BeDoP: Beyond Double Precision

Paul Zimmermann

ANR FastRelax, Lyon, 27 May 2015

The BeDoP project

Research project written in 2014, after interaction with many colleagues (including those involved in FastRelax)

Timeframe over 2016-2020 with one junior researcher, 3 PhD students, 2 post-doctoral students, one development engineer.

Submitted to the European Research Council (ERC) Advanced Grant proposal

Did not pass Step 1 of the evaluation (score B, ranking range 62-74%)

Full project available on `http://www.loria.fr/~zimmerma`

Motivation

Fact 1: petaflop milestone (10^{15} flops) reached in 2008 (IBM's Roadrunner), expect exaflop milestone (10^{18} flops) reached around 2020

Fact 2: hardware floating-point (double precision, about 16 decimal digits) will not change soon

Fact 3: in most applications, rounding errors can increase linearly with the number of operations

Thus more and more applications will require **beyond double precision** in software

$\Longrightarrow$ it is our responsibility to make those computations fast and correct

How: using **formal proof** techniques

The European Exascale Software Initiative (EESI):

> *the ability to perform floating-point arithmetic with different precisions (e.g., 32-, 64-, and 128-bit) will likely be necessary in Exascale systems.*
>
> *...*
>
> *The fundamental challenge of library software design is to develop and provide robust and reliable algorithms and implementations that deliver accurate results or at least compute results with accuracy estimates.*

Reference: Working group report on numerical libraries, solvers and algorithms, `http://www.eesi-project.eu/`, 2011.

Bailey, Barrio, Borwein: *High-precision computation: Mathematical physics and dynamics*, Applied Mathematics and Computation, 2012

several applications already need more than double: evolution of the solar system over billions of years, supernova simulations, climate modeling, studying the fine structure constant of physics, ...

Example

Theorem: Let x be a 5-digit decimal floating-point number. Convert x to the nearest q-bit binary floating-point number, say y. Convert back y to the nearest 5-digit decimal floating-point number, say z. Then if $q \geq 18$ we have $z = x$.

$$x = 3.1415 \rightarrow_{18} y = \frac{205881}{2^{16}} \approx 3.141495 \rightarrow_{5} z = 3.1415$$

$$x = 8.0003 \rightarrow_{17} y = \frac{32769}{2^{12}} \approx 8.000244 \rightarrow_{5} z = 8.00024$$

But we also need that the conversions decimal $\leftrightarrow$ binary are correctly rounded!

Interval Arithmetic (IEEE-1788 standard)

Most likely implementations of IEEE-1788 will be in software, and might include arbitrary precision (cf `libieeep1788` from Marco Nehmeier).

Tight enclosure is not required, but directed rounding should be on the correct side (for infsup representation)

Main target: the GNU MPFR library

- I know it well...

- already used in several applications

- formally ensuring correctness with keeping (or improving) efficiency will be a real challenge

Why formal proof?

Why not?

Already used successfully in big projects: four-color theorem, Feit-Thomson theorem about the classification of finite groups, Intel and AMD hardware processors (Harrison and Russinoff), Flocq library (Boldo and Melquiond), CompCert compiler (Leroy and colleagues), Why3 platform (Filliâtre and colleagues), ...

Will force to simplify the code to make (formal) proof simpler:

- separate the computation of correct rounding (using round/sticky bits)
- design new layers for operations on significands and exponents, avoiding hard-coded bit manipulations as much as possible

Find new bugs? Remove dead code? Improve speed?

GCC Quadruple-Precision Library

Since 2008, GCC provides `__float128` with $+, -, \times, \div$.

Since 2011, some mathematical functions are provided through `libquadmath` (comes from FDLIBM developed by Sun around 1993): `expq`, `logq`, `sinq`, ...

Basic arithmetic on `__float128` seems correctly rounded.

Mathematical functions are not: with GCC 4.9.1, `sqrtq` for 2.0 is off by one ulp. Found errors of up to 10^5 ulps.
Time for multiplying two 1000×1000 matrices, with GCC -O3 (version 4.9.1) on a 3.2Mhz Intel Core i5-4570:

53 bits	double:	0.54s	MPFR: 43.5s (ratio 81)
64 bits	long double:	2.9s	MPFR: 53.2s (ratio 18)
113 bits	__float128:	38.2s	MPFR: 47.1s (ratio 1.2)

Double-double arithmetic (aka expansions)

Represent x as $h + \ell$ where h and ℓ are double-precision numbers.

