FloatX: A C++ Library for Customized Floating-Point Arithmetic

GORAN FLEGAR, Dept. d'Enginyeria i Ciència dels Computadors, Universitat Jaume I, Spain
FLORIAN SCHEIDEGGER, IBM* Research–Zurich, Switzerland
VEDRAN NOVAKOVIĆ, Dept. d'Enginyeria i Ciència dels Computadors, Universitat Jaume I, Spain
GIOVANI MARIANI, IBM* Research–Zurich, Switzerland
ANDRÉS E. TOMÁS, Dept. d'Enginyeria i Ciència dels Computadors, Universitat Jaume I, Spain
A. CRISTIANO I. MALOSSI, IBM* Research–Zurich, Switzerland
ENRIQUE S. QUINTANA-ORTÍ, Dept. d'Informàtica de Sistemes i Computadors, Universitat Politècnica de València, Spain

We present FloatX (Float eXtended), a C++ framework to investigate the effect of leveraging customized floating-point formats in numerical applications. FloatX formats are based on binary IEEE 754 with smaller significand and exponent bit counts specified by the user. Among other properties, FloatX facilitates an incremental transformation of the code, relies on hardware-supported floating-point types as back end to preserve efficiency, and incurs no storage overhead. The paper discusses in detail the design principles, programming interface and datatype casting rules behind FloatX. Furthermore, it demonstrates FloatX's usage and benefits via several case studies from well-known numerical dense linear algebra libraries, such as BLAS and LAPACK; the Ginkgo library for sparse linear systems; and two neural network applications related with image processing and text recognition.

CCS Concepts: • **Mathematics of computing → Mathematical software**; **Arbitrary-precision arithmetic**;

Additional Key Words and Phrases: ACM proceedings, LaTeX, text tagging

ACM Reference Format:
Goran Flegar, Florian Scheidegger, Vedran Novaković, Giovani Mariani, Andrés E. Tomás, A. Cristiano I. Malossi, and Enrique S. Quintana-Ortí. 2019. FloatX: A C++ Library for Customized Floating-Point Arithmetic. *ACM Trans. Math. Softw.* 1, 1, Article 1 (January 2019), 23 pages. https://doi.org/10.1145/3368086

*IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Other product and service names might be trademarks of IBM or other companies.

Authors' addresses: Goran Flegar, Dept. d'Enginyeria i Ciència dels Computadors, Universitat Jaume I, Av. Vicent Sos Baynat, 12071, Castelló de la Plana, Spain, flegar@uji.es; Florian Scheidegger, IBM Research–Zurich , Säumerstrasse 4, CH–8803, Rüschlikon, Switzerland, eid@zurich.ibm.com; Vedran Novaković, Dept. d'Enginyeria i Ciència dels Computadors, Universitat Jaume I, Av. Vicent Sos Baynat, 12071, Castelló de la Plana, Spain, novakoni@uji.es; Giovani Mariani, IBM Research–Zurich, Säumerstrasse 4, CH–8803, Rüschlikon, Switzerland, ova@zurich.ibm.com; Andrés E. Tomás, Dept. d'Enginyeria i Ciència dels Computadors, Universitat Jaume I, Av. Vicent Sos Baynat, 12071, Castelló de la Plana, Spain, tomasan@uji.es; A. Cristiano I. Malossi, IBM Research–Zurich, Säumerstrasse 4, CH–8803, Rüschlikon, Switzerland, acm@zurich.ibm.com; Enrique S. Quintana-Ortí, Dept. d'Informàtica de Sistemes i Computadors, Universitat Politècnica de València, Camino de Vera s/n, 46022, València, Spain, quintana@disca.upv.es.

1 INTRODUCTION

In the last years, *transprecision computing* (TC), in which systems are specialized to deliver just the required precision for intermediate computations, has delivered important energy savings in comparison with the conservative principle of guaranteeing numerical precision of each elementary step of the process [Bekas et al. 2012; Ho et al. 2017; Malossi et al. 2018; Mittal 2016; Palem 2014; Palmer 2015; Xu et al. 2015]. A strong tail wind for TC comes from the deployment of low-power devices and the internet-of-things (IoT) in home appliances, healthcare, and transportation, among others. In particular, many of the embedded applications running on this type of power-constrained systems involve numerical computations that consume a considerable amount of energy to perform the actual computations and move data across the device memory hierarchy. However, the numerical requirements for many of these applications can be satisfied by using a significantly lower precision than the conventional double-precision (DP), single-precision (SP), or even half-precision (HP) floating-point formats, as defined by the IEEE 754 standard binary64, binary32, and binary16 [IEEE 2008]. At a different scale, convolutional neural networks (CNNs) and weather forecast simulations also benefit from more versatile floating-point formats [Gupta et al. 2015; Hill et al. 2018; Palmer 2014; Tobias [n. d.]].

While most current hardware architectures offer support for the conventional SP/DP formats only (and a few recent ones extend this to HP), the discussion in the opening paragraph hints the potential benefits of customized floating-point units (FPUs) that operate with non-conventional formats at reduced precision. This is especially the case as some of these applications are important enough to justify the development of application-specific integrated circuits (ASICs), particularly as we progress along the road towards the end of Moore's Law.

In this paper we present a flexible C++ framework, named FloatX (for Float eXtended), to investigate the effect of exploiting customized reduced-precision floating-point formats in numerical applications. Here, reduced or low precision means a type which does not have more significand or exponent bits than IEEE 754 double precision format. Some of the appealing properties of FloatX include:

Programming interface. FloatX is easy-to-use and minimally-intrusive in order to facilitate an incremental transformation of numerical applications.

Performance. FloatX relies on hardware-supported floating-point types as back end to preserve efficiency. Furthermore, it incurs no storage overhead by maintaining the size of the emulated datatype shorter than (or equal to) that of the back-end datatype.

Expected semantics. The arithmetic in FloatX adheres to the round-to-nearest, ties-to-even rule in the standard IEEE 754 whenever feasible, and the interoperability of variables follows the datatype casting convention in C++.

In more detail, our paper makes the following contributions:

- We discuss the design principles, decisions, programming interface and datatype casting rules underlying FloatX.
- We leverage several representative operations from BLAS (Basic Linear Algebra Subprograms) [Dongarra et al. 1990], LAPACK (Linear Algebra Package) [Anderson et al. 1999], and Ginkgo[1] (a modern C++ linear algebra library for the iterative solution of sparse linear systems) to expose the insights that we gained during the integration of these linear algebra (LA) packages[2] on top of FloatX.

[1]https://github.com/ginkgo-project/ginkgo
[2]The choice of LA is not arbitrary. On one hand, LA operations constitute fundamental building blocks appearing in many scientific and engineering applications, including the aforementioned CNNs and weather forecast modelling. On the other

- Some of the aforementioned LA cases are also employed to report the performance overhead due to FloatX's (emulated) arithmetic as well as to illustrate the integration of the FloatX framework in conjunction with multithreaded codes for multicore processors and kernels written in CUDA C++ for graphics processing units (GPUs).
- In addition, we abandon the classical comfort zone of linear algebra to demonstrate the potential benefits of FloatX by analysing the effect of customized floating-point precision on two neural network applications: an image generator code integrated with TensorFlow [Abadi et al. 2016]; and the Bidirectional Long Short-Term Memory (BLSTM) algorithm for optical character recognition.

The rest of the paper is structured as follows. In Section 2 we discuss the differences between FloatX and some related works. Then, in Section 3 we present FloatX's design principles, application programming interface (API), and several key properties such as performance overhead. In Section 4 we review an "easy" case: the porting of the C++ Ginkgo library to operate on FloatX. Next, in Sections 5 and 6 we address two more challenging cases posed by C and Fortran instances of BLAS and LAPACK. In Sections 7 and 8 we analyse the numerical performance of two neural networks applications running on top of FloatX. Finally, in Section 9, we close this paper with a summary and an outline of future research work.

2 RELATED WORK

There exist some packages to experiment with non-native floating-point precision. Among these, the GNU MPFR [Fousse et al. 2007] library[3] is a de-facto standard for *arbitrary* higher than regular precision, used for example in `gcc`. At this initial point, we highlight that FloatX aims to provide a software emulation tool to rapidly explore ideas by experimenting with *reduced-precision* floating-point formats within high-level, complex applications. This is in contrast with many other floating-point emulation tools (see, e.g., the survey in https://www.mpfr.org/) which, similarly to GNU MPFR, provide software support for *extended precision* and, therefore, serve a different purpose. In the following review we target floating-point emulation tools that focus on reduced floating-point precision.

FlexFloat [Tagliavini et al. 2018] is (internally) a C software library (enhanced with C++ wrappers) that enables exploration of numerical effects by tuning both precision and dynamic range of program variables. The purpose of FlexFloat is, therefore, very similar to that of our FloatX. Nonetheless, as we will discuss in the following section, FloatX presents several programming advantages, due to the adoption of C++ as the back-end framework language, that are difficult to attain with a solution based on C.

INTLAB (INTerval LABoratory) [Rump 1999] is a Matlab package that offers the `fl`-numbers, a concept similar to FloatX. From a freely available example[4] it can be concluded that `fl`-numbers have, at most, 26 significand bits (including the implied one); the maximal exponent range is $[-241, 242]$ (both lower than that in FloatX numbers); and global variables exist that control the numbers of significand and exponent bits in effect. That last trait makes INTLAB slightly less flexible than FloatX, where the precision is a property of the data itself.

