Easing development of precision-sensitive applications with a beyond-quad-precision library

Christoph Lauter

Easing Development of Precision-Sensitive Applications with a Beyond-Quad-Precision Library

Christoph Lauter
Sorbonne Universités, UPMC Univ Paris 06,
UMR 7606, LIP6, F-75005, Paris, France
Email: christoph.lauter@lip6.fr

***Abstract*—The IEEE754 Standard offers essentially two binary floating-point formats, binary32 and binary64, natively supported by current hardware. Whenever these precisions do not suffice, developers are restricted to arbitrary precision libraries, such as MPFR.**

These libraries however leave a gap in the mid-precision range (64 to 512 bits). Their very nature –as object code to be linked in– prevents modern compilers from inlining the code or optimizing it e.g. with loop unrolling.

We propose the `libwidefloat` software meant to fill this gap. It offers precisions from 64 through 512 bits. It supports all basic operations ($+, -, \times, /, \sqrt{}, \mathrm{FMA}$, comparisons etc.) It is fully implemented in header files for automatic optimization.

I. Introduction

The IEEE754 Standard for Floating-Point (FP) Arithmetic defines several binary and decimal FP formats [1]. The basic ones, such as binary32 (single precision) and binary64 (double precision), are natively supported in hardware by current general-purpose processors. The IEEE754 binary128 (quad precision) format is supported in the current FP environment mainly through software emulation. The additional extended formats defined by IEEE754, with precisions beyond quad-precision, have no practical realization [1].

While binary32 and binary64 gear the majority of numerical software, some applications need support for precisions beyond quad-precision, i.e. with more than 113 bits of significand length. Examples for such applications include astronomy [2], [3], [4], computational physics [5] or experimental mathematics [6]. In some cases the use of extended precision might be overcome by subtle FP accuracy analysis and FP tricks [7], [8]. However, some developers may simply prefer to speed up development with extended precision, used in some compute kernels, even if this means breaking a butterfly on a wheel [9].

As we will discuss in more detail below, such applications typically resort to arbitrary precision FP libraries, such as MPFR[1] or MPFUN[2]. These libraries are properly designed and optimized: they offer very good, if not optimal, performance in the precision range where arbitrary precision overhead becomes negligible [10]. This range typically starts at precisions beyond a couple of hundreds to a thousand bits. This leaves a gap in the mid-precision range between quad-precision and a couple of hundred bits.

[1] http://www.mpfr.org/
[2] http://www.davidhbailey.com/dhbsoftware/

In this article, we propose the `libwidefloat` software designed to filled this gap. The package offers FP types with precisions between 64 and 512 bits, to be chosen at application compile time in 32 bit steps. Our FP types come with a 32 bit wide exponent, which essentially eliminates the need for support of subnormal numbers. They come with support for IEEE754-like infinities, Not-A-Numbers, rounding modes and FP flags [1]. The latter two features can be deactivated (resp. set to default round-to-nearest mode) at application compile time to allow for enhanced performance.

This article is organized as follows: in Section II, we first present existing solutions to extended FP precision, analyzing the causes of their shortcomings in the mid-precision range. We then give an overview of modern compiler techniques that we will leverage to overcome these shortcomings (Section III). In Section IV, we present our `libwidefloat` library, detailing the data-types and operations it supports. We briefly describe the algorithms used in the library in Section V, give experimental results in Section VI and conclude with Section VII.

II. Existing Extended Precision Libraries and their shortcomings

We have claimed that the existing arbitrary precision libraries, such as MPFR, are very well optimized for higher precisions (starting at about 1000 bits) but hindered by considerable overhead in the low- and mid-precision range. This overhead is due to several factors.

