The CORE-MATH Project

Alexei Sibidanov, **Paul Zimmermann**, Stéphane Glondu

Arith 2022 conference, September 12, 2022

Which library is correct?

```
#include <stdio.h>
#include <math.h>

int main()
{
  float x1 = 1.01027, x2 = 1.775031328;
  printf ("sinf(x1)=%.9f sinf(x2)=%.9f\n", sinf (x1), sinf (x2));
}
```

GNU libc 2.36:

sinf(x1)=0.846975386 sinf(x2)=0.979216456

Intel Math Library (oneAPI 2022.0.0):

sinf(x1)=0.846975446 sinf(x2)=0.979216397

Which version is correct?

```
#include <stdio.h>
#include <math.h>
#include <gnu/libc-version.h>

int main() {
  printf("GNU libc version: %s\n", gnu_get_libc_version ());
  double x = -0x1.f8b791cafcdefp+4; printf ("sin(x)=%la\n", sin (x));
}
```

```
$ gcc -fno-builtin sin.c -lm
```

GNU libc version 2.27:

sin(x)=-0x1.073ca87470df9p-3

GNU libc version 2.36:

sin(x)=-0x1.073ca87470dfap-3

Unexpected behavior on different hardware

```
#include <stdio.h>
#include <math.h>
int main() {
  double x = 0x1.01825ca7da7e5p+0;
  printf ("x=%la y=%la\n", x, acosh (x));
}
```

```
$ icc -fno-builtin test_acosh.c # icc version 19.1.3.304
```

sirocco14.plafrim.cluster (Intel Xeon Gold 6142):

x=0x1.01825ca7da7e5p+0 y=0x1.bc8c6186687cbp-4

zonda03.plafrim.cluster (AMD EPYC 7452, same binary):

x=0x1.01825ca7da7e5p+0 y=0x1.bc8c6186687cap-4

Our Dream

```
#include <stdio.h>
#include <math.h>

int main()
{
  float x1 = 1.01027, x2 = 1.775031328;
  printf ("sinf(x1)=%.9f sinf(x2)=%.9f\n", cr_sinf (x1), cr_sinf (x2));
}
```

GNU libc 3.4.5:

sinf(x1)=0.846975386 sinf(x2)=0.979216397

Intel Math Library 23.1.4.217:

sinf(x1)=0.846975386 sinf(x2)=0.979216397

Correct Rounding

Definition

For a mathematical function f, a floating-point format F, the correct rounding of $f(x)$ for $x \in F$ is the unique floating-point number $y \in F$ which is closest to $f(x)$ in the given rounding direction.

Remark: here "closest" means with ties resolved (if any).

Consequence: uniqueness $\Longrightarrow$ reproducibility.

What does IEEE 754 say?

9. Recommended operations

Clause 5 completely specifies the operations required for all supported arithmetic formats.

This clause specifies additional operations that are recommended. In a specific programming environment, these operations might be represented in operator notation or in function notation. The function names used in a specific programming environment might differ from the names of the corresponding mathematical functions or from the names of this standard's corresponding operations.

Table 9.1—Additional mathematical operations

Operation	Function	Domain	Other exceptions; see also 9.2.1
exp **expm1** **exp2**	e^x $e^x - 1$ 2^x		

A conforming operation shall return results correctly rounded for the applicable rounding direction for all operands in its domain.

What does the C standard say?

21 The C functions in the following table correspond to mathematical operations recommended by IEC 60559. However, correct rounding, which IEC 60559 specifies for its operations, is not required for the C functions in the table. 7.32.8 reserves `cr_` prefixed names for functions fully matching the IEC 60559 mathematical operations. In the table, the C functions are represented by the function name without a type suffix.