The Sipe [Lefèvre 2013] C library[5] is a header-only tool for experimenting with floating-point algorithms and very low precision. It supports the correctly-rounded basic arithmetic operations, except divisions and square roots, but with fused-multiply-adds (`FMA`). Compared with FloatX, it

hand, LA is one of traditional areas where the effects of rounding errors and finite precision are better understood than, e.g., in the domain of deep learning, where empirical studies are necessary.

[3]Available at http://www.mpfr.org/ (version 4.0.1, February 2018).

[4]Available at http://www.ti3.tuhh.de/intlab/demos/html/dfl.html as of May 2018.

[5]Available at http://www.vinc17.net/research/sipe/ as of May 2018.

stores the number's value either in a native back-end floating-point type, or as a pair of two integers (for the non-normalized significand and the exponent parts, respectively), while the precision is a property of an operation on the data.

Precimonious [Rubio-González et al. 2013] is a program analysis tool based on LLVM that analyses floating-point program variables in an attempt to lower their precision. This tool recommends the smallest datatype for each variable that produces an accurate enough answer for a representative set of program inputs.

Unum and its new version *posit* [Gustafson and Yonemoto 2017] are variable-size alternatives to the IEEE 754 binary formats. FloatX also provides variable-size formats but those are based in the IEEE 754 standard and the number of bits is fixed prior to any computations. In contrast, the unum formats cloud grow automatically if required by the computation. Furthermore, there is no easy mapping between the unum and IEEE 754 formats.

Berkeley's SoftFloat [Hauser 2019] is a library that provides a floating point IEEE 754 implementation using only integer operations. This implementation is limited to the standard types plus the legacy 80-bit format from Intel/Motorola. In contrast, FloatX uses the floating point hardware to greatly reduce code size with respect to SoftFloat as well as to support any format smaller than the underlying base format. Also, FloatX leverages C++ overloaded operators to ease integration in existing programs.

3 THE FLOATX LIBRARY

3.1 Design goals and interface

In addition to implementing custom *low-precision* floating-point formats, an important design principle of FloatX is to provide an intuitive and easy-to-use interface, and whenever possible, to simplify the conversion of existing codes to use FloatX. Thus, the library is shaped by the following design goals:

(1) FloatX types should be an extension of the standard binary floating-point types (`float` and `double`), and their interface should be equivalent to them. This means that, for any two FloatX objects, `a` and `b`, the following expressions/operations should offer the expected semantics, as in the case of `float` and `double`:
    - `a + b`, `a / b`, ...
    - `a < b`, `a >= b`, ...
    - `a = b`, `a += b`, ...
(2) FloatX types should also be interoperable with built-in numeric types (signed and unsigned integers, built-in floating-point types) – the expressions above should be valid even if `a` and `b` are different FloatX types, or if one of them is a built-in type. This implies that FloatX types should support a set of implicit conversions compatible with standard numeric promotions and numeric conversions of built-in types.
(3) The size of a FloatX object should never be larger than the size of a `double`. This simplifies the porting of existing codes to operate on top of FloatX, as it is then possible to *embed* a FloatX value into the storage space originally used for a `double`. This ensures that parts of the code which just move or read data, but perform no floating-point operations, do not have to be modified.

Listing 1 provides a small example that demonstrates the features and interface of FloatX. The example demonstrates several properties in a small fragment of code, but was not conceived to perform any useful computation.

Lines 1, 2, and 7 show how `floatx` numbers can be constructed from built-in types (floating-point numbers and integers) and read from C++ streams. Lines 8 and 9 show how these objects are used

```
 1  flx::floatx<7, 12> a = 1.2; // 7 exponent bits, 12 sign. bits
 2  flx::floatx<7, 12> b = 3;   // 7 exponent bits, 12 sign. bits
 3  flx::floatx<10, 9> c;       // 10 exponent bits, 9 sign. bits
 4  float              d = 3.2;
 5  double             e = 5.2;
 6
 7  std::cin >> c;
 8  c        = a + b;           // decltype(a + b) == floatx<7, 12>
 9  bool t   = a < b;
10  a       += c;
11  d        = a / c;           // decltype(a / c) == floatx<10, 12>
12  e        = c - d;           // decltype(c - d) == floatx<10, 23>
13  c        = a * e;           // decltype(a * e) == floatx<11, 52>
14  auto f   = a + c;           // decltype(f) == floatx<10, 12>
15  std::cout << c;
```

Listing 1. Sample code using FloatX.

to perform basic arithmetic and relational operations. Lines 10–13 demonstrate the interoperability between different `floatx` and built-in types. The comments on the right specify the return type of the operation. (Note that `T == U`, where `T` and `U` are types, is used to convey that these two types are the same, i.e., that `std::is_same<T, U>::value` evaluates to true.) Lines 8 and 11–13 also show that `floatx` types can be implicitly converted to other `floatx` types or to built-in types, while Line 14 shows that the `floatx` types behave as expected when used in combination with the modern C++ `auto` keyword. Finally, Line 15 shows how `floatx` types can be written to an output stream.

3.2 The choice of C++

Given the restrictions imposed by the design goals stated at the beginning of this section, the language of choice needs to support a powerful datatype system, which enables programmers to define their own types and operators on those types. In addition, the language has to support some sort of “type arithmetic”, which is general enough to specify numeric promotion and implicit conversion rules for custom types. Furthermore, the design goal of supporting custom types that are (almost) as precise as `double` and fit into the same storage space as `double` means that custom types are not allowed to incur any memory overhead and that the information about the precision of the type cannot be maintained as additional data, but has to become a part of the type itself.

These requirements discard C as a possible candidate. In contrast, C++ fits the job perfectly as this programming language supports both user-defined types and operator overloading, and the abstractions built in C++ do not impose any memory overhead. C++ templates provide an easy means to incorporate precision information into the type as well as to define arbitrary conversion rules using this information. The latter is possible since C++ templates (are believed to) form a Turing-complete language [Veldhuizen 2003], evaluated during compilation. In addition, most C++ compilers are capable of inlining function calls and optimizing output expressions that can be evaluated at compile time, which improves the performance of FloatX types.

Applications using C and Fortran 77, languages of wide adoption in high performance scientific computing, can be ported to C++ (and thus, integrated with FloatX) with a reasonable programming effort. Since C++ maintains a good level of compatibility with C, applications written in the latter language can usually be compiled as C++ applications with minimal modifications. Fortran 77 applications (unlike more modern Fortran variants) can be translated into C using the tool `f2c` and then compiled with a C++ compiler.

3.3 The `floatx` class template

To achieve the third design goal specified in the introduction of this section, and to simplify realization of the first and second goals, different precisions in FloatX are implemented as distinct specializations of the `floatx` class template. The number of exponent and significand bits used in the format are encoded into the type via integral template parameters.

FloatX uses a hardware-supported floating-point type as *back end*, which is used to store the data, simplify the implementation, and improve performance of arithmetic and relational operations. The binary representation of any FloatX value is equivalent to the rounded and (whenever possible) normalized representation of the same value in the back-end type. Note that the set of all representable values in a back-end floating-point type `B`, with $S_{\mathtt{B}}$ significand bits and $E_{\mathtt{B}}$ exponent bits, is a superset of all representable values in any floating-point type with $S \leq S_{\mathtt{B}}$ significand bits and $E \leq E_{\mathtt{B}}$ exponent bits. Thus, `B` can be used as back end for any `floatx` type that is, at most, as precise as `B`. There are both positive and negative consequences of this approach.

On the positive side, a comparison of two FloatX values amounts to a comparison of the underlying back-end values, which yields a considerable performance benefit. The textual output is likewise straight-forward. In addition, all arithmetic operations are performed on those back-end values. The result is then rounded in accordance to the precision of the FloatX type and again stored (exactly) in the back-end type. Values such as $\pm\infty$ or NaN are directly stored in the back-end type without any conversion. For efficiency, floating point status flags are not supported by FloatX. Also, FloatX multiplication and addition are not fused in the current implementation, as this optimization is rarely worth the extra template metaprogramming effort.

All arithmetic operations are performed in the back-end type, using the corresponding machine's floating-point arithmetic instruction. The result of any operation is inevitably rounded back to that type, using the machine's rounding mode in effect (usually *round-to-nearest, ties-to-even*), before it is rounded again to the target FloatX type, what is called *double rounding* [Figueroa 1995; Martin-Dorel et al. 2013].

