Computing correctly rounded logarithms with fixed-point operations

Julien Le Maire, Florent de Dinechin, Jean-Michel Muller and Nicolas Brunie
Introduction and context

Algorithm

Results and comparisons

Conclusions
Logarithm, the mathematical version

\[
\ln(a \times b) = \ln(a) + \ln(b)
\]

\[
\ln\left(\frac{b}{a}\right) = a \times \ln(b)
\]

Taylor: for \(x\) small, \(\ln(1 + x) \approx x - x^2/2 + x^3/3 - \ldots\)

\[y = \ln(x)\]
\[ \ln(a \times b) = \ln(a) + \ln(b) \]
Logarithm, the mathematical version

- \( \ln(a \times b) = \ln(a) + \ln(b) \)
- \( \ln(b^a) = a \times \ln(b) \)

\[ y = \ln(x) \]
Logarithm, the mathematical version

- \( \ln(a \times b) = \ln(a) + \ln(b) \)
- \( \ln(b^a) = a \times \ln(b) \)
- Taylor: for \( x \) small, \( \ln(1 + x) \approx x - x^2/2 + x^3/3... \)

\[ y = \ln(x) \]
The floating point version of the natural logarithm is called $\log$ (you will also find $\log_2$ and $\log_{10}$ and a few others)

$$\forall x \in F_{64} \quad \log(x) = \circ (\ln(x))$$
Logarithm, the floating-point version

The floating point version of the natural logarithm is called \( \log \) (you will also find \( \log_2 \) and \( \log_{10} \) and a few others)

\[
\forall x \in \mathbb{F}_{64} \quad \log(x) = \circ (\ln(x))
\]

An experiment

Implementing the *floating-point* logarithm function

- using only *integer* arithmetic
- for *performance*

(previous work motivated by *lack of FP hardware*)
Why using fixed-point arithmetic?

- **1960**: 32-bit mainstream integer
- **1980**: 32-bit IEEE-754 (64 bits) or mainstream floating-point
- **2000**: 64-bit floating-point, but only 52-bit precision

If you can predict the value of the exponent, exponent bits are wasted bits.

J. Le Maire, F. de Dinechin, J.-M. Muller and N. Brunie, Computing correctly rounded logarithm with fixed-point operations.
Why using fixed-point arithmetic?

- 64-bit floating-point, but only 52-bit precision
- if you can predict the value of the exponent, exponent bits are wasted bits.
modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication 64x64 $\rightarrow$ 128 ($mulq$)
  - count leading zeroes, shifts ($lzcnt$, $bsr$)
Integer better than floating-point?

- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication $64 \times 64 \rightarrow 128$ (mulq)
  - count leading zeroes, shifts (lzcnt, bsr)
- most operations are faster on integers, especially addition
  (which more or less defines the processor cycle time)
- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication $64 \times 64 \rightarrow 128$ (\textit{mulq})
  - count leading zeroes, shifts (\textit{lzcnt, bsr})
- most operations are faster on integers, especially addition
  (which more or less defines the processor cycle time)
- multiprecision faster and simpler using integers
modern 64-bit machines offer all sort of useful integer instructions
- addition
- multiplication 64x64 → 128 \((\text{mulq})\)
- count leading zeroes, shifts \((\text{lzcnt, bsr})\)

most operations are faster on integers, especially addition
(\text{which more or less defines the processor cycle time})
multiprecision faster and simpler using integers
small multiprecision out of the box:
mainstream compilers \((\text{gcc, clang, icc})\) support \text{__int_128}
- addition 128x128 → 128 \((\text{add, adc})\)
- shift on two registers \((\text{shld, shrd})\)
Integer better than floating-point?

- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication $64 \times 64 \rightarrow 128$ \textit{(mulq)}
  - count leading zeroes, shifts \textit{(lzcnt, bsr)}
- most operations are faster on integers, especially addition
  (which more or less defines the processor cycle time)
- multiprecision faster and simpler using integers
- small multiprecision out of the box:
  - mainstream compilers \textit{(gcc, clang, icc)} support \texttt{__int_128}
    - addition $128 \times 128 \rightarrow 128$ \textit{(add, adc)}
    - shift on two registers \textit{(shld, shrd)}

Caveat: integer SIMD/vector support still lagging behind FP
  \textit{(no vector multiplication)}
Outline

Introduction and context

Algorithm

Results and comparisons

Conclusions
Muller and Lefèvre solved the table maker dilemma for \( \ln \)

Computing the log with an error \( \leq 2^{-113} \) enables correct rounding

- Two consecutive floating-point numbers
- Computed logarithm, with error margin
- Real numbers

**CRLibm refinement of Ziv's technique:**
First step: quick-and-dirty evaluation of \( \ln(x) \)
(just accurate enough to ensure correct rounding in most cases)

- Test if rounding can be decided
- If not (rarely), recompute \( \ln(x) \) with the worst-case accuracy

Trade-off between first and second steps:

\[
\text{MeanTime} = \text{Time (1st step)} + \text{Pr[need 2nd step]} \cdot \text{Time (2nd step)}
\]

Best so far:
\[
\text{Time (2nd step)} \approx 10 \times \text{Time (1st step)}
\]

This work:
\[
\text{Time (2nd step)} \approx 2 \times \text{Time (1st step)}
\]
Muller and Lefèvre solved the table maker dilemma for ln

Computing the log with an error $\leq 2^{-113}$ enables correct rounding

two consecutive floating-point numbers

computed logarithm, with error margin

real numbers
Muller and Lefèvre solved the table maker dilemma for \( \ln \)

Computing the log with an error \( \leq 2^{-113} \) enables correct rounding

---

CRLibm refinement of Ziv's technique:

- First step: quick-and-dirty evaluation of \( \ln(x) \)
  (just accurate enough to ensure correct rounding in most cases)
- Test if rounding can be decided
- If not (rarely), recompute \( \ln(x) \) with the worst-case accuracy

---

![Diagram showing the relationship between real numbers, two consecutive floating-point numbers, and the computed logarithm, with error margin.](image-url)
Muller and Lefèvre solved the table maker dilemma for $\ln$

Computing the log with an error $\leq 2^{-113}$ enables correct rounding

CRLibm refinement of Ziv’s technique:
- First step: quick-and-dirty evaluation of $\ln(x)$ (just accurate enough to ensure correct rounding in most cases)
- Test if rounding can be decided
- If not (rarely), recompute $\ln(x)$ with the worst-case accuracy

Trade-off between first and second steps:
$\text{MeanTime} = \text{Time(1st step)} + \text{Pr[need 2nd step]} \cdot \text{Time(2nd step)}$
Muller and Lefèvre solved the table maker dilemma for \( \ln \)

Computing the log with an error \( \leq 2^{-113} \) enables correct rounding

CRLibm refinement of Ziv’s technique:
- First step: quick-and-dirty evaluation of \( \ln(x) \)
  (just accurate enough to ensure correct rounding in most cases)
- test if rounding can be decided
- if not (rarely), recompute \( \ln(x) \) with the worst-case accuracy

Trade-off between first and second steps:
\[
\text{MeanTime} = \text{Time(1st step)} + \text{Pr[need 2nd step]} \cdot \text{Time(2nd step)}
\]

Best so far: \( \text{Time(2nd step)} \approx 10 \times \text{Time(1st step)} \)
This work: \( \text{Time(2nd step)} \approx 2 \times \text{Time(1st step)} \)
The big picture

1. Filter special cases (negative numbers, $\infty$, ...)
2. Argument range reduction
3. Polynomial approximation
4. Solution reconstruction
5. Error evaluation and rounding test
6. If more accuracy needed:
   Rerun the steps 3 and 4 with the worst-case accuracy.
First argument range reduction

\[ \text{input} = 2^E \cdot (1 + x) \]
\[ \ln(\text{input}) = E \cdot \ln(2) + \ln(1 + x) \]
First argument range reduction

\[ \text{input} = 2^E \cdot (1 + x) \]
\[ \ln(\text{input}) = E \cdot \ln(2) + \ln(1 + x) \]

Evaluation algorithm:
- approximate \( \ln(1 + x) \) with a polynomial \( p(x) \)
  degree needed: at least 26
- evaluate \( E \cdot \ln(2) \)
- add both terms
Tang’s range reduction

