MANCHESTER
1824

The University of Manchester

SIMULATING LOW PRECISION FLOATING-POINT ARITHMETIC

Higham, Nicholas J. and Pranesh, Srikara

2019

MIMS EPrint: **2019.4**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

SIMULATING LOW PRECISION FLOATING-POINT ARITHMETIC*

NICHOLAS J. HIGHAM† AND SRIKARA PRANESH*

Abstract. The half precision (fp16) floating-point format, defined in the 2008 revision of the IEEE standard for floating-point arithmetic, and a more recently proposed half precision format bfloat16, are increasingly available in GPUs and other accelerators. While the support for low precision arithmetic is mainly motivated by machine learning applications, general purpose numerical algorithms can benefit from it, too, gaining in speed, energy usage, and reduced communication costs. Since the appropriate hardware is not always available, and one may wish to experiment with new arithmetics not yet implemented in hardware, software simulations of low precision arithmetic are needed. We discuss how to simulate low precision arithmetic using arithmetic of higher precision. We examine the correctness of such simulations and explain via rounding error analysis why a natural method of simulation can provide results that are more accurate than actual computations at low precision. We provide a MATLAB function `chop` that can be used to efficiently simulate fp16 and bfloat16 arithmetics, with or without the representation of subnormal numbers and with the options of round to nearest, directed rounding, stochastic rounding, and random bit flips in the significand. We demonstrate the advantages of this approach over defining a new MATLAB class and overloading operators.

AMS subject classifications. 65F05, 65G50, 65Y04

Key words. floating-point arithmetic, half precision, low precision, IEEE arithmetic, fp16, bfloat16, subnormal numbers, mixed precision, simulation, rounding error analysis, round to nearest, directed rounding, stochastic rounding, bit flips, MATLAB

1. Introduction.

The 1985 IEEE standard 754 for floating-point arithmetic [26] brought to an end the turbulent period in scientific computing when different hardware manufacturers provided different floating-point arithmetics, some of which were better behaved than others. A particular problem had been that the lack of a guard digit in some floating-point arithmetics could cause algorithms that otherwise work perfectly to fail [22, sec. 2.4]. For the two decades following the introduction of the standard, virtually all floating-point hardware implemented IEEE single precision (fp32) or double precision (fp64) arithmetic.

The 2008 revision of the standard [27] defined a 16-bit half precision (fp16) storage format, but did not define operations on it. However, manufacturers have provided implementations of half precision arithmetic, using the natural extension of the rules for single and double precision. Half precision is available, for example, on the NVIDIA P100 (2016) and V100 (2017) GPUs, the AMD Radeon Instinct MI25 GPU (2017), and the ARM NEON architecture [1].

With regards to future chips, the Fujitsu A64FX Arm processor that will power Japan's first exascale computer will employ half precision [10]. Also the next generation AI chips of IBM will support an 8-bit floating-point format along with half precision [11], [50].

Another form of half precision arithmetic called bfloat16[1] was introduced by Google in its tensor processing units and will be supported by Intel in its forthcoming Nervana Neural Network Processor [40]. The same floating-point format is

*Version of March 20, 2019. **Funding**: This work was supported by MathWorks, Engineering and Physical Sciences Research Council grant EP/P020720/1, and the Royal Society. The opinions and views expressed in this publication are those of the authors, and not necessarily those of the funding bodies.

†School of Mathematics, University of Manchester, Manchester, M13 9PL, UK (`nick.higham@manchester.ac.uk`, `srikara.pranesh@manchester.ac.uk`).

[1] https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

suggested for ultra low power computing in [46], where it is called binary16alt.

While machine learning provides much of the impetus for the development of half precision arithmetic in hardware [14], [45], half precision is also attractive for accelerating general purpose scientific computing. It is already being used in weather forecasting and climate modelling [4], [7], [18], [39], [47] and the solution of linear systems of equations [2], [3], [15], [16], [17], [23]. The Summit machine at Oak Ridge National Laboratory, which leads the June and November 2018 Top 500 lists (https://www.top500.org), has a peak performance of 143.5 petaflops in the LINPACK benchmark[2], a benchmark that employs only double precision. For a genetics application that uses half precision, the same machine has a peak performance of 2.36 exaflops [12]. This difference in the peak performances highlights the importance of using low precision arithmetic to exploit the hardware. Library software that makes efficient use of these low precision formats is being developed—for example, MAGMA[3] [8], [48], [49].

While low-precision enabled hardware can provide substantial computational benefits, these are specialist devices to which not everyone has ready access. Furthermore, one may wish to experiment with new floating-point formats not yet available in hardware. In this work we investigate and propose ways to simulate low precision floating-point arithmetics. Our approach is general, but with a particular focus on numerical linear algebra kernels and implementation in MATLAB.

We propose in section 3 two simulation strategies, in both of which computations are carried out at higher precision and then rounded to lower precision. In the first variant, Simulation 3.1, we perform every scalar operation—addition, subtraction, multiplication, and division—in high precision and round the results to low precision. We explain why double rounding could vitiate this approach and identify precisions for which it provides a valid simulation. In the second variant, Simulation 3.2, we perform certain vector, matrix, or tensor kernels entirely in high precision and then round the final answer to low precision. Using rounding error analysis we analyze the accuracy and backward stability of matrix-matrix multiplication and solution of a triangular system when the second form of simulation is used. We show that the simulation provides results with forward errors that are essentially independent of n and for triangular systems are also independent of the condition number of the triangular matrix, so the results will generally be more accurate than for true low precision computations.

In section 4 we discuss the pros and cons of Moler's fp16 and vfp16 classes (data types) for MATLAB, which are instances of Simulation 3.2,. In section 5 we propose an alternative approach to simulating low precision arithmetic that stores data in single or double precision but rounds results to match the target precision and range. We provide a MATLAB function `chop` that rounds its argument to fp16, bfloat16, fp32, fp64, or a user-defined format with round to nearest, round towards plus or minus infinity, round towards zero, or stochastic rounding, and with optional support of subnormal numbers. The function also allows random bit flips in the significand in order to simulate soft errors, which could be due to running a processor at a reduced voltage [38]; such bit flips are explored in [47], for example. Computing reliably in the presence of soft errors is of interest in training of large scale neural networks [9]. In section 6 we demonstrate the performance advantages of `chop` over fp16/vfp16 and give some examples of the insights that can be gained by experimenting with low

[2] https://www.top500.org/lists/2018/11/

[3] https://icl.utk.edu/magma/software/index.html

TABLE 2.1

Parameters for bfloat16, fp16, fp32, and fp64 arithmetic: number of bits in significand and exponent and, to three significant figures, unit roundoff u, smallest positive (subnormal) number $x^s_{\min}$, smallest normalized positive number $x_{\min}$, and largest finite number $x_{\max}$.

