Towards a correctly-rounded and fast power function in binary64 arithmetic

Tom Hubrecht, Claude-Pierre Jeannerod, Paul Zimmermann

Arith 2023 conference, Portland, September 5th

```
$ cat test.c
#include <stdio.h>
#include <math.h>
int main () {
   double x = 0x1.0fbcd29b56829p-1;
   double y = 0x1.57347b643eac2p-1;
   printf ("x^y = %la\n", pow (x, y)); }
$ cc -fno-builtin test.c -lm
```

If you get `0x1.4ed9af3ea18c3p-1`, maybe you are using the GNU libc.

If you get `0x1.4ed9af3ea18c4p-1`, maybe you are using the Intel math library.

(The correctly rounded result is that of GNU libc in that case, but for $x =$ `0x1.30b3e414e3d3bp-1` and $y =$ `0x1.a6c0a38da8066p-1` we get the converse.)

Previous Work

- books of Markstein and Beebe
- MathLib library (Ziv, 1991)
- LIBMCR (Sun, 2004)
- CRLIBM (De Dinechin, Lauter and al., 2006)
- more recently RLIBM and LLVM libc, but do not provide (yet) a binary64 power function

MathLib

Only for rounding to nearest even.

Integrated in GNU libc, slow path removed after 2.27 (2018).

Algorithm is not detailed.

No longer maintained.

Non official copies still exist, no bug found.

LIBMCR

Only for rounding to nearest even.

Algorithm is not detailed.

pow does not terminate for some inputs, for example `x=0x1.470574d68e0afp+1`, `y=0x1.02e0706205c0ep+1`.

Wrong results for some inputs, for example `x=0x1.f80b060553772p-1` and `y=0x1.99cp+13`, gives `0x1.00001p+0` instead of `0x1.a2e7cca9cfd72p-297` (49 identical bits after the round bit).

No longer maintained.

CRLIBM

Experimental, and only for rounding to nearest even.

Algorithm partially detailed in Lauter's PhD thesis.

Wrong result for `x=0x1.524ebae943097p+1` and `y=0x1.ep-2`, gives -5 instead of `0x1.93bd0cd47eb5fp+0` (64 identical bits after the round bit).

No longer maintained.

Our contribution

- an open-source implementation, integrated into CORE-MATH
- covers all IEEE 754 rounding modes, not only to nearest-even
- an extended paper with full proofs of the 1st phase
- full details of the 2nd and 3rd phases with the code
- performance: outperforms previous work, not far from incorrectly rounded math libraries

Performance

Intel Core i7-8700 cycles with GCC 12.2.0, rounding to nearest-even:

	GNU libc 2.36	MathLib	LIBMCR	CRLIBM	this work
rec. throughput	*43*	123	256	211	66 (60)
latency	*79*	166	285	275	111 (99)

How to compute x^y?

Using $x^y = 2^{y \log_2 x}$ (Markstein, Beebe) looks quite interesting, but the Taylor expansions of $\log_2(1+t)$ and 2^t at $t = 0$ have less simple coefficients.

We prefer to use $x^y = e^{y \log x}$, with more complex argument reduction, but simpler Taylor expansions.

Exact and midpoint cases

x^y is exact if exactly representable in binary64.

x^y is midpoint if exactly representable with 54 bits (but not with 53).

Detecting and dealing with exact/midpoint cases is crucial.

Efficient algorithms from Lauter and Lefèvre (2009).

Workflow

1. 1st phase computes double-double approximation; if rounding test succeeds, return result
2. 2nd phase computes approximation with 128-bit arithmetic; if rounding test succeeds, return result; otherwise, check for exact/midpoint cases
3. 3rd phase computes approximation with 256-bit arithmetic; if rounding test succeeds, return result; otherwise, print error message

Notations

We denote by $\circ(a+b)$ or $\circ(a \cdot b)$ the correct rounding (with the current rounding mode) of $a+b$ and $a \cdot b$ respectively.

We denote by $\circ(a \cdot b + c)$ the correct rounding of $a \cdot b + c$, which can be efficiently computed with an FMA (fused-multiply-add) if available in hardware, or emulated in software (`__builtin_fma`).