$1 + x$: fractional part on 52 bits

$1 + x_1$: fractional part on 7 bits

Define $1 + y = r_x \cdot (1 + x)$

Then $\ln(1 + x) = \ln(1 + y) + \ln(1 + r_x)$
A table, addressed by $x_1$, the 7 most significand bits of $x$, stores

\[
r_x \approx \frac{1}{1 + x_1} \approx \frac{1}{1 + x} \quad \text{and} \quad \ln \left( \frac{1}{r_x} \right)
\]
A table, addressed by $x_1$, the 7 most significant bits of $x$, stores

$$r_x \approx \frac{1}{1 + x_1} \approx \frac{1}{1 + x} \quad \text{and} \quad \ln \left( \frac{1}{r_x} \right)$$

- Define $1 + y = r_x \cdot (1 + x) \approx 1$
- Then $\ln(1 + x) = \ln(1 + y) + \ln\left( \frac{1}{r_x} \right)$
Tang’s range reduction algorithm

1 + x: fractional part on 52 bits
1 + x₁: fractional part on 7 bits
rₓ: fractional part on 18 bits
1 + y: fractional part on 64 bits plus 6 implicit zeros

- Extract the index x₁
- Read, from a table addressed by x₁, both rₓ and ln(\(\frac{1}{r_x}\))
- compute \(y = r_x \cdot (1 + x) - 1\) (exactly)
- approximate \(\ln(1 + y)\) with a polynomial \(p(y)\)

add it all:

\[
\ln(\text{input}) \approx E \cdot \ln(2) + p(y) + \ln\left(\frac{1}{r_x}\right)
\]
Tang’s range reduction

\[ y = r_x \cdot (1 + x) - 1 \]

With \(1 + x\) on 53 bits we can tabulate \(r_x\) on 18 bits:

- the exact product would need 71 bits
- but we can predict the 7 leading bits
- ... so we can let them overflow quietly and use a \(64 \times 64 \rightarrow 64\) multiplication.
Two levels of Tang reduction

\[1 + x: \quad \frac{1}{2^{52}}\]
fractional part on 52 bits

\[1 + x_1: \quad \frac{1}{2^7}\]
fractional part on 7 bits

\[r_x: \quad \frac{1}{2^9}\]
fractional part on 9 bits

\[1 + y: \quad \frac{100000}{2^{60}}\]
fractional part on 60 bits including 5 zeros

\[1 + y_1: \quad \frac{100000}{2^{13}}\]
fractional part on 13 bits

\[r_y: \quad \frac{1}{2^{15}}\]
fractional part on 15 bits

\[1 + z: \quad \frac{1000000000000}{2^{64}}\]
fractional part on 64 bits plus 12 implicit zeros

\[x \in [0, 1)\]

\[y \in [0, 2^{-6.41504})\]

\[z \in [0, 2^{-12.6747})\]

\[x_1 \text{ takes } 64 \text{ different values}\]

\[y_1 \text{ takes } 96 \text{ different values}\]

the whole reduction of \(x\) to \(z\) is computed exactly in 64-bit int.
A few Pareto points in the design space

<table>
<thead>
<tr>
<th>Table size (bytes)</th>
<th>degree 1st</th>
<th>degree 2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>39,936</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>12,288</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4,032</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2,240</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2,016</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>900</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>594</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>298</td>
<td>7</td>
<td>14</td>
</tr>
</tbody>
</table>
Why stop at two levels of reduction?

Answer is: diminishing return.

For a target accuracy of $2^{-60}$:

<table>
<thead>
<tr>
<th></th>
<th>interval of x</th>
<th>degree needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>No reduction</td>
<td>$[-1/2, 1/2]$</td>
<td>29</td>
</tr>
<tr>
<td>1 level</td>
<td>$[-2^{-7}, 2^{-7}]$</td>
<td>8</td>
</tr>
<tr>
<td>2 levels</td>
<td>$[-2^{-12}, 2^{-12}]$</td>
<td>4</td>
</tr>
<tr>
<td>3 levels</td>
<td>$[-2^{-18}, 2^{-18}]$</td>
<td>3</td>
</tr>
</tbody>
</table>

Adding more levels will cost more operations than it saves...
We want to approximate $\log(1 + z)$ on an interval around 0. Use the (now standard) tool set to obtain it.