First, the very nature of arbitrary precision means that the length of the limb arrays that hold the FP significands of the different formats is not statically known. As a consequence, all variables having a type defined by one of these arbitrary precision libraries need not only be declared but also be initialized, using a call to a certain initialization function [11]. This function actually attributes a precision setting to the arbitrary precision variable and calls a memory allocation function such as `malloc` to provide space for the FP variable's significand. All accesses to a program's arbitrary precision variables' significands are in consequence indirect, requiring pointers to be followed onto the stack. Together with the dynamic nature of the significand length, this mostly prevents compilers from performing any reasonable optimization involving the arbitrary precision FP significands. In addition, for certain applications the requirement to allocate the arbitrary precision data on the

memory heap and not hold them on the stack induces important overhead, as memory allocation may have significant cost with respect to computations for smaller precisions.

Second, the fact that existing arbitrary precision software comes packaged as software libraries adds to their overhead. While operations on IEEE754 binary32 and binary64 data-types are "inlined" as basic operations into the code, each and every use of an arbitrary precision operation involves a function call with all its cost in terms of register usage, variable spills, instruction cache misses etc. In addition, the correspondence between function names and the code behind them gets lost to optimization. Typically, when calling the arbitrary precision library, the compiler does not know of the functions' use cases and when compiling the application code, it does not anything about the library functions but their name. Due to the use of assembly in the underlying libraries such as GMP[3], the arbitrary precision libraries even push this issue further to a point where the compiler can almost no optimization, even when using Link-Time-Optimization [12], [11].

III. Leveraging Compiler Optimization Techniques

In order to take the observations described in the previous Section II into account, the following requirements to a software package for mid-precision FP arithmetic seem sensible.

First, the piece of software should not be packaged as a library to be linked in but should come in the form of code to be directly integrated into the application code. With modern compilers, such as `gcc`, this can be achieved by condensing the mid-precision FP library into header files that contain the functions provided by the software package declared `static` and `inline`. This way, the compiler[4] will be able to inline the FP functions into the application, propagating constants as far as possible, eliminating all unnecessary parts (such as NaN handling where no NaNs are possible) and optimizing loops by unrolling them.

Second, the software package should not use any micro-optimizations resorting to inline assembly or linking with objects written in assembly. While it is tempting for performance optimization at a function level, the use of assembly prevents compilers from propagating the information about the program it needs for optimization through the assembly sequence [12]. In most cases, modern compilers have sufficient support for doubled-precision integer arithmetic, in particular full 64×64 bit multiplication, through builtins (such as the `__uint128_t` type in `gcc`[5]), so that resorting to assembly is no longer necessary.

Finally, all wide integer and FP data-types to be declared by a FP software package optimized for the mid-precision range should be statically sized, in order to be held directly on the stack. In addition, the data should be properly aligned in order to avoid cache misses. As a matter of course, the endianess of wide integers, represented as limbs of a statically sized array should match the system's natural endianess, allowing the compiler to make use of up-converting and down-converting memory move operations [13]. In order to help the compiler, all accesses to these data-types involving pointers (or indexing in an array) should be done in a manner that allows for optimization. For instance, functions taking references on such data-types should be declared in a way that makes the compiler understand[6] that the pointers are not aliased (using the `restrict` keyword) and constant (using `const`) [14], [15].

As a matter of course, the last requirement of using statically sized types in a FP mid-precision software package supporting more than one precision comes at the cost of a combinatorial explosion of functions to be provided. Indeed, if the ease of mixing FP types of different precisions that is provided by arbitrary precision libraries is to be maintained, where an arbitrary precision library provides one function for, say, addition, a software package with statically sized types has to provide several functions, one per combination of input and output types that is possible and sensible. In our work on `libwidefloat`, we have made extensive use of macro-programming and code generation to overcome this issue of combinatorial explosion.

IV. The `libwidefloat` library

The `libwidefloat` software was implemented with the shortcomings of the existing arbitrary precision libraries and modern compiler technology in mind. The software is implemented with two header files only, using modern C99/C11 [14], [15] but no assembly in order not to break the compiler's opportunities to optimize the code. All functions are declared in a way that allows them to be inlined and then optimized in their use context by the compiler.

