CPFloat: A C library for simulating low-precision arithmetic

Fasi, Massimiliano and Mikaitis, Mantas

2020

MIMS EPrint: **2020.22**

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

CPFloat: A C library for simulating low-precision arithmetic*

Massimiliano Fasi† Mantas Mikaitis‡

Abstract

One can simulate low-precision floating-point arithmetic via software by executing each arithmetic operation in hardware and then rounding the result to the desired number of significant bits. For IEEE-compliant formats, rounding requires only standard mathematical library functions, but handling subnormals, underflow, and overflow demands special attention, and numerical errors can cause mathematically correct formulae to behave incorrectly in finite arithmetic. Moreover, the ensuing implementations are not necessarily efficient, as the library functions these techniques build upon are typically designed to handle a broad range of cases and may not be optimized for the specific needs of rounding algorithms. CPFloat is a C library for simulating low-precision arithmetics. It offers efficient routines for rounding, performing mathematical computations, and querying properties of the simulated low-precision format. The software exploits the bit-level floating-point representation of the format in which the numbers are stored, and replaces costly library calls with low-level bit manipulations and integer arithmetic. In numerical experiments, the new techniques bring a considerable speedup (typically one order of magnitude or more) over existing alternatives in C, C++, and MATLAB. To our knowledge, CPFloat is currently the most efficient and complete library for experimenting with custom low-precision floating-point arithmetic available in any language.

Key words. low-precision arithmetic, floating-point arithmetic, mixed precision, IEEE 754 standard, binary16, bfloat16, round-to-nearest, directed rounding, round-to-odd, stochastic rounding.

1 A plethora of floating-point formats and rounding modes

The 2019 revision of the IEEE 754 standard for floating-point arithmetic [32] specifies three basic binary formats for computation: binary32, binary64, and binary128. Most 64-bit CPUs equipped with a floating-point unit natively support binary32 and binary64 arithmetic, and 32-bit CPUs can simulate binary64 arithmetic very efficiently by relying on highly optimized software libraries. The binary128 format, introduced in the 2008 revision [31] of the original IEEE 754 standard [30], has not gained much popularity among hardware manufacturers, and over ten years after having been standardized

*Version of October 22, 2022.

Funding: The work of the first author was supported by the Royal Society, by Istituto Nazionale di Alta Matematica INdAM–GNCS Project 2020, and by Wenner-Gren Foundations grant UPD2019-0067. The work of the second author was supported by an EPSRC Doctoral Prize Fellowship and by EPSRC grant EP/P020720/1. This work was initiated while the two authors were at the Department of Mathematics, The University of Manchester, Oxford Road, Manchester, M13 9PL, United Kingdom, and was partly carried out when the first author was at the School of Science and Technology, Örebro University, Örebro, SE-701 82, Sweden.

†Department of Computer Science, Durham University, Upper Mountjoy Campus, Stockton Road, Durham DH1 3LE, United Kingdom, massimiliano.fasi@durham.ac.uk.

‡School of Computing, University of Leeds, Woodhouse Lane, Leeds LS2 9JT, United Kingdom, m.mikaitis@leeds.ac.uk.

it is supported only by two IBM processor families: the POWER9, which implements version 3.0 of the Power Instruction Set Architecture [36], and the z/Architecture [39].

In fact, the low-precision requirements of artificial intelligence applications have steered the hardware market in the opposite direction, and a number of fewer-than-32-bit formats have been proposed in recent years. The first widely available 16-bit floating-point format was binary16. Despite having been defined in the last two revisions of the IEEE 754 standard only as an interchange format, binary16 has been supported as an arithmetic format by all NVIDIA microarchitectures since Pascal [45] and all AMD architectures since Vega [51]. Google has recently introduced the bfloat16 data type [34], a 16-bit format with approximately the same dynamic range as binary32.

The latest Armv8 CPUs support a wide variety of floating-point formats, including binary32, binary64, bfloat16 [1, Sec. A1.4.5], binary16, and an alternative custom half-precision format [1, Sec. A1.4.2]. The latter is a 16-bit format based on binary16 that extends the dynamic range from $[-65{,}504, +65{,}504]$ to $[-131{,}008, +131{,}008]$ by reclaiming the 2,048 bit patterns (about 3%) used for infinities and NaNs (Not a Number).

The latest NVIDIA graphics card microarchitectures offer new low-precision formats, aimed at machine learning, that break with the 16-bit tradition. The A100 GPUs, which are based on the Amprere microarchitecture [46], support a 19-bit floating-point format, called TensorFloat-32, which combines the precision of binary16 with the exponent range of binary32. The latest H100 GPUs, based on the Hopper microarchitecture [47], provide two 8-bit formats: E4M3, with 4 bits of precision and dynamic range $[-240, 240]$, and E5M2, which trades one bit of precision for the much larger dynamic range $[-57{,}344, 57{,}344]$.

In Section 4, we discuss the general framework that underlies all these floating-point formats. This framework is a straightforward extension of the IEEE 754 standard [30, 31, 32] and is well established in the literature [27, 44].

Such a broad range of floating-point formats poses a major challenge to those trying to develop mixed-precision algorithms for scientific computing, because studying the numerical behavior of an algorithm in different working precisions may require access to a number of high-end hardware platforms. To alleviated this issue, two families of techniques for simulating low-precision floating-point arithmetic via software have been proposed over the years.

On the one hand, one can use (signed) integers to represent the exponent and significand of the low-precision numbers. All mathematical functions can then be evaluated by explicitly operating on the integers that make up these representations. This solution allows for maximum flexibility, and it is typically used to simulate arbitrary-precision arithmetics, where exponent and significand can have arbitrarily many digits.

If only low-precision arithmetic is of interest, however, relying on the floating-point arithmetics available in hardware will typically lead to more efficient software simulators. In fact, one can perform each arithmetic operation using a floating-point format available in hardware, check that the result is representable in the low-precision format of interest, and then round the significand to the desired number of significant bits. In this context, simulating low-precision floating-point arithmetic only requires the ability to round a high-precision number to lower precision, and users can leverage existing hardware and mathematical libraries. This method is more convenient: it leads to software that is easier to maintain, since less code is necessary, and is typically more efficient, as most of the computation is performed using thoroughly optimized high-precision mathematical routines that are already available. Here, we follow this second approach, which is the one most often encountered in the literature, as the survey of existing software for simulating low-precision floating-point arithmetic in Section 2 illustrates.

Our contribution is two-fold. First, we discuss how the operations underlying the rounding of a floating-point number x to lower precision can be performed directly on the binary floating-point representation of x. We present the new algorithms in Section 5, and in Section 6 we explain how to implement them efficiently using bitwise operations and integer arithmetic. Second, we introduce

`CPFloat`, a header-only C library that implements our algorithms and can be used to simulate low-precision binary floating-point arithmetic. The name of the library is a shorthand for *Custom-Precision Floats*. Section 3 lists the functions available in the library and provides a minimal, self-contained code snippet showing how `CPFloat` can be used as a full arithmetic library with custom precision floating-point formats. Section 7 discusses some implementation details and describes how the library was tested for correctness.

The floating-point representation in the simulated (target) format exists only implicitly: in practice, all numbers are represented in a (storage) format that is natively supported by the compiler. The storage formats currently available in CPFloat are binary32 and binary64.

`CPFloat` is not the first library for simulating low-precision arithmetic in C: the GNU MPFR library [19], for example, allows the programmer to work with arbitrarily few, as well as arbitrarily many, bits of precision. Unlike MPFR, `CPFloat` is intended only as a simulator for low-precision floating-point arithmetics. Internally, floating-point values are represented using binary32 or binary64 numbers, thus only formats with at most 53 bits of precision and an exponent range no broader than that of binary64 can be simulated. This narrower aim provides scope for the wide range of optimizations we discuss, which in turn yield more efficient implementations.

We provide a MEX interface that allows users to call `CPFloat` directly from their MATLAB and Octave codes. We use this to compare the performance of the new library with that of the MATLAB function `chop` [28] and of the MATLAB toolboxes FLOATP_Toolbox [41] and INTLAB [54]. Our experimental results, discussed in Section 8, show that the new codes bring a considerable speedup over these existing alternatives, as long as the matrices being rounded are large enough to offset the overhead of calling C code from MATLAB.

The library can be used to prototype and test mixed-precision algorithms, as well as to simulate custom-precision hardware. Recently, for example, we have used `CPFloat` to develop an algorithm for computing matrix–matrix products on NVIDIA GPUs equipped with tensor cores [13]. These mixed-precision matrix multiply–accumulate units internally use different precisions and rounding modes [14]. Using a software simulation, we could evaluate the numerical behavior of custom variations of the tensor cores, which we obtained by changing the rounding mode used in some key operations.

2 Related work

A number of packages to simulate low-precision floating-point arithmetic via software are available. These are summarised in Table 1, where we report the main programming language in which the software is written and detail what storage formats, target formats, and rounding modes are supported.

The most comprehensive software package for simulating arbitrary-precision floating-point arithmetic is the GNU MPFR library [19], which extends the formats in the IEEE 754 standard to any precision and to an arbitrarily large exponent range. The library is written in C, but interfaces for most programming languages are available. GNU MPFR is a de facto standard for working with arbitrary precision, and is typically used to perform computations that require high accuracy, rather than to simulate low-precision arithmetics.

Sipe is a header-only C mini-library designed to simulate low precision efficiently. This software supports round-to-nearest with ties-to-even and round-to-zero, and numbers can be stored either as a pair of signed integers representing the significand and the exponent of the floating-point value [37], or as a value in a native floating-point format. The latest version of the library [38] supports `float`, `double`, `long double`, or `__float128` (from the GCC `libquadmath` library).

Dawson and Düben [11] recently developed a Fortran library, called rpe, for simulating reduced-precision floating-point arithmetics in large numerical simulations. In rpe, the reduced-precision floating-point values are stored as binary64 numbers, a solution that provides a very efficient technique for simulating floating-point formats with the same exponent range as binary64 and no more

Table 1: Synoptic table of available software packages for simulating low-precision floating-point arithmetic. The first three columns report the name of the package, the main programming language in which the software is written, and what storage formats are supported. The following three columns describe the parameters of the target formats: whether the number of bits of precision in the significand is arbitrary (A) or limited to the number of bits in the significand of the storage format (R); whether the exponent range can be arbitrary (A), must be a—possibly restricted—sub-range of the exponent range of the storage format (S), or can be a sub-range only for built-in types (B); whether the target format supports subnormal numbers (Y), does not support them (N), supports them only for built-in types (B), supports them by default but allows the user to switch the functionality off (F), or does not support them by default but allow the user to turn the functionality on (O). The following column lists the floating-point formats that are built into the system, if any. The last seven columns indicate what rounding modes the software supports: round-to-nearest with ties-to-even (RNE), ties-to-zero (RNZ), or ties-to-away (RNA), round-toward-zero (RZ), round-to-$+\infty$ and round-to-$-\infty$ (RUD), round-to-odd (RO), and the two variants of stochastic rounding discussed in Section 4 (SR). Fully supported rounding modes are denoted by ✓, while **!** is used for rounding modes that can be simulated at a higher computational cost. The abbreviations bf16, tf32, fp16, fp32, and fp64 denote the formats bfloat16, TensorFloat-32, binary16, binary32, and binary64, respectively.

Package name	Primary language	Storage format	Target format				RNE	RNZ	RNA	RZ	RUD	RO	SR
			p	e	s	built-in							
GNU MPFR	C	custom	A	A	O		✓		!	✓	✓		
Sipe	C	multiple	R	S	Y		✓			✓			
rpe	Fortran	fp64	R	B	B	fp16	✓						
FloatX	C++	fp32/fp64	R	S	Y		✓						
INTLAB	MATLAB	fp64	R	S	Y		✓			✓	✓		
chop	MATLAB	fp32/fp64	R	S	F	fp16/bf16	✓			✓	✓		✓
FLOATP	MATLAB	fp64	R	A	N		✓			✓	✓		✓
QPyTorch	Python	fp32	R	S	N		✓	✓	✓				✓
CPFloat	C	fp32/fp64	R	S	F	fp16/bf16/tf32	✓	✓	✓	✓	✓	✓	✓

than 53 bits of precision.

FloatX [16] is a header-only C++ library for simulating low-precision arithmetic that supports binary32 and binary64 as storage formats. This software is more flexible than rpe, in that it allows the user to choose not only the number of significant digits, but also the number of bits used to represent the exponent in the target format.

The only rounding mode currently implemented by both rpe and FloatX is round-to-nearest with ties-to-even, which may be too restrictive when one wants to simulate hardware units where truncation or stochastic rounding are available.

The `flround` function in the INTLAB toolbox for MATLAB and Octave [54] allows the user to round binary64 values to lower-precision formats. The routine implements the binary version of [55, Alg. 3] and rounds a number x by first computing $x' = x + C$, where C is a suitably chosen constant, and then returning $y = x' - C$. Addition and subtraction are performed in floating-point arithmetic using the format in which x is stored and are thus performed efficiently when the operations can be performed in hardware. This strategy requires that the exponent range of x be limited, in order to avoid overflow in the computation of C: in binary64, for example, the maximum available exponent is 1023, but the exponent of x cannot exceed 970 when the `flround` function is used. The toolbox supports round-to-nearest with ties-to-even and the three directed rounding modes prescribed by the IEEE 754 standard. Furthermore, the full arithmetic library `fl` is available on `fl-type` objects which are rounded binary64 values to some lower custom precision representation.

Higham and Pranesh [28] proposed `chop`, a MATLAB function for rounding arrays of binary32 or binary64 numbers to lower precision. This solution is more efficient and flexible than the `fp16` and `vfp16` MATLAB data types proposed by Moler [42], as the user can specify not only the boundaries of the dynamic exponent range and the number of bits in the significand, but also the rounding mode to be used and whether subnormals are supported. `chop` supports six rounding modes: the four default rounding modes prescribed by the IEEE 754-2019 for single operations and two variants of stochastic rounding. This function can be used only from within the MATLAB programming environment, and the underlying algorithms, which rely on mathematical operations involving the exponent and significand of the represented floating-point numbers, are not necessarily suitable for efficient implementations in a low-level language such as C. For example, `chop` uses the built-in MATLAB functions `abs`, `sign`, `ceil`, `floor`, `log2`, and `pow2`, which may not be optimized for the narrow range of inputs required in order disassemble and reassemble floating-point numbers.