	Signif.	Exp.	u	$x^s_{\min}$	$x_{\min}$	$x_{\max}$
bfloat16	8	8	3.91×10^{-3}	9.18×10^{-41} [a]	1.18×10^{-38}	3.39×10^{38}
fp16	11	5	4.88×10^{-4}	5.96×10^{-8}	6.10×10^{-5}	6.55×10^{4}
fp32	24	8	5.96×10^{-8}	1.40×10^{-45}	1.18×10^{-38}	3.40×10^{38}
fp64	53	11	1.11×10^{-16}	4.94×10^{-324}	2.22×10^{-308}	1.80×10^{308}

[a] In Intel's bfloat16 specification subnormal numbers are not supported [28], so any number less than $x_{\min}$ in magnitude is flushed to zero. The value shown would pertain if subnormals were supported.

precision computations in MATLAB.

2. Low precision floating-point arithmetics. We assume that IEEE standard single precision and double precision are available in hardware and that we wish to simulate arithmetics of lower precision. Table 2.1 shows the arithmetics of particular interest in this work. Even lower precisions, including 8-bit (quarter precision) formats [32], [34], [46], [50] and a 9 bit format [25], are also of interest, but they are not standardized.

We wish to simulate arithmetic of the appropriate precision and range, with various rounding modes, and make the support of subnormal numbers optional. IEEE arithmetic supports subnormal numbers, but bfloat16 as defined by Intel does not [28]. The main reason for a hardware designer not to support subnormal numbers is to simplify the implementation, but when they are handled in software subnormal numbers can degrade the performance of a code[4]. One question that simulation can help address is to what extent subnormal numbers make a difference to the behavior of an algorithm for particular types of data.

3. Simulation models and their properties. We discuss two different ways to simulate low precision arithmetic. We need to achieve both the correct precision and the correct range. Correct range refers not only to ensuring that numbers too large for the target range overflow, but also that numbers in the subnormal range are correctly represented with less precision than normalized numbers. Our approach is to carry out all computations in single precision or double precision, which we refer to as higher precision, and round back to the target precision. This can be done at two different levels of granularity: by rounding every individual scalar operation or by allowing a group of operations to be carried out in higher precision and then rounding the result to lower precision. We investigate how these two possibilities affect the accuracy and stability of computations.

3.1. Rounding every operation. The most natural way to simulate low precision arithmetic is as follows. Here, rounding means any of the following four IEEE arithmetic rounding modes: round to nearest with ties broken by rounding to an even least significant bit, round towards plus infinity or minus infinity, and round towards zero. We will refer to the low precision format of interest as the target format, and we assume that the operands are given in the target format. In practice, numbers in

[4] https://devblogs.nvidia.com/cuda-pro-tip-flush-denormals-confidence/, https://en.wikipedia.org/wiki/Denormal_number

the target format might actually be stored in a higher precision format, padded with zero bits (we will use this approach in Section 5).

SIMULATION 3.1. *Simulate every scalar operation by converting the operands to fp32 or fp64, carrying out the operation in fp32 or fp64, then rounding the result back to the target format.*

By computing a result first in fp32 or fp64 and then rounding to the target format we are rounding the result twice. It is well known that double rounding can give a result different from rounding directly to the target format. A simple example with round to nearest in base 10 is rounding 1.34905 to two significant figures: rounding first to three significant figures gives 1.35 and then rounding to two significant figures gives 1.4, but rounding directly to two significant figures gives 1.3. Fortunately, for the operations of addition, subtraction, multiplication, division, and square root rounding first to fp32 or fp64 and then to bfloat16 or fp16 gives the same result as rounding directly to bfloat16 or fp16, even allowing for subnormal numbers. This is shown by results in [13] and [42], which essentially require that the format used for the first rounding has a little more than twice as many digits in the significand as the target format (this condition is needed for round to nearest, but not for directed rounding). Since double rounding does not change results in simulating bfloat16 or fp16 via fp32 or fp64, we can conclude that Simulation 3.1 is equivalent to carrying out the computation wholly in bfloat16 or fp16, provided that the range is correctly handled.

If the low precision numbers are stored in a special format then Simulation 3.1 can be expensive because every scalar operation incurs the overhead of converting between different floating-point formats.

A subtlety arises in elementary function evaluations, in that Simulation 3.1 is likely to provide a correctly rounded result whereas a computation entirely at the target precision may be unable to do so.q For example, in an exponentiation x^y it is hard to guarantee a correctly rounded result without using extra precision [37, sec. 13.2]. However, since elementary functions are not included in the IEEE standard the computed results will in any case depend upon the implementation.

Simulation 3.1 has been used to develop a bfloat16 data type and basic arithmetic operations on it for Julia [30]. A C++ header file for fp16 is available [41]; we note that it does not convert to fp16 after every scalar operation, but retains a number in fp32 as long as possible, in order to reduce the overhead caused by repeated conversions. This implementation therefore is not of the form Simulation 3.1. SciPy has a float16 data type[5], which appears to use Simulation 3.1 with computations in single precision.

Another instance of Simulation 3.1 is the rpe (reduced floating-point precision) library of Dawson and Düben [6], which provides a derived type and overloaded operators for Fortran and was developed for use in weather and climate modeling. It emulates the specified precision but in general uses the exponent range of double precision.

3.2. Rounding kernels. We now consider a weaker form of simulation. Here, "kernel" can be a vector, matrix, or tensor operation.

SIMULATION 3.2. *Simulate isolated scalar operations as in Simulation 3.1. Implement appropriate kernels by carrying out the kernel in fp32 or fp64 then rounding the result back to the target precision.*

With this form of simulation we could compute the product of two matrices A and

[5]https://docs.scipy.org/doc/numpy-1.15.4/user/basics.types.html

B by converting A and B to fp64, computing the product in fp64, and then rounding the result back to the target format.