- **Sollya:**
  - finds a machine-efficient polynomial $P(z)$
  - computes a safe bound on the approximation error $P(z) - \ln(1 + z)$

- **Gappa:** bounds the accumulation of rounding errors when evaluating $P(z)$ in C

We obtain a Coq proof of the error:

computed approximation of $\ln(1 + z)$, with relative error margin
Reconstructing the solution

\[ \text{input} = 2^e \cdot (1 + x) \]
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{r_x} \cdot (1 + y) \]
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z) \]
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z) \]

\[ \ln(\text{input}) = e \cdot \ln(2) + \ln(r_x^{-1}) + \ln(r_y^{-1}) + \ln(1 + z) \]
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z) \]

\[ \ln(\text{input}) = e \cdot \ln(2) + \ln(r_x^{-1}) + \ln(r_y^{-1}) + \ln(1 + z) \]

"If we can predict the exponents, exponent bits are wasted bits"
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z) \]

\[ \ln(\text{input}) = e \cdot \ln(2) + \ln(r_x^{-1}) + \ln(r_y^{-1}) + \ln(1 + z) \]

- \(e \cdot \ln(2)\): 
- \(\ln(r_x^{-1})\): 
- \(\ln(r_y^{-1})\): 
- \(P(z) \approx \ln(1 + z)\): 
- sum:

\[ 11 \quad 0 \quad -12 \quad -53 \quad -117 \]

“If we can predict the exponents, exponent bits are wasted bits”
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z) \]

\[ \ln(\text{input}) = e \cdot \ln(2) + \ln(r_x^{-1}) + \ln(r_y^{-1}) + \ln(1 + z) \]

"If we can predict the exponents, exponent bits are wasted bits"
Reconstructing the solution

\[ \text{input} = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z) \]

\[ \ln(\text{input}) = e \cdot \ln(2) + \ln(r_x^{-1}) + \ln(r_y^{-1}) + \ln(1 + z) \]

"If we can predict the exponents, exponent bits are wasted bits"

J. Le Maire, F. de Dinechin, J.-M. Muller and N. Brunie
Computing correctly rounded logarithm with fixed-point operations
Error evaluation

\[ e \cdot \ln(2): \]
\[ \ln(r_x^{-1}): \]
\[ \ln(r_y^{-1}): \]
\[ P(z) \approx \ln(1 + z): \]
\[ \text{sum:} \]

11 0 -12 -53 -117
\[ e \cdot \ln(2): \]
\[ \ln(r_x^{-1}): \]
\[ \ln(r_y^{-1}): \]
\[ P(z) \approx \ln(1 + z): \]
\[ \text{sum:} \]
\[ \epsilon < (|e|) \cdot 2^{-117} \]
$e \cdot \ln(2)$:

$\ln(r_x^{-1})$:

$\ln(r_y^{-1})$:

$P(z) \approx \ln(1 + z)$:

sum:

$\epsilon < (|e| + 1 + 1) \cdot 2^{-117}$
\[ \epsilon < \left( |e| + 1 + 1 + P(z) \cdot 2^{-59} \right) \cdot 2^{-117} \]
Simple technique: compute the two bounds of the interval, and see if they round to the same mantissa

(two additions, a xor and a shift)
Rounding test

Simple technique: compute the two bounds of the interval, and see if they round to the same mantissa

(two additions, a xor and a shift)

real numbers
Simple technique: compute the two bounds of the interval, and see if they round to the same mantissa

(two additions, a xor and a shift)

---

For comparison, the proof of the floating-point-based rounding test (invented by Ziv and used in CRLibm) is an 18-page paper that took 20 years to publish...
Second step

- Use 3 words instead of 2 for the precomputed $\ln(2)$
- Use a much more accurate polynomial:
  - with coefficients on 128 bits instead of 64
    (but $z$ is still only a 64-bit number)
  - and using a higher degree polynomial
Outline