Double rounding could fail when the first step results in a value that is just at the same distance from two values in the reduced precision. In the following discussion X stands for an arbitrary bit and single/double arrows are reduced/high precision roundings respectively. One example of double rounding failure is when the bit addition carries all the way up to the last bit before the reduced significand:

$$\begin{tikzcd}
\mathrm{X}.\overbrace{\underbrace{\mathrm{X}\cdots\mathrm{X}1}_{p}\,01\cdots1}^{q}\,1\mathrm{X}\cdots\mathrm{X} \arrow[r, "\text{truncation}"] \arrow[d, Rightarrow, "\text{addition}"] & \mathrm{X}.\underbrace{\mathrm{X}\cdots\mathrm{X}1}_{p} \\
\mathrm{X}.\overbrace{\underbrace{\mathrm{X}\cdots\mathrm{X}1}_{p}\,10\cdots0}^{q} \arrow[r, "\text{addition}"] & \mathrm{X}.\underbrace{\mathrm{X}\cdots\mathrm{X}0}_{p}
\end{tikzcd}$$

An additional example of double rounding failure is when the original value is close to the middle point in high precision:

$$\begin{array}{lcl}
X.\overbrace{\underbrace{X\cdots X0}_{p}\,10\cdots 0}^{q}\,0X\cdots X & \longrightarrow & X.\underbrace{X\cdots X1}_{p} \\
\qquad\quad\Downarrow & & \\
X.\overbrace{\underbrace{X\cdots X0}_{p}\,10\cdots 0}^{q} & \longrightarrow & X.\underbrace{X\cdots X0}_{p}
\end{array}$$

In this case, at least one bit after $q + 1$ must be set in the original value. The same situation arises if the last bits after q in the original value are exactly $10 \dots 0$. In both examples the error using the double rounded value is

$$|a - \text{round}_p(\text{round}_q(a))| \leq \frac{\epsilon_p}{2} + \frac{\epsilon_q}{2},$$

which is approximately the expected error for a significand of p bits, that is $\epsilon_p/2$, when $p \ll q$. This should be the typical case for FloatX as, in most cases, we expect the user is interested in formats which are "smaller" than single precision. Formats close to double precision (more than 44 significand bits) are not quite as interesting because they will provide much lower benefits from the hardware point of view.

The correct result without double rounding can be obtained by exploiting only back-end arithmetic [Rump 2016]. Also, the iterative correction proposed in [Ziv 1991] could be used for any elementary mathematical function. FloatX does not employ either approach because both have a large overhead not worth the extra accuracy. An alternative way to avoid double rounding is to employ round-to-odd [Boldo and Melquiond 2008]. This rounding mode is very efficient and without bias, but the maximum error is the same as with truncation, and twice as large as for round-to-nearest. FloatX does not implement round-to-odd because is not in the IEEE 754 standard.

The rounding routine currently integrated in FloatX has a limitation in that it only supports `double` as the back-end type, and only "*round to nearest, ties to even*" rounding mode. A more general rounding routine with support for different rounding modes and back-end types is planned in the future.

Listing 2 shows the outline of the `floatx` class template and demonstrates how the desired binary representation is achieved. Listing 3 is a verbatim copy of FloatX's top-level rounding routine, which demonstrates the steps required to transform a value of the back-end type into a value of FloatX type.

The rounding process constructs the new value by converting the exponent and mantissa from the original value (the sign is unchanged). No conversion is required for Infinity and NaN values, nor if the FloatX type is configured with the same parameters as those of the back-end type.

The first step in the rounding process is to extract the bits for the mantissa, exponent, and sign using bitwise operations. This process is more complicated for the exponent because of the bias. Next, each part of the number is fitted into the target format. If the exponent is too small, the number will be denormalized; if it is too large, the result will be Infinity. The mantissa is rounded to the nearest value and corrected, together with the exponent, if it is too large. Finally, the rounded value is constructed from the original sign and the revised mantissa and exponent.

3.4 Operations on `floatx` objects

As already noted in Subsection 3.3, all operations on `floatx` values are trivially implemented in terms of operations on the back-end datatype and the rounding routine. In consequence:

```
template <int exp_bits, int sig_bits,
          typename backend_float = double>
class floatx {
private:
    backend_float data;
};
```

Listing 2. Basic structure of `floatx`.

```
backend_float enforce_rounding(backend_float value) noexcept {
  const auto exp_bits = get_exp_bits(self());
  const auto sig_bits = get_sig_bits(self());

  if (exp_bits == float_traits<backend_float>::exp_bits &&
      sig_bits == float_traits<backend_float>::sig_bits)
    return value;

  bits_type bits = reinterpret_as_bits(value);
  auto sig       = (bits & backend_sig_mask) >> backend_sig_pos;
  auto raw_exp   = bits & backend_exp_mask;
  const auto sgn = bits & backend_sgn_mask;

  int exp        = (raw_exp >> backend_exp_pos) - backend_bias;
  const int emax = (1 << (exp_bits - 1)) - 1;
  const int emin = 1 - emax;

  if (!is_nan_or_inf(bits))
  {
    if (is_small(exp, emin))
      convert_subnormal_mantissa_and_exp(bits, sig_bits, emin, exp,
                                         sig, raw_exp);
    else
      sig  = round_nearest(sig, backend_sig_bits - sig_bits);
    if (significand_is_out_of_range(sig))
      fix_too_large_mantissa(sig_bits, exp, sig, raw_exp);
    if (exponent_is_out_of_range(exp, emax))
      bits = assemble_inf_number(sgn);
    else
      bits = assemble_regular_number(sgn, sig, raw_exp);
  }
  return reinterpret_bits_as<backend_float>(bits);
}
```

Listing 3. Main FloatX rounding routine.

- Comparing two `floatx` values is equivalent to comparing their representations in the back-end type.
- Any arithmetic operation on two `floatx` values is equivalent (up to problems with double rounding) to the equivalent arithmetic operation on their back-end representations, followed by a rounding of the result.
- Printing a `floatx` value to a stream is equivalent to printing its back-end representation.
- Reading a `floatx` value from a stream is equivalent to reading it as a back-end value and rounding the result.
- Converting a numeric type into a `floatx` type is equivalent to converting it into the back-end type and rounding the result.
- Converting a `floatx` type into a numeric type is equivalent to converting its back-end representation into that numeric type.
- Converting between two `floatx` types is equivalent to re-rounding the back-end representation.

As a result, the non-trivial part of FloatX remains hidden in the implementation, exposing an intuitive interface to the user which makes FloatX (semantics) behave as expected and supports a flat learning curve.

Creating `floatx` objects and casting between `floatx` objects (as well as between built-in and other `floatx` objects) should be as easy as creating and casting between built-in objects. In order to attain this, FloatX defines a set of converting constructors for creating `floatx` objects and casting to `floatx` objects, as well as a set of conversion operators for casting from `floatx` objects to built-in types.

To support arithmetic and relational operations using the same syntax as that of built-in types, the library provides a complete set of arithmetic and relational operator overloads for `floatx` numbers. In addition, stream input and output operator overloads are also provided to simplify writing and reading `floatx` objects to and from C++ streams.

Supporting interoperability between distinct types (i.e., accommodating operations involving operands of different types) is more difficult. This requires extending the standard numeric promotion and conversion rules to include `floatx` numbers. The C++ definition of these rules relies on the concept of a *common type*, a rigorous definition of which, including the set of implicit conversions and promotions, can be found in the C++ standard [ISO 2017].

In short, the common type for two floating-point types `F` and `G` is the more precise of the two, and the common type of an integral type `I` and a floating-point type `F` is the floating-point type `F`. All binary operations are performed by first converting both operands to their common type, and then performing the operation with converted values. The result of the operation is of the common type. In addition, if the integer being converted is out of the representation range of the common type (this can happen when converting a large integer to a low-precision floating-point type), the result of the operation is unspecified.

Applying these rules literally to FloatX is impossible, since there are pairs of `floatx` types such that neither of them can be considered as more precise (e.g., `floatx<7, 9>` and `floatx<10, 6>`). Thus, FloatX has to extend these rules in a way which does not modify their definition for standard types, while maintaining the desirable properties of those types for FloatX's emulated types. Some of the properties that we consider crucial are the following:

(1) The common type of two operands of the same type T is type T itself; that is, $\mathsf{common\_type}(\mathsf{T}, \mathsf{T}) = \mathsf{T}$.
(2) For any two floating-point types R and S, and objects of those types a and b, respectively, consider the statement `c = a + b;` (equivalently, any basic arithmetic operation). If c is of type $\mathsf{T} \in \{\mathsf{R}, \mathsf{S}\}$, the final result stored in c is equivalent (up to effect of double rounding) to the infinitely-precise result of $\mathsf{a} + \mathsf{b}$, properly rounded to type T.
(3) For any two floating-point types R and S, $\mathsf{common\_type}(\mathsf{R}, \mathsf{S})$ is the smallest type which has at least as many exponent and significand bits as both R and S.

To remove ambiguities with the standard specification of a common type, FloatX uses the following definition.

Definition 1 (Common type for `floatx`). *Let E_S and E_T denote the number of bits reserved for the biased representation of the exponent in the floating-point types* S *and* T, *and let M_S and M_T stand for the number of bits in the significand (mantissa) of the same types.*
For two floating-point types, S *and* T, *FloatX defines* `common_type(S,T)` *as a type with* $\max\{E_\mathsf{S}, E_\mathsf{T}\}$ *bits reserved for the exponent and* $\max\{M_\mathsf{S}, M_\mathsf{T}\}$ *bits for the significand.*
For a floating-point type S *and an integral type* I, `common_type(S,I)` = S.

Note that this definition satisfies all of the above mentioned desirable properties and keeps the original semantics of a common type unchanged for the built-in types. Since the FloatX definition extends the original C++ definition, FloatX only needs to implement the extensions for operations where at least one of the operands is of a type provided by FloatX. This is done by providing more general operator overloads which can take any operands, as long as at least one of them is a FloatX-provided type, and then convert them into their common type.