The `libwidefloat` header starts by declaring statically sized data-types for `libwidefloat` FP values with precisions between 32 and 512 bits, by 32 bit increments. All these types support the basic IEEE754 FP data, such as positive and negative real numbers, signed zeros, signed infinities and Not-A-Number (NaN) data. The significand length is adapted in size according to the type's precision; the exponent width is kept at 32 bits for all precisions. Since this exponent width is pretty wide, we chose not to provide support for subnormal numbers in the `libwidefloat` types. The NaNs are signed but do not provide support for payload. They are all quiet NaNs; signaling NaNs are not supported.

Even though the `libwidefloat` types appear to be IEEE754-compliant from a programmer's perspective, their encoding is an ad-hoc one and does not respect the provisions of the IEEE754 Standard on extended precision types [1].

The `libwidefloat` software has provision for all five IEEE754 exception flags (inexact, invalid, overflow, underflow and division-by-zero) as well as for all four IEEE754 rounding modes (round-to-nearest-ties-to-even, round-down, round-up, round-towards-zero) [1]. The software can be configured at compile time in a way that allows this FP state, made of the flags and the rounding-mode, to be either

- held in a global variable,

[3] see https://gmplib.org/

[4] see e.g. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[5] see e.g. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fint128.html

[6] see e.g. https://gcc.gnu.org/onlinedocs/gcc/Restricted-Pointers.html

- passed as a handle to any of the libwidefloat functions or
- omitted, assuming round-to-nearest as the rounding-mode and setting no flags.

The code is written in a way such that the compiler is able to eliminate all unnecessary computation steps that are not required when no flags are to be set and the rounding mode is statically set to round-to-nearest.

Currently, libwidefloat includes 1303 functions covering the following types of operations:

- Conversion between all combination of libwidefloat types. The up-conversions are exact, the down-conversions are correctly rounded.
- Conversion from and to the IEEE754 binary32 and binary64 formats [1]. These conversions are correctly rounded and signal all required exceptions, in particular inexact when rounding is necessary. These exceptions are signaled in and the rounding mode is taken from the FP environment of the type the conversion converts to. For instance, when converting from IEEE754 to libwidefloat, rounding is done according to the libwidefloat rounding mode and not the IEEE754 one.
- Conversion from and to signed and unsigned integer types. These conversions are provided in all variants (signaling inexact or not, taking the rounding mode from an immediate or from the FP state) that are defined by the IEEE754 Standard.
- Rounding to integral as defined by the IEEE754 Standard [1].
- Signaling and quiet comparisons, returning the comparison result as a enumerated type [1].
- Homogeneous-type absolute value and negation operations. The heterogeneous variants, where the operands' type differs from the one of the result can be provided through composition with a conversion operation [1].
- Operations to access the libwidefloat FP environment with respect to flags and rounding modes.
- And finally, addition, Subtraction, Multiplication, Division, Fused-Multiply-And-Add (FMA) and Square Root with correct rounding. All these operations are provided in a homogeneous variant, where all operands and the result have the same type and in all heterogeneous variants, where the result is of a smaller type, such that all correctly rounded heterogeneous operations required by IEEE754 can be provided through composition with the conversion operations [1].

V. Arithmetical Algorithms used in libwidefloat

There is no real novelty in the algorithms used in libwidefloat. In order to manage the combinatorial explosion due to support for the 16 formats covering the precisions from 32 to 512 bits in homogeneous and heterogeneous operations, the final rounding step towards any of the supported formats has been factored into final rounding functions. These final rounding functions receive an intermediate result that has undergone already some rounding with respect to the mathematically exact result.

We shall hence explain the techniques used for intermediate and final rounding first, before detailing the algorithms behind multiplication, addition, FMA, division and square root.

Let us just note that libwidefloat also contains conversion functions from and to integer and the IEEE754 binary32 and binary64 formats. These conversions have been coded in ad-hoc ways, which are not described inhere.