IEC 60559 operation	C function	Clause
exp	`exp`	7.12.6.1, F.10.3.1
expm1	`expm1`	7.12.6.6, F.10.3.6
exp2	`exp2`	7.12.6.4, F.10.3.4
exp2m1	`exp2m1`	7.12.6.5, F.10.3.5
exp10	`exp10`	7.12.6.2, F.10.3.2
exp10m1	`exp10m1`	7.12.6.3, F.10.3.3
log	`log`	7.12.6.11, F.10.3.11
log2	`log2`	7.12.6.15, F.10.3.15
log10	`log10`	7.12.6.12, F.10.3.12
logp1	`log1p`, `logp1`	7.12.6.14, F.10.3.14
... continued ...		

Hard-to-Round (HR) Cases

Binary32:

$$\mathtt{pow}(\mathtt{0x1.46ee2p+67}, \mathtt{-0x1.acbb3ap-7})$$
$$= \mathtt{0x1.15f75e\color{red}{000000000000000}58b...p-1}$$

Binary64:

$$\mathtt{hypot}(\mathtt{0x1.6p+0}, \mathtt{0x1.2c2fc595456a7p-26})$$
$$= \mathtt{0x1.60000000000008\color{red}{000000000000}141...}$$

Decimal64:

$$\mathtt{exp}(0.09407822313572878)$$
$$= 1.0986456820663385\color{red}{0000000000000000}278...$$

How to Round Correctly?

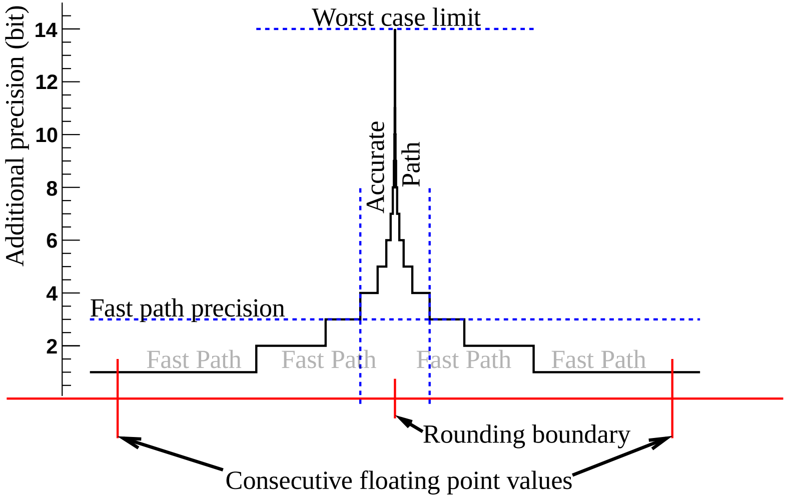
Ziv's "onion peeling" Strategy.

Assume the target type has n bits.

Let $\circ()$ the current rounding mode.

- [quick phase/fast path] compute an approximation y with say $n + 10$ correct bits, and maximal error ε
- [rounding test] if $\circ(y - \varepsilon) = \circ(y + \varepsilon)$, return that number
- [if any] deal with exact or midpoint cases
- [accurate phase/slow path] otherwise compute an approximation y' with more than m correct bits, where m is a bound on the hardest-to-round cases, then $\circ(y')$ is always CR (Table Maker's Dilemma)

A figure is better than thousand words



MathLib/libultim (Ziv et al., IBM, 1991)

Provides the following binary64 functions: `acos`, `asin`, `atan`, `atan2`, `exp`, `exp2`, `log`, `log2`, `cos`, `sin`, `tan`, `cot`, `pow`

Slow path based on multiple-precision arithmetic with up to 768 bits.

Correct rounding only for rounding to nearest.

Integrated in GNU libc 2.27 (except acos, exp2, log2 and cot).

Slow path removed progressively after GNU libc 2.27.

440,000 cycles for binary64 pow in the "768-bit" path.

Non-official copy: `https://github.com/dreal-deps/mathlib`.