The common type is determined at compile time, using a combination of meta-programming techniques including: trait classes for built-in and FloatX-provided types, SFINAE (Substitution Failure Is Not An Error) [ISO 2017], and the `std::enable_if` standard C++ utility. For implementation details, refer to the FloatX source code[6]. The common type resulting from such an overload is always a FloatX-provided type with the significand and exponent bits set to the values specified as in Definition 1. The back-end type is set to the common type of back-end types of the operands (if one of the operands is a built-in type `T`, its back-end type is considered to also be `T`).

3.5 The `floatxr` class template

One downside of `floatx` is that the "range" and "precision" of a type need to be known at compile time. Since the goal of `floatx` is to ease and accelerate the experimentation with low precision, this limit can sometimes be too restrictive. There are applications (e.g., Jacobi linear solvers [Anzt et al. 2015]) for which it is interesting to start with a low precision and increase the number of bits in small steps as the algorithm progresses. In this case, having a different type for each step will make the actual implementation very complex. In addition, the intent "*to evaluate an algorithm with all significand sizes between S_1 and S_2, and return the optimal one*" is difficult to express, as it would either require the user to recompile the code for each size or use template meta-programming to write the compile-time equivalent of a loop.

For this reason we introduce another class template, `floatxr`, which allows the user to set the precision at runtime. Listing 4 displays the data layout of this type.

```
template <typename backend_float = double,
          typename metadata_type = unsigned int>
class floatxr {
private:
    backed_float data;
    metadata_type exp_bits;
    metadata_type sig_bits;
};
```

Listing 4. Basic structure of `floatxr`.

Unfortunately, specifying the precision at runtime comes at the cost of increased memory footprint, and as such, `floatxr` does not fulfill the third design goal specified in the beginning of this section.

On the positive side, `floatxr` and `floatx` can be implemented to use the same code base, so there is only one rounding routine and one set of operators to maintain (and the compiler just optimizes the same generic function templates more efficiently when instantiating them for `floatx` than for `floatxr`). Also, `floatxr` types support the same operations and conversions as `floatx` types, and interoperate with `floatx` and built-in types. As stated in the following definition, the latter requires a minor revision of the common type, since the precision of `floatxr` is not known at compile-time.

Definition 2 (Common type for `floatxr`). *For any two types* `S` *and* `T`*:*

- *If at least one of* `S` *and* `T` *is a* `floatxr` *type, their common type is a* `floatxr` *type whose back-end type is the common type of* `S` *and* `T`*'s back-end types.*
- *Otherwise, the common type is deduced as specified in Definition 1.*

[6] Available at https://github.com/oprecomp/FloatX

3.6 Notes on concurrency

FloatX variables maintain no state (e.g., there are no routines for setting or global variables holding the *current* working precision, since this is either a property of the types involved in an operation, and/or a property of the data). The rounding operation itself also depends only on the back-end and the destination types' properties.

An important advantage of the stateless `floatx` compared with stateful `floatxr`, is that the framework can be safely used from many concurrent threads of execution, in a sense that no arithmetic or relational operation on `floatx` variables involves accessing (let alone changing) any other non-constant data. The threads in question can either be operating system threads on a CPU or CUDA threads on a GPU. It should also be noted that the rounding operation and the functions employed in FloatX consist of the same code in both the CPU and the GPU cases, up to certain integer compiler intrinsics and the necessary CUDA device function attributes.

Since changing the precision of a `floatxr` variable at run-time requires modifying its two metadata components and re-rounding the value in the back-end component, neither read nor write access to the variable from another thread should be allowed while the operation is in progress. In general, there are no special guarantees of atomicity for any FloatX type or operation. For example, an assignment of a `floatx` variable `b` to `a` involves: creating a temporary `floatx` with the back-end value set to the one of `b`, re-rounding of that value in-place, and finally replacing `a`'s back-end component with the temporary's (up to any compiler optimizations that might be applied). The users should therefore rely on standard mutual exclusion primitives to avoid data races during the concurrent access to FloatX variables of any type.

3.7 Advanced properties and performance of FloatX

An additional advantage of the stateless FloatX is that the memory size and the alignment requirements are both identical to those of the back-end type. Thus, in certain read-only scenarios, an array of FloatX variables can be directly passed, with just a pointer typecast, to a routine expecting an array of the back-end type. A use case might be a pretty-printer routine, or a writer routine for a custom file format, possibly written in a different language.

The easiest way to compute the mathematical functions for FloatX is to obtain the result using the back-end function, and round the result into a target FloatX type. Of course, that could sporadically introduce results which are not correctly rounded, due to double rounding, where the routine itself is expected to be correctly rounded (e.g., `sqrt`).

When representing a native floating-point type as a FloatX type (in particular, `floatx<11, 52>` represents `double`, and `floatx<8, 23>` is equivalent to `float`), the final rounding operation is unnecessary (no-op) when that machine's type is equivalent to the back-end one, or can otherwise be simplified by employing two native datatype conversions. In both cases, the simplification can be triggered at compile time. The former case yields arithmetic performance close or equivalent (zero-overhead achieved by code inlining) to using the native back-end type directly, including the possibility of automatic vectorization and other optimizations not generally applicable to FloatX types. The latter case has not yet been implemented.

Overhead. It is evident from the control flow in Listing 3 that the complexity of rounding depends on the value being rounded as well as on the type constraints. Therefore, the performance of the code is inherently dependent on the data itself. For example, the rounding operation from Listing 3, fully optimized and inlined after an addition `a + b`, with the variables declared as `floatx<7, 12> a` and `floatx<10, 9> b`, requires around 80 assembly instructions on an Intel Haswell architecture with the `gcc` 7.2.1 C++ compiler. However, not all of those instructions are executed in each rounding

<5,10>	<8,23>*	<10,13>	<11,44>	<11,52>**	float*	double**
128.8	132.6	127.7	149.4	1714.8	1999.1	1715.2

Table 1. MFLOP/s for DOT BLAS-1 operation with various FloatX and native types. Numerically-equivalent datatypes are identified using the same positive amount of asterisks in the superscript.

<8,23>*	<9,14>	<10,13>	<11,28>	<11,52>**	float*	double**
128.4	125.1	125.7	149.8	2316.8	5751.0	2652.0

Table 2. MFLOP/s for GEMM BLAS-3 operation with various FloatX and native types. Numerically-equivalent datatypes are identified using the same positive amount of asterisks in the superscript.

operation, due to possibly different code paths taken in each case. Furthermore, the alternative code paths make code vectorization almost impossible.

Our next experiments provide an evaluation of the practical overhead introduced by FloatX when using non-native datatypes, using the routines for the legacy [7] BLAS-1 `DOT` and BLAS-3 `GEMM` converted to C++ and integrated with FloatX.

The *testing machine* for all experiments performed in this paper (except when the target is a GPU) was an Intel Xeon E5-2630 v3 (Haswell) system, running at 2.40 GHz, with the `gcc` 7.3.0 C++ compiler and a 64-bit GNU/Linux.

Table 1 summarizes the performance of the sequential `DOT` routine, with two vectors of $n = 10^9$ pseudo-random elements in $(-1, 1)$ of a particular type each. The millions of floating-point operations per second (MFLOP/s) is displayed for each type specified in the header ("`floatx`" prefix was omitted for brevity). The conclusion from this experiment is that all `floatx` types exhibit a similar (but not the same) performance hit compared with the back-end type, except when equivalent to it and the optimization described above is in effect. However, when the exponent range is the same as the one of the back-end type (11 bits for `double`), the rounding procedure from Listing 3 is faster than in the other cases.

Table 2 summarizes the overhead observed with the BLAS-3 `GEMM` multiplication of the form $C = 2AB - C$, when using FloatX versus the native datatypes in the reference implementation of this routine in BLAS converted to C++. The entries of matrices A, B and C, of order $n = 3000$, were randomly generated. The computation consists of multiplications and subtractions, for a total of $2n^3 + O(n^2)$ floating-point operations. The choice of types displayed in Table 2 covers the minimal and maximal observed performance (125.1 and 149.8 MFLOP/s, respectively) within a test subset of 20 "proper" (i.e., not particularly optimized, like `floatx<11, 52>`) FloatX types, confirming further that the complexity of the rounding operation varies with the datatype of the parameters as well as with the data values themselves.

4 THE GINKGO FRAMEWORK

Our first example illustrates the effort to integrate a C++ framework, such as Ginkgo, with FloatX. Given that the programming language for Ginkgo matches that of FloatX, and the advanced features of Ginkgo, we can naturally expect that this integration is smoother than that of BLAS and LAPACK (to be discussed in the next two sections).

[7] Available at http://www.netlib.org/blas.

Ginkgo is a single-node high performance linear algebra library, designed to transparently support heterogeneous systems consisting of various devices. It currently contains a single-threaded reference module for CPUs and a highly optimized CUDA module for NVIDIA GPUs. An OpenMP-based multithreaded module for CPUs is under development. The library is designed as a linear operator algebra, primarily focusing on the iterative solution of sparse linear systems of equations. However, other linear transformations also fit the Ginkgo framework.

