Meta-implementation of vectorized logarithm function in binary floating-point arithmetic

Hugues de Lassus Saint-Geniès, Nicolas Brunie, Guillaume Revy

Meta-implementation of vectorized logarithm function in binary floating-point arithmetic

Hugues de Lassus Saint-Geniès*†, Nicolas Brunie‡, and Guillaume Revy*†
*Univ. Perpignan Via Domitia, DALI, Perpignan, France
†LIRMM, Univ. Montpellier, CNRS (UMR 5506), Montpellier, France
‡Kalray, Montbonnot-Saint-Martin, France

***Abstract*—Besides scalar instructions, modern micro-architectures also provide support for vector instructions. They enable to treat packed inputs (typically 4 or 8) in a single instruction. The challenge is now to write vector programs to support mathematical functions like $\sin$, $\cos$, $\exp$, $\log$, $\cdots$ which efficiently exploit those vector instructions. This article focuses on the design of vectorized implementation of $\log(x)$ function, and more particularly on its automation for different formats and micro-architectures. First it rewrites a classic range reduction in a branchless fashion so as to use at best recent micro-architecture features, like `rcp` (reciprocal) instruction, and to treat all inputs in the same flow. Second it details rigorously how to achieve "faithfully rounded" implementations. Third it shows how to automate this implementation process using the MetaLibm framework, on SSE/AVX and AVX2 supporting micro-architectures. Finally we illustrate that this process enables to achieve high throughput implementations for the binary32 and binary64 formats in a fully automated way.**

Keywords: logarithm function, vector micro-architectures, faithful rounding, automated design

I. Introduction

Nowadays, most modern micro-architectures embrace new hardware support for vector instructions in floating-point arithmetic, besides scalar instructions. They enable to treat packed inputs (typically 4 or 8) in a single instruction, thus potentially increasing the performance of floating-point programs. The challenge is now to use these instructions properly to write efficient vectorized implementations of mathematical functions. These are widely used in scientific programs manipulating floating-point computations, for which performance improvement is crucial (like in physics, for example).

In this article, we deal with the vectorized implementation of the logarithm function, $\log(x)$, in floating-point arithmetic, with a particular focus on its automation through the MetaLibm framework [1], [2]. It enables to describe the implementation of a function using a meta-language and to generate C codes optimized for different micro-architectures. For performance purpose, we target "faithfully rounded" implementations, that is, with 1-ulp accuracy. Even if, since 2008, the IEEE 754 standard recommends the correct rounding for scalar elementary functions like $\log(x)$, no output accuracy is required nor recommended for functions with vector arguments [3], and "faithful rounding" is often sufficient in high-performance computing context, for example in physics [4].

Several vectorized implementations of mathematical functions have already been proposed. The commercial Intel SVML library[1] provides intrinsic support for vectorized mathematical functions, but it is not freely available. Other, non-commercial implementations have been proposed. In 2015, Intel contributed the GNU Libmvec to glibc 2.22.[2] AMD also contributed the proprietary ACML_MV (no longer available) and the AMD LibM.[3] All these libraries support a set of vectorized mathematical functions. GNU Libmvec claims reasonable testing to pass 4-ulp maximum relative error, while ACML_MV was announced returning results accurate to at most 1 ulp, but with unpredictable behavior for subnormal inputs. Unfortunately, GNU Libmvec is written in assembly code, which reduces portability. The VDT is another open source effort [4]: in terms of accuracy, $\log(x)$ function is announced with at most 2 bits different compared to AMD LibM. Two recent projects are Yeppp[4] and SLEEF,[5] the latter providing 3-ulp vectorized function versions. Finally in 2016, a new SIMD vector library is proposed [5]. Written in high-level C language, $\log(x)$ is accurate to at most 1.0825 ulp. Note that the accuracy of these implementations is not guaranteed since, to our knowledge, it is obtained by non-exhaustive testing.