We now consider what can be said about the accuracy and backward stability of two particular kernels under Simulation 3.2: matrix multiplication and the solution of a triangular system.

Denote the lower precision by fp_low and the higher precision (fp32 or fp64) by fp_high. We assume round to nearest, use the standard model of floating-point arithmetic [22, sec. 2.2], and ignore the possibility of overflow and underflow. We denote by u_ℓ and u_h the unit roundoffs for the lower and higher precisions and write

$$\gamma_\ell(n) = \frac{nu_\ell}{1 - nu_\ell}, \quad \gamma_h(n) = \frac{nu_h}{1 - nu_h}$$

for $nu_\ell < 1$ and $nu_h < 1$.

Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$, and let $C = AB$. The product $\widehat{C}$ computed in precision fp_high satisfies [22, sec. 3.5]

$$|C - \widehat{C}| \le \gamma_h(n)|A||B|, \tag{3.1}$$

where the absolute value is taken componentwise. Rounding $\widehat{C}$ to $\widetilde{C}$ in fp_low, we have [22, Thm. 2.2]

$$\widetilde{C} = \widehat{C} + \Delta C, \quad |\Delta C| \le u_\ell|\widehat{C}|. \tag{3.2}$$

Hence

$$|C - \widetilde{C}| = |C - \widehat{C} + \widehat{C} - \widetilde{C}| \le \gamma_h(n)|A||B| + u_\ell|\widehat{C}|.$$

From (3.1) we obtain

$$|\widehat{C}| \le (1 + \gamma_h(n))|A||B|$$

and then

$$|C - \widetilde{C}| \le \big(\gamma_h(n) + u_\ell(1 + \gamma_h(n))\big)|A||B|.$$

Hence we certainly have

$$|C - \widetilde{C}| \le 2u_\ell|A||B| \tag{3.3}$$

as long as $\gamma_h(n)(1 + u_\ell) + u_\ell \le 2u_\ell$, which is equivalent to

$$n \le \frac{u_\ell}{u_h(1 + 2u_\ell)}. \tag{3.4}$$

When fp_low and fp_high are fp16 and fp64, respectively, this bound is 4.3×10^{12}, which covers all n of current practical interest for dense matrix multiplication. The bound (3.3) is to be compared with the bound

$$|C - \widehat{C}| \le \gamma_\ell(n)|A||B| \tag{3.5}$$

that holds for the computation done entirely in fp_low. The difference is that in (3.3) the constant does not depend on n, as long as (3.4) holds. Looked at another way, the constant in (3.5) exceeds 1 once $nu_\ell \ge 1$ (indeed the bound is only valid for $nu_\ell < 1$), whereas the constant in (3.3) is of order u_ℓ and the bound holds for n up to a much larger value. This demonstrates that the results from Simulation 3.2 will in general be more accurate than for actual low precision computation.

Now consider the solution of a triangular system $Tx = b$, where $T \in \mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^n$. Solving by substitution in fp_high we know that the computed solution $\widehat{x}$ satisfies the componentwise backward error result [22, Thm. 8.5]

$$(T + \Delta T)\widehat{x} = b, \quad |\Delta T| \le \gamma_h(n)|T|. \tag{3.6}$$

Rounding to fp_low gives

$$\widetilde{x} = (I + D)\widehat{x}, \quad D = \mathrm{diag}(\delta_i), \quad |\delta_i| \le u_\ell. \tag{3.7}$$

First, we consider the backward error. We have

$$|T\widetilde{x} - b| = |T\widehat{x} + TD\widehat{x} - b| = |-\Delta T\widehat{x} + TD\widehat{x}| \le (\gamma_h(n) + u_\ell)|T||\widehat{x}|,$$

which implies that [22, Thm. 7.3]

$$(T + \Delta T_h)\widetilde{x} = b, \quad |\Delta T_h| \le (u_\ell + \gamma_h(n))|T|. \tag{3.8}$$

If (3.4) holds then we certainly have $|\Delta T_h| \le 2u_\ell|T|$.

The conclusion is that $\widetilde{x}$ has the same componentwise form of backward error as if it had been computed entirely in fp_low, but with no dependence on n as long as (3.4) holds.

Turning to the forward error, (3.6) implies [22, Thm. 7.4]

$$\frac{\|x - \widehat{x}\|_\infty}{\|x\|_\infty} \le \frac{\mathrm{cond}(T, x)\gamma_h(n)}{1 - \mathrm{cond}(T)\gamma_h(n)}, \tag{3.9}$$

where

$$\mathrm{cond}(T, x) = \frac{\|\,|T^{-1}||T||x|\,\|_\infty}{\|x\|_\infty}, \quad \mathrm{cond}(T) = \|\,|T^{-1}||T|\,\|_\infty.$$

Combined with (3.7) this gives

$$\frac{\|x - \widetilde{x}\|_\infty}{\|x\|_\infty} \lesssim \frac{\mathrm{cond}(T, x)\gamma_h(n)}{1 - \mathrm{cond}(T)\gamma_h(n)} + u_\ell.$$

Hence, since $\mathrm{cond}(T, x) \le \mathrm{cond}(T)$, as long as $\gamma_h(n)\,\mathrm{cond}(T) \lesssim u_\ell$, that is,

$$n\,\mathrm{cond}(T) \lesssim \frac{u_\ell}{u_h} \tag{3.10}$$

we will have

$$\frac{\|x - \widetilde{x}\|_\infty}{\|x\|_\infty} \lesssim 2u_\ell, \tag{3.11}$$

and the right-hand side of (3.10) is 4.3×10^{12} for fp_low as fp16 and fp_high as fp64. This is to be compared with the bound

$$\frac{\|x - \widehat{x}\|_\infty}{\|x\|_\infty} \le \frac{\mathrm{cond}(T, x)\gamma_\ell(n)}{1 - \mathrm{cond}(T)\gamma_\ell(n)}$$

(analogous to (3.9)) that we would expect for a solution computed entirely in fp_low. Hence when (3.10) is satisfied we can expect $\widetilde{x}$ to be more accurate than it would be

were it to be computed entirely in fp_low, for two reasons: the constant term in (3.11) does not depend on n and the condition number does not appear in the bound.