CRLIBM (Muller et al., 2004-2006)

Provides `exp`, `expm1`, `log`, `log1p`, `log2`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `sinpi`, `cospi`, `tanpi`, `atanpi`, and `pow` (incomplete).

All 4 rounding modes: `exp_rn`, `exp_rz`, `exp_ru`, `exp_rd`.

Assumes rounding precision to double, and rounding mode to nearest-even (`crlibm_init`).

- use of modern instructions (FMA)
- knowledge of HR cases $\Longrightarrow$ better tuning of slow path
- use of triple-double arithmetic

For double-precision `exp`, [6] reports a max/avg ratio of 6500 for MathLib, against only 6.6 for CR-LIBM.

[6] Daramy-Loirat, Defour, de Dinechin, Gallet, Gast, Lauter, Muller, CR-LIBM: A library of correctly rounded elementary functions in double-precision, 2006.

RLIBM (2020-)

Santosh Nagarakatte, M. Aanjaneya, J.P. Lim, S. Park.

Provides binary32 `cosh`, `cospi`, `exp`, `exp10`, `exp2`, `log`, `log10`, `log2`, `sinh`, `sinpi`.

All rounding modes (in RLIBM-ALL).

Provides not only IEEE binary formats, but also posits.

Use new approach based on linear programming, to find polynomials that yield correct rounding.

Does this approach scale for binary64?

`https://people.cs.rutgers.edu/~sn349/rlibm/`

LLVM-libc (2022-)

Authors: Tue Ly, Siva Chandra, Kirill Okhotnikov.

Supported by Google.

Goal is to provide only correctly-rounded routines.

LLVM 14.0.6 already provides (correctly-rounded) binary32 `log`, `log10`, `log2`, `hypot`, and binary64 `hypot`.

Reserved Names

The current draft of the C2x standard (N3047) contains (page 451):

> *Function names that begin with* `cr_` *are potentially reserved identifiers and may be added to the* `<math.h>` *header. The* `cr_` *prefix is intended to indicate a correctly rounded version of the function.*

C2x also contains new functions: `exp2m1`, `exp10m1`, `log2p1`, `rsqrt`, `sinpi`, `cospi`, `tanpi`, `asinpi`, `acospi`, `atanpi`, `atan2pi`.

The CORE-MATH methodology and expertise

Compute exact, midpoint and HR cases (including for bivariate binary32 functions): BaCSeL software tool.

Implement a quick phase with about 10-15 extra bits wrt the target precision with small or no tables and optimal minimax polynomials: Sollya software tool.

Analyze the maximum error of the quick phase and deduce the rounding test bound: tight error analysis.

Tune the accurate phase accuracy with knowledge of the HR cases, with some exceptional inputs if needed.

Check correctness on the exact, midpoint and HR cases, for all rounding modes: GNU MPFR software tool.

HR cases for `atan2`

y, x is a HR case for `atan2` iff $\mathrm{atan}(y/x)$ is near a 25-bit number z in the binary32 range.

Thus y/x is near $\tan z$.

Algorithm:

- for each 25-bit number z in the binary32 range:
- compute the continued fraction of $\tan z$
- take the largest convergent y/x such that both y and x are representable on 24 bits
- then y, x is a hard-to-round case for `atan2`

Worst non-trivial case $(x, y \neq 2^k)$ is:

$$\mathtt{atan2}(\text{0x1.3ee9f4p}+37, \text{0x1.7e87d2p}+23) = \text{0x1.921ae9}{\color{red}\text{00000000000000}}\text{8b4...p}+0$$

Methodology in action: binary64 cube root

- computation of exact cases
- computation of HR-cases
- fast path
- accurate path

Binary64 cube root: exact cases

We want

$$y^3 = x$$

with both x and y binary64 numbers.

Wlog, we can assume $1 \leq y < 2$.

Write $y = m \cdot 2^e$ with m odd.

Necessarily $m \leq \lfloor 2^{53/3} \rfloor = 208063$, otherwise m^3 does not fit into 53 bits.