FastTwoSum

```
Input: $a, b \in \mathbb{F}$ with $a = 0$ or $|a| \geq |b|$
Output: $h, \ell$ such that $h + \ell$ approximates $a + b$
  1: $h \leftarrow \circ(a + b)$
  2: $t \leftarrow \circ(h - a)$                    ▷ always exact
  3: $\ell \leftarrow \circ(b - t)$
```

Theorem (Zimmermann, 2023)

Whatever the IEEE 754 rounding mode, in the absence of underflow/overflow, the output h, ℓ of FastTwoSum satisfies

$$|h + \ell - (a + b)| \leq 2^{-105}|h|,$$

and if the exponents of a, b differ by at most 53, $h + \ell = a + b$.

FastSum

```
Input: $a, b_h, b_\ell \in \mathbb{F}$ with $a = 0$ or $|a| \geq |b_h|$
Output: $h, \ell$ such that $h + \ell$ approximates $a + b_h + b_\ell$
1: $h, t \leftarrow \mathrm{FastTwoSum}(a, b_h)$
2: $\ell \leftarrow \circ(t + b_\ell)$
```

Lemma

In no underflow nor overflow, the pair (h, ℓ) computed by Algorithm FastSum satisfies

$$\left| h + \ell - (a + b_h + b_\ell) \right| \leq 2^{-105} |h| + \mathrm{ulp}(\ell).$$

Approximation of $\log(1+z)$ around zero

Algorithm 1 Algorithm p_1

```
Input: $z \in \mathbb{F}$, $|z| \le 33 \cdot 2^{-13}$
Output: $p_h + p_\ell$ approximating $\log(1+z) - z$
1: $w_h, w_\ell \leftarrow \mathrm{ExactMul}(z, z)$
2: $t \leftarrow \circ(\texttt{0x1.0001f0c80e8cep-3} \cdot z + \texttt{0x1.2494526fd4a06p-3})$
3: $u \leftarrow \circ(\texttt{0x1.55555553d1eb4p-3} \cdot z + \texttt{0x1.999999981f535p-3})$
4: $v \leftarrow \circ(\texttt{0x1.0000000000003p-2} \cdot z + \texttt{0x1.5555555555558p-2})$
5: $u \leftarrow \circ(tw_h + u)$
6: $v \leftarrow \circ(uw_h + v)$
7: $u \leftarrow \circ(vw_h)$
8: $p_h \leftarrow -0.5 \cdot w_h$
9: $p_\ell \leftarrow \circ(uz - 0.5 \cdot w_\ell)$
```

Lemma

Given $|z| \leq 33 \cdot 2^{-13}$ *with* z *an integer multiple of* 2^{-61}, *the double-double approximation* $p_h + p_\ell$ *to* $\log(1+z) - z$ *returned by Algorithm p_1 satisfies*

$$|p_h + p_\ell - (\log(1+z) - z)| < 2^{-75.492},$$

with $|p_h| < 2^{-16.9}$ *and* $|p_\ell| < 2^{-25.446}$. *If* $z \neq 0$, *and assuming further* $|z| \leq 32 \cdot 2^{-13}$, *the relative error satisfies*

$$\left| \frac{z + p_h + p_\ell}{\log(1+z)} - 1 \right| < 2^{-67.441}.$$

Proved using the Coq proof assistant (by Laurence Rideau and Laurent Théry).