FLOATP_Toolbox [41] is a MATLAB toolbox for simulating reduced-precision fixed-point and floating-point arithmetics. This library uses binary64 as storage format, supports the same six rounding modes as `chop` [28], and implements a number of mathematical functions, such as `log`, `exp`, or `sin`, for example. The functionalities of the FLOATP_Toolbox can be used in two ways: either by calling the routines that work directly on the binary64 data structure, or by relying on the methods of the `floatp` class, which override a number of built-in MATLAB functions. A similar library for simulating posit arithmetic has been developed by the same author [26, 12].

The QPyTorch library is a low-precision arithmetic simulator, written in PyTorch, whose primary goal is to facilitate the training of neural networks in low precision without the overhead of building low-precision hardware [60]. The core design principles of this library are analogous to those of `chop`, in that numbers are stored in binary32 before as well as after rounding. QPyTorch supports custom floating-point formats that can fit into the binary32 format, arbitrary-precision fixed-point formats no wider than 24 bits, and block floating-point formats [59]. Infinities, NaNs, and subnormals are not supported for efficiency's sake and because, as the authors point out, these special values do not appear when training neural networks and thus may not be supported by the underlying low-precision hardware. The supported rounding modes are stochastic rounding and round-to-nearest with ties-to-even.

`VPREC-libm`[4] is a related tool developed to evaluate the accuracy/performance trade-offs of numerical code. The library instruments the code so that calls to mathematical functions are intercepted at runtime and performed in simulated low precision. Each function is evaluated in binary128 arithmetic by using the GCC `libquadmath` library and the result is then rounded to the target precision.

The packages SERP [56] and FASE [49] utilize dynamic binary translation to seamlessly change the precision of an executable without requiring any alteration of the source code. A downside of this approach is that the user cannot control which rounding mode a given operation will use, and the target format appears to be a global setting which cannot be changed on a per-operation basis—this significantly limits the simulation environment, and notably does not allow for the simulation of mixed-precision operations, which are becoming more and more common in hardware devices.

The algorithms presented here complement existing software by proposing efficient techniques for implementing rounding using low-level bitwise instructions. Our library is intended as a software package that enables the use of these rounding functionalities in low-level languages such as C and C++, but high-level languages that allow the user to call C routines, either directly or indirectly, can also benefit from it.

3 Features and examples

At the core of `CPFloat` are the efficient rounding routines described in Section 5. These convert a number from one floating-point format (the storage format, which will be either binary32 or binary64) to a second, custom format (the target format). The representation in the target format is implicit, as

Listing 1: CPFloat example

```c
#include <stdio.h>
#include "cpfloat_binary64.h"

#define N 3

int main () {
  // Allocate the data structure for target formats and rounding parameters.
  optstruct *fpopts = init_optstruct();

  // Set up the parameters for binary16 target format.
  fpopts->precision = 11;                 // Bits in the significand + 1.
  fpopts->emax = 15;                      // The maximum exponent value.
  fpopts->subnormal = CPFLOAT_SUBN_USE;   // Support for subnormals is on.
  fpopts->round = CPFLOAT_RND_TP;         // Round toward +infinity.
  fpopts->flip = CPFLOAT_NO_SOFTERR;      // Bit flips are off.
  fpopts->p = 0;                          // Bit flip probability (not used).
  fpopts->explim = CPFLOAT_EXPRANGE_TARG; // Limited exponent in target format.

  // Validate the parameters in fpopts.
  int retval = cpfloat_validate_optstruct(fpopts);
  printf("The validation function returned %d.\n", retval);

  // Initialize C array with arbitrary elements.
  double X[N] = { (double)5/3, M_PI, M_E };
  double Y[N] = { 1.5, 1.5, 1.5 };
  double Z[N];
  printf("X in binary64:\n  %.15e %.15e %.15e\n", X[0], X[1], X[2]);

  // Round the values of X to the binary16 format and store in Z.
  cpfloat(Z, X, N, fpopts);
  printf("X rounded to binary16:\n  %.15e %.15e %.15e\n", Z[0], Z[1], Z[2]);

  // Round the sum of X and Y.
  cpf_add(Z, X, Y, N, fpopts);
  printf("Sum rounded to binary16:\n  %.15e %.15e %.15e\n", Z[0], Z[1], Z[2]);

  // Round the product of X and Y.
  cpf_mul(Z, X, Y, N, fpopts);
  printf("Product rounded to binary16:\n  %.15e %.15e %.15e\n", Z[0], Z[1], Z[2]);

  // Round the logarithm of X.
  cpf_log(Z, X, N, fpopts);
  printf("Log rounded to binary16:\n  %.15e %.15e %.15e\n", Z[0], Z[1], Z[2]);

  // Round the 2-argument arctangent of X and Y.
  cpf_atan2(Z, X, Y, N, fpopts);
  printf("Angle rounded to binary16:\n  %.15e %.15e %.15e\n", Z[0], Z[1], Z[2]);

  free_optstruct(fpopts);
}
```

the rounded numbers are still stored as either binary32 or binary64 values. Relying on these rounding routines, CPFloat provides functions to simulate custom low-precision arithmetic when operating on arrays of scalars. We now describe its user interface, using Listing 1 as reference. In the example, we use double as storage format, thus we include the header file cpfloat_binary64.h; the header file cpfloat_binary32.h should be included when using float arrays.

The target numerical format is specified by a C structure of type optstruct. The structure should be initialized by calling

```
optstruct *init_optstruct()
```

as done on line 8. This function allocates the memory underlying the optstruct correctly, but the target format that the structure represents is still undefined. The parameters of the target format should be initialized explicitly as shown on lines 11–17.

In Listing 1, we do not showcase all the functionalities of CPFloat, and in particular we do not inject soft errors in the low-precision results. The library supports two injection modes, which can be selected by changing the value of the flip field of the optstruct structure. If flip is set to CPFLOAT_FRAC_SOFTERR, then with probability p a single bit flip can strike the fraction of the low-precision rounded number. This is the injection mode used in chop [28].

If flip is set to CPFLOAT_FP_SOFTERR, on the other hand, the single bit flip can strike any bit of the low-precision floating-point representation. Soft errors are typically modelled in this way in high-performance computing studies [2] as well as in numerical simulations [52, Sec. V]. For reproducibility, errors are sometimes introduced by hands at specific points of the execution rather than at random [57]; in CPFloat, this can be achieved by setting p to 1.0 and switching the value of the flip field between CPFLOAT_FP_SOFTERR (soft errors on) and CPFLOAT_NO_SOFTERR (soft errors off).

The parameters in an optstruct structure can be validated with

```
int cpfloat_validate_optstruct(optstruct *fpopts)
```

if the storage format is binary64, or

```
int cpfloat_validate_optstructf(optstruct *fpopts)
```

if the storage format is binary32. The convention of appending an f to the function name when the storage format is binary32 is common in C, and is used throughout the library whenever a binary64 and a binary32 variant are offered. The two functions return a negative integer if the fields of the input do not represent a valid floating-point format, zero if they do, and a positive integer if they represent a valid format which may yield an unexpected behavior such as potentially harmful double rounding (see Section 5). Some of the fields of an optstruct are pointers, and the function

```
void free_optstruct(optstruct *fpopts)
```

should be used as shown on line 49 to free the memory correctly.

The functions that make up the library can be divided into three groups: functions for rounding, functions for computing in low-precision arithmetic, and functions for probing the low-precision floating-point format. We discuss these assuming arrays of binary64 values—for binary32 arrays it suffices to change all double to float and add an f at the end of the function name.

A double array can be rounded to low precision using the functions

```
int cpfloat(double *X, const double *A, const size_t n, optstruct *fpopts)
int cpf_fpround(double *X, const double *A, const size_t n, optstruct *fpopts)
```

which convert the n elements in the array A to the format specified by the parameters in fpopts. We note that cpfloat is just an alias for cpf_fpround which ensures backward compatibility with previous versions of the library—the two functions rely on the same implementation. We also remark that X and A may point to the same memory location, in which case the rounding is performed in place.

The second group contains functions for simulating the four elementary arithmetic operations and many mathematical functions in low precision. The computation is first performed in the storage format, using the corresponding C operators or C mathematical library (`math.h`) functions, and the result is then rounded to the specified low-precision format.

The third and last group comprises functions that work on the implicit target-format representation. These can be used to extract the exponent and fraction of the number rounded to low precision, to compute the number that in the target format is closest to a given value in the storage format, and to check whether a value in the storage format would become subnormal, normal, infinite, or NaN once rounded to the target format.

We list all functions available in `CPFloat` in Table 2, and refer the reader to the package documentation for additional details.

Table 2: CPFloat functions if `double` is used as storage format. The names of the corresponding functions for `float` are obtained by appending the suffix "`f`": for example, the function `cpfloat_mulf` can be used to multiply two `float` arrays elementwise. All functions require two extra arguments to define the number of elements in the input vectors (`numelem`) and the floating-point representation (`fpopts`). All mathematical functions return as output numbers in the target format.

Function	Description
`cpfloat_validate_optstruct`	Validate fields of `optstruct struct` variable.
`cpfloat`	Round `double` array to target format (legacy function name).
`cpf_fpround`	Round `double` array to target format.
`cpf_add`	Sum in target format.
`cpf_sub`	Difference in target format.
`cpf_mul`	Product in target format.
`cpf_div`	Ratio in target format.
`cpf_cos`	Trigonometric cosine.
`cpf_sin`	Trigonometric sine.
`cpf_tan`	Trigonometric tangent.
`cpf_acos`	Inverse trigonometric cosine.
`cpf_asin`	Inverse trigonometric sine.
`cpf_atan`	Inverse trigonometric tangent.
`cpf_atan2`	Two-argument arctangent.
`cpf_cosh`	Hyperbolic cosine.
`cpf_sinh`	Hyperbolic sine.
`cpf_tanh`	Hyperbolic tangent.
`cpf_acosh`	Inverse hyperbolic cosine.
`cpf_asinh`	Inverse hyperbolic sine.
`cpf_atanh`	Inverse hyperbolic tangent.
`cpf_exp`	Exponential.
`cpf_frexp`	Exponent and normalized fraction in target format.
`cpf_ldexp`	Scale number by power of 2.
`cpf_log`	Natural logarithm.
`cpf_log10`	Base-10 logarithm.
`cpf_modf`	Integral and fractional part.
`cpf_exp2`	Base-2 exponential.
`cpf_expm1`	$\exp(x) - 1$.
`cpf_ilogb`	Integral part of logarithm of absolute value.
`cpf_log1p`	Natural logarithm of number shifted by one.
`cpf_log2`	Base-2 logarithm.

Table 2: (continued)

Function	Description
`cpf_scalbn`	Scale number by power of `FLT_RADIX`.
`cpf_scalbln`	Scale number by power of `FLT_RADIX`.
`cpf_pow`	Real powers.
`cpf_sqrt`	Square root.
`cpf_cbrt`	Cube root.
`cpf_hypot`	Hypotenuse of a right-angle triangle.
`cpf_erf`	Error function.
`cpf_erfc`	Complementary error function.
`cpf_tgamma`	Gamma function.
`cpf_lgamma`	Natural logarithm of absolute value of gamma function.
`cpf_ceil`	Ceiling function.
`cpf_floor`	Floor function.
`cpf_trunc`	Integer truncation.
`cpf_round`	Closest integer (round-to-nearest).
`cpf_lround`	Closest integer (round-to-nearest).
`cpf_llround`	Closest integer (round-to-nearest).
`cpf_rint`	Closest integer with specified rounding mode.
`cpf_lrint`	Closest integer with specified rounding mode.
`cpf_llrint`	Closest integer with specified rounding mode.
`cpf_nearbyint`	Closest integer with specified rounding mode.
`cpf_remainder`	Remainder of floating point division.
`cpf_remquo`	Remainder and quotient of rounded numbers.
`cpf_copysign`	Compose number from magnitude and sign.
`cpf_nextafter`	Next number in specified direction in target format.
`cpf_nexttoward`	Next number in specified direction in target format.
`cpf_fdim`	Positive difference.
`cpf_fmax`	Element-wise maximum.
`cpf_fmin`	Element-wise minimum.
`cpf_fpclassify`	Categorize floating-point values.
`cpf_isfinite`	Check whether value is finite in target format.
`cpf_isinf`	Check whether value is infinite in target format.
`cpf_isnan`	Check if value is not a number in target format.
`cpf_isnormal`	Check whether value is normal in target format.
`cpf_fabs`	Absolute value.
`cpf_fma`	Fused multiply-add.

The example in Listing 1 is available in the `CPFloat` source code repository on GitHub.[1] As we explain in more detail in Section 7, `CPFloat` can leverage OpenMP to run multiple threads in parallel, but the runtime overhead causes the parallel version of the code to be slower than the sequential one on arrays with very few elements. To alleviate this issue, we use a sequential version of the code for small arrays and switch to the parallel variant only when the number of elements exceeds a machine-dependent threshold which we determine with an auto-tuning procedure. For the C library, the auto-tuning can be triggered with the command

```
$ make autotune
```

[1] https://github.com/north-numerical-computing/cpfloat

The example in Listing 1 can be compiled with

```
$ make example
```

which produces the binary `bin/example`. When executed, this program produces the following output.

```
The validation function returned 0.
X in binary64:
  1.666666666666667e+00 3.141592653589793e+00 2.718281828459045e+00
X rounded to binary16:
  1.666992187500000e+00 3.142578125000000e+00 2.718750000000000e+00
Sum rounded to binary16:
  3.167968750000000e+00 4.644531250000000e+00 4.218750000000000e+00
Product rounded to binary16:
  2.500000000000000e+00 4.714843750000000e+00 4.078125000000000e+00
Log rounded to binary16:
  5.112304687500000e-01 1.145507812500000e+00 1.000000000000000e+00
Angle rounded to binary16:
  7.329101562500000e-01 4.455566406250000e-01 5.043945312500000e-01
```

4 Floating-point storage formats and rounding

A family of binary floating-point numbers $\mathcal{F}\langle p, e_{\min}, e_{\max}, \mathfrak{s}_n\rangle$ is a finite set that includes a subset of the real line and a handful of values with a special meaning. In our notation, the three integer parameters p, $e_{\min}$, and $e_{\max}$ represent the number of binary digits of precision, the minimum exponent, and the maximum exponent, respectively, and the Boolean flag $\mathfrak{s}_n$ indicates whether subnormals are supported. A real number $x := (s, m, e)$ in $\mathcal{F}\langle p, e_{\min}, e_{\max}, \mathfrak{s}_n\rangle$ can be written as

$$x = (-1)^s \cdot m \cdot 2^{e-p+1}, \tag{4.1}$$

where s is the sign bit, set to 0 if x is positive and to 1 otherwise, the integral siginificand m is a natural number not greater than $2^p - 1$, and the exponent e is an integer between $e_{\min}$ and $e_{\max}$ inclusive.

