# How Slow is Quadruple Precision?

Paul Zimmermann, INRIA, Nancy, France

May 7, 2020

ICERM Workshop on Variable Precision in Mathematical and Scientific Computing

**Workshop Abstract:** [...] Exascale computing has also exposed the need for even greater precision than IEEE 64-bit double in some cases, because greatly magnified numerical sensitivities often mean that one can no longer be certain that results are numerically reliable. One remedy is to use IEEE 128-bit quad precision in selected portions of the computation, which is now available via software in some compilers, notably the gfortran compiler. As a single example, researchers at Stanford have had remarkable success in using quad precision in multiscale linear programming applications in biology. [...]

# Plan of the Talk

- the IEEE-754 binary128 format
- a toy example: the double pendulum
- can we do better?
- conclusion and perspectives

# The IEEE-754 Binary128 Format

Encoding:

| sign $s$ | exponent $e$ | significand $m$ |
|---|---|---|
| 1 bit | 15 bits | 112 bits |

Decoded value (except special numbers):

$$x = (-1)^s \cdot 2^{e-16383} \cdot (1 + \frac{m}{2^{112}})$$

Smallest absolute value: $x_{\mathrm{min}} \approx 6.5 \cdot 10^{-4966}$

Largest absolute value: $x_{\mathrm{max}} \approx 5.9 \cdot 10^{4931}$

Accuracy about 34 decimal digits.

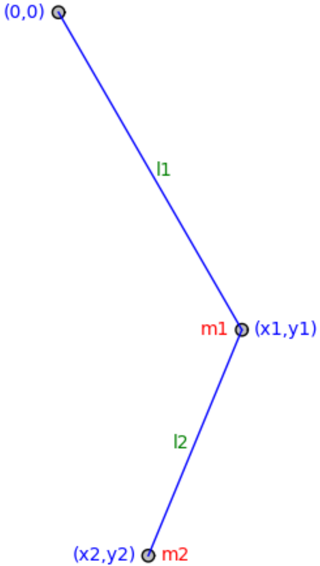
# Hardware and Software Support

Currently, only the IBM Power9 supports binary128 in hardware.

Several compilers/libraries support binary128 in software:

- **GNU libc/libquadmath** (`_Float128`);
- **Intel Math library** (`_Quad`);
- Berkeley's SoftFloat by John Hauser;
- Oracle Studio;
- ASquadmath by Alexei Sibidanov (not publicly available).

# Example: the Double Pendulum




$\theta_1$ ($\theta_2$): angle of the 1st (2nd) pendulum wrt the vertical axis

$$u = {\theta_2'}^2 \ell_2 + {\theta_1'}^2 \ell_1 \cos(\theta_1 - \theta_2)$$

$$v = g(2m_1 + m_2) \sin\theta_1$$

$$w = m_2 g \sin(\theta_1 - 2\theta_2)$$

$$x = {\theta_1'}^2 \ell_1 (m_1 + m_2)$$

$$y = g(m_1 + m_2) \cos\theta_1$$

$$z = {\theta_2'}^2 \ell_2 m_2 \cos(\theta_1 - \theta_2)$$

$$d = 2m_1 + m_2 - m_2 \cos(2\theta_1 - 2\theta_2)$$

$$\theta_1'' = \frac{-v - w - 2\sin(\theta_1 - \theta_2) m_2 u}{\ell_1 d}$$

$$\theta_2'' = \frac{2\sin(\theta_1 - \theta_2)(x + y + z)}{\ell_2 d}$$

# Testing Framework

Pendulum lengths: $\ell_1 = \ell_2 = 1$.

Masses: $m_1 = m_2 = 2$.

Acceleration due to gravity: $g = 9.81$.

Initial conditions: $\theta_1(0) = \theta_2(0) = \pi/2$, with $\theta_1'(0) = \theta_2'(0) = 0$.

Integration time: 20 seconds.

Method: Euler's scheme, with 50 000 steps per second ($h = 1/50000$):

$$\begin{aligned} \theta_i'(t+h) &= \theta_i'(t) + h\theta_i''(t) \\ \theta_i(t+h) &= \theta_i(t) + h\theta_i'(t) \end{aligned}$$

Source code:

`http://www.loria.fr/~zimmerma/double_pendulum.html`

# Performance Comparison

miriel038.plafrim.cluster: Intel Xeon E5-2680, 2.50GHz

Ratio to the reference time of glibc/double (220ms):

| | gcc 9.2.0<br>glibc 2.17 | icc 19.0.4.243<br>gcc version 9.2.0 compat. |
|---|---|---|
| single | 0.5 | 0.4 [1] |
| double | **1** | 0.5 |
| quadruple | 62 [2] | 10 [3] |

[1] results differ with optimization level 0 ($x_2 = -0.654694$, $y_2 = 0.631660$), level 1 ($x_2 = -1.343469$, $y_2 = 0.625392$), and levels 2 or 3 ($x_2 = -1.182620$, $y_2 = 0.601759$)