Approximation of $\log x$

Algorithm 2 Algorithm log_1

```
Input: a binary64 value $x \in [\alpha, \Omega]$
Output: $h + \ell$ approximating $\log x$
 1: write $x = t \cdot 2^e$ with $t \in (1/\sqrt{2}, \sqrt{2})$ and $e \in \mathbb{Z}$
 2: $i \leftarrow \lfloor 2^8 t \rfloor$                                ▷ $i$ integer, $181 \le i \le 362$
 3: $r \leftarrow \mathrm{INVERSE}_i, \quad \ell_1, \ell_2 \leftarrow \mathrm{LOGINV}_i$
 4: $z \leftarrow \circ(rt - 1)$                                  ▷ exact, $|z| \le 33 \cdot 2^{-13}$
 5: $t_h \leftarrow \circ(e\,\mathrm{LOG2H} + \ell_1)$
 6: $t_\ell \leftarrow \circ(e\,\mathrm{LOG2L} + \ell_2)$
 7: $h, \ell \leftarrow \mathrm{FastSum}(t_h, z, t_\ell)$
 8: $p_h, p_\ell \leftarrow$ p_1$(z)$
 9: $h, \ell \leftarrow \mathrm{FastSum}(h, p_h, \circ(\ell + p_\ell))$
10: if $e = 0$ and $|\ell| > 2^{-24}|h|$ then $h, \ell \leftarrow \mathrm{FastTwoSum}(h, \ell)$
```

Lemma

Given $x \in [\alpha, \Omega]$, Algorithm log_1 computes (h, ℓ) such that $|\ell| \leq 2^{-23.89}|h|$, and

$$|h + \ell - \log x| \leq \varepsilon_{\log} \cdot |\log x| \tag{1}$$

with $\varepsilon_{\log} = 2^{-73.527}$ if $x \notin (1/\sqrt{2}, \sqrt{2})$, and $\varepsilon_{\log} = 2^{-67.0544}$ otherwise.

Proved using the Coq proof assistant (by Laurence Rideau and Laurent Théry).

Multiplication by y

Input: a double-double value $h+\ell$, and a double y
Output: $r_h + r_\ell$ approximating $y(h+\ell)$

```
1: $r_h, s \leftarrow \mathrm{ExactMul}(y, h)$
2: $r_\ell \leftarrow \circ(y\ell + s)$
```

Lemma

If x, h, ℓ are as in previous Lemma and if $2^{-969} \le |yh| \le 709.7827$, then (r_h, r_ℓ) satisfy $|r_h| \in [2^{-970}, 709.79]$, $|r_\ell| \le 2^{-14.4187}$, $|r_\ell/r_h| \le 2^{-23.8899}$, $|r_h + r_\ell| \le 709.79$, and

$$|r_h + r_\ell - y \log x| \le \varepsilon_{mul}$$

with $\varepsilon_{mul} = 2^{-63.799}$ if $x \notin (1/\sqrt{2}, \sqrt{2})$, and $\varepsilon_{mul} = 2^{-57.580}$ otherwise.

Final exponentiation

See proceedings for details of Algorithm exp_1.

Lemma

In the case $\rho_1 \le r_h \le \rho_2$*, if* $|r_\ell / r_h| < 2^{-23.8899}$ *and* $|r_\ell| < 2^{-14.4187}$*, then the value* $e_h + e_\ell$ *returned by Algorithm exp_1 satisfies*

$$\left| \frac{e_h + e_\ell}{\exp(r_h + r_\ell)} - 1 \right| < 2^{-74.16}.$$

Moreover, $|e_\ell / e_h| \le 2^{-41.7}$.

Main algorithm for 1st phase

Input: two binary64 numbers x, y with $x > 0$
Output: the correct rounding of x^y, or FAIL

```
1: $\ell_h, \ell_\ell \leftarrow \mathrm{log\_1}(x)$
2: $r_h, r_\ell \leftarrow \mathrm{mul\_1}(\ell_h, \ell_\ell, y)$
3: $e_h, e_\ell \leftarrow \mathrm{exp\_1}(r_h, r_\ell)$
4: if $\sqrt{2}/2 < x < \sqrt{2}$ then $\varepsilon \leftarrow \mathrm{RU}(2^{-57.579})$ else $\varepsilon \leftarrow \mathrm{RU}(2^{-63.797})$
5: $u \leftarrow \circ(e_h + \circ(e_\ell - \varepsilon e_h))$, $v \leftarrow \circ(e_h + \circ(e_\ell + \varepsilon e_h))$
6: if $u = v$ then return $u$ else return FAIL
```

Theorem

The value returned by Algorithm phase_1 (if not FAIL) is correctly rounded.