A. Intermediate and Final Rounding

In order to provide correct rounding $\circ(y) = \circ(f(x))$ for a function f, an intermediate result $\tilde{y}$ must be computed in such a way that $\circ(\tilde{y}) = \circ(y)$ [16]. For all operations supported by libwidefloat, computing such an approximation $\tilde{y}$ is pretty easy: it is either possible to compute the exactly representable result $f(x)$ (multiplication), or to compute a good approximation together with a flag indicating inexactness. Indeed, for division and square root, the backward error can be exactly computed and compared to zero to yield the inexactness predicate. For addition and FMA, it is just necessary to control the part of the significand that gets discarded during the alignment shift.

However, when computing the approximation $\tilde{y}$ out of the exact result or a better approximation assorted with an inexact indication, pure truncation is not enough. Such a rounding scheme would suffer from the so-called double-rounding issue [16], [17].

In order to overcome the double-rounding issue, libwidefloat uses rounding to odd for the intermediate results $\tilde{y}$. This particular rounding mode is due to Boldo and Melquiond [17] and can be though of as a intermediate rounding longer than final precision where the last few additional bits represent the classical round, guard and sticky bits [18].

B. Multiplication

As already stated, multiplication in libwidefloat is based on an exact multiplication of the input significands, considered to be integer. At the current state of the library development, naïve schoolbook multiplication using $\mathcal{O}(n^2)$ machine multiplications is used. The partial products are first represented in a carry-save format and then reduced [19].

The use of Karatsuba or Toom-Cook Multiplication [16] is left for future work but might prove necessary.

C. Addition, Subtraction and Fused-Multiply-Add

When implemented in software, Floating-Point Addition and Subtraction reduce to the same algorithm. The FMA operation can also share this part of code; the only difference is in whether one of the arguments to addition comes directly from one of the operands or from an intermediate exact product.

The core algorithm for addition and subtraction used in libwidefloat is a classical one [16], [18] and very simple: after reordering of the FP operands for exponent, the significand of the lesser operand is shifted right for alignment, unless

this shift would be larger than the output precision. The aligned significands are then added resp. subtracted. A normalization step then ensures correct construction of a FP significand in the case of elimination for subtractions. In software, due to overhead in branching and code bloat, it is sensible to run that normalization step in any case and not to distinguish between a near-path and a far-path.

D. Division and Square Root

The algorithms for division and square root are both based on an approximation to $1/\sqrt{z}$ in `libwidefloat`. We compute division x/y as $x/(\sqrt{y})^2$ and square root $\sqrt{x}$ as $x \times 1/\sqrt{x}$.

An approximation to $1/\sqrt{x}$ can easily be computed using the Newton-Raphson iteration

$$y_{n+1} = y_n \cdot \frac{1}{2} \cdot \left(3 - x \cdot y_n^2\right)$$

seeded with a small table for y_0 [16]. In `libwidefloat` we use a 6 bit approximation for y_0.

In an extended-precision software environment, it is useful not to perform the Newton iteration on software-simulated FP variables but purely on integer variables, using a fixed-point representation. We use this approach in `libwidefloat`. However, we chose to generate that code sequence instead of writing it manually in order not to have to determine all alignment shifts in that fixed-code point by hand.

VI. Experimental results

Our current implementation of `libwidefloat` provides 1303 user-available functions. We have tested it for performance against MPFR 3.1.2-p3 based on GMP 6.0.0. We used `gcc` 4.9.2 on a 64bit Linux system featuring a 4 core Intel i7-5500U running at 2.4 GHz. The optimization level was set to `-O3`.

There is no one-to-one relation between the functions provided by `libwidefloat` and those provided by MPFR. We therefore do not give performance results per function. We rather wrote two benchmark applications and compiled them against both `libwidefloat` and MPFR. The first benchmark application consists of a LU-decomposition, triangular system solving and residue computation. The second one uses secant's method to approximate the zero of a high-degree polynomial.