Suppose now that T is upper triangular and that we implement back substitution for $Tx = b$ as

$$x_i = \mathrm{fl}_{\mathrm{low}} \left(\frac{b_i - \mathrm{fl}_{\mathrm{low}} \left(\mathrm{fl}_{\mathrm{high}} \left(\sum_{j=i+1}^{n} t_{ij} x_j \right) \right)}{t_{ii}} \right),$$

that is, the underlying inner product is evaluated in fp_high but the subtraction and division are in fp_low. It is easy to see that we now obtain a backward error result of the same form as (3.8) but with a different constant that again is independent of n unless n is very large, but we cannot conclude that a small normwise forward error is obtained. So this mode of evaluation is closer to evaluation wholly in fp_low than if we execute the substitution in fp_high and then round to fp_low.

In summary, our analysis of Simulation 3.2 shows that the results it yields can be more accurate than those from true low precision computation in two respects. First, by evaluating the whole kernel at precision fp_high we obtain a forward error bound for matrix multiplication and a backward error bound for the solution of triangular systems that have essentially no dependence on n. Second, the forward error for $Tx = b$ will be essentially independent of the condition number. The second phenomenon no longer holds if only the underlying inner products are evaluated at the higher precision.

Our conclusion is that care is needed in deciding at what level of granularity to apply the higher precision. The lower the level, the more realistic the simulation will be.

4. Moler's fp16 and vfp16 classes. We now focus on simulating low precision arithmetic in MATLAB. The `fp16` half-precision MATLAB class of Moler [32], [34] introduces a new data type fp16 that implements the IEEE fp16 storage format and overloads some basic functions for fp16 arguments. It uses Simulation 3.2. Although it was designed to be economical in coding rather than to have good performance, the fp16 class has proved very useful in research [3], [23], [24].

The fp16 class includes the following two functions (which live in the `@fp16` directory).

```
function z = plus(x,y)
   z = fp16(double(x) + double(y));
end

function [L,U,p] = lu(A)
    [L,U,p] = lutx(A);
end
```

These functions do the computations in double precision and round the result back to fp16. The `plus` function is used for matrix addition and is invoked when MATLAB encounters the `+` operator with fp16 operands. The `lutx` function called from `lu` is a "textbook" implementation of LU factorization with partial pivoting, implemented as a straightforward MATLAB program file in [32]. When this function is called with an fp16 argument all the operations within the function are done by the overloaded fp16 operators. The key lines of `lutx` are

```
      % Compute multipliers
      i = k+1:n;
```

```
      A(i,k) = A(i,k)/A(k,k);

      % Update the remainder of the matrix
      j = k+1:n;
      A(i,j) = A(i,j) - A(i,k)*A(k,j);
```

For fp16 variables, the multiplication in the last of these lines calls the function `mtimes` in the `@fp16` directory.

```
function z = mtimes(x,y)
    z = fp16(double(x) * double(y));
end
```

Because `mtimes` is being called to evaluate a rank-1 matrix, and hence it performs one multiplication per entry and no additions, that line faithfully simulates fp16 arithmetic. However, an expression `A*B` with fp16 matrices `A` and `B` will in general be evaluated as multiple double precision multiplications and additions with a final rounding to fp16, that is, as in Simulation 3.2 with matrix multiplication as the kernel.

The solution of linear systems (as invoked by backslash) is done by the function `mldivide`, which has the form

```
function z = mldivide(x,y)
   z = fp16(double(x) \ double(y));
end
```

Here again, the whole solution process is carried out in double precision then rounded to fp16, so this is Simulation 3.2 with linear system solution as the kernel. This is not equivalent to carrying out the computation in fp16, as we saw in the last section in the special case of triangular systems.

From our analysis in the previous section we conclude that in the fp16 class both matrix multiplication and linear system solutions enjoy error bounds that are essentially independent of n and the forward error for the latter will be condition number independent (for all n of practical interest), so we can expect to obtain more accurate results than for actual fp16 computation.

Moler's fp16 class has the advantage that it is a true MATLAB class and so program files (ones containing sequences of MATLAB commands) can work seamlessly when provided with fp16 input arguments. However, arithmetic in this class is very slow because of both the overhead of object orientation in MATLAB and the cost of converting to and from the fp16 storage format. In the experiments with linear systems in [23], [24] the dimension was restricted to a few hundred in order to obtain reasonable computing times.

Moler has also written a class `vfp16` that allow the partitioning of a 16-bit word between significand and exponent to be varied, in particular allowing bfloat16 to be simulated [36]. This class also allows subnormals to be supported or not, and it allows fused multiply-adds (FMAs) to be done within the inner products inside a matrix multiplication.

5. Parametrized rounding. We now develop a different strategy to simulate low precision arithmetic in MATLAB. Instead of defining a new data type, we store all variables in double precision or single precision but ensure that they have the precision and range corresponding to the lower precision. Computations are performed in the storage format but rounded to the target precision.

Table 5.1
Rounding modes supported by MATLAB function `chop`*, as specified by the function's* `options.round` *parameter.*

`options.round`	Rounding modes
1	Round to nearest, with tie broken by rounding to the number with an even last bit
2	Round towards ∞
3	Round towards $-\infty$
4	Round towards zero
5	Stochastic rounding, mode 1: round to the next larger or smaller floating-point number x with probability equal to 1 minus the distance to x
6	Stochastic rounding, mode 2: round up or down with equal probability.

This strategy is facilitated by our MATLAB function `chop`. It takes an fp64 or fp32 argument and rounds it to a precision chosen from among bfloat16, fp16, fp32, fp64, or a user-defined format, using one of the rounding modes listed in Table 5.1. We support stochastic rounding because of its increasing adoption in machine learning [14] and the need for better understanding of its behavior. The user-defined format allows selection of, for example, the nonstandard fp8 and fp16 formats proposed in [50]. See the appendix for a link to `chop` and examples of its usage.

We note that MATLAB has an undocumented function `feature('setround',mode)`, (as used in [43], for example), which sets the rounding mode on the processor. Here, `mode` can specify any of the first four rounding modes in Table 5.1. This function works for the built-in fp32 and fp64 data types only, so is not useful within `chop`, but we have used it to test `chop`.

The `chop` function allows subnormal numbers to be supported or flushed to zero. The reason we allow fp64 as an output option is that by asking for subnormals not to be supported one can explore the effect of flushing subnormals to zero in fp64. The `chop` function can also create random bit flips in the significand, which enable soft errors to be simulated.