We provide here high throughput "faithfully rounded" implementations of $\log(x)$, in the binary32 and binary64 formats, and with a guarantee on the output accuracy. We propose also a meta-implementation of this function in the MetaLibm framework, enabling to write different optimized binary32 versions. This work being parametrized by the precision and code generation being based on MetaLibm backends, this can easily be extended to any other precision. Our functions do not handle special inputs, producing unpredictable outputs in these cases, and the subnormal inputs are considered in the main flow. This results in branchless programs, which avoids an expensive fallback treatment for these inputs (like most of the vector libraries presented above do) at a small overhead.

Therefore the main contributions of the article are:

- A branchless range reduction, designed to use at best some special features available on SSE/AVX and AVX2 instruction sets,
- A rigorous description of how to achieve "faithful rounding" in a guaranteed fashion,

[1]See https://software.intel.com/en-us/node/524289.
[2]See https://sourceware.org/glibc/wiki/libmvec.
[3]See https://developer.amd.com/amd-cpu-libraries/amd-math-library-libm/.
[4]See http://www.yeppp.info/.
[5]See http://sleef.org.

- The meta-implementation of this range reduction in the MetaLibm framework, parametrized by the precision, and enabling to generate different function versions, optimized for various instruction sets and vector sizes.

The article is organized as follows: Section II details the range reduction we use to achieve required accuracy, while Section III gives some guidelines on how to build accurate enough programs. Then Section IV provides some implementation details. In Section V, we present the work done in the MetaLibm framework. And Section VI gives some numerical results, before a conclusion in Section VII.

II. Algorithm for $\log(x)$ with faithful rounding

We consider here that the input x is neither a special input (NaN, $\pm\infty$, ± 0, or $x \leq 0$), nor the particular input $x = 1$. All the floating-point numbers manipulated, as well as the floating-point operations carried out, are in precision $p \geq 2$.

A. Range reduction

Let $x \neq 1$ be a positive floating-point number as defined in [3] with the sign $s = 0$:

$$x = m \cdot 2^e, \tag{1}$$

where $m \in [1, 2 - 2^{1-p}]$ and $e \in \{e_{\min}, \cdots, e_{\max}\}$. Many techniques have already been designed for the implementation of $\log(x)$. They are generally based on table lookup and/or polynomial evaluations [6]–[13]. Here let r_i denote an approximation of $1/m$. Then using $\log(x) = e \cdot \log(2) + \log(m)$, the $\log(x)$ function is computed as follows:

$$\log(x) = (e + \tau) \cdot \log(2) - \log(2^\tau \cdot r_i) + \log(1 + u),$$

where $u = r_i \cdot m - 1$. Here $\tau = [m > \sqrt{2}]$ is introduced to avoid the catastrophic cancellation that may occur while computing $e \cdot \log(2) + \log(m)$ when $e = -1$ and $m \approx 2$ as in [13]. Note that this rewriting enables to avoid code branches, and to take advantage of the `rcp` instruction, returning an approximation of the reciprocal and available on recent vector micro-architectures.

Furthermore when x is a subnormal floating-point number, a usual way consists in rescaling x in the normal range. Hence let x' be a floating-point number defined as:

$$x' = x \cdot 2^\lambda \quad \text{with} \quad \lambda \in \{0, \cdots, p-1\}, \tag{2}$$

where λ is the number of leading zeros in the binary representation of m. It follows that $x' = 2^{e'} \cdot m' \geq 2^{e_{\min}}$ and

$$\log(x) = (e' + \tau - \lambda) \cdot \log(2) - \log(2^\tau \cdot r_i) + \log(1 + u), \tag{3}$$

with $m' \in [1, 2 - 2^{1-p}]$, represented exactly with only one precision-p floating-point number. The quantity r_i is thus an approximation of $1/m'$ obtained with the `rcp` instruction truncated on i bits, and $u = r_i \cdot m' - 1$. In addition the value $-\log(2^\tau \cdot r_i)$ is retrieved from a lookup table using the $i > 0$ most significant fraction bits of the significand of r_i.