In order to ensure a unique representation for all numbers in $\mathcal{F}\langle p, e_{\min}, e_{\max}, \mathfrak{s}_n\rangle \setminus \{0\}$, it is customary to normalize the system by assuming that if $x \geq 2^{e_{\min}}$ then $2^{p-1} \leq m \leq 2^p - 1$, that is, the number is represented using the smallest possible exponent and the largest possible significand. In such systems, the number $(s, m, e) \in \mathcal{F}\langle p, e_{\min}, e_{\max}, \mathfrak{s}_n\rangle \setminus \{0\}$ is normal if $m \geq 2^{p-1}$, and subnormal otherwise. The exponent of subnormals is always $e_{\min}$, and in a normalized system any number $x = (s, m, e) \neq 0$ has a unique p-digit binary expansion $(-1)^s \cdot \widetilde{m} \cdot 2^e$, where

$$\widetilde{m} = m \cdot 2^{1-p} = d_0 + \frac{d_1}{2} + \cdots + \frac{d_{p-1}}{2^{p-1}} = d_0.d_1 \ldots d_{p-1}, \tag{4.2}$$

for some $d_0, d_1, \ldots, d_{p-1} \in \{0, 1\}$, is called the normal significand of x. One can verify that if x is normal then $d_0 = 1$ and $1 \leq \widetilde{m} < 2$, whereas if x is subnormal then $d_0 = 0$ and $0 < \widetilde{m} < 1$. Conventionally, the signed zero values +0 and −0 are considered neither normal nor subnormal. In a normalized system, the smallest subnormal is $x_{\text{minsub}} := 2^{e_{\min}-p+1}$, whereas the smallest and largest positive normal numbers are $x_{\min} := 2^{e_{\min}}$ and $x_{\max} := 2^{e_{\max}}(2 - 2^{1-p})$, respectively. Their negative counterparts can be obtained by observing that a floating-point number system is symmetric with respect to 0. In our notation, subnormals are kept if $\mathfrak{s}_n =$ **true**, and rounded to either the closest smallest floating-point number or the zero of appropriate sign if $\mathfrak{s}_n =$ **false**.

The floating-point numbers in $\mathcal{F}\langle p, e_{\min}, e_{\max}, \mathfrak{s}_n\rangle$ can be represented efficiently as binary strings. The sign is stored in the leftmost bit of the representation, and the following b_e bits are used to store the exponent. Under the IEEE 754 assumption that $e_{\min} = 1 - e_{\max}$, the most efficient way of representing the exponent is obtained by setting $e_{\max} = 2^{b_e-1} - 1$ and using a representation biased by

$e_{\max}$, so that $00\cdots01_2$ represents the smallest allowed exponent and $11\cdots10_2$ represents the largest. The trailing $p-1$ bits are used to store the significand of x. It is not necessary to store the value of d_0 explicitly, as the IEEE standard uses an encoding often referred to as "implicit bit": d_0 is assumed to be 1 unless the binary encoding of the exponent is the all-zero string, in which case $d_0 = 0$ and x represents either a signed 0, if $m = 0$, or a subnormal number, otherwise. If the exponent field contains the reserved all-one string, then the number represents $+\infty$ or $-\infty$ if the fraction field is set to 0, and a NaN otherwise. Infinities are needed to express values whose magnitude exceeds that of the largest positive and negative numbers that can be represented in $\mathcal{F}\langle p, e_{\min}, e_{\max}, \mathfrak{s}_n\rangle$, whereas NaNs represent the result of invalid operations, such as taking the square root of a negative number, dividing a zero by a zero, or multiplying an infinity by a zero. These are needed in order to ensure that the semantics of all floating-point operations is well specified and that the resulting floating-point number system is closed.

A rounding is an operator that maps a real number x to one of the floating-point numbers closest to x in a given floating-point family. The original IEEE 754 standard [30, Sec. 4] defines four rounding modes for binary formats: the default round-to-nearest with ties-to-even, and three directed rounding modes, round-toward-$+\infty$, round-toward-$-\infty$, and round-toward-zero. The 2008 revision of the standard [31] introduces a fifth rounding mode, round-to-nearest with ties-to-away, but states that it is not necessary for an implementation of a binary format to be IEEE compliant. Finally, the latest revision of the standard [32] recommends the use of round-to-nearest with ties-to-zero for augmented operations [32, Sec. 9.5]. Here we also consider two less common rounding strategies, round-to-odd [3] and stochastic rounding [9].

With round-to-odd, we have to set the least significant bit of the rounded number to 1 unless the infinitely-precise number is exactly representable in the target format. This rounding has applications in several domains. In computer arithmetic, for instance, it can be used to emulate a correctly rounded fused multiply–add (FMA) operator when a hardware FMA unit is not available [3]. In general, round-to-odd helps avoid issues associated with double rounding, and is used for this purpose in GCC,[2] for example. The fact that it can be implemented at low cost in hardware has recently prompted interest from the machine learning community [7].

Unlike all rounding modes discussed so far, stochastic rounding is non-deterministic, in the sense that the direction in which to round is chosen randomly and repeating the rounding may yield different results. The simplest choice is to round an infinitely-precise number that is not representable in the target format to either of the two closest representable floating-point numbers with equal probability. A more interesting flavor is obtained by rounding a non-representable number x to either of the closest floating-point numbers with probability proportional to the distance between x and the two rounding candidates. This rounding mode dates back to the fifties [17, 18], and has recently gained prominence owing to the surge of interest in low-precision floating-point formats. It is particularly effective at alleviating swamping in long sums [8] and ordinary differential equation solvers [29, 15], and at counteracting the loss of accuracy observed when the precision used to train neural networks is reduced [25, 58]. Stochastic rounding is not widely available in general-purpose hardware, but has started to appear in some specialized processors, such as the Intel Loihi neuromorphic chips [10] and the Graphcore IPU, an accelerator for machine learning [22]. We refer the reader to the recent survey [9] for more details on this rounding mode.

5 Efficient rounding to a lower-precision format

We simulate low precision aritmetic by first performing each operation in some higher precision available in hardware and then rounding the result to the desired target format. In this section, we discuss how to exploit the binary representation in the storage format to develop efficient algorithms for

[2] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=21718#c25

rounding $x \in \mathcal{F}^{(h)}$ to $\widetilde{x} \in \mathcal{F}^{(\ell)}$, where

$$\mathcal{F}^{(h)} := \mathcal{F}\langle p^{(h)}, e^{(h)}_{\min}, e^{(h)}_{\max}, \mathfrak{s}^{(h)}_{\mathrm{n}}\rangle, \qquad \text{and} \qquad \mathcal{F}^{(\ell)} := \mathcal{F}\langle p, e^{(\ell)}_{\min}, e^{(\ell)}_{\max}, \mathfrak{s}^{(\ell)}_{\mathrm{n}}\rangle, \tag{5.1}$$

and the same superscript notation is used for x_{minsub}, $x_{\min}$, and $x_{\max}$.

Whether this technique can be used to simulate low precision accurately depends on the floating-point parameters of the two formats under consideration. First of all, we need to ensure that any number in $\mathcal{F}^{(\ell)}$ is representable in $\mathcal{F}^{(h)}$. In most cases, a necessary and sufficeint condition for $\mathcal{F}^{(\ell)} \subset \mathcal{F}^{(h)}$ is that $p \le p^{(h)}$ and $e^{(\ell)}_{\max} \le e^{(h)}_{\max}$. When $\mathfrak{s}^{(\ell)}_{\mathrm{n}} = \textbf{true}$ but $\mathfrak{s}^{(h)}_{\mathrm{n}} = \textbf{false}$, however, this is not sufficient, because a number that is subnormal in $\mathcal{F}^{(\ell)}$ is not necessarily normal in $\mathcal{F}^{(h)}$. In this case, we need to require that $e^{(\ell)}_{\min} \ge e^{(h)}_{\min} + p - 1$ or equivalently that $e^{(\ell)}_{\max} \le e^{(h)}_{\max} - p + 1$.

But this may not be enough when it comes to simulating arithmetic operations. Let $y \in \mathbb{R}$ be the exact result of an arithmetic operation $f(x_1, \ldots, x_N)$, where $x_1, \ldots, x_N \in \mathcal{F}^{(h)}$, and let $\mathrm{fl}_h : \mathbb{R} \to \mathcal{F}^{(h)}$ and $\mathrm{fl}_\ell : \mathbb{R} \to \mathcal{F}^{(\ell)}$ be the functions that round to the high- and low-precision formats, respectively. A correctly rounded floating-point implementation of f will return $\mathrm{fl}_h(y)$, and a faithful low-precision simulation should return $\mathrm{fl}_\ell(y)$.

Is $\mathrm{fl}_\ell(y) = \mathrm{fl}_\ell(\mathrm{fl}_h(y))$ for any value of $x_1, \ldots, x_N$? This is always true for directed rounding, but when round-to-nearest is considered one may have that $\mathrm{fl}_\ell(y) \ne \mathrm{fl}_\ell(\mathrm{fl}_h(y))$ if the intermediate format $\mathcal{F}^{(h)}$ does not have enough extra bits of precision.

If f is one of the arithmetic operators in [32, Sec. 5.4.1], for which the IEEE 754 standard mandates correct rounding, it is known that $p \le p^{(h)}/2 - 1$ will guarantee that double rounding will be innocuous for the four elementary arithmetic operations (+, −, ×, and ÷) and for the square root [53, 55].

If f is a transcendental function, we cannot guarantee that harmful double rounding will not occur, but this is not a problem, as the [32, Sec. 9.2] only recommends that these operations be correctly rounded, and many general-purpose mathematical libraries do not provide correct rounding. This is the case, for example, for the GNU C library[3] [40, Sec. 19.7]), for the OpenCL standard [24, Sec. 7.4] and [23, Sec. 4.4], which the Intel oneAPI math kernel library[4] uses to determine the accuracy of mathematical functions,[5] and for the Radeon open compute math library.[6] Most mathematical libraries do not provide correctly rounded implementations of transcendental functions, and we refer the interested reader to recent work by Innocente and Zimmermann [33] for an experimental investigation of the issue.

We now list the high-level functions we need to operate on floating-point numbers. In the description, x denotes a floating-point number in $\mathcal{F}^{(h)}$, n denotes a positive integer, and i is an integer index between 0 and $p^{(h)} - 1$ inclusive. Unless otherwise stated, these functions are not defined for infinities and NaNs.

- $\textsc{abs}(x)$ returns the absolute value of x. For infinities, we use the definition $\textsc{abs}(\pm\infty) = +\infty$.
- $\textsc{digit}(x, i)$ returns the ith digit of the significand of x from the left. The indexing starts from 0, so that $\textsc{digit}(x, i)$ is d_i in (4.2).
- $\textsc{exponent}(x)$ returns the exponent of x. This is the signed integer e in (4.1) if x is normal, and an integer $\gamma < e_{\min}$ otherwise.
- $\textsc{significand}(x)$ returns the integral significand of x, that is, the positive integer m in (4.1).
- $\textsc{rand}(n)$ returns a string of n randomly generated bits.

[3] https://www.gnu.org/software/libc/

[4] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html

[5] https://www.intel.com/content/www/us/en/develop/documentation/oneapi-dpcpp-cpp-compiler-dev-guide-and-reference/top/optimization-and-programming/intel-oneapi-dpc-c-compiler-math-library.html

[6] https://github.com/RadeonOpenCompute/ROCm-Device-Libs/blob/amd-stg-open/doc/OCML.md#supported-functions

Algorithm 5.1: Round a number from $\mathcal{F}^{(h)}$ to $\mathcal{F}^{(\ell)}$ in (5.1).

```
function CPFloat(x ∈ F^(h), F^(ℓ), roundFun : F^(h) × ℕ⁺ × F^(h) → F^(h))
    if s_n^(ℓ) then
        ζ ← x_minsub^(ℓ)
    else
        ζ ← x_min^(ℓ)
    if abs(x) < x_min^(ℓ) and e_min^(ℓ) > e_min^(h) then
        t ← p − (e_min^(ℓ) − exponent(x))
    else
        t ← p
    x̃ ← roundFun(x, t, ζ)
```

- $\textsc{sign}(x)$ returns -1 if the floating-point number x is negative and $+1$ otherwise. This function behaves as expected for (signed) zeros and infinities, i.e., $\textsc{sign}(\pm 0) = \pm 1$ and $\textsc{sign}(\pm\infty) = \pm 1$.
- $\textsc{tail}(x, i)$ returns the trailing $p^{(h)} - i$ bits of the significand of x as an unsigned integer. For infinities, this function returns 0 by convention, thus $\textsc{tail}(\pm\infty, i) = 0$ for any i.
- $\textsc{trunc}(x, i)$ returns the number x with the last $p^{(h)} - i$ bits of the significand set to zero. Truncating an infinity leaves it unchanged, thus $\textsc{trunc}(\pm\infty, i) = \pm\infty$ for any i.
- $\textsc{ulp}(x, i)$ returns the number $2^{\textsc{exponent}(x)-i+1}$, that is, the gap between x and its successor in a floating-point number system with i bits of precision. As noted by Muller [43], this function corresponds to the unit in the last place as defined by Overton [50, p. 14] and Goldberg [20].

How to implement these efficiently will be discussed in detail in Section 6.

Our rounding strategy is summarized in Algorithm 5.1. The function CPFloat computes the representation of the floating-point number $x \in \mathcal{F}^{(h)}$ in a lower-precision format $\mathcal{F}^{(\ell)}$. In the pseudocode, $\mathbb{N}^+$ denotes the set of positive integers. The input parameter roundFun is a pointer to one of the functions in Algorithm 5.2, 5.4, or 5.5. A call to $\textsc{roundFun}(x, t, \zeta)$ returns the floating-point number $\widetilde{x} \in \mathcal{F}^{(h)}$, that is, x rounded to $\mathcal{F}^{(\ell)}$. The function starts by setting the underflow threshold ζ, which corresponds to the smallest subnormal number if $\mathfrak{s}_{\mathrm{n}}^{(\ell)} =$ **true** and to the smallest normal number if $\mathfrak{s}_{\mathrm{n}}^{(\ell)} =$ **false**. This value will be used by roundFun to round numbers that are too small to be represented. Then the algorithm computes the number t of significant digits in the significand of the binary representation of $\widetilde{x}$. If x falls within the normal range of $\mathcal{F}^{(\ell)}$, then its significand has p significant digits and the algorithm sets $t = p$. If, on the other hand, $|x|$ is between $x_{\mathrm{minsub}}^{(\ell)}$ and $x_{\mathrm{min}}^{(\ell)}$, then the exponent of x is smaller than $e_{\mathrm{min}}^{(\ell)}$ and the number t of significant binary digits may have to be reduced. If $e_{\mathrm{min}}^{(\ell)} = e_{\mathrm{min}}^{(h)}$, then x is subnormal and has the same number of leading zeros in both storage and target formats. Otherwise, the value of t is given by the difference between p and the number of leading zeros in the representation of $\widetilde{x}$, including the zero to the left of the binary point.