Total 104032 exact cases.

Remark: output of cbrt is never in the subnormal range.

Note: we deal with exact cases after the rounding test, outside the critical path.

Binary64 cube root: hard-to-round cases

We used the BaCSeL software tool.

Wlog, we can restrict to $1/2 \leq x < 4$.

We search inputs with at least 44 identical bits after the round bit.

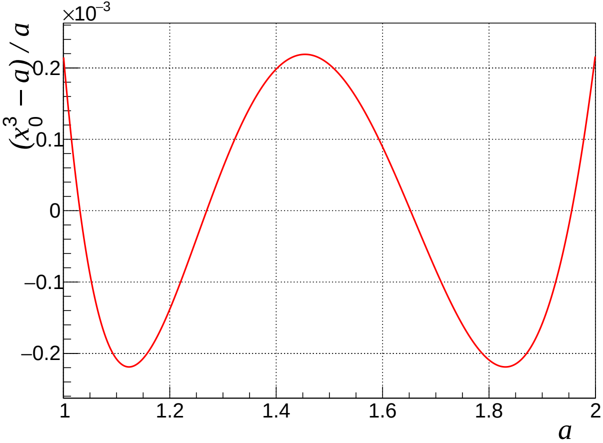
Real time about 2 hours per binade on a 112-core E7-4850 at 2.2Ghz.

We found 1496 such inputs: 491 in $[1/2, 1)$, 501 in $[1, 2)$, 504 in $[2, 4)$.

$$\texttt{cbrt(0x1.9b78223aa307cp+1)} = \text{0x1.79d15d0e8d59b8}\textcolor{red}{\text{00000000000000}}\text{ffc}\ldots$$

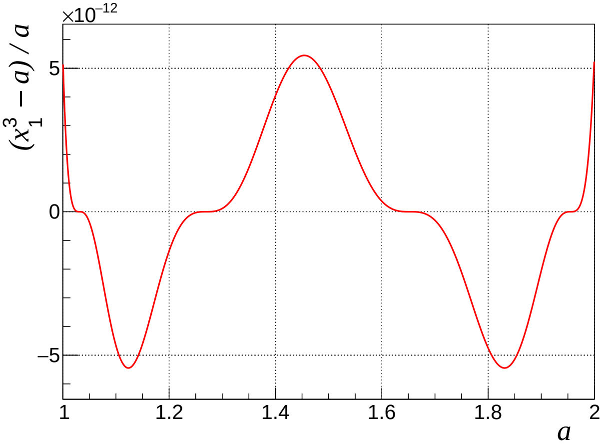
Binary64 cube root: fast path (1/3)

- scale a to $[1, 2)$
- compute an initial 3rd-order minimax approximation x_0 with rel. error $< 0.3 \cdot 10^{-3}$



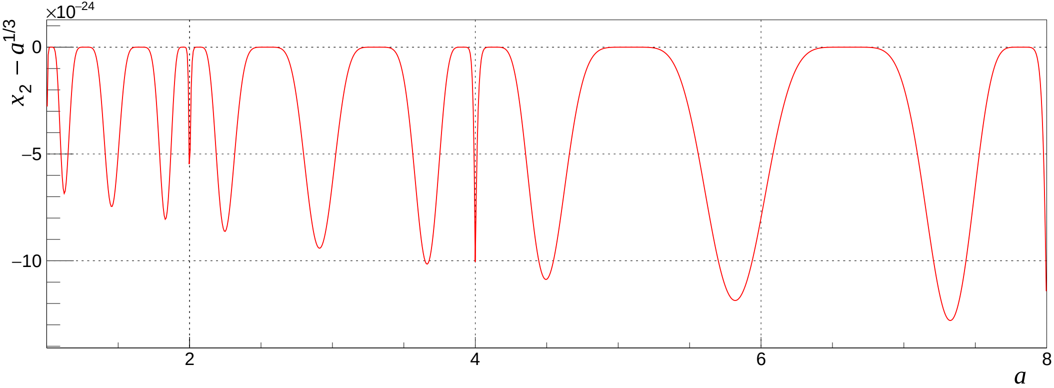
Binary64 cube root: fast path (2/3)