Note the way subnormal inputs are handled here. It is different from the one usually seen in vector function implementations, for which calling a fallback routine for these inputs is often preferred. However, treating subnormal inputs in the general flow has no great impact on performance as discussed in Section VI-D.

B. How to determine r_i, u, τ, and $-\log(2^\tau \cdot r_i)$?

Let us now detail the computation of r_i, u, τ, and $-\log(2^\tau \cdot r_i)$.

Determination of r_i. Let assume an `rcp` instruction available, that returns an approximation of $1/m'$ with a relative error bounded by κ:

$$\texttt{rcp}(m') = 1/m' \cdot (1 + \epsilon_{\text{rcp}}), \quad \text{with} \quad |\epsilon_{\text{rcp}}| \leq \kappa. \tag{4}$$

The quantity r_i is then computed by truncating $\texttt{rcp}(m')$ on $i > 0$ fraction bits. In order to improve and to re-center error enclosure, we first add one half-ulp on i bits. Thus we have

$$r_i = \texttt{rcp}(m') \cdot (1 + \epsilon_{r_i}), \quad \text{with} \quad |\epsilon_{r_i}| \leq 2^{-i-1},$$

since $m', r_i > 0$. It follows that $r_i = 1/m' \cdot (1+\epsilon_{\text{rcp}}) \cdot (1+\epsilon_{r_i})$. Hence:

$$r_i = 1/m' \cdot (1 + \epsilon) \quad \text{with} \quad \epsilon = \epsilon_{\text{rcp}} + \epsilon_{r_i} + \epsilon_{\text{rcp}} \cdot \epsilon_{r_i}, \tag{5}$$

where

$$|\epsilon| \leq \kappa + 2^{-i-1} + \kappa \cdot 2^{-i-1}. \tag{6}$$

This assumption on `rcp` is not restrictive at all. This kind of instructions is available on a many vector micro-architectures, like on Intel's ones since SSE and considered in this article, or on ARM's ones through the VRECPE (Vector Reciprocal Estimate) instruction.[6] If it is not the case, it can be emulated with full division or lookup table, as discussed in Section V. For example, using Intel intrinsics guide,[7] we know that SSE/AVX instruction sets provide a binary32 `rcp` instruction with a maximum relative error less than $1.5 \cdot 2^{-12}$, that is, with $|\epsilon_{\text{rcp}}| \leq 1.5 \cdot 2^{-12}$, while on more recent micro-architectures, AVX-512 instruction set provides also a binary64 `rcp` instruction with $|\epsilon_{\text{rcp}}| \leq 2^{-14}$. On SSE/AVX-supporting micro-architectures, the binary32 instruction can also be used to approximate $1/m'$ for a binary64 input: m' must first be casted to the binary32 format, thus slightly increasing the approximation error: $|\epsilon_{\text{rcp}}| \leq 1.5 \cdot 2^{-12} + 2^{-24} + 1.5 \cdot 2^{-36}$.

Determination of u. Since $u = r_i \cdot m' - 1$, we deduce from (5) that the quantity $u = \epsilon$, that is, u is the error occurring when approximating $1/m'$ by r_i.

Property 1: Let $x \neq 1$ be a positive floating-point number as in (1). The quantity u in (3) can be represented exactly with only one precision-p floating-point number as long as the parameter i is chosen so that:

$$i \leq e_{\max} - 2 \quad \text{and} \quad \kappa \cdot (2^i + 2^{-1}) < 2^{-1}. \tag{7}$$

Proof: Let M and R be the integer significands of m' and r_i, respectively. Since r_i is obtained after truncating $\texttt{rcp}(m')$ on $i > 0$ fraction bits, its significand can be stored on at most $i + 1$ bits. Hence

$$m' = M \cdot 2^{1-p} \quad \text{and} \quad r_i = R \cdot 2^{\delta - i},$$

[6] See ARM Compiler armasm Reference Guide.