In the coming sections we discuss how the function roundFun can be implemented for the six rounding strategies we consider.

5.1 Round-to-nearest

Our algorithm for rounding a floating-point number in $\mathcal{F}^{(h)}$ to the closest floating-point number in the lower-precision format $\mathcal{F}^{(\ell)}$ is given in Algorithm 5.2. The pseudocode describes, in particular, a variant of round-to-nearest known as ties-to-even, whereby a number exactly in-between two floating-point numbers is rounded to the rounding candidate with even significand, that is, a number

Algorithm 5.2: Function for round-to-nearest with ties-to-even.

```
1  function roundToNearest(x ∈ F^(h), t ∈ ℕ+, ζ ∈ F^(h))
2      if abs(x) ≤ ζ then                                          ▷ Underflow.
3          if abs(x) ≤ ζ/2 then
4              x̃ ← sign(x) · 0
5          else
6              x̃ ← sign(x) · ζ
7      else if abs(x) ≥ 2^(e_max^(ℓ)) (2 − 2^(−p)) then            ▷ Overflow.
8          x̃ ← sign(x) · +∞
9      else                                                        ▷ In range.
10         x̃ ← trunc(x, t)
11         if tail(x, p) > 2^(p^(h)−t−1) or (tail(x, p) = 2^(p^(h)−t−1) and digit(x, t − 1) = 1) then
12             x̃ ← x̃ + sign(x) · ulp(x̃, t)
13     return x̃
```

that has a 0 in the $t-1$ position to the right of the binary point. Two other variants, ties-to-zero and ties-to-away, will be briefly examined at the end of this section.

Initially, the function checks if the number to be rounded is too small to be represented in $\mathcal{F}^{(\ell)}$. If subnormals are supported then the smallest representable number $x^{(\ell)}_{\text{minsub}}$ has an odd fraction, and a tie value $x \in \mathcal{F}^{(h)}$ such that $|x| = \zeta/2$ is rounded towards zero. If subnormals are not supported, then x is equidistant from two numbers with even significands, since the significand of the smallest normal number $x^{(\ell)}_{\text{min}}$ in $\mathcal{F}^{(\ell)}$ is even. We still round x to zero, as this behavior is consistent with that of the GNU MPFR library.

Next, a number $x \in \mathcal{F}^{(h)}$ that is too large to be rounded to zero but has absolute value below the threshold ζ is rounded to $\text{sign}(x) \cdot \zeta$. If $|x|$ is larger than the threshold, the algorithm checks whether x is within the dynamic range of $\mathcal{F}^{(\ell)}$: following the IEEE 754 standard [32, Sec. 4.3.1], x will overflow to infinity without changes in sign if $|x| < 2^{e^{(\ell)}_{\text{max}}}(2 - 2^{-p})$.

If x is within the range of numbers representable in $\mathcal{F}^{(\ell)}$, then the algorithm truncates $x \in \mathcal{F}^{(h)}$ to t significant digits to obtain $\widetilde{x} \in \mathcal{F}^{(\ell)}$, the largest number (in absolute value) that satisfies $\text{sign}(\widetilde{x}) = \text{sign}(x)$ and $|\widetilde{x}| \leq |x|$. In general, $\widetilde{x}$ is one of the two floating-point numbers in $\mathcal{F}^{(\ell)}$ closest to x, the other candidate being $\widetilde{x} + \text{sign}(x) \cdot \text{ulp}(\widetilde{x}, t)$. In order to choose a rounding direction, we examine the value of the discarded bits. The unsigned integer $d := \text{tail}(x, t)$ represents the trailing $p^{(h)} - t$ digits of the significand of x. Thus $0 \leq d \leq 2^{p^{(h)}-t} - 1$, and if $d < \gamma := 2^{p^{(h)}-t-1}$ then $\widetilde{x}$ is the result to be returned, whereas if $d > \gamma$ then it is necessary to add (or subtract, if $\widetilde{x}$ is negative) $\text{ulp}(\widetilde{x}, t)$ in order to obtain the correctly rounded value. If $d = \gamma$, then we have a tie and we need to round to the nearest even number. Therefore we add $\text{sign}(x) \cdot \text{ulp}(\widetilde{x}, t)$ if the bit in position $t-1$ of the significand of $\widetilde{x}$ is a 1, and we leave $\widetilde{x}$ unchanged otherwise.

The latest revision of the IEEE 754 standard mentions two other tie-breaking rules for round-to-nearest: ties-to-zero, to be used for the recommended augmented operations, and ties-to-away, which is required for decimal formats. These can be implemented by changing the conditions of the if statements on lines 3 and 11 in Algorithm 5.2 to

```
3  if abs(x) ≤ ζ/2 then
```

[...]

```
11 if tail(x, t) > 2^(p^(h)−t−1) then
```

for ties-to-zero, and to

Algorithm 5.3: Function for round-to-odd.

```
1  function roundToOdd(x ∈ F^(h), t ∈ ℕ⁺, ζ ∈ F^(h))
2      if abs(x) < ζ and x ≠ 0 then                          ▷ Underflow.
3          x̃ ← sign(x) · ζ
4      else if abs(x) > x_max^(ℓ) and abs(x) ≠ +∞ then       ▷ Overflow.
5          x̃ ← sign(x) · x_max^(ℓ)
6      else                                                  ▷ In range.
7          x̃ ← trunc(x, t)
8          if tail(x, p) ≠ 0 then
9              digit(x̃, t − 1) ← 1
10     return x̃
```

3 **if** $\text{ABS}(x) < \zeta/2$ **then**

[...]

11 **if** $\text{TAIL}(x, t) \geq 2^{p^{(h)}-t-1}$ **then**

for ties-to-away.

Note that this implementation preserves the sign of zero, maps infinities to infinities, and does not change the encoding of quiet and signaling NaN values. The same observation is true for the rounding functions in Algorithm 5.3, 5.4, and 5.5.

5.2 Round-to-odd

The function roundToOdd in Algorithm 5.3 implements round-to-odd according to the definition in [3, Sec. 3.1], as this rounding mode is not part of the IEEE standard. Informally speaking, if x is exactly representable in $\mathcal{F}^{(\ell)}$, then the function returns it unchanged, otherwise it returns the number closest to x with an odd significand, that is, a significand with a trailing 1. In the spirit of the algorithms discussed so far, one could obtain $\widetilde{x}$ by truncating $|x|$ to the first t significant digits, checking the parity of the significand of $\widetilde{x}$, and adding $\text{ULP}(\widetilde{x}, t)$ to the result if $\widetilde{x}$ is even. However, in this case we know that the result of the truncation requires a correction only if the least significant digit of $\widetilde{x}$ is a 0, in which case adding $\text{ULP}(\widetilde{x}, t)$ amounts to setting that bit to 1. Therefore, we check the bits obliterated by the truncation, and if $\text{TAIL}(x, t) \neq 0$ then we conclude that x is not exactly representable in $\mathcal{F}^{(\ell)}$ and that the significand of the rounded $\widetilde{x}$ must be odd. We can ensure this by setting the digit in position $t - 1$ of the significand of $\widetilde{x}$ to 1. In practice, this operation will have an effect only if that digit is not already set to 1, and in particular will have no effect when $t = 1$, as the leading digit of the significand is not stored explicitly. The core idea of this algorithm is the same as that of the second of the two methods for round-to-odd discussed in [3, Sec. 3.4].

The algorithm must round $\widetilde{x}$ to the closest odd number in $\mathcal{F}^{(\ell)}$ if it falls within the underflow or the overflow range. Since 0 is even, rounding to 0 is not an option when this rounding mode is used, and numbers too small to be represented must be rounded to the smallest floating-point number with an odd significand of corresponding sign. When subnormals are supported, the smallest representable number $x_{\text{minsub}}^{(\ell)}$ is odd, thus numbers smaller than $x_{\text{minsub}}^{(\ell)}$ in absolute value are simply rounded to $\text{sign}(x) \cdot x_{\text{minsub}}^{(\ell)}$. If subnormals are not supported, on the other hand, the smallest representable number $x_{\text{min}}^{(\ell)}$ is even, and we are faced with a choice. We could round $\widetilde{x}$ such that $|\widetilde{x}| < x_{\text{min}}^{(\ell)}$ to $\text{sign}(x) \cdot \left(x_{\text{min}}^{(\ell)} + \text{ULP}(x_{\text{min}}^{(\ell)}, t)\right)$, which is the closest number with odd significand. This choice, however, feels unnatural: the operator thus defined would not be rounding to either of the floating-point numbers closest to $\widetilde{x}$. In our pseudocode, we prefer to round $\widetilde{x}$ to the rounding candidate largest in magnitude, that is,

$\text{sign}(x) \cdot x_{\text{min}}^{(\ell)}$.

The definition given by Boldo and Melquiond cannot be applied directly to values in the overflow range, as in principle the significand of $\pm\infty$ is neither odd nor even. Since $-x_{\text{max}}^{(\ell)}$ and $x_{\text{max}}^{(\ell)}$ are necessarily odd, we prefer to round values outside the finite range of $\mathcal{F}^{(\ell)}$ to the closest finite number. Being exactly representable, infinities themselves represent an exception to this rule, and the algorithm leaves them unchanged.

5.3 Directed rounding

The functions in Algorithm 5.4 show how to implement the three directed rounding modes prescribed by the IEEE 754 standard. The idea underlying the three functions is similar to that discussed for the function roundToNearest in Algorithm 5.2, the main differences being 1) the use of the sign, which is relevant when the rounding direction is not symmetric with respect to 0, and 2) the conditions under which a unit in the last place has to be added.

We start by discussing the function roundTowardPlusInfinity. First, we check whether x is within the range of numbers that are representable in $\mathcal{F}^{(\ell)}$. Numbers that are too small to be represented are rounded up to 0 if negative and to ζ if positive. Finite positive numbers larger than the largest representable number $x_{\text{max}}^{(\ell)}$ overflow to $+\infty$, whereas negative numbers smaller than the smallest representable number $-x_{\text{max}}^{(\ell)}$ are rounded up to $-x_{\text{max}}^{(\ell)}$, with the only exception of $-\infty$, which is handled below using the fact that $\text{trunc}(-\infty, t) = -\infty$ and $\text{tail}(-\infty, t) = 0$.

Next, the function computes $\widetilde{x}$, that is, the number x with significand truncated to t significant digits, and checks whether $\widetilde{x}$ is smaller than the smallest number representable in $\mathcal{F}^{(\ell)}$. The rounding can be easily performed by noting that the truncation $\widetilde{x}$ is the correct result if x is negative or exactly representable in $\mathcal{F}^{(\ell)}$. Otherwise, $\widetilde{x}$ is incremented by $\text{ulp}(\widetilde{x}, t)$.

The function roundTowardMinusInfinity is identical, modulo some sign adjustments to take the opposite rounding direction into account. The algorithm starts by checking that x is within the range of numbers representable in $\mathcal{F}^{(\ell)}$. Numbers between $-\zeta$ and 0 are rounded down to $-\zeta$, whereas those between 0 and ζ underflow and are flushed to 0. Numbers that are smaller than the smallest number representable in $\mathcal{F}^{(\ell)}$ overflow to $-\infty$, whereas finite numbers greater than the largest number representable in $\mathcal{F}^{(\ell)}$ are rounded to $x_{\text{max}}^{(\ell)}$, the only exception in this case being $+\infty$. In order to round a number that falls within the range of $\mathcal{F}^{(\ell)}$, we first compute $\widetilde{x}$ by truncating the significand of x to t significant digits, and then subtract $\text{ulp}(\widetilde{x}, t)$ from $\widetilde{x}$ if x is negative and not exactly representable with a t-digit significand.

The function roundTowardZero is simpler than the other two considered in this section, as truncation is sufficient to correctly round the significand of x to t significant digits. Underflow and overflow are also easier to handle: finite numbers that are smaller than the smallest representable number in absolute value are flushed to 0, whereas numbers too large to be represented are rounded to the closest representable finite number.

5.4 Stochastic rounding

The functions in Algorithm 5.5 describe how to implement the two variants of stochastic rounding we are concerned with.