One of the main focuses of Ginkgo is extensibility, making it an ideal first candidate for FloatX. Even more so, since in addition to supporting user-defined sparse building blocks (matrix formats, preconditioners, solvers, stopping criteria, etc.), Ginkgo enables easy integration of user-defined arithmetic types. In fact, integrating a FloatX type only required changing 5 lines in Ginkgo's source code. Once this was done the type could be used in the same way as a native type could.

```
 1  #include <ginkgo/ginkgo.hpp>
 2  #include <floatx.hpp>
 3  #include <iostream>
 4
 5  int main() {
 6      using T = flx::floatx<7, 16>;
 7      auto gpu = gko::CudaExecutor::create(0, gko::OmpExecutor::create());
 8      auto A = gko::read<gko::matrix::Ell<T>>(std::cin, gpu);
 9      auto b = gko::read<gko::matrix::Dense<T>>(std::cin, gpu);
10      auto x = gko::read<gko::matrix::Dense<T>>(std::cin, gpu);
11      auto solver =
12          gko::solver::Cg<T>::build()
13              .with_criteria(
14                  gko::stop::Iteration::build().with_max_iters(20u).on(gpu),
15                  gko::stop::ResidualNormReduction<T>::build()
16                      .with_reduction_factor(T{1e-15})
17                      .on(gpu))
18              .on(gpu);
19      solver->generate(give(A))->apply(lend(b), lend(x));
20      write(std::cout, lend(x));
21  }
```

Listing 5. An example of using `floatx<7,16>` in combination with the Ginkgo library.

Our tests with Ginkgo+FloatX included the compilation of the framework with support for several custom FloatX types and a few of the examples included with Ginkgo, modified to solve a sparse linear system using the Conjugate Gradient (CG) method with a matrix stored in ELLPACK format. The purpose of our tests with Ginkgo was not to validate FloatX itself, but to gain some initial experiences on the type of caveats and problems of such integration process. In general, the experience was quite satisfactory, and the CG method was run both on one core of the Intel Haswell CPU and an NVIDIA P100 GPU connected to the server.

As an example, in Listing 5 we display the complete source code needed to iteratively solve a sparse linear system using Ginkgo with FloatX. Line 7 defines the execution space that will be used to run all subsequent operations. In this case, it is the CUDA device with ID 0. Next, in Lines 8–10, a system matrix `A`, right-hand side vector `b` and an initial guess `x` are read from the standard input and saved in Ginkgo's ELLPACK matrix format and dense vector formats that use the specified FloatX types to store the numeric values of the matrix. Then, a CG solver is constructed from the system data in Lines 11–18. The system is solved in Line 19, and the solution written to the standard output in Line 20. All in all, Ginkgo + FloatX is a good example where both libraries are designed with standardized interfaces and are compatible with one another. As a result, there is virtually no difference for the user when Ginkgo is used with FloatX, compared to using Ginkgo with built-in types.

5 BLAS

5.1 Legacy implementation

The legacy implementation of the BLAS API[8] is written in Fortran, though there exist C and C++ wrappers which preserve established array storage and interface conventions while offering some of the advanced features of more modern languages. The first question to obtain an implementation of (the legacy) BLAS that operates with custom types is how to combine a library like (the reference implementation of) BLAS, which is strongly rooted in the old Fortran 77, with a C++ framework such as FloatX. The inauspicious answer is that this is not straight-forward.

Fortunately, the contents of the legacy BLAS can be automatically transformed into plain C routines via a tool such as `f2c` without too much trouble. However, when the goal is to combine the result of `f2c` with FloatX, there appear some caveats worthy of being discussed:

- The `f2c` converter produces a plain C code, which needs to be "genericized" by templating the routines with a type parameter, named after the appropriate `f2c`'s typedef: e.g., `real` or `doublereal`, for single or double precision routines, respectively (our implementation is a generic version of the latter). Here, only the floating-point type needs to become generic while the character, logical, and integer types may be left as `typedef`s.
- In addition, the `f2c`-processed code depends on the `f2c.h` header definitions and the `libf2c` routine library. The header file introduces a couple of macros (e.g., for `abs` and `min`/`max`), for which the standard C++ generic functions would be more appropriate.
- Moreover, `f2c` sometimes declares global static variables, which need to be moved into the translated routines, to get them in scope of the template's typename by which they will be declared.
- The calls to non-generic mathematical functions have to be fixed as well.
- BLAS by itself does not use the Fortran I/O subsystem, and therefore the translated code can be made self-contained, i.e., not dependent on `libf2c`. The only exception is the `XERBLA` routine, which is simple enough to be manually encoded in C++.

Having done so, all of the BLAS has been "genericized" to the point that it becomes a header-only library; that is, no code is compiled into separate object files (except for testing purposes). All routines that are needed by a particular application are instantiated when and where needed, with the appropriate types used in each call (e.g., the native or the FloatX types). This type of approach increases the compile time of the applications noticeably, but we believe that the flexibility of the result outweighs that drawback. For reference, Listing 6 shows examples of generic C++ prototypes for a real `AXPY`[9] and a real `GEMM`.[10]

```
template <typename doublereal>
void daxpy_(integer *n,
            doublereal *da,
            doublereal *dx, integer *incx,
            doublereal *dy, integer *incy);

template <typename doublereal>
void dgemm_(char *transa, char *transb,
            integer *m, integer *n, integer *k,
            doublereal *alpha, doublereal *a, integer *lda,
                               doublereal *b, integer *ldb,
            doublereal *beta,  doublereal *c, integer *ldc,
            ftnlen transa_len, ftnlen transb_len);
```

Listing 6. Generic prototypes for some real BLAS routines.

[8]Available at http://www.netlib.org/blas (version 3.8.0, November 2017).

[9]An update of a vector by another scaled vector, $\mathbf{y} = \mathbf{y} + \alpha\mathbf{x}$.

[10]Matrix multiplication, $\mathbf{C} = \beta\mathbf{C} + \alpha\mathbf{A}^{\mathrm{op}_\mathbf{A}}\mathbf{B}^{\mathrm{op}_\mathbf{B}}$, with $\mathrm{op}_\mathbf{X}$ being an optional transpose (with or without conjugate) of $\mathbf{X}$.

At this point, it is worth mentioning that, in comparison with a recent effort to produce a generic C++ API for BLAS (and LAPACK) [Gates et al. 2017], our "FloatX-ed" routines operate only on a single datatype. In comparison, the proposed C++ BLAS API implementation[11] allows mixing the datatypes in a single call (e.g., an `AXPY` call is "template-ized" with two type parameters, one for vector **x** and another for **y**, while the type of the scalar a is deduced from the two). However, offering a generic BLAS interface was only a secondary goal to us, needed to demonstrate the feasibility of using FloatX with ubiquitous scientific computing kernels.

5.2 Optimized implementations

The experiment in Table 1 exposed the overhead incurred when using software-emulated datatypes instead of their hardware-native counterparts within the legacy BLAS. In practice, numerical applications do not rely on such a plain implementation of the BLAS but on an optimized, hardware-specific counterpart. These high performance instances of BLAS heavily rely on vector `FMA` instructions and inline cache prefetching, at the same time exerting a strong control of data movements across the memory hierarchy to amortize the cost of communication. Some examples of vendor-optimized implementations of the BLAS are included in Intel MKL [Intel 2015], IBM ESSL [IBM 2015] and NVIDIA cuBLAS [NVIDIA 2016]. Unfortunately, all these software packages are black-boxes and their contents cannot be inspected nor modified. Open instances of BLAS that are often competitive in performance include OpenBLAS [OpenBLAS 2015], ATLAS [Whaley and Dongarra 1998] and BLIS [Van Zee and van de Geijn 2015].

Among the previous instances of the BLAS, BLIS (*BLAS-like Library Instantiation Software Framework*) is especially appealing because it drastically reduces the amount of code that needs to be manually optimized for a given architecture. For example, BLIS encodes the `GEMM` kernel as five plain loops around two simple packing routines and a micro-kernel. All code in BLIS is written in C, while only for highly optimized for architecture-specific implementations of BLIS, the micro-kernel is usually written in assembly or with vector intrinsics. Furthermore, BLIS includes OpenMP pragmas to deliver a multi-threaded execution of the routines on a multicore processor; see [Van Zee and van de Geijn 2015] for details.

An interesting question in this case is whether we can migrate a C framework such as BLIS, parallelized with OpenMP, to operate on top of FloatX and still benefit from some of the performance optimizations integrated in the former. In particular, given the implementation of BLIS, we can expect to maintain the tight control over the data movements as well as to take advantage of the multi-threading capabilities of BLIS/OpenMP on a multicore processor. However, is not possible to profit from the optimized implementation of the micro-kernel, which depends on the target architecture and includes vector FMA instructions. The conclusion is that any instance of BLAS, even a cache-aware one such as BLIS, operating on top of FloatX, will offer low performance due to the lack of support at the micro-kernel level.

6 LAPACK

To further illustrate our experience with FloatX, in this section we discuss the integration of the custom floating-point framework with the solver for general linear systems (via the LU factorization) and with the singular value decomposition (SVD), both from LAPACK[12] [Anderson et al. 1999]. The solution of (dense) linear systems was chosen because it is one of the classical examples of a numerical method that can be adapted to combine an inner reduced-precision (RP) solver with an outer extended-precision (EP) iterative refinement [Barrachina et al. 2008; Buttari et al. 2007;

[11] Available at https://bitbucket.org/icl/blaspp as of May 2018.
[12] Available at http://www.netlib.org/lapack (version 3.8.0, November 2017).