- scale a to $[1, 2)$
- compute an initial 3rd-order minimax approximation x_0 with rel. error $< 0.3 \cdot 10^{-3}$
- perform a Newton iteration of order 3 to get x_1 with relative error $< 6 \cdot 10^{-12}$



Binary64 cube root: fast path (3/3)

- scale a to $[1, 2)$
- initial 3rd-order minimax approximation x_0 with rel. error $< 0.3 \cdot 10^{-3}$ (double)
- Newton iteration of order 3 to get x_1 with rel. error $< 6 \cdot 10^{-12}$ (double)
- Newton iteration of order 2 to get x_2 with rel. error $< 1.32 \cdot 10^{-23}$ (double-double)



Binary64 cube root: fast path

At the end of the fast path, $a^{1/3}$ is approximated by $x_2 := x_2^{\mathrm{high}} + x_2^{\mathrm{low}}$.

Lemma

Whatever the rounding mode, we have $|x_2^{\mathrm{low}}| < 2^{-52}$.

Maximal error $|a^{1/3} - x_2| < 1.32 \cdot 10^{-23} < 2^{-76}$.

Round to nearest: check

$$||x_2^{\mathrm{low}}| - 2^{-53}| > 2^{-76}$$

where $2^{-53} = 1/2\mathrm{ulp}(x_2^{\mathrm{high}})$.

Directed modes:

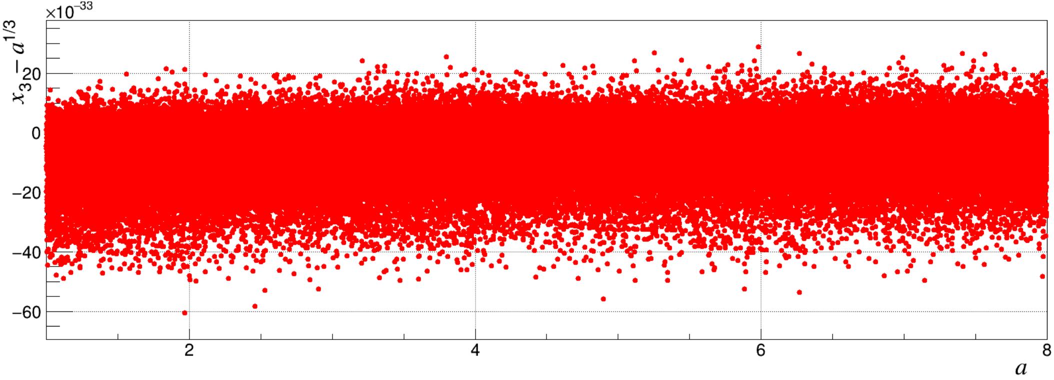
$$|x_2^{\mathrm{low}}| > 2^{-76} \quad \text{and} \quad |x_2^{\mathrm{low}} - 2^{-52}| > 2^{-76}$$

Probability of the accurate path $< 2^{-76}/2^{-52} = 2^{-24}$.

Binary64 cube root: accurate path

The value x_2 at the end of the fast path has about 76 correct bits.

We perform another Newton iteration in double-double, and get x_3 with about 104 bits of accuracy.