All benchmark codes were run in all precisions supported by `libwidefloat` and compared to MPFR, which was run at the same precisions. While we focused on performance testing, we also compared the actual numerical results of `libwidefloat` and MPFR. On all benchmark codes and all precisions, the results were bitwise identical between both libraries, as expected, given that both libraries provide correct rounding.

We ran our LU-decomposition, triangular system solving and residue benchmark for matrix sizes 1000×1000. The matrices and respective right-hand sides were filled with random IEEE754 binary64 values between -10^7 and 10^7. The time needed to allocate and free the different vectors and

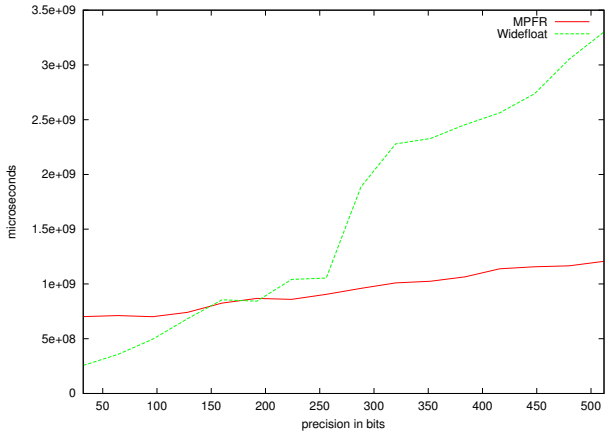

Figure 1. Experimental results: Linear Algebra

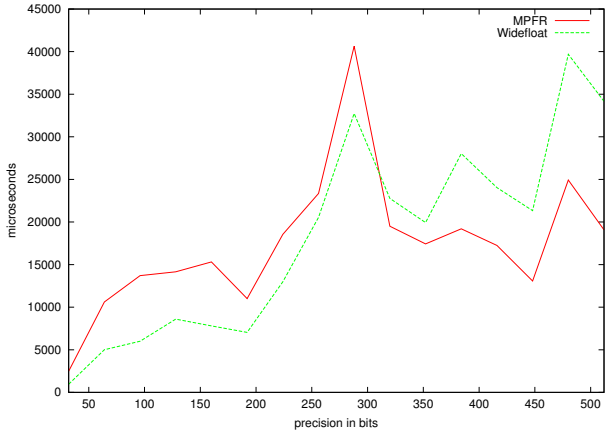

Figure 2. Experimental results: Expression Evaluation

matrices was included in the performance testing. Matrix and vector-allocation for MPFR required calling the MPFR initialization procedures for each element of the matrix resp. vector; `libwidefloat` was not burdened by this requirement.

The results of the LU-decomposition benchmark are given in Figure 1. They show that `libwidefloat` performs well better than MPFR for small precisions but that starting with precisions around 224 bits, the use of Karatsuba-multiplication helps in keeping MPFR's timings low while `libwidefloat` exhibits quadratic behavior. We shall address this issue in our library in future releases.

In our second benchmark, we used secant's method to approximate a zero of a polynomial of degree 150 up to the accuracy that was achievable within the given compute precision (which we made vary for the test.) The polynomial was input as an expression tree and evaluated using a recursive algorithm. This example is typical for numerical Computer Algebra Systems, such as Sollya[7] or Sage[8].

As shown with Figure 2, for that second expression-evaluation benchmark, `libwidefloat` typically performed better with respect to MPFR. However, our library is still

[7] see http://sollya.gforge.inria.fr/
[8] see http://www.sagemath.org/

hindered by the use of a naïve multiplication algorithm for precisions higher than 320 bits.

VII. Conclusion and Future Work

The `libwidefloat` software fills the gap between IEEE754 binary64 (or binary128 where available) and arbitrary precision libraries, which involve a certain amount of overhead in the mid-precision range. Our library leverages modern compilers' optimization techniques, such as constant propagation, code inlining and dead code elimination, to achieve this goal.