Main achievements

Our algorithms work for all IEEE 754 rounding modes.

Heavy use of FMA (either in hardware or emulated).

Crucial usage of FastTwoSum, with error bound for directed rounding modes.

Hardest to round inputs are not known for pow: 3rd phase might return an error message, but will not deliver any incorrectly-rounded result.

Code 2x to 4x faster than previous work.

All IEEE 754 specials cases are dealt with.

Very detailed proofs.

A major achievement since pow is the most difficult binary64 function.

Further work

The algorithms and code can be improved further (and we work on it).

Formal proof of the 1st phase in progress, thanks to Laurence Rideau and Laurent Théry, using the Coq proof assistant.

Simplify the formal proofs using tools presented by Paul Geneau de Lamarlière on Monday.

Approach scales for double-extended and quadruple precision.

An important step towards requiring correct rounding in IEEE 754-2029?

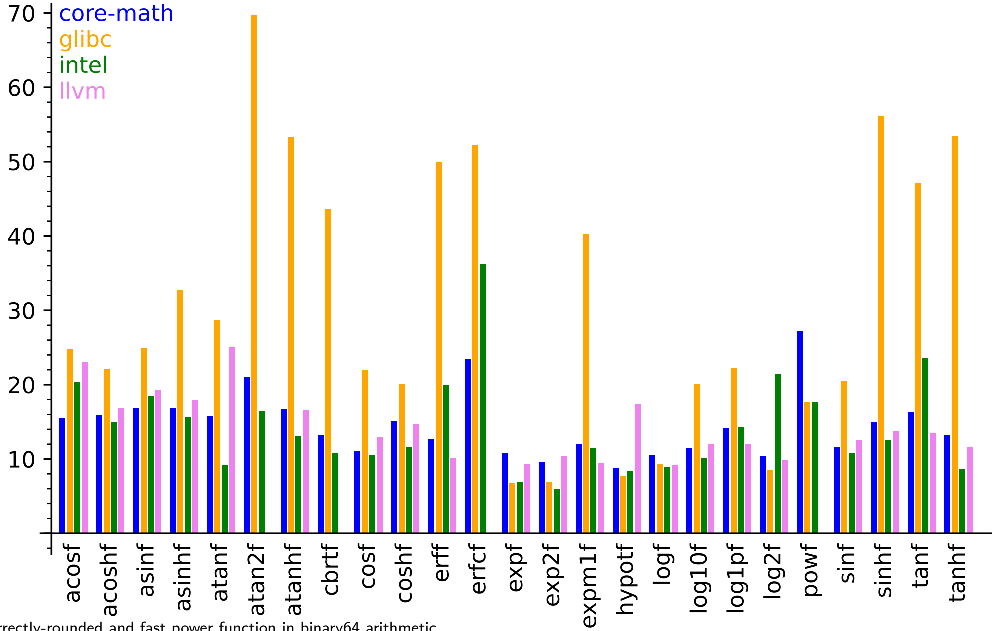
CORE-MATH timings

The following slides give reciprocal throughput and latency of CORE-MATH (commit 81d5ea0), GNU libc 2.37, the Intel math library with icx 2023.2.0 (with `-fp-model=strict -fno-fast-math`), and LLVM libc (revision d099dbb).