The function roundStochastic implements the strategy that rounds $x \in \mathcal{F}^{(h)}$ to one of the two closest floating-point numbers with probability proportional to the distance. First, the algorithm considers numbers in the underflow range, whose rounding candidates are 0 and the threshold value ζ, which equals x_{minsub} if subnormals are supported and x_{min} if they are not. The distance between $|x|$ and 0 depends not only on the significand but also on the magnitude of x, thus the algorithm starts by computing the two values e_x and m_x, which represent the exponent and the integral significand of x, respectively. Being the exponent of a floating-point number in $\mathcal{F}^{(h)}$, e_x can be much smaller than

Algorithm 5.4: Functions for directed rounding modes.

function roundTowardPlusInfinity($x \in \mathcal{F}^{(h)}$, $t \in \mathbb{N}^+$, $\zeta \in \mathcal{F}^{(h)}$)
  if $x > 0$ **and** $x < \zeta$ **then** ▷ Underflow.
    $\widetilde{x} \leftarrow \zeta$
  else if $x \leq 0$ **and** $x > -\zeta$ **then**
    $\widetilde{x} \leftarrow \text{sign}(x) \cdot 0$
  else if $x > x_{\max}^{(\ell)}$ **then** ▷ Overflow.
    $\widetilde{x} \leftarrow +\infty$
  else if $x < -x_{\max}^{(\ell)}$ **and** $x \neq -\infty$ **then**
    $\widetilde{x} \leftarrow -x_{\max}^{(\ell)}$
  else ▷ In range.
    $\widetilde{x} \leftarrow \text{trunc}(x, t)$
    if $\text{tail}(x, p) \neq 0$ **and** $x > 0$ **then**
      $\widetilde{x} \leftarrow \widetilde{x} + \text{ulp}(\widetilde{x}, t)$
  return $\widetilde{x}$
function roundTowardMinusInfinity($x \in \mathcal{F}^{(h)}$, $t \in \mathbb{N}^+$, $\zeta \in \mathcal{F}^{(h)}$)
  if $x < 0$ **and** $x > -\zeta$ **then** ▷ Underflow.
    $\widetilde{x} \leftarrow -\zeta$
  else if $x \geq 0$ **and** $x < \zeta$ **then**
    $\widetilde{x} \leftarrow \text{sign}(x) \cdot 0$
  else if $\widetilde{x} > x_{\max}^{(\ell)}$ **and** $\widetilde{x} \neq +\infty$ **then** ▷ Overflow.
    $\widetilde{x} \leftarrow x_{\max}^{(\ell)}$
  else if $\widetilde{x} < -x_{\max}^{(\ell)}$ **then**
    $\widetilde{x} \leftarrow -\infty$
  else ▷ In range.
    $\widetilde{x} \leftarrow \text{trunc}(x, t)$
    if $\text{tail}(x, p) \neq 0$ **and** $x < 0$ **then**
      $\widetilde{x} \leftarrow \widetilde{x} - \text{ulp}(\widetilde{x}, t)$
  return $\widetilde{x}$
function roundTowardZero($x \in \mathcal{F}^{(h)}$, $t \in \mathbb{N}^+$, $\zeta \in \mathcal{F}^{(h)}$)
  if $\text{abs}(x) < \zeta$ **then** ▷ Underflow.
    $\widetilde{x} \leftarrow \text{sign}(x) \cdot 0$
  else if $\text{abs}(x) \geq x_{\max}^{(\ell)}$ **and** $\text{abs}(x) \neq +\infty$ **then** ▷ Overflow.
    $\widetilde{x} \leftarrow \text{sign}(x) \cdot x_{\max}^{(\ell)}$
  else ▷ In range.
    $\widetilde{x} \leftarrow \text{trunc}(x, t)$
  return $\widetilde{x}$

Algorithm 5.5: Functions for stochastic rounding.

function roundStochastic($x \in \mathcal{F}^{(h)}, t \in \mathbb{N}^+, \zeta \in \mathcal{F}^{(h)}$)
  if abs(x) $< \zeta$ **then** ▷ Underflow.
    $e_{\min} \leftarrow$ exponent(ζ)
    $e_x \leftarrow$ exponent(x)
    $m_x \leftarrow$ significand(x)
    $t \leftarrow \lfloor m_x \cdot 2^{e_x+1-e_{\min}} \rfloor$
    if $t >$ rand($p^{(h)}$) **then**
      $\widetilde{x} \leftarrow$ sign(x) $\cdot \zeta$
    else
      $\widetilde{x} \leftarrow$ sign(x) $\cdot 0$
  else ▷ In range or overflow.
    $\widetilde{x} \leftarrow$ trunc(x, t)
    if tail(x, t) $>$ rand($p^{(h)} - p$) **then**
      $\widetilde{x} \leftarrow \widetilde{x} +$ sign(x) $\cdot$ ulp($\widetilde{x}, t$)
    if $\widetilde{x} \geq 2^{e^{(\ell)}_{\max}}(2 - 2^{-p})$ **then** ▷ Overflow.
      $\widetilde{x} \leftarrow$ sign(x) $\cdot +\infty$
  return $\widetilde{x}$
function roundStochasticEqual($x \in \mathcal{F}^{(h)}, t \in \mathbb{N}^+, \zeta \in \mathcal{F}^{(h)}$)
  if abs(x) $< \zeta$ **and** $x \neq 0$ **then** ▷ Underflow.
    $\widetilde{x} \leftarrow$ sign(x) $\cdot$ randSelect($0, \zeta$)
  else if abs(x) $> x^{(\ell)}_{\max}$ **and** abs(x) $\neq +\infty$ **then** ▷ Overflow.
    $\widetilde{x} \leftarrow$ sign(x) $\cdot$ randSelect($x^{(\ell)}_{\max}, +\infty$)
  else ▷ In range.
    $\widetilde{x} \leftarrow$ trunc(x, t)
    if tail(x, t) $\neq 0$ **then**
      $\widetilde{x} \leftarrow$ randSelect($\widetilde{x}, \widetilde{x} +$ sign(x) $\cdot$ ulp($\widetilde{x}, t$))
  return $\widetilde{x}$
function randSelect($x \in \mathcal{F}^{(h)}, y \in \mathcal{F}^{(h)}$)
  if rand(1) $= 1$ **then**
    return x
  else
    return y

$e_{\min}^{(\ell)}$, in which case it may be necessary to rescale m_x in order to align its exponent to $e_{\min}^{(\ell)}$. This is achieved by multiplying m_x by $2^{e_x+1-e_{\min}^{(\ell)}}$. In the pseudocode we take the floor of the result in order to keep it integer, although this is not strictly necessary: we prefer to work with integer arithmetic here so that the integers generated by the random number generator can be used without any further post-processing. This is desirable not only from a performance point of view, but also because drawing floating-point numbers from the uniform distribution over an interval is not a trivial task, even when a good integer pseudo-random number generator is available [21]. Finally, the algorithm generates a $p^{(h)}$-digit random integer γ, which is used as a threshold to choose the rounding direction: if the discarded bits, interpreted as an unsigned integer, are larger than γ, then x is rounded away from zero, otherwise it is rounded towards zero.

The procedure for numbers in the representable range is easier. In this case it suffices to compute $\widetilde{x}$, the value of x truncated to t significant bits, and then generate a random integer r between 0 and $2^{p^{(h)}-t}$. Since $\textsc{tail}(x,t)$ represents the distance between x and $\widetilde{x}$, we increment the absolute value of $\widetilde{x}$ by $\textsc{ulp}(\widetilde{x},t)$ if $\textsc{tail}(x,t) > r$, and leave it unchanged otherwise. For overflow, we use the threshold value that the IEEE 754 standard recommends for round-to-nearest, and round numbers whose absolute value after rounding is larger than the threshold $2^{e_{\max}^{(\ell)}}(2-2^{-p})$ to infinity, leaving the sign unchanged.

The function roundStochasticEqual deals with the simpler strategy that rounds x up or down with equal probability. Depending on the interval in which x falls, the function selects the two closest representable numbers in $\mathcal{F}^{(\ell)}$ and calls the function randSelect to select one of them with equal probability. In the pseudocode, we use a single bit generated randomly to discriminate between the two rounding directions.

6 Efficient implementation for IEEE-like representation formats

The subroutines used in Section 5 can be implemented efficiently if we assume that the numbers are represented using the floating-point format described in Section 4. First, we need to define the semantics of the operators for bit manipulation that we will rely on. These are available in most programming languages, although the notation varies greatly from language to language. For clarity, we use a prefix notation for all the operators.

Let a and b be strings of n bits. The bits are indexed from left to right, so that a_0 and a_{n-1} denote the leftmost and the rightmost bit of a, respectively. For $i \in \mathbb{N}$, we define the following operators.

- Conjunction: $\mathrm{c} = \textsc{and}(\mathrm{a},\mathrm{b})$ is an n-bit string such that $\mathrm{c}_k = 1$ if a_k and b_k are both set to 1, and $\mathrm{c}_k = 0$ otherwise.
- Disjunction: $\mathrm{c} = \textsc{or}(\mathrm{a},\mathrm{b})$ is an n-bit string such that $\mathrm{c}_k = 1$ if at least one of a_k and b_k is set to 1, and $\mathrm{c}_k = 0$ otherwise.
- Negation: $\mathrm{c} = \textsc{not}(\mathrm{a})$ is an n-bit string such that $\mathrm{c}_k = 1$ if $\mathrm{a}_k = 0$, and $\mathrm{c}_k = 0$ otherwise.
- Logical shift left: $\mathrm{c} = \textsc{lsl}(\mathrm{a}, i)$ is an n-bit string such that $\mathrm{c}_k = 0$ if $k > (n-1) - i$, and $\mathrm{c}_k = \mathrm{a}_{k+i}$ otherwise.
- Logical shift right: $\mathrm{c} = \textsc{lsr}(\mathrm{a}, i)$ is an n-bit string such that $\mathrm{c}_k = 0$ if $k < i$, and $\mathrm{c}_k = \mathrm{a}_{k-i}$ otherwise.

Most of the operations used in Section 5 require extracting a certain subset of the bits in the binary representation of the floating-point number $x \in \mathcal{F}\langle p, e_{\min}, e_{\max}, \mathfrak{s}_\mathrm{n}\rangle$ while zeroing out the remaining ones. This can be achieved by taking the bitwise conjunction between the binary string that represents x and a bitmask, that is, a string as long as the binary representation of x that has ones in the positions corresponding to the bits to be extracted and zeros everywhere else. More generally, the functions in

Section 5 can be implemented using the operators above as follows. In the descriptions, x denotes the binary floating-point representation of x, n denotes a positive integer, and i denotes an integer index between 0 and $p-1$.

- $\textsc{abs}(x)$ can be implemented as $\textsc{and}(\mathsf{x}, \mathsf{m}_{\text{abs}})$, where m_{abs} is constituted by a single leading 0 followed by ones.
- $\textsc{digit}(x, i)$ can be implemented as $\textsc{and}(\mathsf{x}, \mathsf{m}_{\text{digit}})$, where $\mathsf{m}_{\text{digit}}$ has a 1 in the position corresponding to the digit to be extracted and 0 everywhere else. We note that checking whether this digit is 0 or 1 does not require any additional operations in programming languages such as C where 0 is interpreted as **false** and any other integer is interpreted as **true**.
- $\textsc{exponent}(x)$ can be implemented as a sequence of logic and arithmetic operations. The raw bits of the exponents can be extracted with $\mathsf{c} := \textsc{and}(\mathsf{x}, \mathsf{m}_{\text{exp}})$, where m_{exp} has 1 in the positions corresponding to the exponent bits of the binary representation of x. This can be converted into the unsigned integer $\textsc{lsr}(\mathsf{c}, p-1)$, and the signed exponent can be obtained by subtracting the bias of the storage floating-point format. If x is subnormal in $\mathcal{F}\langle p, e_{\min}, e_{\max}, \mathfrak{s}_n\rangle$, then the value computed in this way is $-e_{\max} = e_{\min} - 1$, and the correct value to return in this case is $e_{\min} + \lambda$, where λ is the number of trailing zeros in the significand of x, including the implicit bit.
- $\textsc{significand}(x)$ can be implemented leveraging the function $\textsc{exponent}$. The digits to the right of the radix point can be obtained as $\mathsf{c} := \textsc{and}(\mathsf{x}, \mathsf{m}_{\text{frac}})$ where m_{frac} is the bitmask that has the $p-1$ trailing bits set to 1 and the remaining bits set to 0. If $x_{\min} \le |x| \le x_{\max}$, then $\textsc{exponent}(x) > e_{\min}$ and the implicit bit must be set to 1. This can be achieved by using $\textsc{or}(\mathsf{c}, \textsc{lsl}(1, p-1))$, for instance.
- $\textsc{rand}(n)$ can be implemented by concatenating numbers produced by a pseudo-random number generator. Two m-bit strings a and b can be joined together by $\textsc{or}(\textsc{lsl}(\mathsf{a}, m), \mathsf{b})$, and the unnecessary bits can be set to zero using a suitable bitmask.
- $\textsc{sign}(x)$ is relatively expensive to implement by means of bit manipulation. However, note that we only need to compute the product $\text{sign}(x) \cdot y$ where y is a positive floating-point number. This operation can be implemented as $\textsc{or}(\textsc{and}(\mathsf{x}, \mathsf{m}_{\text{sign}}), \mathsf{y})$, where m_{sign} is the bitmask with a leading 1 followed by zeros and the string y denotes the floating-point representation of y.
- $\textsc{tail}(x, i)$ can be implemented as $\textsc{and}(\mathsf{x}, \mathsf{m}_{\text{tail}})$, where the trailing $p-i$ bits of m_{tail} are set to 1 and the remaining bits are set to 0. This way, bits i to $p-1$ of the significand of x are preserved while the rest of the bits, including those representing the sign and exponent, are set to zero.
- $\textsc{trunc}(x, i)$ can be implemented as $\textsc{and}(\mathsf{x}, \mathsf{m}')$, where $\mathsf{m}' = \textsc{not}(\mathsf{m}_{\text{tail}})$. This way, bits i to $p-1$ of the significand of x are set to zero while the rest of the bits of x, including the exponent and sign bits, are preserved.
- $\textsc{ulp}(x, p)$ is a rather expensive function to implement, because it requires extracting the exponent from the binary representation of x and then performing arithmetic operations on it. Increasing or decreasing x by $\textsc{ulp}(x, p)$, on the contrary, can be achieved efficiently using only one bit shift and one integer arithmetic operation. In particular, it suffices to add to the binary representation of x, seen as an unsigned integer, a number that has 0 everywhere but in the pth digit, that is, the digit in position $p-1$ of the significand. We note that this technique could fail if $x = \pm\infty$, since adding $\textsc{ulp}(x, p)$ in this fashion would turn infinities into NaNs. It is easy to check that this is not a problem in our setting, as we only add or subtract $\textsc{ulp}(x, p)$ when x is finite and nonzero.

It is possible to implement some of the rounding routines even more efficiently by extending to other rounding modes the technique for round-to-nearest with ties-to-even developed on [35, p. 2–17], which manipulates the binary representation of the floating-point number by using only integer arithmetic. As a demonstration, here we show the concrete values of the bitmasks, expressed in hexadecimal notation, that one would use to round a binary32 number y to binary16. These methods generalize easily to other combinations of storage and target formats, and we describe this in general for the conversion of a floating-point number $x \in \mathcal{F}\langle p^{(h)}, e_{\min}^{(h)}, e_{\max}^{(h)}, \mathfrak{s}_{\mathrm{n}}^{(h)} \rangle$ to $\mathcal{F}\langle p, e_{\min}^{(\ell)}, e_{\max}^{(\ell)}, \mathfrak{s}_{\mathrm{n}}^{(\ell)} \rangle$.

We denote the 32-bit string containing the floating-point representation of y by y, and use the uppercase Latin letters X and Y to denote the unsigned integers that can be obtained by interpreting x and y as unsigned integers in radix 2. All the usual underflow and overflow checks are not included here—the aim is to demonstrate the core ideas for performing each type of rounding efficiently. We recall that the sign of a floating-point number can be determined by checking the leftmost bit, and that x (resp. y) is negative if x_0 (resp. y_0) is set and positive otherwise. The rounding modes amenable to this approach can be implemented as follows.