Introduction and context

Algorithm

Results and comparisons

Conclusions
## Implementation parameters of correctly rounded implementations

<table>
<thead>
<tr>
<th></th>
<th>glibc</th>
<th>crlibm-td</th>
<th>crlibm-de</th>
<th>cr-FixP</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree pol. 1</td>
<td>3/8</td>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>degree pol. 2</td>
<td>20</td>
<td>12</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>tables size</td>
<td>13 Kb</td>
<td>8192 bytes</td>
<td>6144 bytes</td>
<td>4032 bytes</td>
</tr>
<tr>
<td>% accurate phase</td>
<td>N/A</td>
<td>1.5</td>
<td>0.4</td>
<td>4.4</td>
</tr>
</tbody>
</table>
### Pentium timing (lower is better)

<table>
<thead>
<tr>
<th>cycles</th>
<th>MKL</th>
<th>glibc</th>
<th>crlibm</th>
<th>cr-de</th>
<th>cr-FixP</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg time</td>
<td>25</td>
<td>90</td>
<td>69</td>
<td>46</td>
<td>49</td>
</tr>
<tr>
<td>max time</td>
<td>25</td>
<td>11,554</td>
<td>642</td>
<td>410</td>
<td>79</td>
</tr>
</tbody>
</table>

### Timing breakdown on two processors

<table>
<thead>
<tr>
<th>cycles</th>
<th>Core i5</th>
<th>Bostan</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>glibc</td>
<td>newlib</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>105</td>
</tr>
<tr>
<td>quick phase alone</td>
<td>42</td>
<td>94</td>
</tr>
<tr>
<td>accurate phase alone</td>
<td>74</td>
<td>181</td>
</tr>
<tr>
<td>both phases (avg time)</td>
<td>49</td>
<td>121</td>
</tr>
<tr>
<td>both phases (max time)</td>
<td>79</td>
<td>225</td>
</tr>
</tbody>
</table>

Slanted means: no correct rounding
### Average and max running time (in processor cycles)

#### Pentium timing (lower is better)

<table>
<thead>
<tr>
<th>cycles</th>
<th>MKL</th>
<th>glibc</th>
<th>crlibm</th>
<th>cr-de</th>
<th>cr-FixP</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg time</td>
<td>25</td>
<td>90</td>
<td>69</td>
<td>46</td>
<td>49</td>
</tr>
<tr>
<td>max time</td>
<td>25</td>
<td>11,554</td>
<td>642</td>
<td>410</td>
<td>79</td>
</tr>
</tbody>
</table>

#### Timing breakdown on two processors

<table>
<thead>
<tr>
<th>cycles</th>
<th>Core i5</th>
<th>Bostan</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>glibc</td>
<td>newlib</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>105</td>
</tr>
<tr>
<td>quick phase alone</td>
<td>42</td>
<td>94</td>
</tr>
<tr>
<td>accurate phase alone</td>
<td>74</td>
<td>181</td>
</tr>
<tr>
<td>both phases (avg time)</td>
<td>49</td>
<td>121</td>
</tr>
<tr>
<td>both phases (max time)</td>
<td>79</td>
<td>225</td>
</tr>
</tbody>
</table>

Slanted means: no correct rounding
Average and max running time (in processor cycles)

Pentium timing (lower is better)

<table>
<thead>
<tr>
<th>cycles</th>
<th>MKL</th>
<th>glibc</th>
<th>crlibm</th>
<th>cr-de</th>
<th>cr-FixP</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg time</td>
<td>25</td>
<td>90</td>
<td>69</td>
<td>46</td>
<td>49</td>
</tr>
<tr>
<td>max time</td>
<td>25</td>
<td>11,554</td>
<td>642</td>
<td>410</td>
<td>79</td>
</tr>
</tbody>
</table>

Timing breakdown on two processors

<table>
<thead>
<tr>
<th>cycles</th>
<th>Core i5</th>
<th>Bostan</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>glibc</td>
<td>newlib</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>105</td>
</tr>
<tr>
<td>quick phase alone</td>
<td>42</td>
<td>94</td>
</tr>
<tr>
<td>accurate phase alone</td>
<td>74</td>
<td>181</td>
</tr>
<tr>
<td>both phases (avg time)</td>
<td>49</td>
<td>121</td>
</tr>
<tr>
<td>both phases (max time)</td>
<td>79</td>
<td>225</td>
</tr>
</tbody>
</table>