[2] time extrapolated on another machine

[3] compiled with `-Qoption,cpp,-extended_float_types`

# What About Accuracy?

Tested with `mpcheck` (`mpcheck.gforge.inria.fr`) based on GNU MPFR. $10^6$ random tests. Rounding to nearest. Error in ulps.

| function | glibc 2.31 | icc 19.0.4.243 |
|---|---|---|
| exp | 0.501 | 0.501 |
| log | 0.871 | 0.501 |
| log2 | 2.14 | 0.501 |
| log10 | 1.43 | 0.501 |
| sin | 1.27 | 0.501 |
| atan | 1.09 | 0.501 |
| acos | 1.13 | 0.501 |
| sinh | 1.83 | 0.501 |
| tanh | 2.30 | 0.501 |
| acosh | 3.24 | 0.501 |
| tgamma | 4.70 | 4090 [1] |

[1] bug reported, for $x = 0x3.08e1f38ddd769117414bf11b45dcp+8$

If we replace all calls to sinf128 by the following (same for cosf128):

```
static _Float128 my_sinf128 (_Float128 x)
{
  return (_Float128) 0.5;
}
```

the total time is divided by 18.1 with glibc, by 7.1 with the Intel Math Library.

Conclusion: the main bottleneck are the mathematical functions.

# Can We Do Better?

On our double pendulum example, quadruple precision is 20 times slower than double precision with the Intel Math Library, and 62 times with the GNU library.

Can we do better?

Challenge: implement a fast `exp` function in quadruple precision.

Target processor: `x86_64`.

# Exercise: Implement `expf128` for `x86_64`

The GNU libm takes on average about 3200 cycles.

The Intel Math Library takes on average 280-430 cycles.

Goal: save a factor of 10 over the GNU libm.

Everything is allowed.

Accuracy constraint: should be about as accurate as the glibc function.

Time constraint: at most one week of design/coding/testing (March 23-27, 2020).

Principle 1: avoid all operations on `_Float128`, even addition and multiplication.

Instead, extract the `_Float128` input into a special `binary128` structure, do all computations on `binary128`, and unpack at the end.

# The binary128 structure

Encoding:

| sign $s$ | exponent $e$ | $m_0$ | $m_1$ |
|---|---|---|---|
| `int` | `int` | `uint64_t` | `uint64_t` |
| $s \in \{-1, 1\}$ | $-16493 \le e \le 16383$ | $m_0 < 2^{64}$ | $m_1 < 2^{64}$ |

Decoding:

$$x = s \cdot 2^e \cdot \left( \frac{m_1}{2^{64}} + \frac{m_0}{2^{128}} \right)$$

Encoding similar to GNU MPFR, with no implicit bit.

No systematic normalization ($m_1$ can be smaller than $2^{63}$).

Corollary: we get $128 - 113 = 15$ extra bits of accuracy.

# Algorithm for binary128 exponential

1. extract $x$ into a `binary128` structure, say $y$
2. check for special values, overflow, underflow
3. write $y = i \log 2 + j \log 2 \cdot 2^{-8} + k \log 2 \cdot 2^{-16} + r$ with $-128 \le j, k < 128$ and $|r| \le \log 2 \cdot 2^{-17}$
4. $y \leftarrow y - i \log 2$
5. $y \leftarrow y - u_j - v_k$ $\qquad u_j \approx j \log 2 \cdot 2^{-8}$, $v_k \approx k \log 2 \cdot 2^{-16}$
6. $e_{jk} \leftarrow f_j \cdot g_k$ $\qquad f_j \approx \exp(u_j)$, $g_k \approx \exp(v_k)$
7. now $|y| \le \log 2 \cdot 2^{-17}$
8. $z \leftarrow y(p_4 + y(p_5 y + p_6))$ $\qquad$ [64-bit arithmetic only]
9. $z \leftarrow p_1 + y(p_2 + y(p_3 + z))$
10. $y \leftarrow e_{jk} + y \cdot z \cdot e_{jk}$
11. return $\mathrm{unpack}(y, i)$ $\qquad$ [multiplies by $2^i$]

The coefficients $p_1, p_2, ..., p_6$ were generated by the Sollya tool.

$$p(x) = 1 + p_1 x + p_2 x^2 + \cdots + p_6 x^6$$

They minimize the relative error of $p(x) - \exp x$ for $|x| \leq \log 2 \cdot 2^{-17}$, with the following constraints:

$p_1, p_2, p_3$ fit on 128 bits

$p_4, p_5, p_6$ fit on 64 bits

$$\begin{aligned}
p_1 &= \text{0x1.00000000000000000000000000000000ap+0} \\
p_2 &= \text{0x8.00000000000000000000000006af3f78p-4} \\
p_3 &= \text{0x2.aaaaaaaaaaaaaaaaaaa80cd5b9d88f6p-4} \\
p_4 &= \text{0xa.aaaaaaaaaaaaaap-8} \\
p_5 &= \text{0x2.2222222224dce8p-8} \\
p_6 &= \text{0x5.b05b43776501cp-12}
\end{aligned}$$

Relative error $< 2^{-121.33}$ (not counting rounding errors).