- Round-to-nearest with ties-to-even: isolate the bit in position $p-1$ of the significand of x, and then compute $\textsc{trunc}(X + \textsc{lsr}(\mathsf{m}_{\mathrm{tail}}, 1) + \textsc{digit}(x, p-1), p)$. When rounding a binary32 number to binary16, the formula becomes $\textsc{trunc}(Y + \mathrm{0x7FFF} + \mathsf{y}_{15}, 16)$.
- Round-to-nearest with ties-to-away: return $\textsc{trunc}(X + \textsc{lsl}(\mathrm{0x1}, p^{(h)} - p - 1), p)$, which in our example becomes $\textsc{trunc}(Y + \mathrm{0x8000}, 16)$.
- Round-to-nearest with ties-to-zero: return $\textsc{trunc}(X + \textsc{lsr}(\mathsf{m}_{\mathrm{tail}}, 1), p)$, which in our example becomes $\textsc{trunc}(Y + \mathrm{0x7FFF}, 16)$.
- Round-toward-$+\infty$: return $\textsc{trunc}(X, p)$ if x_0 is set and $\textsc{trunc}(X + \mathsf{m}_{\mathrm{tail}}, p)$ otherwise. For our example, return $\textsc{trunc}(Y, 16)$, if y_0 is set and $\textsc{trunc}(Y + \mathrm{0xFFFF}, 16)$ if not.
- Round-toward-$-\infty$: return $\textsc{trunc}(X, p)$ if x_0 is not set and $\textsc{trunc}(X + \mathsf{m}_{\mathrm{tail}}, p)$ otherwise. For our example, return $\textsc{trunc}(Y, 16)$ if y_0 is not set and $\textsc{trunc}(Y + \mathrm{0xFFFF}, 16)$ otherwise.
- Round-toward-zero: return $\textsc{trunc}(X, p)$. For our example, return $\textsc{trunc}(Y, 16)$.
- Stochastic rounding with proportional probabilities: return $\textsc{trunc}(X + \textsc{rand}(p^{(h)} - t), p)$. For our example, return $\textsc{trunc}(Y + \textsc{rand}(13), 13)$.

7 Implementation and validation of the code

Our C implementation of the algorithms discussed in Section 5 and Section 6 is available on GitHub. The code can be compiled as a static or dynamic library, but we also provide the option to use `CPFloat` as a header-only library.

A header-only library allows the user to take advantage of the inlining feature of the C language, for maximum efficiency, and it also enhances the portability of the code, as packaging of the binaries and installation of the library are not required. To alleviate the main drawback of this approach, that is, a longer compilation time, we divided the library into two separate units, one for each supported storage format.

In order to achieve better performance on large data, our functions work directly on C arrays. All the algorithms discussed in Section 5 are embarrassingly parallel, and each element of an array can be rounded independently from all the others. Therefore, our code can leverage the OpenMP library, if available on the system in use.

In general, OpenMP brings significant gains in terms of performance, but greatly increases the execution time for arrays with just few elements. This well-known phenomenon is caused by the

additional overhead of synchronization and loop scheduling [5], which is negligible for large arrays but can be significant when only a small amount of work is allocated to each OpenMP thread. The impact of this overhead is hard to quantify in general, as it depends on the hardware platform as well as the number of OpenMP threads and the compiler used [6]. Our library contains both a parallel and a sequential version of each function, but we were unable to provide a single threshold that would allow the code to switch automatically from one variant to the other for optimal performance. Thus we devised a simple auto-tuning strategy that tries to determine the optimal threshold for the system in use by timing the rounding function on several arrays of different lengths and performing a binary search.

For generating the pseudo-random numbers required for stochastic rounding, we rely on algorithms from the family of permuted congruential generators developed by O'Neill [48], who provides a pure C implementation available on GitHub.[7] In our code we use the functions `pcg32_random_r` and `pcg64_random_r` to generate 32-bit and 64-bit random numbers, respectively; we initialize the random number generators with `pcg32_srandom_r` and `pcg64_srandom_r`, and advance them with `pcg32_advance_r` and `pcg64_advance_r`, respectively. As initial state, we use the current time as returned by `time(NULL)`. Use of the—considerably slower—default C pseudo-random number generator is also supported.

In order to validate our code experimentally, we wrote a suite of extensive unit tests. We describe in detail how we tested the rounding routines—all other functions in the library are tested by relying on them. We considered two storage formats, binary32 and binary64, which are available in C via the native data types `float` and `double`, respectively, and two target formats, binary16 and bfloat16. For each combination of storage and target formats, we performed three types of tests.

First we checked that all the numbers that can be represented exactly in the target format, including subnormals and special values such as infinities and NaNs, are not altered by any of the rounding routines. As the target formats we consider do not have an unduly large cardinality, we can test that this property is true for all representable numbers.

To check the correctness of the code when rounding is necessary, exhaustive testing is not an option, as the storage formats contain too many distinct values. In this case, we opted for testing only a set of representative values. For deterministic rounding, the correctness of the function can be assessed by checking that the output of the rounding routine matches the value predicted by the definition. For each pair of numbers $x_1, x_2 \in \mathcal{F}^{(\ell)}$ such that x_1 and x_2 are consecutive in $\mathcal{F}^{(\ell)}$ and $x_1 < x_2$, we considered five values in $\mathcal{F}^{(h)}$: $\mathrm{nextafter}(x_1, +\infty)$, $\mathrm{nextafter}(x_m, -\infty)$, x_m, $\mathrm{nextafter}(x_m, +\infty)$, and $\mathrm{nextafter}(x_2, -\infty)$. Here $\mathrm{nextafter}(x, y)$ denotes the next number in $\mathcal{F}^{(h)}$ after x in the direction of y, and x_m denotes the mid point between x_1 and x_2. We used the same technique for numbers in the underflow range, whereas for testing the correctness of overflow we used the values $\mathrm{nextafter}(\pm x^{(\ell)}_{\mathrm{max}}, \pm\infty)$, $\mathrm{nextafter}(\pm x^{(\ell)}_{\mathrm{bnd}}, \mp\infty)$, $\pm x^{(\ell)}_{\mathrm{bnd}}$, $\mathrm{nextafter}(\pm x^{(\ell)}_{\mathrm{bnd}}, \pm\infty)$, where $x^{(\ell)}_{\mathrm{bnd}} := 2^{e^{(\ell)}_{\mathrm{max}}}(2 - 2^{-p})$ is the IEEE 754 threshold for overflow in round-to-nearest.

This technique would not work for stochastic rounding, as each value that is not representable in $\mathcal{F}^{(\ell)}$ can be rounded to two different values. We produced a test set by taking, for each pair of numbers $x_1, x_2 \in \mathcal{F}^{(\ell)}$ such that x_1 and x_2 are consecutive in $\mathcal{F}^{(\ell)}$ and $x_1 < x_2$, the numbers $(3x_1 + x_2)/4$, $(x_1 + x_2)/2$, and $(x_1 + 3x_2)/4$. We rounded each number 1,000 times and confirmed that the rounding routines always return either x_1 or x_2, and that the empirical probability distribution matches the expected one. We validated the correctness of the implementation for values in the underflow range by using the same technique, whereas for inputs in the overflow range we repeated the test on the three values: $(3x^{(\ell)}_{\mathrm{max}} + x^{(\ell)}_{\mathrm{bnd}})/4$, $(x^{(\ell)}_{\mathrm{max}} + x^{(\ell)}_{\mathrm{bnd}})/2$, and $(x^{(\ell)}_{\mathrm{max}} + 3x^{(\ell)}_{\mathrm{bnd}})/4$.

The Makefile target

```
$ make ctest
```

runs the test suite for the C implementations.

[7] https://github.com/imneme/pcg-c

We designed a MEX interface for MATLAB and Octave which is in charge of parsing and checking the input, allocating the output, and calling our library to perform the rounding. In order to show that the interface is fully compatible with `chop`, we designed a set of tests by modifying the default test suite for `chop`.[8] These tests can be run with

```
$ make mtest
```

in MATLAB, and with

```
$ make otest
```

in Octave.

8 Performance evaluation

The experiments were run on a machine equipped with two 12-core Intel Xeon CPU E5-2690 v3 running at 2.60GHz. Exclusive node access was used to avoid timing artifacts and ensure that all 24 CPU threads were available for parallel runs. The C code was compiled with version 9.3.0 of the GNU Compiler Collection (GCC) with the optimization flag `-O3` and the architecture option `-march=native`. The MATLAB experiments were run using the 64-bit GNU/Linux version of MATLAB 9.10 (R2021a). For our parallel implementations, we used version 4.5 of the OpenMP library. Source code and scripts to reproduce the experiments discussed in this section are available on GitHub.[9]

We compare three versions of our codes.

- `cpfloat_seq` denotes the sequential C codes in our library. Rounding is performed using the algorithms in Section 6.
- `cpfloat` denotes the C codes in our library that leverage OpenMP and employ the auto-tuning technique discussed in Section 7 to switch between sequential and parallel implementations. Rounding is performed using the algorithms in Section 6.
- `cpfloat_ml` denotes our MEX interface to `cpfloat` compiled in MATLAB. For large matrices, this function relies on the parallel version of our C codes.

As the numerical validation of the code has already been discussed in Section 7, here we focus on timings. We time the C or C++ code by comparing the value returned by the function `clock_gettime` with `CLOCK_MONOTONIC` before and after the execution, and take the median of 1,000 repetitions in order to reduce the influence of possible outliers. For the MATLAB code, we rely on the function `timeit`, which runs a portion of code several times and returns the median of the measurements.

8.1 Performance of the C interface

In this section we compare the performance of `CPFloat`, GNU MPFR, and FloatX by considering the following implementations.

- `chop_mpfr` denotes the codes that rely on the GNU MPFR library.[10] As storage format, we use the GNU MPFR custom data type `mpfr_t`. For rounding, our implementation sets the precision and exponent range of MPFR to the precision and exponent range of the target format and then converts the binary64 input using the function `mpfr_set_d`. Arithmetic operations use `mpfr_t` arrays for both input and output.

[8] https://github.com/higham/chop/blob/master/test_chop.m
[9] https://github.com/north-numerical-computing/cpfloat_experiments
[10] https://www.mpfr.org/

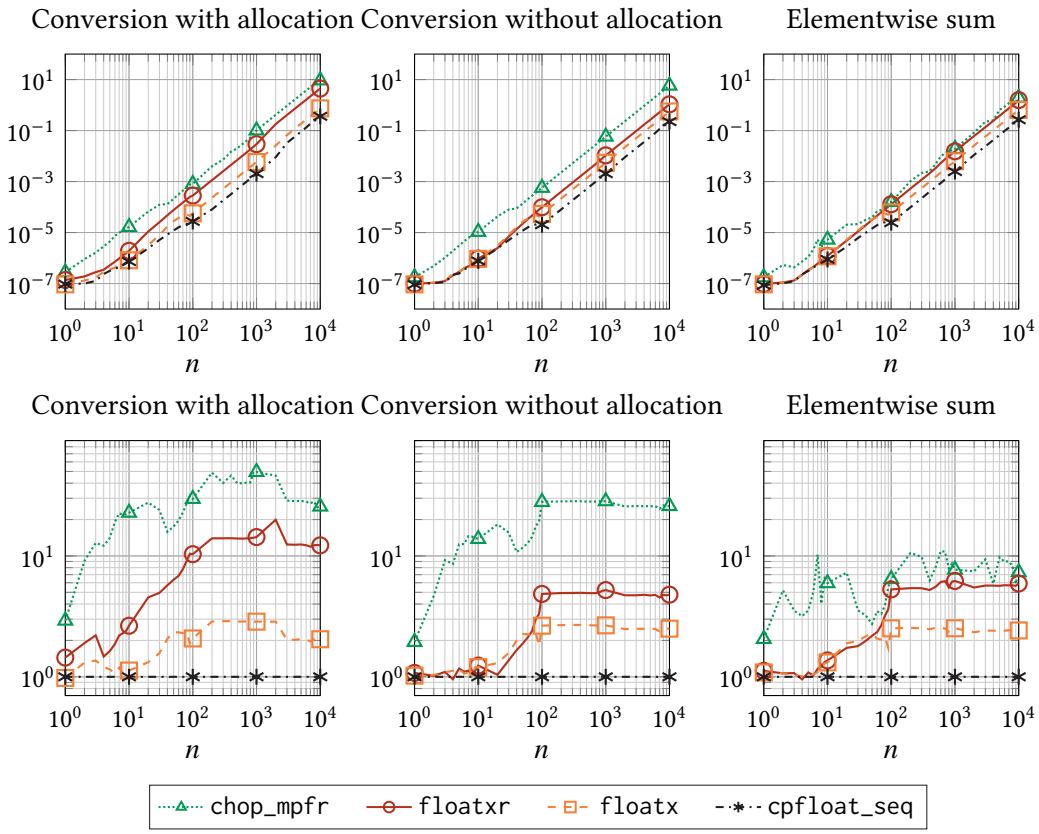

Figure 1: Top: execution time, in seconds, of `chop_mpfr`, `floatxr`, `floatx`, and `cpfloat_seq` to convert matrices of order n from binary64 to binary16 (left and middle) and to compute the sum of two matrices of size n in binary16 (right) using round-to-nearest with even-on-ties. Bottom: Corresponding slowdown plots, where `cpfloat_seq` is taken as baseline. The timings include the allocation of the output vector only in the left-most panel.

- `floatx` denotes the codes that use the `floatx` class from the FloatX library.[11] Arrays of binary64 numbers are converted to a lower-precision target format by invoking the constructor of the `floatx` class. This code requires that the parameters of the target format be specified at compile time, as the `floatx` class uses C++ templates that are instantiated only for the low-precision formats declared in the source code. For arithmetic operations, we use arrays of `floatx` objects as input and output.

- `floatxr` denotes the codes that rely on the `floatxr` class in the FloatX library. Conversion is performed using the class constructor. This function is more flexible than `floatx` in that the number of bits of precision and the maximum exponent allowed for the target format can be specified at runtime. As with the other implementations, for arithmetic operations we assume that input and output are arrays of `floatxr` objects.

Figure 1 compares the time required by `chop_mpfr`, `floatxr`, `floatx`, and `cpfloat_seq` to perform two operations: converting a square matrix from the storage format to the target format and computing the sum of two matrices in the target format. For format conversion, we report timings both including (left) and excluding (middle) the time needed to allocate the output vector. For the sum we report only the time needed to execute the operation (right), assuming that the memory to store the result has already been allocated.