Slanted means: no correct rounding
### Pentium timing (lower is better)

<table>
<thead>
<tr>
<th>cycles</th>
<th>MKL</th>
<th>glibc</th>
<th>crlibm</th>
<th>cr-de</th>
<th>cr-FixP</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg time</td>
<td>25</td>
<td>90</td>
<td>69</td>
<td>46</td>
<td>49</td>
</tr>
<tr>
<td>max time</td>
<td>25</td>
<td>11,554</td>
<td>642</td>
<td>410</td>
<td>79</td>
</tr>
</tbody>
</table>

### Timing breakdown on two processors

<table>
<thead>
<tr>
<th>cycles</th>
<th>Core i5</th>
<th>Bostan</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>glibc</td>
<td>newlib</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>105</td>
</tr>
<tr>
<td>quick phase alone</td>
<td>42</td>
<td>94</td>
</tr>
<tr>
<td>accurate phase alone</td>
<td>74</td>
<td>181</td>
</tr>
<tr>
<td>both phases (avg time)</td>
<td>49</td>
<td>121</td>
</tr>
<tr>
<td>both phases (max time)</td>
<td>79</td>
<td>225</td>
</tr>
</tbody>
</table>

Slanted means: no correct rounding
Outline

Introduction and context

Algorithm

Results and comparisons

Conclusions
Conclusions

- Better range reduction thanks to a wider format
- ... leading to improvements in polynomial degree and table size
- Faster multiprecision due to higher precision
- Able to reuse some computations of the fast step in the accurate step
- Alternative rounding test for accurate step
- Probability to launch 2nd step is high, but this is acceptable since 2nd step is so cheap

Competitive against state-of-the-art
Worst case improved compared to others implementations
Second step-only version is a viable alternative

Limitations:
- Require 64 bits integer support
- No support for vectorization
Conclusions

- Better range reduction thanks to a wider format
- ... leading to improvements in polynomial degree and table size
- Faster multiprecision due to higher precision
- Able to reuse some computations of the fast step in the accurate step
- Alternative rounding test for accurate step
- Probability to launch 2nd step is high, but this is acceptable since 2nd step is so cheap

- Competitive against state-of-the-art
- Worst case improved compared to others implementations
- Second step-only version is a viable alternative

Limitations:
- Require 64 bits integer support
- No support for vectorization
Conclusions

- Better range reduction thanks to a wider format
- ... leading to improvements in polynomial degree and table size
- Faster multiprecision due to higher precision
- Able to reuse some computations of the fast step in the accurate step
- Alternative rounding test for accurate step
- Probability to launch 2nd step is high, but this is acceptable since 2nd step is so cheap
- Competitive against state-of-the-art
- Worst case improved compared to others implementations
- Second step-only version is a viable alternative

Limitations:
- Require 64 bits integer support
- No support for vectorization
Any question?
Bonus: a floating-point in, fixed-point out variant

Some code

details on the solution reconstruction
Floating-point in, fixed-point out

- output: fixed-point, 11 bits integer part, 53 bit fractional part
  
<table>
<thead>
<tr>
<th>integer part</th>
<th>fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- target absolute accuracy $2^{-52}$

<table>
<thead>
<tr>
<th>output format</th>
<th>absolute accuracy</th>
<th>table size</th>
<th>Core i5 cycles</th>
<th>Bostan cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fix64</td>
<td>$2^{-52}$</td>
<td>2304</td>
<td>24</td>
<td>66</td>
</tr>
<tr>
<td>Fix128</td>
<td>$2^{-116}$</td>
<td>4032</td>
<td>60</td>
<td>179</td>
</tr>
<tr>
<td>double (libm)</td>
<td>$2^{-42}$</td>
<td></td>
<td>90</td>
<td>105</td>
</tr>
</tbody>
</table>

- Fix64 is the code of the first step only, without the conversion to float.

  - tweak: poly degree 3 only for abs. accuracy $2^{-59}$

- Fix128 is the code of the second step only, without the conversion to float.
Motivation

TKF91 : DNA sequence alignment algorithm

- dynamic programming algorithm:
  
alignment as a path within a 2D array.
- borders of an array initialized with log-likelihoods
- then array filled using recurrence formulae
  
  that involve only max and + operations.