[7] See https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

Table I
SUITABLE VALUES FOR i FOR DIFFERENT COMBINATIONS (e_{MAX}, κ).

$e_{\max}$	κ	$i \in \cdots$
127	$1.5 \cdot 2^{-12}$	$[1, 10]$
1023	2^{-14}	$[1, 12]$
1023	$1.5 \cdot 2^{-12} + 2^{-24} + 1.5 \cdot 2^{-36}$	$[1, 10]$

with $\delta \in \{-1, 0\}$. Using $u = r_i \cdot m' - 1$, we rewrite u as:

$$\begin{aligned} u &= R \cdot M \cdot 2^{\delta - i + 1 - p} - 1 \\ &= 2^{\delta - i + 1 - p} \cdot \left(R \cdot M - 2^{p - 1 + i - \delta}\right). \end{aligned}$$

Since $p \geq 2$ and $i > 0$, then if $i \leq e_{\max} - 2$, we deduce that $\delta - i + 1 - p \in [e_{\min} + 1 - p, 1 - p]$ is the exponent of a floating-point number. Moreover the value U defined as:

$$U = R \cdot M - 2^{p + i - 1 - \delta}$$

is an integer. Using (6), since $u = \epsilon$, we deduce that:

$$|U| \leq \kappa \cdot 2^p \cdot (2^i + 2^{-1}) + 2^{p-1}.$$

Therefore if i is chosen so that $\kappa \cdot (2^i + 2^{-1}) < 2^{-1}$, then $|U| < 2^p$, and u is a precision-p floating-point number, which concludes the proof. ∎

From Property 1, we know that for certain values of i, the quantity u fits exactly in only one floating-point number in precision p. These values are given in Table I, for different combinations $(e_{\max}, \kappa)$. As a consequence, in order to implement the sequence computing u, a first choice may be to use an `fma`, returning exactly u in a single instruction, and available on recent micro-architectures. If it is not available, an alternative is to use multi-word arithmetic, like double-double arithmetic in [14], [15]. This is discussed in Section VI.

Determination of τ and $-\log(2^\tau \cdot r_i)$. The value $-\log(2^\tau \cdot r_i)$ is an approximation of $-\log(2^\tau / m')$, and it is retrieved from a lookup table, indexed only by the i most significant fraction bits of r_i, and not by τ. It follows that this table is built such that for the entries corresponding to $\tau = 0$, it contains approximations of $-\log(1/m')$, while for the others ($\tau = 1$), it contains approximations of $-\log(2/m')$.

As a consequence, each cell in the table must represent m' values for which τ is the same. In this sense, τ cannot be decided using m', but it must be decided using the i most significant fraction bits of r_i, as well, denoted by `idx` below. And since r_i is a floating-point number, τ is computed as:

$$\tau = \begin{cases} 1 & \text{if } \texttt{idx} \leq \lfloor (2/\sqrt{2} - 1) \cdot 2^i \rfloor, \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

This way, when $m' \approx 1$, we may have $\texttt{idx} = 0$ and $\tau = 1$, while τ should be 0. Hence in this particular case, the value of τ must be explicitly set to 0.

Note that τ in (8) may not correspond exactly to the definition of τ in Section II-A: this is not an issue since we have a certain latitude for the value involved in the test. Usually $\sqrt{2}$ is chosen to re-center the fraction around 1, but other values around $\sqrt{2}$ do also work.