Binary64 cube root: accurate path

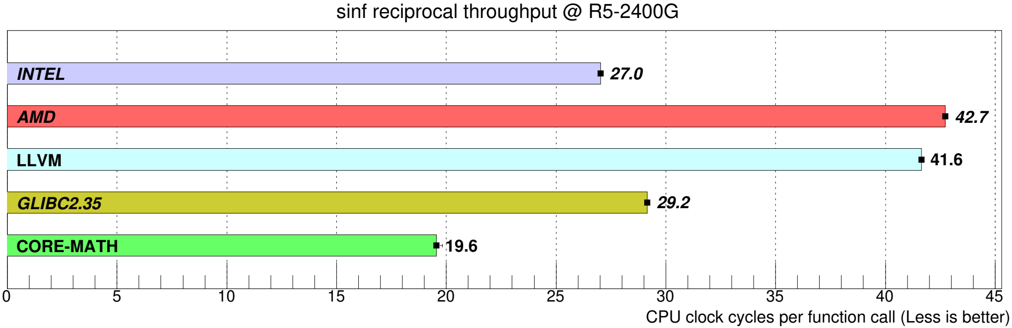
For a few hard-to-round cases, the accurate path does not return a correctly rounded value.

We hardcode the correct result for them (only 9 values), where z is the input value reduced to $[1, 8)$:

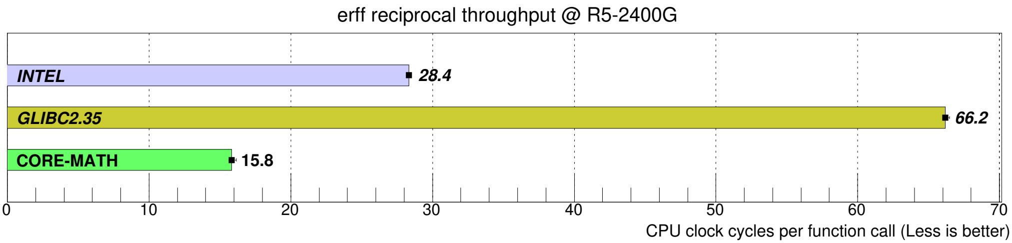
```
if (abs(z) == 0x1.9b78223aa307cp+1)
    y = copysign (0x1.79d15d0e8d59cp+0, z);
```

Performance comparison: binary32 sine function

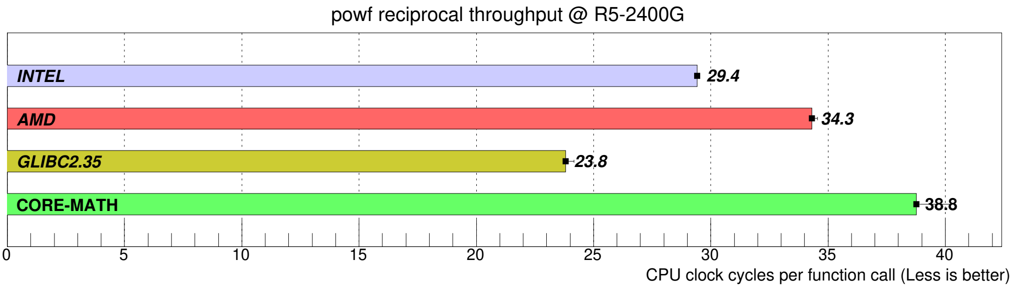
Intel Math Library from icx 2021.1, AMD libm 3.9, LLVM 14.0.6, GNU libc 2.35, CORE-MATH a9d7d84