Pros: arithmetic on h and ℓ is very fast

Cons: exponent limitation (10^{-324} to 10^{308})

Implementations:

- QD package from Bailey and colleagues (includes also quad-double), however no guarantee of maximal rounding error.
- FastRelax Expansions?

implementation	precision	correct rounding	formal proof
hardware $+,-,\times,\div$	53 bits	yes	yes
libc math. functions	53 bits	no/yes	no
libgcc $+,-,\times,\div$	113 bits	yes	no
libquadmath math. functions	113 bits	no	no
MPFR $+,-,\times,\div$	arbitrary	yes	no
MPFR math. functions	arbitrary	yes	no

BeDoP Scientific Roadmap

- Research Target 1: Formalising Low-Level Arbitrary-Precision Floating-Point Arithmetic
- Research Target 2: Formalising Quadruple-Precision Arithmetic
- Research Target 3: Formalising Arbitrary-Precision Arithmetic

Research Target 1: Formalising Low-Level Arbitrary-Precision Floating-Point Arithmetic

- Task RT1-1: Design the MPS Language Interface
- Task RT1-2: Efficient Implementation of the MPS Language
- Task RT1-3: Formally Prove the Correctness of the MPS Implementation

The MPS language

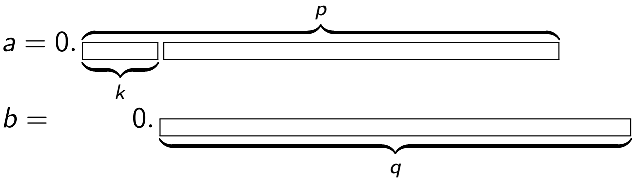
In GMP, large integers (`mpz`) are based on the basic layer `mpn`

The MPS layer will be similar to `mpn` for floating-point operations

$$x = 0.\underbrace{\overbrace{\boxed{\mathbf{1100110}}}^{\text{upper bits}}\,\boxed{\mathbf{0101001}}\boxed{\mathbf{1101101}}\,\overbrace{\boxed{\mathbf{011}}}^{\text{lower}}}_{p\text{ bits}}\,\overbrace{\boxed{0000}}^{\text{guard}} \quad \times \quad 2^e$$

```
void mps_add (void *a, long p,
              void *b, long q, long k,
              int *rnd, int *stck)
```

`mps_add` adds $\{b, q\}$ shifted by k bits towards the least significant bits to $\{a, p\}$:



At exit, `rnd` and `stck` are set to the *round* and *sticky* bits

For the (common) case where $\mathrm{prec}(b) \le \mathrm{prec}(a)$:

```
int add (mps a, mps b, mps c, int rnd_mode)
{
   int rnd, stck;
   long k = b->exp - c->exp; /* assumed non-negative */

   mps_cpy (a->ptr, a->prec, b->ptr, b->prec); /* exact */
   mps_add (a->ptr, a->exp, c->ptr, c->exp, k, &rnd, &stck);
   return round (a, rnd, stck, rnd_mode);
}
```

Using GMP, `mps_add` might use `mpn_add` and `mpn_rshift`.

Formally Prove the Correctness of MPS

```
Theorem mps_add_correct :
  forall a p b q k rnd stck mem mem',
  mem' = eval mem (mps_add a p b q k rnd stck) ->
  let c_exact = value mem a p + value mem b q / 2^k in
  no_overlap a p b q -> value mem' a p = round (c_exact p) /\
  is_round_bit (value mem' rnd) c_exact /\
  is_sticky_bit (value mem' stck) c_exact.
```

Research Target 2: Formalising Quadruple-Precision Arithmetic

- Task RT2-1: Efficient Quadruple-Precision Routines
- Task RT2-2: Formal Proof of the Quadruple-Precision Routines
- Task RT2-3: Validate Quadruple-Precision Routines on Large-Scale Applications

Efficient Quadruple-Precision Routines

Table Maker's Dilemma (TMD) not solved in general!

Use Ziv's onion peeling strategy, with initial precision tuned to minimize the average time.

Implementation: either contribute to `libquadmath`, or build a special quadruple-precision layer in MPFR:

```
__float128 sinq (__float128 x)
```

Formal Proof of the Quadruple-Precision Routines

```
Theorem sinq_correct :
   forall (x : binary128) rnd_mode,
   sinq x rnd_mode = round (sin x) binary128 rnd_mode.
```

Research Target 3: Formalising Arbitrary-Precision Arithmetic

- Task RT3-1: Efficient Arbitrary-Precision Routines
- Task RT3-2: Formal Proof of Arbitrary-Precision Routines
- Task RT3-3: Validate Arbitrary-Precision Routines on Large-Scale Applications

Efficient Arbitrary-Precision Routines

For example for the hyperbolic sine integral (Shi function):

```
int mpfr_shi (mpfr_t y, mpfr_t x, mpfr_rnd_t rnd_mode)
```

Formal Proof of Arbitrary-Precision Routines

```
Theorem shi_correct :
   forall (x : arbitrary_fp) (y : arbitrary_fp)
          rnd_mode mem mem',
   mem' = eval mem (mpfr_shi y x rnd_mode) ->
   y = round (shi x) (precision y) rnd_mode.
```