall RE and Zucker J (2013) Lattice sums arising from the Poisson equation. *Journal of Physics A: Mathematical and Theoretical* 46: 115201.

Bailey DH, Borwein JM, Kimberley J and Ladd W (2016) Computer discovery and analysis of large Poisson polynomials. *Experimental Mathematics* 26: 349–363. DOI:https://www.tandfonline.com/doi/full/10.1080/10586458.2016.1180565.

Bailey DH, Borwein JM, Martin U, Salvy B and Taufer M (2014) Opportunities and challenges in 21st century experimental mathematical computation. URL `https://www.davidhbailey.com/dhbpapers/ICERM-2014.pdf`. ICERM workshop report.

Bailey DH and Broadhurst DJ (2000) Parallel integer relation detection: Techniques and applications. *Mathematics of Computation* 70(236): 1719–1736.

Bailey DH, Burgess N, Dongarra J, Fox A, Hittinger JAF and Rubio-Gonzaalez C (2019) Variable precision in mathematical and scientific computing. URL `https://www.davidhbailey.com/dhbpapers/icerm-varprec-prop.pdf`. Proposal for an ICERM workshop.

Buttari A, Dongarra J, Kurzak J, Luszczek P and Tomov S (2008) Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Transactions on Mathematical Software* 34: 17. DOI:http://doi.acm.org/10.1145/1377596.1377597.

Demmel J and Koev P (2005) The accurate and efficient solution of a totally positive generalized Vandermonde linear system. *SIAM Journal of Matrix Analysis Applications* 27: 145–152.

Diffenderfer J, Fox A, Hittinger J, Sanders G and Lindstrom P (2018) Error analysis of ZFP compression for floating-point data. URL https://arxiv.org/abs/1805.00546.

Ferguson HRP, Bailey DH and Arno S (1999) Analysis of PSLQ, an integer relation finding algorithm. *Mathematics of Computation* 68(225): 351–369.

Fousse L, Hanrot G, Lefevre V, Pelissier P and Zimmermann P (2007) MPFR: A multiple precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software* 33(2). DOI: https://doi.org/10.1145/1236463.1236468.

Gustafson J and Yonemoto I (2017) Beating floating point at its own game: Posit arithmetic. URL http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf.

Haidar A, Tomov S, Dongarra J and Higham NJ (2018) Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In: *Proceedings of SC18*. Piscataway, NJ, USA: IEEE Press, p. 47. DOI:https://doi.org/10.1145/3148226.3148237.

Hida Y, Li XS and Bailey DH (2001) Algorithms for quad-double precision floating point arithmetic. In: *15th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, pp. 155–162.

Higham NJ and Mary T (2018) A new approach to probabilistic rounding error analysis. URL http://eprints.maths.manchester.ac.uk/2673/. MIMS Preprint 2018.33.

Lindstrom P (2014) Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20: 2674–2683. DOI:https://doi.org/10.1109/TVCG.2014.2346458.

Lindstrom P, Lloyd S and Hittinger J (2018) Universal coding of the reals: Alternatives to IEEE floating point. In: *Proceedings of Next Generation Arithmetic (CoNGA '18)*. New York, NY: ACM, p. 5. DOI:https://doi.org/10.1145/3190339.3190344.

Ma D and Saunders M (2017) Solving multiscale linear programs using the simplex method in quadruple precision. In: Al-Baali M, Grandinetti L and Purnama A (eds.) *Recent Developments in Numerical Analysis and Optimization*. New York, NY: Springer. URL http://web.stanford.edu/group/SOL/reports/quadLP3.pdf.

Nguyen C, Rubio-Gonzalez C, Mehne B, Sen K, Iancu C, Demmel J, Kahan W, Lavrijsen W, Bailey DH and Hough D (2016) Floating-point precision tuning using blame analysis. In: *38th International Conference on Software Engineering (ICSE 2016)*. Washington, DC: IEEE Computer Society.

NVIDIA (2019) Deep learning SDK documentation. Technical report, NVIDIA. URL https://docs.nvidia.com/deeplearning/sdk/index.html.

Rubio-Gonzalez C, Nguyen C, Nguyen HD, Demmel J, Kahan W, Sen K, Bailey DH, Iancu C and Hough D (2013) Precimonious: Tuning assistant for floating-point precision. In: *Proceedings of SC13*. New York, NY: ACM.