Chopping refers to rounding towards zero, which was the form of rounding used in certain computer architectures prior to the introduction of the IEEE standard. The less specific name `round` would therefore be preferable to `chop`, but `round` is a built-in MATLAB function that rounds to the nearest integer or to a specified number of decimal places. The `chop` function is consistent with earlier MATLAB functions of the same name. The original Fortran version of MATLAB had a function `chop` [33], [35]:

> "The result of each floating point operation may also be 'chopped' to simulate a computer with a shorter word length ... the statement CHOP(p) causes the p least significant octal or hexadecimal digits in the result of each floating point operation to be set to zero."

This function was implemented by using the Fortran binary operator `.AND.` to mask the result of every arithmetic operation. Our function `chop` builds on a function of the same name in the Matrix Computation Toolbox [19] that has been successfully used to simulate single precision arithmetic[6] with round to nearest in, for example, [5], [20], [21], [22].

Applying `chop` to the result of every arithmetic operation can be cumbersome.

[6]MATLAB first provided single precision arithmetic in MATLAB 7.0 (Release 14) in 2004.

For example, the scalar computation `x = a + b*c` would be rewritten

```
x = chop(a + chop(b*c))
```

However, if an algorithm is expressed in a vectorized fashion, just a few invocations of `chop` may be required. Moler's function `lutx`, discussed in the previous section, is a good example. To produce a modified function `lutx_chop` that uses `chop` we need to change just two lines, as shown in this extract:

```
% Compute multipliers
i = k+1:n;
A(i,k) = chop(A(i,k)/A(k,k));

% Update the remainder of the matrix
j = k+1:n;
A(i,j) = chop(A(i,j) - chop(A(i,k)*A(k,j)));
```

The expression `chop(A(i,k) * A(k,j)))` forms a rank-1 matrix in double precision and then rounds every element to the target precision.

For optimal efficiency we need to minimize the number of calls to `chop`, and this may require reformulating a computation. For example, consider the multiplication $C = AB$ of $n \times n$ matrices. If we code at the scalar level, $C(i,j) = \sum_{k=1}^{n} A(i,j)B(k,j)$, then $O(n^3)$ calls to `chop` will be required, as `chop` will be applied to the result of every scalar multiplication and addition. Using the formulation $C(:,j) = \sum_{k=1}^{n} A(:,k)B(k,j)$ requires $O(n^2)$ calls, but the outer product formulation $C = \sum_{k=1}^{n} A(:,k)B(k,:)$ requires just $O(n)$ calls to `chop` (analogously to `lutx_chop`) and so is the most efficient way to code matrix multiplication with `chop`.

We discuss the key features of the function `chop`. The general form of a call is `chop(x,options)`, where `options` is a structure used to define the arithmetic format, the rounding mode, whether subnormal numbers are supported, and whether random bit flips are required. The second input argument, `options`, can be omitted as long as the required options are set up on the initial call to `chop`. A persistent variable in the function stores the selected options for future use during a session. Importantly, all arithmetic operations in `chop` are vectorized, so best efficiency is obtained by calling it with a vector or matrix argument. The rounding operation is done in the separate function `roundit`.

A significant advantage of `chop` over the use of a class is that since all data is maintained in single precision or double precision the overheads of conversions to and from other formats are avoided.

The simple nature of `chop` provides some benefits. Mixed precision arithmetic is easily implemented, simply by using different `options` arguments on different invocations of `chop`. An FMA `y = a + b*c` can be simulated using `chop(a + b*c)`: assuming the target precision is single or less and that `a`, `b`, and `c` are outputs from `chop` at that precision, the product `b*c` will be computed exactly at double precision and so `a + b*c` will be obtained correctly rounded to double-precision and will then be correctly rounded to the target precision.

6. Experiments. We now describe experiments with the `chop` function that illustrate some of the points made in the previous sections and show how insight into low precision computations can be gained.

Our experiments were carried out in MATLAB R2018b on a machine with an Intel Core i7-6800K CPU with 6 cores running at 3.4–3.6GHz. Our codes are available at

Table 6.1

The error measure $\min\{\, \epsilon > 0 : |C - \widehat{C}| \le \epsilon |A||B| \,\}$ *for random* $n \times n$ A *and* B *from the uniform* $[0,1]$ *distribution, where* $C = AB$ *and* $\widehat{C}$ *is the computed product. The product* C *is computed entirely in fp16 or in fp64 and then rounded to fp16 ("fp64* $\to$ *fp16").*

n	fp16	fp64 $\to$ fp16
500	1.19e-02	4.88e-04
1000	2.30e-02	4.88e-04
1500	4.07e-02	3.67e-04
2000	4.60e-02	4.88e-04
2500	6.88e-02	4.29e-04

Table 6.2

Time in seconds for LU factorizing with partial pivoting an $n \times n$ *matrix* `randn(n)` *using Moler's* `lutx` *function. The function* `lutx_chop` *is* `lutx` *modified with calls to* `chop`*, as discussed in section 5, with fp16 arithmetic and subnormals supported.*

n	`lutx` with fp16 class	`lutx_chop` at fp16	`lutx` in double
50	6.3	3.8e-2	1.6e-3
100	4.9e1	4.9e-2	3.2e-3
250	7.2e2	3.4e-1	3.7e-2
500	5.8e3	3.4	4.1e-1

https://github.com/SrikaraPranesh/LowPrecision_Simulation.

We note that the defaults for `chop` are to support subnormals for fp16, fp32, and fp64 and not to support them for bfloat16.

The first experiment compares two implementations of matrix multiplication in fp16 arithmetic: the first carries out every operation in fp16 and the second computes the product in fp64 and then rounds to fp16. The error bound in (3.3) for the latter computation is constant, whereas for fp16 the error bound in (3.5) is proportional to n. The results in Table 6.1 for random fp16 A and B confirm the behavior suggested by the bounds, with the fp64 $\to$ fp16 errors approximately constant and the fp16 errors growing proportional to n.