[11] https://github.com/oprecomp/FloatX

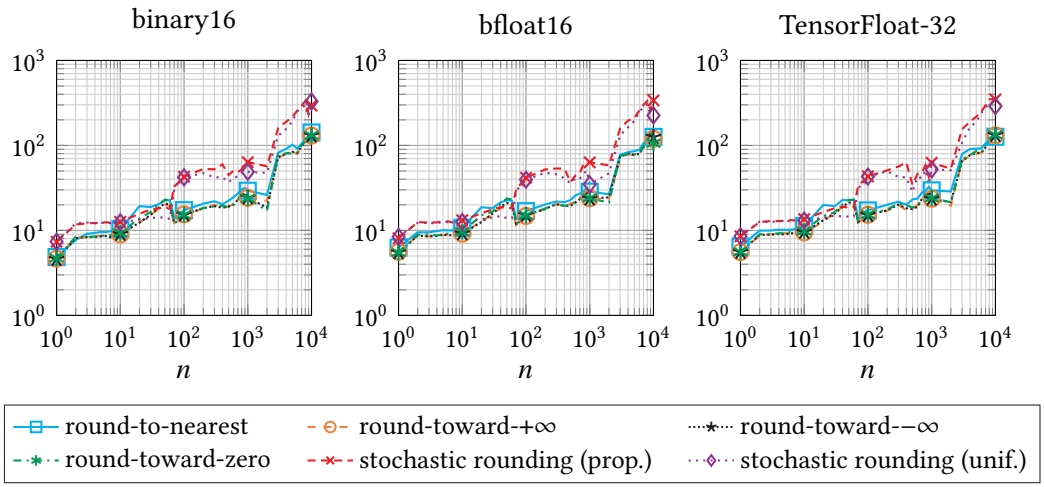

Figure 2: Ratio of the execution times of `chop` to that of `cpfloat_ml` on $n \times n$ matrices of normal floating-point numbers stored in binary64. Target formats are binary16 (left), bfloat16 (center), and TensorFloat-32 (right).

As storage and target formats, we use binary64 and binary16, respectively. We repeated the experiment simulating bfloat16 and TensorFloat-32 arithmetic—we do not reproduce these results here as they are indistinguishable from those for binary16. We use only round-to-nearest with ties-to-even, as the FloatX library currently does not support any other rounding modes. We observe, however, that `chop_mpfr` also supports directed rounding as prescribed by the IEEE 754 floating-point standard.

The plots in the top row report the median timing of 100 runs for each algorithm. The plots in the bottom row present the same data as slowdown with respect to the timings of `cpfloat_seq`. Unsurprisingly, the execution time of the four algorithms grows quadratically with the order of the matrix to be converted and thus linearly with the number of entries in the matrix. For matrices of small size, the performance of `floatx` and `cpfloat_seq` are essentially indistinguishable, and for `floatxr` there is only a difference when the time to allocate the memory is considered. For matrices with 100 or more elements ($n > 10$), `cpfloat_seq` is the fastest of the four implementations we consider. In this regime, `floatx` is typically two to three times slower than `cpfloat_seq`, whereas for the two other algorithms the performance varies depending on the operation being considered. The slowdown factor of `floatxr` can get well over ten for conversion if the time needed to allocate the output is factored in, but it goes below five when allocations are not considered. The performance of `chop_mpfr`, on the other hand, seems to depend mostly on what operation is performed: the slowdown factor is mostly over 20 for data conversion but is generally below 10 for sums.

We remark that `chop_mpfr`, `floatxr`, and `cpfloat_seq` are more flexible than `floatx`, as the latter requires the parameters of the target format be known at compile time, in order for the compiler to instantiate the templates appropriately.

8.2 Performance of the MATLAB interface

In this section we compare the performance of `cpfloat_ml` with that of existing MATLAB alternatives.

First, we consider the MATLAB function `chop`, which is available on GitHub.[12] Figure 2 reports the speedup of `cpfloat_ml` over `chop`. In each plot we consider the conversion of square matrices of size n between 1 and 10,000 using the six rounding modes implemented in `chop`. We consider binary64 as storage format, and binary16 (left), bfloat16 (center), and TensorFloat-32 (right) as target formats.

[12] https://github.com/higham/chop

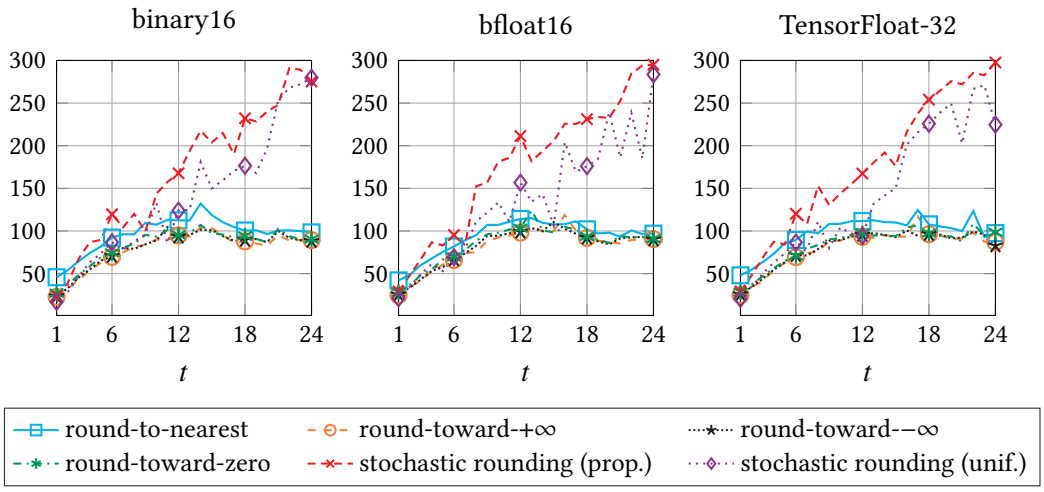

Figure 3: Ratio of the execution times of `chop` to that of `cpfloat_ml` on a 10,000 × 10,000 matrix of normal floating-point numbers stored in binary64 as the number of threads increases. Target formats are binary16 (left), bfloat16 (center), and TensorFloat-32 (right).

The input data is obtained by manipulating the entries of an $n \times n$ matrix X of pseudo-random numbers uniformly distributed in $(0, 1)$. To generate a matrix of normal numbers, we add to each entry of X the constant value $x_{\min}^{(\ell)}$, which guarantees that x_{ij}, for $i, j = 1, \ldots, n$, is distributed uniformly in the interval $\left(x_{\min}^{(\ell)}, 1 + x_{\min}^{(\ell)}\right)$.

In all cases, the speedup is greater than one and increases with the size of the input matrix. In other words, `cpfloat_ml` is always faster than `chop`, and particularly so for larger matrices. The two rounding modes for which the new algorithms bring the most significant gains are the two flavors of stochastic rounding. This is expected: generating pseudo-random numbers accounts for a large fraction of the execution time for these rounding modes, and by using a more efficient pseudo-random number generator the new algorithms have a great advantage over `chop`. The remaining four rounding modes show a very similar speedup, although the curves for round-to-nearest are generally slightly favorable.

We repeated the experiment using binary32 instead of binary64 as storage format, and considering matrices whose entries are subnormal, rather than normal, in the target format. The results of these experiments are not included here as they are not noticeably different from those in Figure 2.

The timings used to generate Figure 2 were obtained allowing MATLAB to use all 24 computational threads available on the system whereon the experiment was run. In order to assess how the better performance of `cpfloat_ml` depends on the number of threads used, we run a strong scaling experiment. We took matrices of size $n = 10{,}000$ (the largest value considered in Figure 2) and measured the speedup as the number t of computational threads increases. We did this by setting the maximum number of computational threads that MATLAB is allowed to use by means of the function `maxNumCompThreads`. The ratios between the execution times of `chop` and `cpfloat_ml` for binary16, bfloat16, and TensorFloat-32 are reported in Figure 3.

We note that the execution time of both functions increases as the number of threads is reduced, which confirms that they can both exploit parallelism and take advantage of additional computational threads available. On such large matrices, `cpfloat_ml` is always at least one order of magnitude faster than `chop`. For directed rounding modes, the speedup is just over 20× if a single thread is used, it increases until $t \leq 12$, which is the maximum number of cores of one of the available CPUs, and then settles down just below 100×. The speedup for round-to-nearest oscillates between 50× and 120×, but follows a similar trend. For the two stochastic rounding modes, on the other hand, the speedup does not stop increasing at $t = 12$, and reaches just below 300× for $t = 24$.

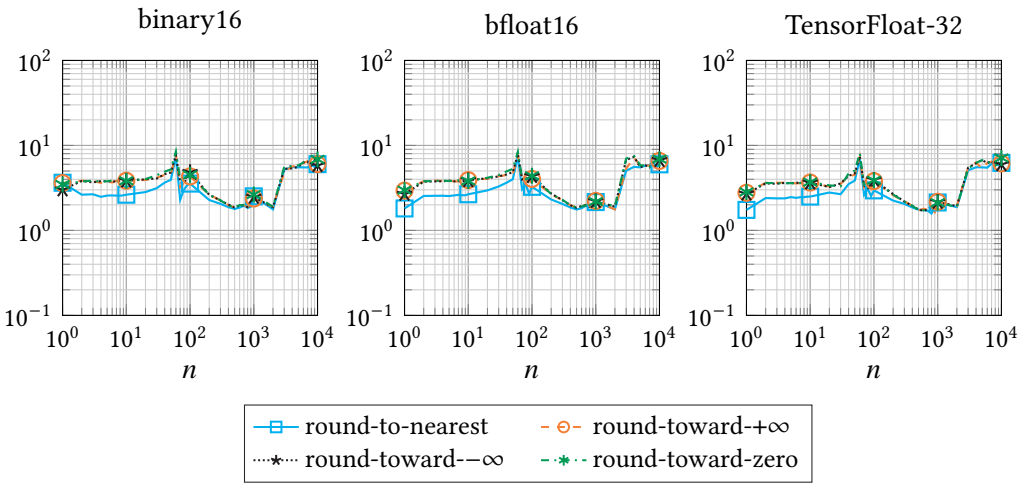

Figure 4: Ratio of the execution times of `flround` from INTLAB V12 to that of `cpfloat_ml` on $n \times n$ matrices of normal floating-point numbers stored in binary64.

The data shows that `chop` and `cpfloat_ml` can both take advantage of additional computational resources, but the latter is more efficient than the former at doing so.

We compared the performance of `cpfloat_ml` and of the MATLAB function `f_d_dec2floatp` from the FLOATP_Toolbox[13] in a similar way. This library is less efficient than `chop` at the task we examine: it is typically over 100 times slower than `cpfloat_ml` at simulating binary16 and bfloat16 arithmetic, and always over 1,000 slower at simulating TensorFloat-32 arithmetic.

Finally, we run a similar comparison with INTLAB's `flround` function, part of the `fl` custom precision arithmetic library [55] of INTLAB.[14] The results are shown in Figure 4. This rounding function supports only binary64 as a storage format and does not support rounding to odd or stochastic rounding modes. These experiments show that `cpfloat_ml` is faster than INTLAB's `flround` by up to 9×.

We remark that `chop`, the FLOATP_Toolbox, and INTLAB are entirely written in MATLAB, whereas `CPFloat` relies on optimized C code. Therefore, the performance observations in this section should be understood as a comparison of implementations available in MATLAB, rather than comparison among the underlying algorithms.

8.3 Overhead of the MATLAB interface

As a final test, we consider the overhead introduced by MATLAB when calling the underlying C implementation of the rounding algorithms. In Figure 5 we compare the execution time required to convert a matrix to binary16 with a direct call to the C code (first column) or with a call to the MEX interface in MATLAB (second column). As the performance of the two algorithms is very similar and the data in the two series is hard to compare directly, we provide the speedup in the third column. We repeat the experiment for both binary32 (top row) and binary64 (bottom row). We remark that the C functions were tuned by using the `make autotune` command, whereas for the MEX interface we used `cpfloat_autotune`, a MATLAB function included in the software package.

The raw timings show that in our implementations stochastic rounding is the slowest rounding mode. The performance of the other rounding modes is so similar that the lines are hard to distinguish for both the C and the MEX interface. The data in the right-most column shows that for both storage formats we consider, the overhead of the MEX interface is significant for small matrices, but becomes

[13] https://gerard-meurant.pagesperso-orange.fr/floatp.zip
[14] https://www.tuhh.de/ti3/rump/intlab/

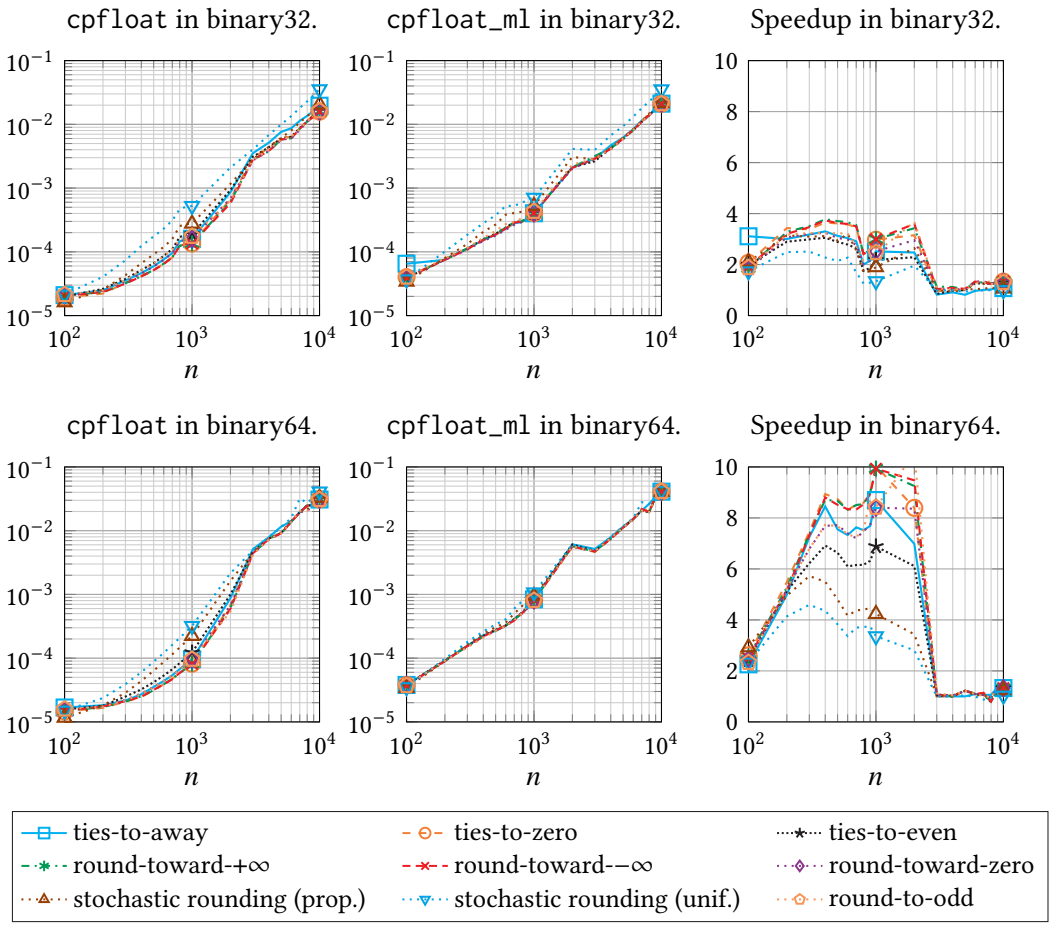

Figure 5: Execution time, in seconds, of `cpfloat` (first column) and `cpfloat_ml` (second column) on matrices of increasing order n and target format binary16. The third column represents the ratio of the execution time in the first column to that in the second.

negligible for matrices of order 3,000 or more.