Higham 2002]. With FloatX, one can then investigate customized datatypes for the inner RP solver, which can for example feature a range of representation (i.e., number of bits in the exponent) as wide as that of DP (to avoid underflow/overflow problems), but a constrained precision (i.e., number of bits in the significand) similar to that of HP. The SVD was selected because it has been described as the "Swiss Army knife of matrix decompositions", due to its many applications [O'Leary 2006], and also for being representative of the type of unitary/orthogonal one-sided and two-sided factorizations for (dense) linear least squares problems and the solution of (dense) eigenvalue problems.

Migration. Porting a routine from LAPACK to C++ using `f2c` shares many of the issues discussed for the BLAS in the previous section. An additional obstacle in the case of LAPACK are the operations on character strings (e.g., concatenation of two strings), which get translated into calls to the `libf2c` routines, and have to be re-coded manually to avoid reliance on that library. The same is true for the power function (`x**y`, i.e., x^y) and the logarithms, which are used in the reference LAPACK code to compute certain numerical bounds in a portable fashion, but are not essential to the algorithms themselves.

A hidden stumbling point that one should be aware of when using LAPACK routines that require a floating-point workspace and allow for querying its optimal size is that the query result (intrinsically an integer) is returned as a floating-point number in the first element of the workspace array. Should that floating-point type be too narrow to exactly represent the integer value, all sorts of issues can and will arise with the large matrix dimensions and especially with the workspace of a quadratic size in terms of them. If the integer to floating-point conversion is done via truncation instead of rounding towards $+\infty$, a too-small workspace length can be returned; otherwise, it may become unnecessarily large. In the worst case, the length can even be a useless ∞. That is however a design decision of LAPACK that cannot be changed without breaking its API.

Furthermore, the LAPACK codes evolve faster than the underlying BLAS routines, with a tendency to introduce the more modern Fortran language constructs. Unfortunately, some of these constructs are beyond those recognized by `f2c`, such as the `EXIT` statement for exiting the innermost enclosing loop, or the newer Fortran intrinsics. Those cases have to be re-written, either by "downdating" the Fortran sources, or by providing an alternative implementation in C++.

Routine `LAMCH`. The main non-straightforward part in the conversion of LAPACK to C++ is implementing a generic `LAMCH` function. This routine is expected to return certain parameters derived from the floating-point type's limits and representation, such as the type's ε (the smallest positive number to satisfy $1+\varepsilon \neq 1$ in a selected rounding mode), the smallest and largest normalized positive number of the type, the type's radix (providing support for non-radix-2 arithmetic), etc. For the native types, those quantities can be taken or easily computed from the values provided by the standard C++ template class `std::numeric_limits`, but for FloatX types they have to be deduced from the type's exponent and significand widths, as discussed next.

Listing 7 shows how to derive the quantities for FloatX types. These expressions are valid for any floating-point type `T` with radix 2, one bit for the sign, a biased exponent, and a significand that includes an implicit bit for the normalized numbers. Here we assume that the range of values of `T` is a subset of the range of values of some (here, back-end) type `B` with which it is easy to manipulate.

Figure 1 explains the meaning of the mnemonics for the `LAMCH` quantities, and offers concrete values for the example `floatx<5, 10>` (equivalent to a 16-bit HP type). The quantities that are derived from the basic ones do not have a mnemonic, while our extension `t` of the standard quantities serves to simplify a bound required in the `LARTG` routine for computing the Givens rotations.

```
template <typename T>
typename std::enable_if<(flx::float_traits<T,void>::is_floatx &&
    !flx::float_traits<T,void>::is_runtime),T>::type
dlamch_(const char *const cmach, ftnlen cmach_len) {
  using B = typename flx::float_traits<T>::backend_float;

  static const int E = flx::float_traits<T,void>::exp_bits;
  static const int M = flx::float_traits<T,void>::sig_bits;
  static const int bias = (1 << (E - 1)) - 1;
  static const int tinyE = 1 - bias;
  static const int hugeE = ((1 << E) - 2) - bias;

  T ret = 0;
  switch (toupper(cmach[0])) {
  case 'E': ret = std::scalbn(B(1.0), -(M + 1)); break; // 2\^{}-(M+1)
  case 'S': ret = std::scalbn(B(1.0), tinyE); break; // 2^tinyE
  case 'B': ret = FLT_RADIX; break; // assumed to be 2
  case 'P': ret = std::scalbn(B(1.0), -M); break; // 2^-M
  case 'N': ret = (M + 1); break;
  case 'R': ret = 1; break; // always rounding-to-nearest
  case 'M': ret = tinyE + 1; break;
  case 'U': ret = std::scalbn(B(1.0), tinyE); break; // 2^tinyE
  case 'L': ret = hugeE + 1; break;
  case 'O':
// fma(x,y,z) = x*y + z; here, x*y alone could overflow, but fma()
// comes to the rescue, since only one rounding should take place.
    ret = std::fma(std::scalbn(B(1.0), hugeE), B(2.0),
        -std::scalbn(B(1.0), (hugeE - M))); break;
  case 'T': // non-standard extension for SAFMN2 from xLARTG
    ret = std::scalbn(B(1.0), (tinyE + (M + 1))/2); break;
  default: // non-standard return -0 to signify undefined result
    ret = -ret;
  }
  return ret;
}
```

Listing 7. LAMCH for FloatX.

```
E: Epsilon                 = 0.000488281  |  M: Minimum exponent    = -13
S: Safe minimum            = 6.10352e-05  |  U: Underflow threshold = 6.10352e-05
B: Base                    = 2            |  L: Largest exponent    = 16
P: Precision               = 0.000976562  |  O: Overflow threshold  = 65504
N: # of digits in mantissa = 11           |     1 / safe minimum    = 16384
R: Rounding mode           = 1            |  t: SAFMN2 for xLARTG   = 0.5
```

Fig. 1. An example of the output from LAMCH for `floatx<5, 10>`.

For example, the machine precision ε (mnemonic E) is computed as $2^{-(M+1)}$, since LAPACK uses ε for the maximum rounding error. In rounding-to-nearest, this is half the distance 2^{-M} between the number 1.0 and the next one exactly representable in T.

Another interesting LAMCH quantity is the largest finite representable number (mnemonic O). This value is computed by taking 2 to a power equal to the largest exponent available for the finite numbers, multiplying it by 2 (which would lead to overflow if not done carefully and the exponent range is the same for T and its back-end type), and subtracting the difference between that and the required value, going back into the finite range. The FMA operation behaves as if there was no rounding taking place after the intermediate multiplication.

Note that the FMA operation cannot be autogenerated for FloatX by a C++ compiler, so all testing comparisons have been done by disabling the automatic generation of such machine instructions for the native types as well (with `-mno-fma` compiler flag in GNU's gcc), but the `fma` function from the standard library is available where explicitly needed, as is the case in the computation above.

7 GENERATION OF SYNTHETIC IMAGES USING REDUCED PRECISION IN GANS

The next two sections aim to demonstrate the possibilities and benefits of leveraging FloatX from practical applications. We integrate an arithmetic-consistent casting operator into the Tensor-Flow [Abadi et al. 2016] framework, and use different number formats to generate synthetic images with the generator of a Generative Adversarial Network (GAN) [Goodfellow et al. 2014]. The generator receives a random latent vector as input and is trained to generate synthetic data similar to the images stemming from the training data.

Neural network and numerical analysis setup. In this demonstration, we study the inference phase of the state-of-the-art generator network used in DCGAN [Radford et al. 2015] and reused in BAGAN [Mariani et al. 2018] trained on the CelebA [Liu et al. 2015] dataset. The generator produces a 64×64-pixel synthetic face image based on a 100-dimensional random latent vector. We study the quality of the generated image depending on the reduced precision floating-point format used to communicate information between all compute operations within the generator network. For the evaluation, we implemented a cast layer that maps IEEE 754 floating point values to reduced floating-point datatypes using FloatX, and maps them back to IEEE 754 floating point values. In this study, we quantized weights, inputs and outputs to the same global format to compress the model and to compress the amount of data that is required to flow from one compute operation to the next.

Numerical results. In the following $T_{w,t}$ denotes a FloatX type `floatx<`w`,` t`>` used to generate low precision computing pipeline variants from the same randomly generated latent vectors. Figure 2 shows the obtained image quality in terms of the PSNR (peak signal-to-noise ratio) between the approximated and the reference generated image. We observe a sharp change in quality when the number of bits used to encode the exponent exceeds 3. Using too few exponent bits fails to produce real-looking images, since the dynamic range of the compute domain is too restricted. In contrast, exceeding the dynamic range does not alter the results significantly. For a large enough exponent, adding more mantissa bits gives a smooth numerical improvement of the average PSNR value over the full range. Figure 3 shows faces generated with reference precision and with low precision number formats. As known, images exceeding a PSNR value of around 50 dB are visually equivalent. Number formats as small as $T_{4,1}$, consisting only of four exponent bits, one mantissa bit, and one sign bit (for a total of 6 bits) generate visually equivalent faces as the IEEE 754 32-bit floating point baseline. With this reduced type, all weights of the model can be compressed by a ratio of 5.3×when storing all values with 6 bits instead of 32 bits.