Table 6.2 gives the times to LU factorize with partial pivoting an $n \times n$ matrix with random fp16 entries from the normal $(0, 1)$ distribution using Moler's `lutx` function called by his fp16 class and using `lutx_chop` with fp16 arithmetic and with subnormal numbers supported. Results generated by the `lutx` function and our function `lutx_chop` are bitwise identical. We see that `lutx_chop` is of order 1000 times faster than `lutx`. Clearly, the code using the fp16 class is spending almost all its time on overheads and not on floating-point arithmetic. The last column of the table shows that the conversions to fp16 in `chop` cause a factor 10 or so slowdown compared with running the code in double precision. This is comparable to the typical slowdown reported for the rpe library [6], which is compiled Fortran rather than semi-compiled MATLAB.

The ability to vary the precision with which a code executes can be useful in a variety of situations. As a simple example, consider the harmonic series $1 + 1/2 + 1/3 + \cdots$. The series diverges, but when summed in the natural order in floating-point arithmetic it converges, because the partial sums grow while the addends decrease, and eventually the addends are small enough that they do not change the partial sum. Table 6.3 shows the computed harmonic series for five different precisions, along with how many terms are added before the sum becomes constant. The results for the first four arithmetics were computed with MATLAB using the `chop` function (and the bfloat16 and fp16 results have been reproduced in Julia [44]). The arithmetic denoted

TABLE 6.3

Computed harmonic series $\sum_{i=1}^{\infty} 1/i$ in five precisions of floating-point arithmetic and number of terms for the computed sum to converge.

Arithmetic	Sum	Terms
fp8	3.5000	16
bfloat16	5.0625	65
fp16	7.0859	513
fp32	15.404	2097152
fp64	34.122	$2.81 \cdots \times 10^{14}$

TABLE 6.4

Computed harmonic series $\sum_{i=1}^{\infty} 1/i$ in fp16 and bfloat16 arithmetic for six rounding modes, with number of terms for the computed sum to converge. The stochastic rounding modes are defined in Table 5.1.

	bfloat16		fp16	
Rounding mode	Sum	Terms	Sum	Terms
To nearest	5.0625	65	7.0859	513
Towards ∞	2.2×10^{12}	5013	∞	13912
Towards $-\infty$	4.0000	41	5.7461	257
Towards 0	4.0000	41	5.7461	257
Stochastic, mode 1	4.1875	38	6.1797	277
Stochastic, mode 2	4.0938	34	6.2148	257

by fp8 is not standard, but is a form of quarter precision implemented in Moler's fp8 class [32], [34]; it apportions 5 bits to the significand and 3 to the exponent. The fp64 value is reported by Malone [31], based on a computation that took 24 days.

In Table 6.4 we focus on bfloat16 and fp16 and show the computed harmonic series for all the different rounding modes supported by `chop`. We see significant variation of the result with the rounding mode.

Next, we show the value of being able to control whether or not subnormal numbers are supported. We evaluate the sum $\sum_{i=1}^{n} 1/(n-i+1)^2$ for $n = 10^2, 10^3, 10^4$ in fp16 arithmetic with and without subnormal numbers, using the `chop` function. As Table 6.5 shows, for $n = 10^3, 10^4$ the sum is larger when subnormal numbers are supported than when they are flushed to zero. This is because the terms are summed from smallest to largest, so that the smaller terms can contribute to the sum, even if they are subnormal [21], [22, sec. 4.2].

The next experiment emphasizes how quickly bfloat16 and fp16 can lose accuracy and the importance, again, of subnormal numbers. We solve the ordinary differential equation $y' = -y =: f(x, y)$ over $[0, 1]$, with $y(0) = 0.01$, using Euler's method, $y_{k+1} = y_k + hf(x_k, y_k)$, $y_0 = y(0)$, where $h = 1/n$, with several n between 10 and 10^5. The absolute (global) errors $|y(1) - \widehat{y}_n|$ are plotted in Figure 6.1. Since the global error in Euler's method is of order h, we expect a linear decrease in the error as n increases until rounding error starts to dominate, at which point the error will increase linearly, and the theory suggests that the optimal h is of order $u^{1/2}$, corresponding to an optimal n of order $u^{-1/2}$ [29, pp. 374–376]. This is roughly what we see in Figure 6.1: note that $u^{-1/2} = 16$ for bfloat16. However, perhaps surprising is that when subnormal numbers are flushed to zero in fp16 the errors start to grow much sooner than when subnormals are supported. In fact, once $n \gtrsim 100$ the $hf(x_k, y_k)$ term underflows and the computed y_k are constant when subnormals are not supported.

Our final experiment compares the different rounding modes in the solution of linear systems. We generate 100 linear systems $Ax = b$, with $A \in \mathbb{R}^{100 \times 100}$ and b having random fp16 elements from the normal (0,1) distribution. Each system is

Table 6.5
The computed sum $\sum_{i=1}^{n} 1/(n-i+1)^2$ *in fp16.* $\widehat{s}_1$*: with subnormal numbers,* $\widehat{s}_2$*: without subnormal numbers.*

n	$\widehat{s}_1$	$\widehat{s}_2$	$\widehat{s}_1 - \widehat{s}_2$
100	1.635742e+00	1.635742e+00	0.00e+00
1000	1.644531e+00	1.637695e+00	6.84e-03
10000	1.645508e+00	1.637695e+00	7.81e-03

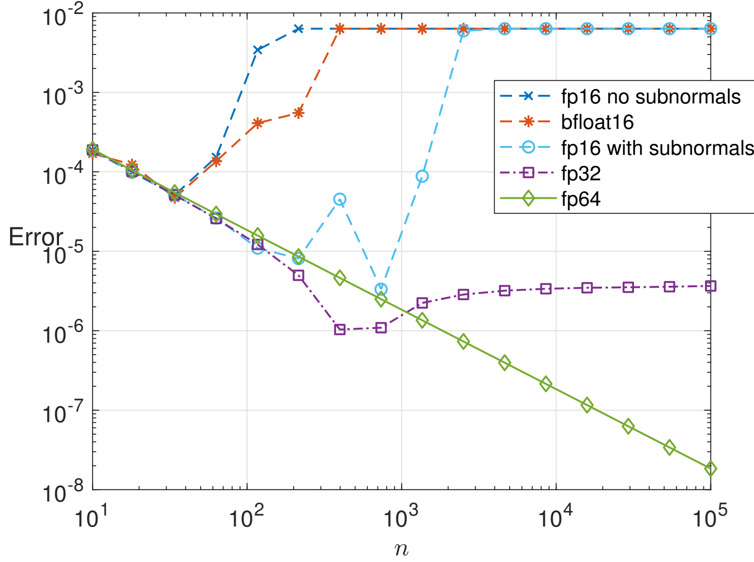

Figure 6.1. *Absolute errors in Euler's method for* $y' = -y$*,* $y(0) = 0.01$*, over* $[0, 1]$*, in different simulated arithmetics, with stepsize* $h = 1/n$*.*

solved by LU factorization with partial pivoting and substitution using `lutx_chop`, with fp16 simulation and each of the supported rounding modes. Table 6.6 shows the mean, maximum, and minimum backward errors $\|b - A\widehat{x}\|_1/(\|A\|_1\|x\|_1 + \|b\|_1)$ for the computed solutions $\widehat{x}$. As might be expected, round to nearest produces the smallest backward errors.