All current implementations of this algorithm use a floating-point array, but

- int64 + and max are 1-cycle, vectorizable, and exact operations;
- absolute accuracy of initialization logs: up to $2^{-42}$ with FP log, $2^{-52}$ with FixP log.
Only partial experiments

- Improvement in accuracy measured
- No noticeable improvement in performance
Bonus: a floating-point in, fixed-point out variant

Some code

details on the solution reconstruction
Special cases: business as usual

```c
/* reinterpret x to manipulate its bits more easily */
uint64_t inputbits = ((union {
    double d; uint64_t u;
}){input}).u;
int xe = inputbits >> 52;

/* filter the special cases: !(x is normalized and 0 < x < +Inf) */
if (0x7FEu <= ((unsigned)xe - 1u)) {
    /* x = +0:  raise a DivideByZero, return -Inf */
    if (((xbits & ~(1ull << 63)) == 0) return -1.0/0.0;
    /* x < 0.0:  raise a InvalidOperation, return a qNaN */
    if (((xbits & (1ull << 63)) != 0) return (x-x)/0;
    /* x = qNaN:  return a qNaN
      x = sNaN:  raise a InvalidOperation, return a qNaN
      x = +Inf:  return +Inf */
    if (xe != 0) return x+x;
    /* x subnormal: change x to a normalized number */
    else {
        int u = clz64(xbits) - 12;
        xbits <<= u + 1;
        xe -= u;
    }
}
```

Only interesting line: the subnormal management
Argument reduction

```c
/* input = 2^xe * (1 + X) */
uint64_t x = inputbits & 0xFFFFFFFFFFFFFull;

/* 1 + X = (1/Ri) * Y */
uint8_t x1 = x >> (52 - ARG_REDUC_1_PREC);
uint64_t y = argReduc1_inv[x1] * (x + (1ull << 52));
__builtin_prefetch(& argReduc1_log[x1], 0, 0);

/* Y = (1/S) * (1 + Z) */
/* with dZ = dz/2^(52 + ARG_REDUC_1_SIZE + ARG_REDUC_2_SIZE)
and 1/S = argReduc2[si].val/2^ARG_REDUC_2_SIZE */
uint8_t y1 = (y >> (52 + ARG_REDUC_1_SIZE - ARG_REDUC_2_PREC))
           - (1u << ARG_REDUC_2_PREC);
uint64_t z = argReduc2_inv[y1] * y;  // +1 part removed by overflow
__builtin_prefetch(& argReduc2_log[y1], 0, 0);
```
/\* Polynomial approximation of log(1+Z)/Z \approx P(Z) and Z\times P(Z) */

static const uint64_t a4 = UINT64_C(0x3ffc147cb4539237);
static const uint64_t a3 = UINT64_C(0x5555553dc70f0dfd);
static const uint64_t a2 = UINT64_C(0x7fffffffffd574fd);
static const uint64_t a1 = UINT64_C(0xffffffffffffffffa);

uint64_t pz = a1 - ( highmul(z, a2 - ( highmul(z, a3 - ( highmul(z, a4) >> 12) ) >> 12) ) >> 12);

uint128_t zpz = fullmul(z, pz);

/* Polynomial approximation without shifts:
   replace highmul(z, a) >> 12 with highmul(z, a >> 12) */

static const uint64_t a4 = UINT64_C(0x0000000003ffc147);
static const uint64_t a3 = UINT64_C(0x0000005555553dc6);
static const uint64_t a2 = UINT64_C(0x0007fffffffffd57);
static const uint64_t a1 = UINT64_C(0xffffffffffffffffa);

uint64_t pz = a1 - highmul(z, a2 - highmul(z, a3 - highmul(z, a4)));

uint128_t zpz = fullmul(z, pz);
Reconstructing the solution

/* Compute part of the result that don’t depend on Z
   \[(x e \log (2) + \log (1/R_i) + \log (1/S_i))\] */

uint128_t cstpart = fullimul(xe, log2fw_mid)
    + UINT128((int64_t)xe * log2fw_high, 0) // fullmul not needed
    + UINT128(argReduc1[ri].log_hi, argReduc1[ri].log_lo)
    + UINT128(argReduc2[si].log_hi, argReduc2[si].log_lo);