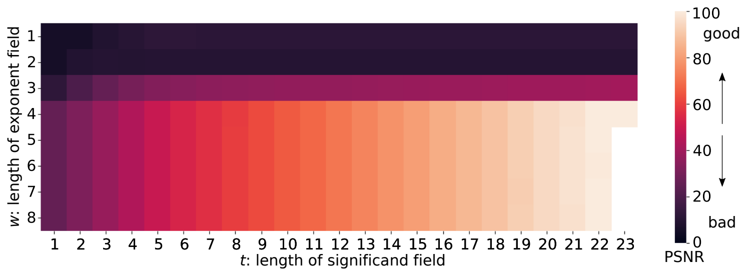

Fig. 2. Average quality of the generated faces when operating the generator with quantized weights, inputs and outputs. For one test, the full network is using a global format $T_{w,t}$ in all quantization steps and the quality is assessed by comparing the result with the reference faces computing the PSNR measured in dB. For exponent fields below three bits, a major harm is caused in the output. Increasing the number of exponent bits beyond 4 bits does not significantly alter the results as the dynamic range of the computing domain is covered well enough.

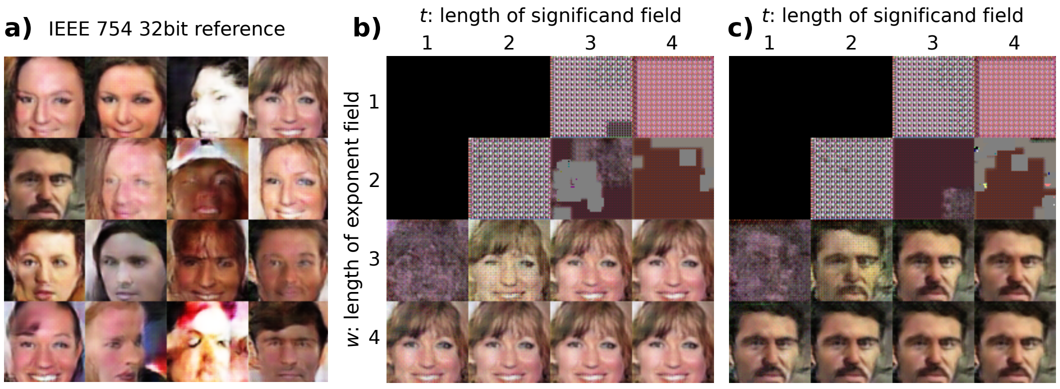
a) IEEE 754 32bit reference
b)
c)
t: length of significand field
1
2
3
4
w: length of exponent field

Fig. 3. a) One batch of 16 randomly generated synthetic faces when using the BAGAN generator network with the 32-bit floating point reference datatype. b) and c) show two representative samples produced with different formats $T_{w,t}$. Results obtained with number formats $T_{3,3}$, $T_{3,4}$, $T_{4,1}$, $T_{4,2}$, $T_{4,3}$, and $T_{4,4}$ look alike and are visually indistinguishable from the full precision reference.

8 REDUCED PRECISION FOR BLSTM

We integrated FloatX and studied the overall recognition accuracy on the optical character recognition (OCR) tasks by using a Bidirectional Long Short-Term Memory (BLSTM) architecture [Rybalkin et al. 2017]. We performed our evaluation based on the data and C implementation provided by the authors of the BLSTM [Rybalkin et al. 2017], and conclude that BLSTM profits from compact floating-point formats with negligible impacts on the final accuracy.

The trained BLSTM [Rybalkin et al. 2017] model solves the optical character recognition problem by processing scanned text images column-wise. The BLSTM updates two LSTM cells in a forward and backward passes, merges the results with a dense layer, and finally predicts with a softmax output layer. Even though matrix-vector operations dominate the BLSTM [Rybalkin et al. 2017], nonlinear activation functions within the recurrent update cause a non-trivial error propagation behaviour of the BLSTM inference. Due to the presence of recurrent computations within BLSTM, artefacts due to numerical approximation might potentially accumulate and propagate through the output and yield wrong results. However, the error resilience of deep learning methods [Hill et al. 2018; Rybalkin et al. 2017] allows recovering from imprecision caused by numerical representations. With FloatX we demonstrate for all internal representations and arithmetic operations that a reduction of the average bit-width down to 12.2 bit is enough to stay within the strict accuracy requirements.

Figure 4 shows a sharp transition between correct and incorrect operation when running at different precisions $T_{w,t}$. The sharp boundary between these two types of operations is due to the inherent error resilience of deep learning-based computations and stands in contrast to regular linear solvers that exhibit a smooth error dependency on bit widths. Due the presence of the accuracy plateau from $T_{6,6}$ up to $T_{11,52}$ (*double*), we identify $T_{6,6}$ as the optimal global data type, which is considerably narrower than $T_{8,23}$ (*float*) used to execute the baseline. Table 3 presents accuracies obtained for selected operation points. Operating with a 16-bit format encoded as $T_{8,7}$ [Tagliavini et al. 2017] outperforms the $T_{5,10}$ (*half*) format. Individually reducing input encoding for weights (W) and image (I) down to $T_{3,1}$ has a marginal effect on the result. Since simultaneously using $T_{3,1}$

[14] [Rybalkin, Wehn, Yousefi, and Stricker 2017]
[15] [Tagliavini, Mach, Rossi, Marongiu, and Benini 2017]

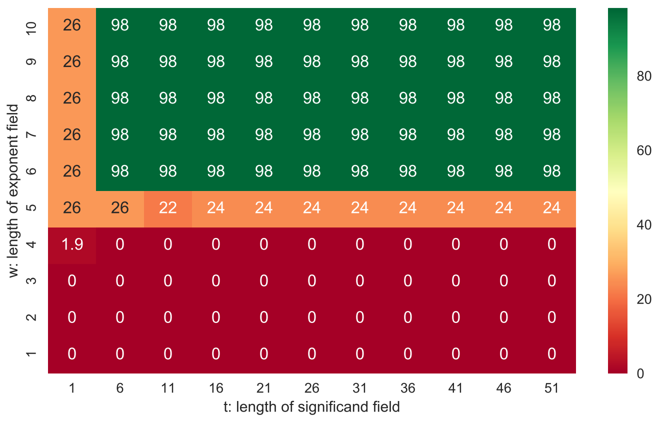

Fig. 4. Recognition accuracy of BLSTM when operating with type $T_{w,t}$. The baseline runs with $T_{8,23}$, which explains part of the plateau at around 98% accuracy. BLSTM operates close-to-perfect with further reduced formats.

Setting	Weights	Images	Operations	Accuracy
See ref.[13]	$T_{8,23}$	$T_{8,23}$	100% in $T_{8,23}$ (*float*)	98.2337%
See ref.[14], Fig. 6	16 bit fixed		See ref.[14] IV-C	97.9794%
See ref.[14], Fig. 6	5 bit fixed		See ref.[14] IV-C	97.5821%
$T_{5,10}$ (*half*)	$T_{5,10}$	$T_{5,10}$	100% in $T_{5,10}$ (*half*)	21.4392%
$T_{6,6}$, see Figure 4	$T_{6,6}$	$T_{6,6}$	100% in $T_{6,6}$	98.0536%
$T_{8,7}$, see ref.[15]	$T_{8,7}$	$T_{8,7}$	100% in $T_{8,7}$	98.1890%
Quantized W	$T_{3,1}$	$T_{8,23}$	100% in $T_{8,23}$	98.0692%
Quantized I	$T_{8,23}$	$T_{3,1}$	100% in $T_{8,23}$	98.1181%
Quantized W&I	$T_{3,1}$	$T_{3,1}$	100% in $T_{8,23}$	96.7730%
Quantized W&I	$T_{4,1}$	$T_{4,1}$	100% in $T_{8,23}$	98.1660%
Modified MAC (average 15.7 bit)	$T_{4,1}$	$T_{4,1}$	40.7% in $T_{5,2}$, 40.7% in $T_{5,10}$ 18.6% in $T_{8,23}$	98.1905%
Proposed (average 12.2 bit)	$T_{4,1}$	$T_{4,1}$	40.7% in $T_{5,2}$, 40.7% in $T_{5,10}$ 18.6% in $T_{6,6}$	97.9969%

Table 3. Impact of reduced precision on BLSTM recognition accuracy: reference results, global scans, input quantization effects and proposed configurations.

for weights and images reduces accuracy, we used $T_{4,1}$ which enables fair accuracy. Profiling shows that MAC operations from dot products contribute more than 80% of all executed operations. We suggest to use $T_{5,1}$ arithmetic for multiplication and a $T_{5,10}$ arithmetic for accumulation. The last configuration shows that replacing the remaining parts with narrow formats allows computing with an average bit-width of 12.2 bit and still getting an accuracy of about 98%.

9 SUMMARY AND FUTURE WORK

This paper presents FloatX, a framework developed in modern C++ designed with three main goals: *i)* easy-to-use and minimally-intrusive API which facilitate an incremental transformation of numerical applications; *ii)* offer reasonable performance while incurring no storage overhead; and *iii)* follow "natural" C++ rules such as "round-to-nearest, ties-to-even" and datatype casting conventions.