7. Conclusions. We have analyzed the use of higher precision arithmetic with rounding to simulate lower precision arithmetic, showing that it is valid way to simulate bfloat16 or fp16 using fp32 or fp64 arithmetic. It is essential to keep in mind the limitations pointed out in section 3, notably that results of greater accuracy than expected can be obtained if one carries out sequences of operations at higher precision before rounding back to the target format.

Our MATLAB function `chop` rounds to a specified target format, including fp16 or bfloat16 or a user-specified format, respecting precision and range, and offers six different rounding modes and optional support for subnormal numbers. It incurs a performance penalty of about an order of magnitude when applied to a pure MATLAB program file for LU factorization, compared with execution in double precision, which compares favorably with the overhead of Moler's fp16/vfp16 MATLAB class.

Software simulation provides a useful tool for understanding the effects of lowering the precision in numerical computations, as well as the benefits of representing subnormal numbers instead of flushing them to zero, as the examples in section 6

Table 6.6

Backward errors for 100 random linear systems $Ax = b$ with $A \in \mathbb{R}^{100\times100}$ solved in fp16 arithmetic using LU factorization with partial pivoting and substitution. The stochastic rounding modes are defined in Table 5.1.

	Round to nearest	Round towards $+\infty$	Round towards $-\infty$	Round towards zero	Stochastic rounding 1	Stochastic rounding 2
Mean	5.24e-04	3.47e-03	3.50e-03	3.45e-03	6.86e-04	9.09e-04
Min	3.52e-04	1.87e-03	2.05e-03	1.94e-03	4.75e-04	6.55e-04
Max	7.74e-04	6.92e-03	5.72e-03	6.51e-03	1.44e-03	6.51e-03

demonstrate. It also provides a convenient way to explore floating-point formats not supported in hardware as well as stochastic rounding, which is becoming popular in machine learning.

Appendix A. The MATLAB function `chop`.

The function `chop` uses the function `roundit`. Both functions are available from https://github.com/higham/chop. Test codes `test_chop` and `test_roundit` thoroughly test these functions. There are two main usages of `chop`. First, one can pass `options` with every call:

```
options.precision = 's'; options.round = 5; options.subnormal = 1;
...
A(i,j) = chop(A(i,j) - chop(A(i,k) * A(k,j),options),options);
```

Here, `options.precision = 's'` specifies that the precision is single, `options.round = 5` specifies stochastic rounding, mode 1 (see Table 5.1), and `options.subnormal = 1` specifies that subnormal numbers are not flushed to zero. For full details of the options see the link given at the start of the section. The above usage is rather tedious and produces cluttered code. Instead we can set up the arithmetic parameters on a call of the form `chop([],options)` and exploit the fact that subsequent calls with just one input argument will reuse the previously specified `options`:

```
options.precision = 's'; options.subnormal = 1; chop([],options)
...
A(i,j) = chop(A(i,j) - chop(A(i,k)*A(k,j)));
```

The current value of `options` stored inside the function (in a persistent variable, whose value is retained during the session until the function is cleared with `clear chop`) can be obtained with

```
[~,options] = chop
```

REFERENCES

[1] *ARM Architecture Reference Manual. ARMv8, for ARMv8-A Architecture Profile*. ARM Limited, Cambridge, UK, 2018. Version dated 31 October 2018. Original release dated 10 April 2013.

[2] Erin Carson and Nicholas J. Higham. A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM J. Sci. Comput.*, 39(6): A2834–A2856, 2017.

[3] Erin Carson and Nicholas J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J. Sci. Comput.*, 40(2):A817–A847, 2018.

[4] Matthew Chantry, Tobias Thornes, Tim Palmer, and Peter Düben. Scale-selective precision for weather and climate forecasting. *Monthly Weather Review*, 147(2):645–655, 2019.

[5] Anthony J. Cox and Nicholas J. Higham. Accuracy and stability of the null space method for solving the equality constrained least squares problem. *BIT*, 39(1):34–50, 1999.

[6] Andrew Dawson and Peter D. Düben. rpe v5: An emulator for reduced floating-point precision in large numerical simulations. *Geoscientific Model Development*, 10(6):2221–2230, 2017.

[7] Andrew Dawson, Peter D. Düben, David A. MacLeod, and Tim N. Palmer. Reliable low precision simulations in land surface models. *Climate Dynamics*, 51(7):2657–2666, 2018.

[8] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. Accelerating numerical dense linear algebra calculations with GPUs. In *Numerical Computations with GPUs*, Volodymyr Kindratenko, editor, Springer International Publishing, Cham, 2014, pages 3–28.

[9] Sanghamitra Dutta, Ziqian Bai, Tze Meng Low, and Pulkit Grover. CodeNet: Training large scale neural networks in presence of soft-errors. *arXiv e-prints*, page 54, 2019. arXiv:1903.01042.

[10] Michael Feldman. Fujitsu reveals details of processor that will power Post-K supercomputer. https://www.top500.org/news/fujitsu-reveals-details-of-processor-that-will-power-post-k-supercomputer, August 2018. Accessed November 22, 2018.

[11] Michael Feldman. IBM takes aim at reduced precision for new generation of AI chips. https://www.top500.org/news/ibm-takes-aim-at-reduced-precision-for-new-generation-of-ai-chips/, December 2018. Accessed January 8, 2019.

[12] Michael Feldman. Record-breaking exascale application selected as Gordon Bell finalist. https://www.top500.org/news/record-breaking-exascale-application-selected-as-gordon-bell-finalist/, September 2018. Accessed January 8, 2019.