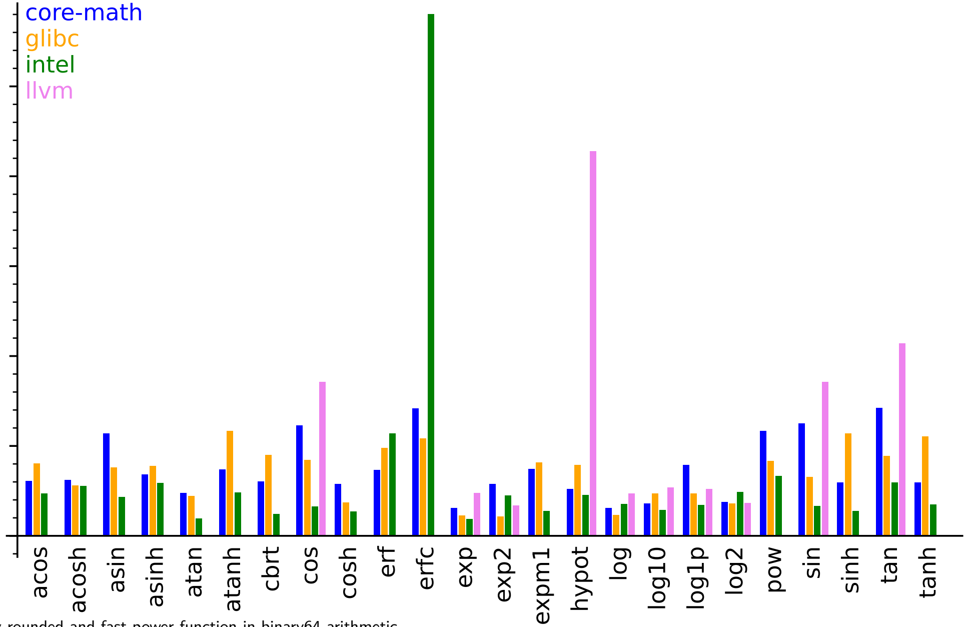
Hardware: Intel Xeon Silver 4214.

Compiler (except for Intel math library): gcc 13.2.0.

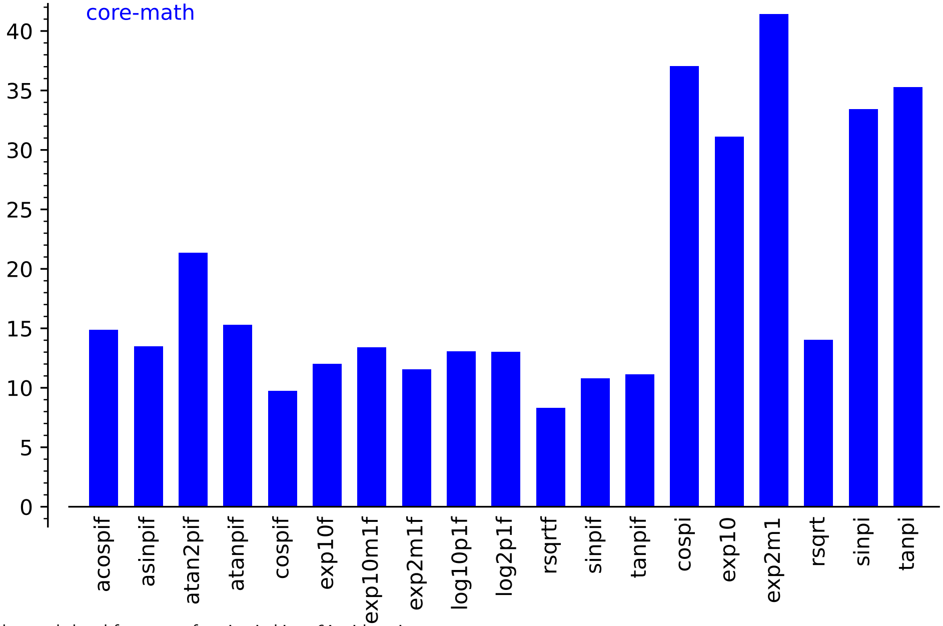
reciprocal throughput of C99 binary32 functions