Our results seem to suggest that MATLAB code that requires the functionalities of `cpfloat_ml` should be translated into C in order to obtain the maximum efficiency. We would like to stress, however, that this translation might bring only a minor performance gain, and in fact not be worth the effort, unless the `cpfloat` function is used extensively on matrices of small size. In fact, the overhead of the MEX interface is modest in absolute terms, and is noticeable only in cases when the overall execution time of both `cpfloat` and `cpfloat_ml` is below 5 milliseconds. This suggests that, in most cases, switching to a pure C implementation would bring only a marginal benefit, if any.

9 Summary and future work

Motivated by the growing number of tools and libraries for simulating low-precision arithmetic, we considered the problem of rounding floating-point numbers to low precision in software. We developed low-level algorithms for a number of rounding modes, explained how to implement them efficiently using bit manipulation, and discussed how to validate their behavior experimentally by means of exhaustive testing. We developed `CPFloat`, an efficient C library that implements all the algorithms discussed here and can be used from within MATLAB and Octave by means of a MEX interface we provide. When used in C, `CPFloat` can act as a full custom-precision floating-point arithmetic library, as it supports elementary arithmetic operations and mathematical functions such as those available

in `math.h` for binary32 and binary64. Our experimental results show that the new implementations outperform existing alternatives in C, C++, and MATLAB.

Traditionally, floating-point arithmetic has been the most widely adopted technique for working with non-integer numbers in high-performance scientific computing, but alternative methods have recently begun to gain popularity. In particular, we believe that the techniques we developed here could be adapted to posit arithmetic [26, 12], a generalization of the IEEE 754 floating-point number format, and to fixed-point arithmetic, a de facto standard technique for working with reals on systems that are not equipped with a floating-point unit. This will be the subject of future work.

Acknowledgments

The authors are grateful to Nicholas J. Higham, for discussions about the MATLAB function chop and for insightful observations on early drafts of this document, to Laura Morgenstern and Anne Reinarz, for feedback on the manuscript, and to Theo Mary, for testing the MEX interface to CPFloat and for reporting bugs affecting the software. The authors thank the anonymous referees of ACM Transactions on Mathematical Software for their comments that substantially improved the quality of the paper and led to a change in the scope of the library.

References

[1] Arm Limited, *Arm architecture reference manual*, Tech. Report ARM DDI 0487F.c (ID072120), Mar. 2020.

[2] G. Aupy, A. Benoit, A. Cavelan, M. Fasi, Y. Robert, H. Sun, and B. Uçar, *Coping with silent errors in HPC applications*, in https://doi.org/10.1007/978-3-319-46376-6, A. Adamatzky, ed., Springer-Verlag, Cham, Switzerland, Nov. 2016, pp. 269–292.

[3] S. Boldo and G. Melquiond, *Emulation of a FMA and correctly rounded sums: Proved algorithms using rounding to odd*, IEEE Trans. Comput., 57 (2008), p. 462–471.

[4] E. Brun, D. Defour, P. de Oliveira Castro, M. Istoan, D. Mancusi, E. Petit, and A. Vaquet, *A study of the effects and benefits of custom-precision mathematical libraries for HPC codes*, IEEE Trans. Emerg. Topics Comput., 9 (2021), p. 1467–1478.

[5] J. M. Bull, *Measuring synchronisation and scheduling overheads in OpenMP*, in Proceedings of First European Workshop on OpenMP, Wiley, Sept. 1999, pp. 99–105.

[6] J. M. Bull, F. Reid, and N. McDonnell, *A microbenchmark suite for OpenMP tasks*, in Proceedings of the 8th International Workshop on OpenMP, vol. 7312 of Lecture Notes in Computer Science, Berlin, Heidelberg, June 2012, Springer-Verlag, p. 271–274.

[7] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, *Bfloat16 processing for neural networks*, in Proceedings of the 26th IEEE Symposium on Computer Arithmetic, Institute of Electrical and Electronics Engineers, June 2019, pp. 88–91.

[8] M. P. Connolly, N. J. Higham, and T. Mary, *Stochastic rounding and its probabilistic backward error analysis*, SIAM J. Sci. Comput., 43 (2021), pp. A566–A585.

[9] M. Croci, M. Fasi, N. J. Higham, T. Mary, and M. Mikaitis, *Stochastic rounding: Implementation, error analysis and applications*, Roy. Soc. Open Sci., 9 (2022).

[10] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, *Loihi: A neuromorphic manycore processor with on-chip learning*, IEEE Micro, 38 (2018), pp. 82–99.

[11] A. Dawson and P. D. Düben, *rpe v5: An emulator for reduced floating-point precision in large numerical simulations*, Geosci. Model Dev., 10 (2017), pp. 2221–2230.

[12] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen, *Posits: The good, the bad and the ugly*, in Proceedings of the Conference for Next Generation Arithmetic, ACM Press, Mar. 2019, pp. 1–10.

[13] M. Fasi, N. J. Higham, F. Lopez, T. Mary, and M. Mikaitis, *Matrix multiplication in multiword arithmetic: Error analysis and application to GPU tensor cores*, MIMS EPrint 2022.3, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Jan. 2022. To appear in SIAM J. Sci. Comput.

[14] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, *Numerical behavior of NVIDIA tensor cores*, PeerJ Comput. Sci., 7 (2021), pp. e330(1–19).

[15] M. Fasi and M. Mikaitis, *Algorithms for stochastically rounded elementary arithmetic operations in IEEE 754 floating-point arithmetic*, IEEE Trans. Emerg. Topics Comput., 9 (2021), p. 1451–1466.

[16] G. Flegar, F. Scheidegger, V. Novaković, G. Mariani, A. E. Tomás, A. C. I. Malossi, and E. S. Quintana-Ortí, *FloatX: A C++ library for customized floating-point arithmetic*, ACM Trans. Math. Software, 45 (2019), p. 1–23.

[17] G. E. Forsythe, *Round-off errors in numerical integration on automatic machinery*, Bull. Amer. Math. Soc., 56 (1950), pp. 55–64.

[18] G. E. Forsythe, *Reprint of a note on rounding-off errors*, SIAM Rev., 1 (1959), p. 66–67.

[19] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, *MPFR: A multiple-precision binary floating-point library with correct rounding*, ACM Trans. Math. Software, 33 (2007), pp. 13:1–13:15.

[20] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Comp. Surv., 23 (1991), p. 5–48.

[21] F. Goualard, *Generating random floating-point numbers by dividing integers: A case study*, in Computational Science – ICCS 2020, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, eds., Lecture Notes in Computer Science, Cham, Switzerland, 2020, Springer-Verlag, p. 15–28.

[22] Graphcore Limited, *IPU programmer's guide*, 2020.

[23] K. O. W. Group, *The OpenCL C++ 1.0 Specification*, The Khronos Group, July 2019. Version V2.2-11.

[24] K. O. W. Group, *The OpenCL C 2.0 Specification*, The Khronos Group, July 2019. Version V2.2-11.

[25] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, *Deep learning with limited numerical precision*, in Proceedings of the 32nd International Conference on Machine Learning, F. Bach and D. Blei, eds., vol. 37 of Proceedings of Machine Learning Research, PMLR, July 2015, pp. 1737–1746.

[26] J. L. Gustafson and I. T. Yonemoto, *Beating floating point at its own game: Posit arithmetic*, Supercomputing Frontiers and Innovations, 4 (2017), pp. 71–86.

[27] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second ed., 2002.

[28] N. J. Higham and S. Pranesh, *Simulating low precision floating-point arithmetic*, SIAM J. Sci. Comput., 41 (2019), pp. C585–C602.

[29] M. Hopkins, M. Mikaitis, D. R. Lester, and S. Furber, *Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations*, Philos. Trans. R. Soc. A, 378 (2020).

[30] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, Oct. 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[31] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985)*, Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, Aug. 2008.

[32] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*, Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, July 2019.

[33] V. Innocente and P. Zimmermann, *Accuracy of mathematical functions in single, double, extended double and quadruple precision*, Tech. Report hal-03141101, version 2, Inria, Sept. 2022.

[34] Intel Corporation, *BFLOAT16—Hardware Numerics Definition*, Nov. 2018. White paper. Document number 338302-001US.

[35] Intel Corporation, *Intel architecture instruction set extensions and future features programming reference*, Mar. 2020.

[36] International Business Machines Corporation, *Power ISA version 3.0 B*, 2017.

[37] V. Lefèvre, *SIPE: Small integer plus exponent*, in Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Institute of Electrical and Electronics Engineers, Apr. 2013, pp. 99–106.

[38] V. Lefèvre, *Sipe: A Mini-Library for Very Low Precision Computations with Correct Rounding*, Tech. Report hal-00864580, version 1, Inria, Sept. 2013.

[39] C. Lichtenau, S. Carlough, and S. M. Mueller, *Quad precision floating point on the IBM z13*, in Proocedings of the 23rd IEEE Symposium on Computer Arithmetic, Institute of Electrical and Electronics Engineers, July 2016, pp. 87–94.

[40] S. Loosemore, e. R. M. Stallman, R. McGrath, A. Oram, and U. Drepper, *The GNU C Library Reference Manual*, Free Software Foundation, for version 2.36 ed., Aug. 2022.

[41] G. Meurant, *FLOATP_Toolbox*, 2020. Matlab software, variable precision floating point arithmetic.

[42] C. B. Moler, *"Half precision" 16-bit floating point arithmetic*. Blog post, Dec. 2017.

[43] J.-M. Muller, *On the definition of* $\mathrm{ulp}(x)$, Tech. Report RR-5504, LIP RR-2005-09, Inria, LIP, May 2005.

[44] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*, Birkhäuser, 2nd ed., 2018.

[45] NVIDIA Corporation, *NVIDIA Tesla P100 architecture*, Tech. Report WP-08019-001_v01.1, 2016.

[46] NVIDIA Corporation, *NVIDIA A100 tensor core GPU architecture*, tech. report, 2020.

[47] NVIDIA Corporation, *NVIDIA H100 tensor core GPU architecture*, 2020.

[48] M. E. O'Neill, *PCG: A family of simple fast space-efficient statistically good algorithms for random number generation*, Tech. Report HMC-CS-2014-0905, Harvey Mudd College, Sept. 2014.

[49] J. Osorio, A. Armejach, E. Petit, G. Henry, and M. Casas, *FASE: A fast, accurate and seamless emulator for custom numerical formats*, in Proceedings of the 2022 IEEE International Symposium on Performance Analysis of Systems and Software, May 2022, pp. 144–146.

[50] M. L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[51] Radeon Technologies Group, *Radeon's next-generation Vega architecture*, tech. report, Advanced Micro Devices, 2017. File no longer available on the AMD website. Archived version at `https://en.wikichip.org/w/images/a/a1/vega-whitepaper.pdf`.

[52] A. Reinarz, J.-M. Gallard, and M. Bader, *Influence of a-posteriori subcell limiting on fault frequency in higher-order DG schemes*, in Proceedings of the 8th IEEE/ACM Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS), Piscataway, NJ, USA, Nov. 2018, Institute of Electrical and Electronics Engineers, pp. 79–86.

[53] P. Roux, *Innocuous double rounding of basic arithmetic operations*, J. Formaliz. Reason., 7 (2014), pp. 131–142.

[54] S. M. Rump, *INTLAB — INTerval LABoratory*, in Developments in Reliable Computing, T. Csendes, ed., Dordrecht, Netherlands, Sept. 1999, Springer-Verlag, pp. 77–104.

[55] S. M. Rump, *IEEE754 precision-k base-β arithmetic inherited by precision-m base-β arithmetic for $k < m$*, ACM Trans. Math. Software, 43 (2017), p. 1–15.

[56] J. O. Ríos, A. Armejach, E. Petit, G. Henry, and M. Casas, *Dynamically adapting floating-point precision to accelerate deep neural network training*, in Proceedings of the 20th IEEE International Conference on Machine Learning and Applications, Dec. 2021, pp. 980–987.

[57] P. Samfass, T. Weinzierl, A. Reinarz, and M. Bader, *Doubt and redundancy kill soft errors—towards detection and correction of silent data corruption in task-based numerical software*, in Proceedings of the 11th IEEE/ACM Workshop on Fault Tolerance for HPC at eXtreme Scale, Piscataway, NJ, USA, Nov. 2021, Institute of Electrical and Electronics Engineers.

[58] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, *Training deep neural networks with 8-bit floating point numbers*, in Advances in Neural Information Processing Systems, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds., vol. 31, Curran Associates, Dec. 2018, pp. 7675–7684.

[59] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, Notes on Applied Science No. 32, Her Majesty's Stationery Office, 1963. Also published by Prentice-Hall, Englewood Cliffs, NJ, USA. Reprinted by Dover, New York, 1994.

[60] T. Zhang, Z. Lin, G. Yang, and C. De Sa, *QPyTorch: A low-precision arithmetic simulation framework*, in Proceedings of the 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition, Piscataway, NJ, USA, Dec. 2019, Institute of Electrical and Electronics Engineers, pp. 11–13.