Our experience gained with the integration of numerical LA libraries such as BLAS, LAPACK, and Ginkgo, offers a variety of insights and serves to identify a number of problems that other users of FloatX may also encounter. In principle, the difficulties of porting an application to operate on top of FloatX largely depend on the design of the application. In general though, we found that many caveats were due to the use of C or Fortran. We hope that our efforts in this line can help others to rapidly take advantage and enjoy working with FloatX. The last group of applications aim to illustrate only two among the many numerical analyses that are enabled by FloatX. We recognize that the number of possibilities is much larger and we expect it will be fueled by the advance of IoT.

FloatX is a living open project. Therefore, it is natural to think of many appealing extensions to it. Here we name only a few:

- FloatX is a real floating-point arithmetic framework, and therefore, it is natural to ask whether it could support complex numbers over FloatX types and how.
- The only back-end type for FloatX is `double`. From the storage and performance points of view, depending on the emulated datatype, `single` or even `half` could be more efficient.
- As mentioned earlier in the manuscript, the development of a general rounding routine with support for different rounding mode is an open research line.

AVAILABILITY AND ACKNOWLEDGEMENTS

FloatX is distributed under an Apache Licence 2.0 and available at https://github.com/oprecomp/FloatX.

This work was supported by the CICYT projects TIN2014-53495-R and TIN2017-82972-R of the MINECO and FEDER, and the EU H2020 project 732631 "OPRECOMP. Open Transprecision Computing".

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

Edward Anderson, Zhaojun Bai, L. Susan Blackford, James Demmesl, Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, Anne Greenbaum, Alan McKenney, and Danny C. Sorensen. 1999. *LAPACK Users' guide* (3rd ed.). SIAM.

Hartwig Anzt, Jack Dongarra, and Enrique S. Quintana-Ortí. 2015. Adaptive Precision Solvers for Sparse Linear Systems. In *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing (E2SC '15)*. ACM, New York, NY, USA, Article 2, 10 pages. https://doi.org/10.1145/2834800.2834802

S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. 2008. Solving Dense Linear Systems on Graphics Processors. In *Proceedings of the 14th International Euro-Par Conference (Lecture Notes in Computer Science, 5168)*, E. Luque, T. Margalef, and D. Benítez (Eds.). Springer, 739–748.

C. Bekas, A. Curioni, and I. Fedulova. 2012. Low-cost Data Uncertainty Quantification. *Concurr. Comput. : Pract. Exper.* 24, 8 (June 2012), 908–920. https://doi.org/10.1002/cpe.1770

S. Boldo and G. Melquiond. 2008. Emulation of a FMA and Correctly Rounded Sums: Proved Algorithms Using Rounding to Odd. *IEEE Trans. Comput.* 57, 4 (April 2008), 462–471. https://doi.org/10.1109/TC.2007.70819

Alfredo Buttari, Jack J. Dongarra, Julie Langou, Julien Langou, Piotr Luszcek, and Jakub Kurzak. 2007. Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems. *Int. J. of High Performance Computing & Applications* 21, 4 (2007), 457–486.

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 16, 1 (March 1990), 1–17.

Samuel A. Figueroa. 1995. When is Double Rounding Innocuous? *SIGNUM Newsl.* 30, 3 (July 1995), 21–26. https://doi.org/10.1145/221332.221334

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2, Article 13 (June 2007). https://doi.org/10.1145/1236463.1236468

Mark Gates, Piotr Luszczek, Ahmad Abdelfattah, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. 2017. *C++ API for BLAS and LAPACK.* Technical Report 2, ICL-UT-17-03. Revision 02-21-2018.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems.* 2672–2680.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15).* JMLR.org, 1737–1746. http://dl.acm.org/citation.cfm?id=3045118.3045303

Gustafson and Yonemoto. 2017. Beating Floating Point at Its Own Game: Posit Arithmetic. *Supercomput. Front. Innov.: Int. J.* 4, 2 (June 2017), 71–86. https://doi.org/10.14529/jsfi170206

John Hauser. Accessed March 2019. Berkeley SoftFloat project home page. http://www.jhauser.us/arithmetic/SoftFloat.html.

Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (second ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. xxx+680 pages.

Parker Hill, Babak Zamirai, Shengshuo Lu, Yu-Wei Chao, Michael Laurenzano, Mehrzad Samadi, Marios Papaefthymiou, Scott Mahlke, Thomas Wenisch, Jia Deng, Lingjia Tang, and Jason Mars. 2018. Rethinking Numerical Representations for Deep Neural Networks. *arXiv e-prints* (Aug 2018). arXiv:1808.02513

N. M. Ho, E. Manogaran, W. F. Wong, and A. Anoosheh. 2017. Efficient floating point precision tuning for approximate computing. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC).* 63–68. https://doi.org/10.1109/ASPDAC.2017.7858297

IBM. 2015. Engineering and Scientific Subroutine Library. http://www-03.ibm.com/systems/power/software/essl/.

IEEE. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935

Intel. 2015. Math Kernel Library. https://software.intel.com/en-us/intel-mkl.

ISO. 2017. ISO International Standard ISO/IEC 14882:2017(E) – Programming Language C++. https://isocpp.org/std/the-standard. Visited June 2018..

V. Lefèvre. 2013. SIPE: Small Integer Plus Exponent. In *2013 IEEE 21st Symposium on Computer Arithmetic.* 99–106. https://doi.org/10.1109/ARITH.2013.22

Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2015. Deep learning face attributes in the wild. In *Proceedings of the IEEE International Conference on Computer Vision.* 3730–3738.

A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, and N. Wehn. 2018. The transprecision computing paradigm: Concept, design, and applications. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE).* 1105–1110. https://doi.org/10.23919/DATE.2018.8342176

Giovanni Mariani, Florian Scheidegger, Roxana Istrate, Costas Bekas, and Cristiano Malossi. 2018. BAGAN: Data Augmentation with Balancing GAN. *arXiv preprint arXiv:1803.09655* (2018).

Érik Martin-Dorel, Guillaume Melquiond, and Jean-Michel Muller. 2013. Some issues related to double rounding. *BIT Numerical Mathematics* 53, 4 (01 Dec 2013), 897–924. https://doi.org/10.1007/s10543-013-0436-2

Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4, Article 62 (March 2016), 33 pages. https://doi.org/10.1145/2893356

NVIDIA. 2016. cuBLAS. https://developer.nvidia.com/cublas.

D. O'Leary. 2006. Matrix factorization for information retrieval. Lecture Notes for a course on Advanced Numerical Analysis. University of Maryland. Available at https://www.cs.umd.edu/users/oleary/a600/yahoo.pdf.

OpenBLAS 2015. http://www.openblas.net.

Krishna V. Palem. 2014. Inexactness and a future of computing. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 372, 2018 (2014). https://doi.org/10.1098/rsta.2013.0281 arXiv:http://rsta.royalsocietypublishing.org/content/372/2018/20130281.full.pdf

Tim Palmer. 2015. Build imprecise supercomputers. *Nature* 526 (2015), 32–33.

T. N. Palmer. 2014. More reliable forecasts with less precise computations: a fast-track route to cloud-resolved weather and climate simulators? *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 372, 2018 (2014). https://doi.org/10.1098/rsta.2013.0391 arXiv:http://rsta.royalsocietypublishing.org/content/372/2018/20130391.full.pdf

Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning Assistant for Floating-point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 27, 12 pages. https://doi.org/10.1145/2503210.2503296

S.M. Rump. 1999. INTLAB - INTerval LABoratory. In *Developments in Reliable Computing*, Tibor Csendes (Ed.). Kluwer Academic Publishers, Dordrecht, 77–104. http://www.ti3.tuhh.de/rump/.

Siegfried M. Rump. 2016. IEEE754 Precision-k Base-β Arithmetic Inherited by Precision-m Base-β Arithmetic for $k < m$. *ACM Trans. Math. Softw.* 43, 3, Article 20 (Dec. 2016), 15 pages. https://doi.org/10.1145/2785965

Vladimir Rybalkin, Norbert Wehn, Mohammad Reza Yousefi, and Didier Stricker. 2017. Hardware architecture of Bidirectional Long Short-Term Memory Neural Network for Optical Character Recognition. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1390–1395.

G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benin. 2018. A transprecision floating-point platform for ultra-low power computing. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1051–1056. https://doi.org/10.23919/DATE.2018.8342167

Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. 2017. A Transprecision Floating-Point Platform for Ultra-Low Power Computing. *arXiv preprint arXiv:1711.10374* (2017).

Thornes Tobias. [n. d.]. Can reducing precision improve accuracy in weather and climate models? *Weather* 71, 6 ([n. d.]), 147–150. https://doi.org/10.1002/wea.2732 arXiv:https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/wea.2732

Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3 (2015), 14:1–14:33.

Todd L. Veldhuizen. 2003. *C++ templates are Turing complete.* Technical Report.

R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of SC'98*.

Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. 2015. Approximate Computing: A Survey. *IEEE Design & Test* 33 (01 2015), 8–22.

Abraham Ziv. 1991. Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit. *ACM Trans. Math. Softw.* 17, 3 (Sept. 1991), 410–423. https://doi.org/10.1145/114697.116813