/* Polynomial approximation of \(\log(1+Z)/Z \approx P(Z)\) and \(ZP(Z)\) */

/* Assemble the two parts, compute sign, mantissa and exponent */

uint128_t longres = cstpart + (zpz >> (11 + IMPLICIT_ZEROS));
uint64_t sign = -(HI(longres) >> 63); // 0 or ~0
// if sign != 0, this is longres = ~longres (~a = ~a + 1)
// to avoid the +1 approx, do:
// longres = ((int64_t)sign + longres) ^ UINT128(sign, sign);
longres ^= UINT128(sign, sign);

int u = clz64(HI(longres)) + 1;
int exponent = 11 - u;
uint64_t mantissa = (HI(longres) << u) | (LO(longres) >> (64 - u))
Rounding test and conversion

/* Compute the maximal absolute error (aligned with longres):  
   + 2 + abs(xe) for xe*\log(2), \log(1/Ri) and \log(1/Si)  
   + 1 + zpz>>(POLYNOMIAL_PREC+IMPLICIT_ZEROS+11) for the polynomial  
If result*(1 + maxRelErr) are not rounded to the same number, we have  
uint64_t maxAbsErr = 3 + abs(xe) + (HI(zpz) >> (POLYNOMIAL_PREC + 
uint64_t maxRelErr = (maxAbsErr >> (64 - u)) + 1; 
if (((mantissa + maxRelErr) ^ (mantissa - maxRelErr)) >> 11) {
    return log_rn_accurate (cstpart, z, xe);
}

/* Assemble the computed result */
uint64_t resultbits = ((uint64_t)sign << 63)  
+ ((uint64_t)(exponent+1023) << 52)  
+ (mantissa >> 12)  
+ (((mantissa >> 11) & 1); /* round to nearest */
return (union { uint64_t u; double d; }){ resultbits }.d;
Bonus: a floating-point in, fixed-point out variant

Some code

details on the solution reconstruction
Reconstructing the solution

\[ e \cdot \ln(2) : \]

\[ \ln(r_x^{-1}) : \]

\[ \ln(r_y^{-1}) : \]

\[ P(z) \approx \ln(1 + z) : \]

\[ \text{sum:} \]

11  0  -11  -53  -117
Reconstructing the solution

\[ e \cdot \ln(2): \]

\[ \ln(r_x^{-1}): \]

\[ \ln(r_y^{-1}): \]

\[ P(z) \approx \ln(1 + z): \]

sum:

1 floating-point fraction

11 0 -11 -53 -117
Reconstructing the solution

\[ e \cdot \ln(2): \]

\[ \ln(r_x^{-1}): \]

\[ \ln(r_y^{-1}): \]

\[ P(z) \approx \ln(1 + z): \]

\[ \text{sum:} \]

1 floating-point fraction

J. Le Maire, F. de Dinechin, J.-M. Muller and N. Brunie

Computing correctly rounded logarithm with fixed-point operations
Reconstructing the solution

\[ e \cdot \ln(2): \]

\[ \ln(r_x^{-1}): \]

\[ \ln(r_y^{-1}): \]

\[ P(z) \approx \ln(1 + z): \]

\[ \text{sum:} \]

1 floating-point fraction
Reconstructing the solution

\[ e \cdot \ln(2): \]

\[ \ln(r_x^{-1}): \]

\[ \ln(r_y^{-1}): \]

\[ P(z) \approx \ln(1 + z): \]

\[ \text{sum:} \]

J. Le Maire, F. de Dinechin, J.-M. Muller and N. Brunie
Computing correctly rounded logarithm with fixed-point operations
Reconstructing the solution

\[ e \cdot \ln(2): \]

\[ \ln(r_x^{-1}): \]

\[ \ln(r_y^{-1}): \]

\[ P(z) \approx \ln(1 + z): \]

sum:

1 floating-point fraction

J. Le Maire, F. de Dinechin, J.-M. Muller and N. Brunie