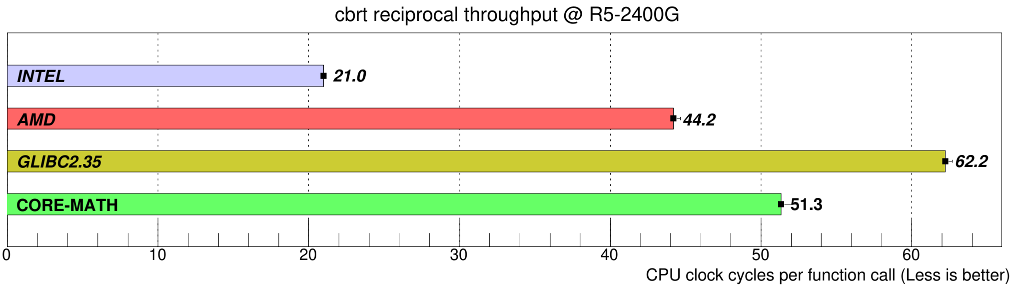
Binary32 error function



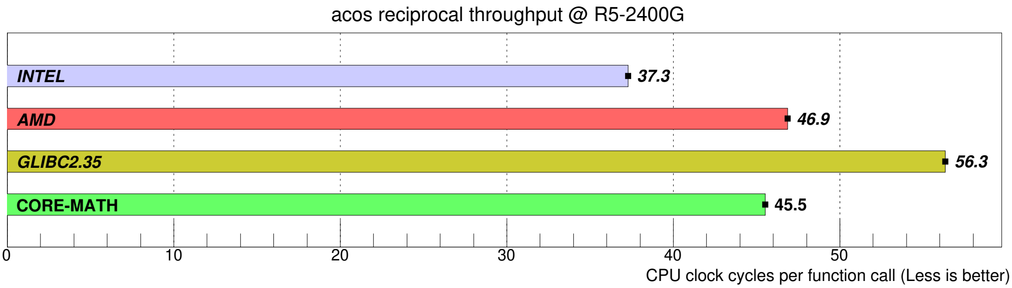
Binary32 power function



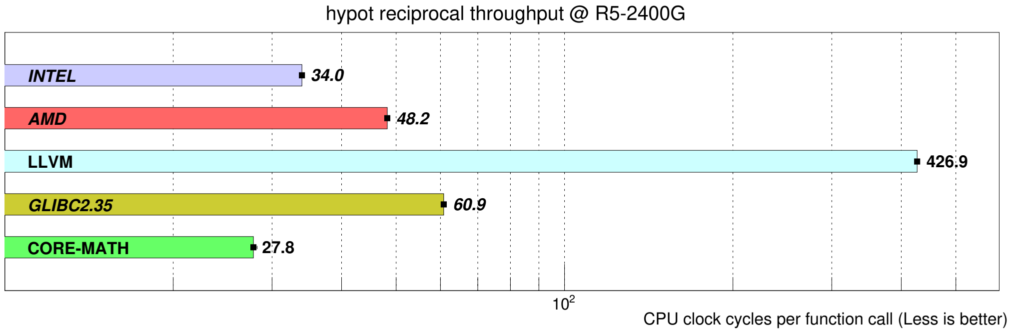
Binary64 cube root



Binary64 arc-cosine



Binary64 hypot



Current progress

See `https://core-math.gitlabpages.inria.fr/`

MIT license to make integration easier.

binary32: all C99 functions implemented plus `exp10`

binary64: `acos`, `cbrt`, `exp`, `exp2`, `hypot`. In review: `asin`, `exp10`, `pow`

Conclusion

We present for the first time implementations of correctly rounded routines as fast as in the best current math libraries or even faster.

Full set of C99 binary32 functions ready for integration (either as `expf` or `cr_expf`).

Full set of C99 binary64 functions planned for end of 2023.

Time for IEEE-754 to require correct rounding!

Not yet another libm, but aimed at integration in existing libms.

If you want CORE-MATH routines to be optimized for your particular hardware, please contact us.

The CORE-MATH perf.sh tool

On a AMD EPYC 7282:

```
$ ./perf.sh atanf
GNU libc version: 2.36
GNU libc release: stable
17.592 # CORE-MATH
31.617 # GNU libc 2.36
```

```
$ PERF_ARGS=--latency ./perf.sh atanf
61.456 # CORE-MATH
72.358 # GNU libc 2.36
```

```
$ LIBM=libllvmlibc-14.0.6.a ./perf.sh log10f
10.754 # CORE-MATH
18.513 # GNU libc 2.36
9.952  # LLVM 14.0.6
```