At the current state, `libwidefloat` supports all basic operations, like e.g. addition, FMA, square root or comparisons. Development is ongoing: we are working on a C++ wrapper to allow for an easy drop-in-replacement of the IEEE754 binary64 format by `libwidefloat` formats. Further, we plan to use GNU's `printf`/`scanf` extension techniques to allow for easy input/output operations with `libwidefloat`. The development of the elementary intrinsic functions like the exponential, the logarithm or the trigonometric operations is also part of future work.

Experimentally, we see that `libwidefloat`'s performance still needs some improvement in the range between 256 and 512 bits. The current implementation of the library lacks sufficient performance due to the use of a naïve (schoolbook) multiplication algorithm. We shall extend the library to use the Karatsuba or Toom-Cook multiplication algorithms in that higher precision range.

References

[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008.

[2] J. Laskar, P. Robutel, F. Joutel, M. Gastineau, A. Correia, and B. Levrard, "A long-term numerical solution for the insolation quantities of the earth," *Astronomy & Astrophysics*, vol. 428, no. 1, pp. 261–285, 2004.

[3] J. Laskar, "High precision astronomical solution for paleoclimate studies," in *EGU General Assembly Conference Abstracts*, vol. 15, 2013, p. 11302.

[4] A. Farrés Basiana, J. Laskar, S. Blanes, F. Casas Pérez, J. Makazaga, and A. Murua, "High precision symplectic integrators for the solar system," 2013.

[5] A. Wittig and M. Berz, "Rigorous high precision interval arithmetic in cosy infinity," *Proceedings of the Fields Institute*, 2009.

[6] D. H. Bailey and J. M. Borwein, "High-precision computation and mathematical physics," 2015.

[7] S. Graillat, "Contribution à l'amélioration de la précision et à la validation des algorithmes numériques," Habilitation à Diriger des Recherches, Université Pierre et Marie Curie, 2013.

[8] S. Boldo, "Formal verification of tricky numerical computations," in *16th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, Würzburg, Germany, Sep. 2014, invited talk. [Online]. Available: http://www.scan2014.uni-wuerzburg.de/book_of_abstracts/

[9] D. Thomas, "A general-purpose method for faithfully rounded floating-point function approximation in fpgas," in *Computer Arithmetic (ARITH), 2015 IEEE 22nd Symposium on*, June 2015, pp. 42–49.

[10] K. R. Ghazi, V. Lefevre, P. Theveny, and P. Zimmermann, "Why and how to use arbitrary precision," *Computing in Science and Engineering*, vol. 12, no. 3, p. 5, 2010.

[11] 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. Softw.*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[12] T. Glek and J. Hubicka, "Optimizing real world applications with GCC link time optimization," *CoRR*, vol. abs/1010.2196, 2010. [Online]. Available: http://arxiv.org/abs/1010.2196

[13] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual, Instruction Set Reference, A-Z, Vol. 2*.

[14] ISO, *ISO/IEC 9899:1999: Programming Languages — C*, Dec. 1999, available in electronic form for online purchase at `http://webstore.ansi.org/` and `http://www.cssinfo.com/`. [Online]. Available: http://anubis.dkuug.dk/JTC1/SC22/open/n2620/n2620.pdf;http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/n897.pdf;http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+9899%3A1999;http://www.iso.ch/cate/d29237.html

[15] ——, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, December 2011. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

[16] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Boston: Birkhäuser, 2010.

[17] S. Boldo and G. Melquiond, "When double rounding is odd," in *Proceedings of the 17th IMACS World Congress on Computational and Applied Mathematics*, Paris, France, 2005.

[18] M. D. Ercegovac and T. Lang, *Digital arithmetic*. San Francisco (Calif.): Morgan Kaufmann Oxford, 2004.

[19] D. Defour and F. d. Dinechin, "Software carry-save for fast multiple-precision algorithms," in *35th International Congress of Mathematical Software*, Beijing, China, 2002, updated version of LIP research report 2002-08.