[13] Samuel A. Figueroa. When is double rounding innocuous? *SIGNUM News*, 30(3):21–26, 1995.

[14] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *JMLR: Workshop and Conference Proceedings*, 2015, pages 1737–1746.

[15] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Panruo Wu, Srikara Pranesh, Stanimire Tomov, and Jack Dongarra. The design of fast and energy-efficient linear solvers: On the potential of half-precision arithmetic and iterative refinement techniques. In *Computational Science—ICCS 2018*, Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack Dongarra, and Peter M. A. Sloot, editors, Springer International Publishing, Cham, 2018, pages 586–600.

[16] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18 (Dallas, TX), Piscataway, NJ, USA, 2018, pages 47:1–47:11. IEEE Press.

[17] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17 (Denver, CO), November 2017, pages 10:1–10:8.

[18] Sam Hatfield, Peter Düben, Matthew Chantry, Keiichi Kondo, Takemasa Miyoshi, and Tim Palmer. Choosing the optimal numerical precision for data assimilation in the presence of model error. *Journal of Advances in Modeling Earth Systems*, 10(9):2177–2191, 2018.

[19] Nicholas J. Higham. The Matrix Computation Toolbox. http://www.maths.manchester.ac.uk/~higham/mctoolbox.

[20] Nicholas J. Higham. The accuracy of solutions to triangular systems. *SIAM J. Numer. Anal.*, 26(5):1252–1265, 1989.

[21] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput.*, 14(4):

783–799, 1993.

[22] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002. xxx+680 pp. ISBN 0-89871-521-0.

[23] Nicholas J. Higham and Theo Mary. A new preconditioner that exploits low-rank approximations to factorization error. *SIAM J. Sci. Comput.*, 41(1):A59–A82, 2019.

[24] Nicholas J. Higham, Srikara Pranesh, and Mawussi Zounon. Squeezing a matrix into half precision, with an application to solving linear systems. MIMS EPrint 2018.37, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, November 2018. 15 pp. Revised March 2019.

[25] S. i. O'uchi, H. Fuketa, T. Ikegami, W. Nogami, T. Matsukawa, T. Kudoh, and R. Takano. Image-classifier deep convolutional neural network training by 9-bit dedicated hardware to realize validation accuracy and energy efficiency superior to the half precision floating point format. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pages 1–5.

[26] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Institute of Electrical and Electronics Engineers, New York, 1985.

[27] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985)*. IEEE Computer Society, New York, 2008. 58 pp. ISBN 978-0-7381-5752-8.

[28] Intel Corporation. BFLOAT16—hardware numerics definition, November 2018. White paper. Document number 338302-001US.

[29] Eugene Isaacson and Herbert Bishop Keller. *Analysis of Numerical Methods*. Wiley, New York, 1966. xv+541 pp. Reprinted by Dover, New York, 1994. ISBN 0-486-68029-0.

[30] Julia Computing. Bfloat16s. https://github.com/JuliaComputing/BFloat16s.jl.

[31] David Malone. To what does the harmonic series converge? *Irish Math. Soc. Bulletin*, 71: 59–66, 2013.

[32] Cleve B. Moler. Cleve Laboratory. http://mathworks.com/matlabcentral/fileexchange/59085-cleve-laboratory.

[33] Cleve B. Moler. MATLAB users' guide. Technical Report CS81-1 (revised), Department of Computer Science, University of New Mexico, Albuquerque, New Mexico, August 1982. 60 pp.

[34] Cleve B. Moler. "Half precision" 16-bit floating point arithmetic. http://blogs.mathworks.com/cleve/2017/05/08/half-precision-16-bit-floating-point-arithmetic/, May 2017.

[35] Cleve B. Moler. The historic MATLAB users' guide. https://blogs.mathworks.com/cleve/2018/02/05/the-historic-matlab-users-guide/, February 2018.

[36] Cleve B. Moler. Variable format half precision floating point arithmetic. https://blogs.mathworks.com/cleve/2019/01/16/variable-format-half-precision-floating-point-arithmetic/, January 2019.

[37] Jean-Michel Muller. *Elementary Functions: Algorithms and Implementation*. Third edition, Birkhäuser, Boston, MA, USA, 2016. xxv+283 pp. ISBN 978-1-4899-7981-0.

[38] Krishna Palem and Avinash Lingamneni. Ten years of building broken chips: The physics and engineering of inexact computing. *ACM Trans. Embed. Comput. Syst.*, 12(2s):87:1–87:23, 2013.

[39] T. N. Palmer. More reliable forecasts with less precise computations: A fast-track route to cloud-resolved weather and climate simulators? *Phil. Trans. R. Soc. A*, 372(2018), 2014.

[40] Naveen Rao. Beyond the CPU or GPU: Why enterprise-scale artificial intelligence requires a more holistic approach. https://newsroom.intel.com/editorials/artificial-intelligence-requires-holistic-approach, May 2018. Accessed November 5, 2018.

[41] Christian Rau. Half 1.12. IEEE 754-based half-precision floating point library. http://half.sourceforge.net/index.html, March 2017.

[42] Pierre Roux. Innocuous double rounding of basic arithmetic operations. *Journal of Formalized Reasoning*, 7(1):131–142, 2014.

[43] Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

[44] Viral Shah. Comment on Nicholas J. Higham, "Half Precision Arithmetic: fp16 Versus bfloat16". https://nickhigham.wordpress.com/2018/12/03/half-precision-arithmetic-fp16-versus-bfloat16/#comment-5466, December 2018.

[45] Alexey Svyatkovskiy, Julian Kates-Harbeck, and William Tang. Training distributed deep recurrent neural networks with mixed precision on GPU clusters. In *MLHPC'17: Proceedings of the Machine Learning on HPC Environments*, ACM Press, New York, 2017, pages 10:1–10:8.

[46] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benin. A trans-

precision floating-point platform for ultra-low power computing. In *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2018, pages 1051–1056.

[47] Tobias Thornes, Peter Düben, and Tim Palmer. On the use of scale-dependent precision in earth system modelling. *Quart. J. Roy. Meteorol. Soc.*, 143(703):897–908, 2017.

[48] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[49] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, Atlanta, GA, April 19-23 2010, pages 1–8. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[50] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, Curran Associates, Inc., 2018, pages 7686–7695.