# Generic binary128 Routines

[$a, b, c$ stand for binary128 structures, $m$ stands for some $m_1 \cdot 2^{-64} + m_0 \cdot 2^{-128}$ with $m_1, m_0 < 2^{64}$]

extract_binary128: extract a _Float128 into binary128

unpack: unpack a binary128 into a _Float128

normalize: shift $m_1, m_0$ and adjust $e$ so that $2^{63} \le m_1 < 2^{64}$

align_binary128: shift $m$ so that $e = 0$ (assumes $e \le 0$ initially)

sub_inplace: $a \leftarrow a - c$, assuming $e_a = e_c$

add_inplace: $a \leftarrow a + c$

mul: $a \leftarrow \mathrm{high}(b \cdot c)$

addu: $a \leftarrow b + m \cdot 2^{e_b}$, assuming no carry

shift_right, shift_left: shift $m_a$ by $k$ bits and update $e_a$

# Specific Routines

`reduce`: $a \leftarrow a - i \log 2$, $i$ integer, $\log 2$ precomputed on 192 bits

# Accuracy of binary128 exp

Test done by Alexei Sibidanov, on $10^5$ random inputs in $[-10, 10]$.

Correctly rounded results:

| Oracle Studio 12.6 | Intel Math 19.0.5.281 | libquadmath 9.2.1 | ASquadmath | Paul's exp |
|---|---|---|---|---|
| 99615 | 99997 | **99999** | **99999** | 99951 |

All other results are wrong by one ulp.

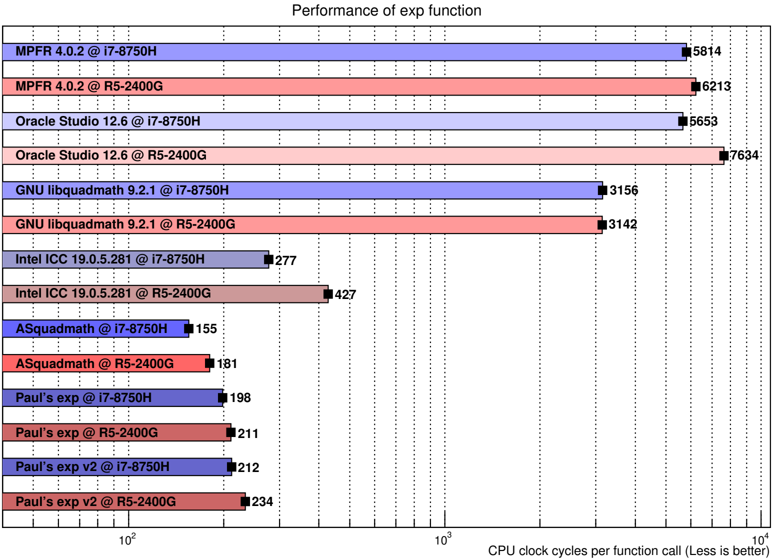
# Performance of binary128 exp

Test done by Alexei Sibidanov, on an AMD Ryzen 5 2400G.

Average number of cycles (measured with `perf stat`):

| MPFR 4.0.2 | Oracle Studio 12.6 | Intel Math 19.0.5.281 | libquadmath 9.2.1 | ASquadmath | Paul's exp |
|---|---|---|---|---|---|
| 6213 | 7634 | 427 | 3142 | **181** | 234 |

Goal of saving a factor of 10 over the GNU libm is reached!




[credit Alexei Sibidanov]

# Conclusion

Quadruple precision is indeed slow, but we can do much better!

We saved a factor of 10 with little effort, probably we can save another factor of 2 with more effort.

Use of integer operations is the key for efficiency.

The generic `binary128` routines can be reused for other functions.

# Perspectives

Implement addition, subtraction, multiplication, division directly with correct rounding for the `binary128` type, to avoid converting to/from `_Float128`.

Design an exponential function with correct rounding. The slow path would use similar integer-only arithmetic, with four 64-bit words (256 bits of accuracy), assuming the hard-to-round cases are known.

# The libm detector

```
https://homepages.loria.fr/PZimmermann/libm-detector/

$ gcc libm-detector.c -lm
$ ./a.out
Mathematical Library Detector, version 1.0
Probably libm shipped with GNU libc, version >= 2.29

$ icc libm-detector.c
$ ./a.out
Mathematical Library Detector, version 1.0
Probably Intel Math Library
```

# Afterthoughts

Performance is nice.

What about reproducibility (cf. David Bailey's talk)?

Currently developers/users mostly care about efficiency.

Shouldn't we instead seek for bit-to-bit reproducibility, and then only for performance?

IEEE 754 only recommends correct rounding for math functions. Should it require correct rouding?

Maybe we get some answer in Jason Riedy's talk Potential Directions for Moving IEEE-754 Forward at 3:45pm.

# References

Sollya: an environment for the development of numerical codes, Sylvain Chevillard, Mioara Maria Joldes and Christoph Lauter, Third International Congress on Mathematical Software (ICMS), LNCS 6327, 2010.

A new quadruple precision math library, Alexei Sibidanov, 50 pages, personal communication, February 2020.

Source code for `expf128` available here:

`https://homepages.loria.fr/PZimmermann/glibc-contrib/`