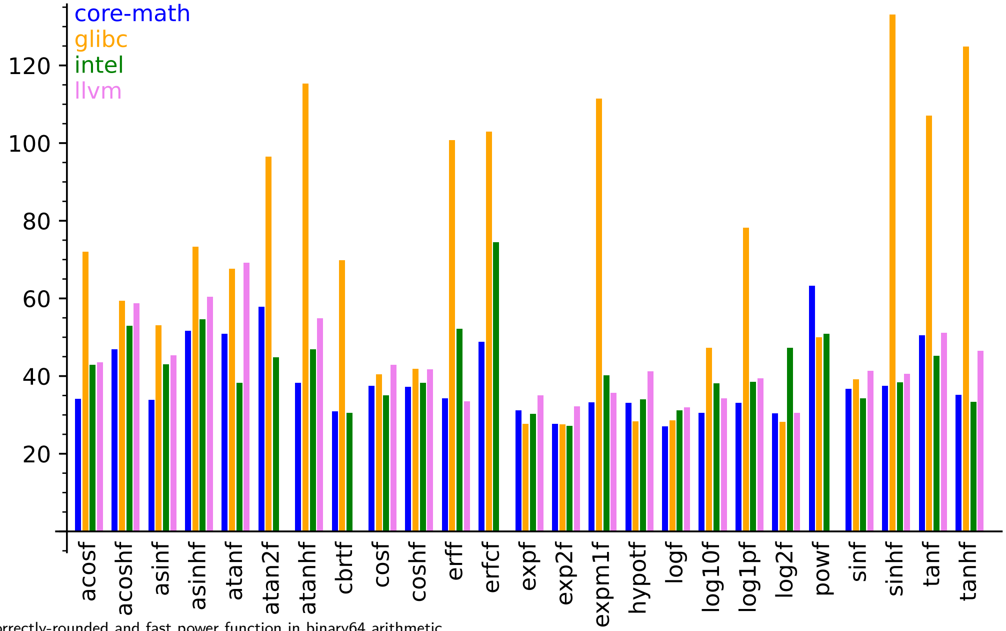
reciprocal throughput of C99 binary64 functions

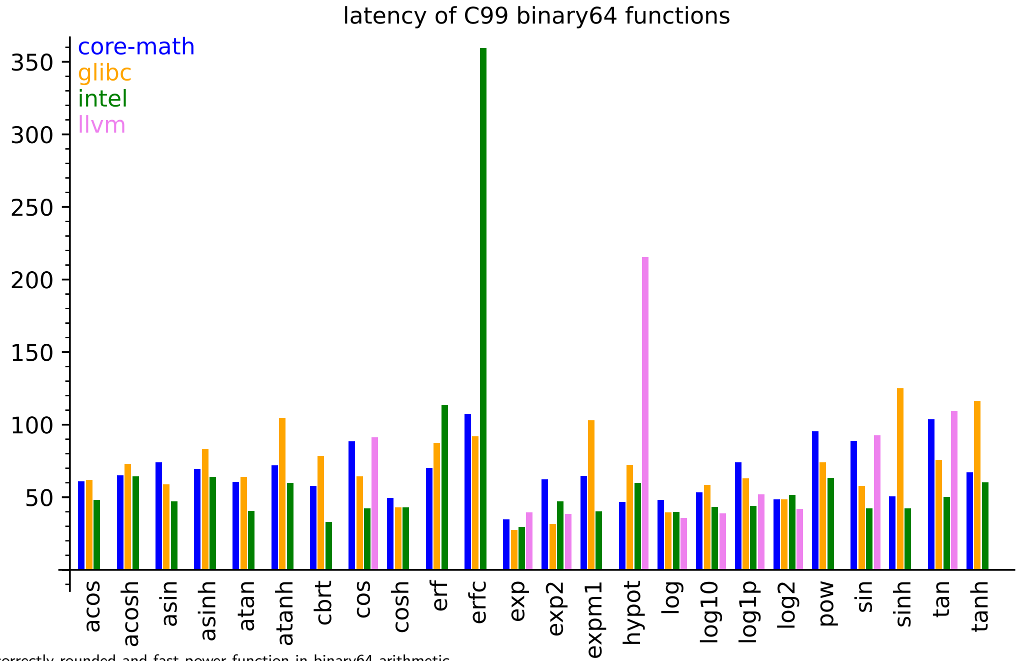


reciprocal throughput of new C23 functions

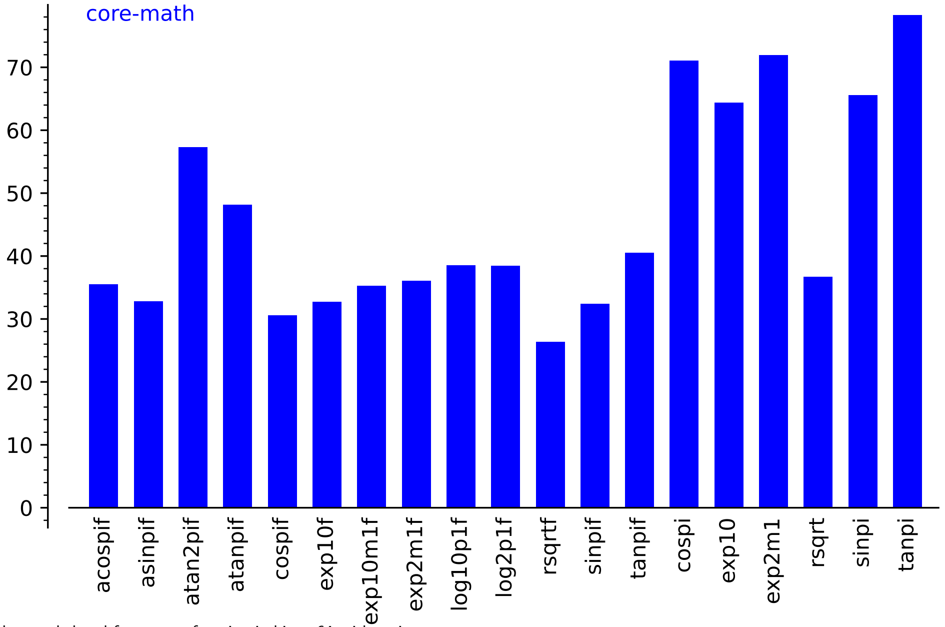


latency of C99 binary32 functions

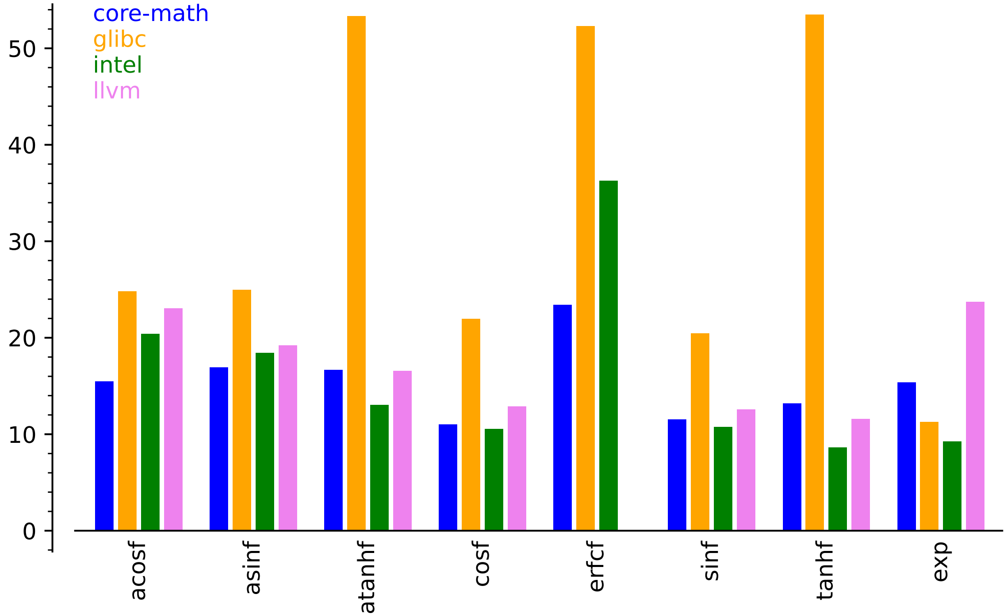



latency of C99 binary64 functions
core-math
glibc
intel
llvm
350
300
250
200
150
100
50
acos
acosh
asin
asinh
atan
atanh
cbrt
cos
cosh
erf
erfc
exp
exp2
expm1
hypot
log
log10
log1p
log2
pow
sin
sinh
tan
tanh

latency of new C23 functions



reciprocal throughput of selected functions


core-math
glibc
intel
llvm
acosf
asinf
atanhf
cosf
erfcf
sinf
tanhf
exp