C. How to ensure faithful rounding?

Let r be an approximation of $\log(x)$, with $x \neq 1$ a positive floating-point number. For accuracy purpose, the value r is represented as the unevaluated sum of two floating-point numbers r_h and r_ℓ in precision p, that is

$$r = r_h + r_\ell \quad \text{with} \quad |r_\ell| < \tfrac{1}{2} \operatorname{ulp}(r_h) \leq 2^{-p} \cdot |r_h|. \tag{9}$$

Our goal is to compute r so that r_h is a "faithful rounding" of $\log(x)$, that is, either $\operatorname{RD}(\log(x))$ or $\operatorname{RU}(\log(x))$. To do this, following [16], a sufficient condition is:

$$|r_h - \log(x)| < \operatorname{ulp}(\log(x)). \tag{10}$$

Using the triangular inequality, we have:

$$|r_h - \log(x)| \leq |(r_h + r_\ell) - \log(x)| + |r_\ell|.$$

Hence from (9) and assuming $\operatorname{ulp}(\log(x)) = \operatorname{ulp}(r_h)$, we deduce that if

$$|(r_h + r_\ell) - \log(x)| \leq \tfrac{1}{2} \operatorname{ulp}(r_h), \tag{11}$$

then (10) holds. When $\log(x) \in]2^{k-1} \cdot (2 - 2^{-p}), 2^k[$, with $k \in \mathbb{Z}$, we have $\operatorname{ulp}(\log(x)) = \frac{1}{2} \operatorname{ulp}(r_h)$: these cases are ignored at implementation time, and the accuracy of the result for these particular inputs is checked *a posteriori* by exhaustive testing.

Now let us distinguish three cases:

- Case 1: When $e' + \tau - \lambda = 0$, and $x' \in \mathcal{X}$ with
$$\mathcal{X} = \left[\frac{1 + \kappa}{1 + 2^{-i-1} - 2^{1-p}}, \frac{1 - \kappa}{1 - 2^{-i-2}} \right],$$
we show that $\log(2^\tau \cdot r_i) = 0$, and $\log(x) = \log(1 + u)$. In this case, since $|\log(x)| > 2^{-p}$, we have $|r_h| \geq 2^{-p}$ and $\operatorname{ulp}(r_h) \geq 2^{-2p+1}$. (See [13, Prop. 3] for details.)
- Case 2: When $e' + \tau - \lambda = 0$ and $x' \notin \mathcal{X}$:
$$|\log(x)| > \min$$
with $\min = \min(|\log(\inf(\mathcal{X}))|, |\log(\sup(\mathcal{X}))|)$. Then
$$|r_h| \geq \min \quad \text{and} \quad \operatorname{ulp}(r_h) \geq \operatorname{ulp}(\min).$$
- Case 3: Otherwise, $e' + \tau - \lambda \neq 0$: thus we have $|\log(x)| > \log(\sqrt{2})$. Hence without loss of generality, we deduce that $|\log(x)| > 2^{-2}$, then $|r_h| \geq 2^{-2}$ and $\operatorname{ulp}(r_h) \geq 2^{-p-1}$.

It follows that if

$$|r - \log(x)| \leq \theta \text{ with } \theta = \begin{cases} 2^{-2p} & \text{in Case 1,} \\ \operatorname{ulp}(\min)/2 & \text{in Case 2,} \\ 2^{-p-2} & \text{in Case 3,} \end{cases} \tag{12}$$

then (11) holds, and r_h is a "faithful rounding" of $\log(x)$. For example, for $(p, i) = (24, 8)$, we have:

$$\min \approx 6.1 \cdot 10^{-4} \quad \text{and} \quad \operatorname{ulp}(\min) = 2^{-34}.$$

Distinguishing these three cases will help us in certifying the error entailed by the evaluation of the program in finite precision as explained in Section IV-B.

III. Guidelines to build accurate enough program

This section gives some guidelines to build a program that returns a "faithful rounding" of $\log(x)$. It is based on the evaluation of a particular polynomial. Then, this section presents how to constraint some coefficients of this polynomial, so that the special input 1 can also be handled in the general flow.

A. Program to handle the general flow

Let $x \neq 1$ be a floating-point number. Our goal is now to build a program that computes $r = r_h + r_\ell$ as in (12). Following [13], we use a 3-step process:

- We consider $\log(x)$ as the exact result of the function F defined as in (3):

  $$F(x) = (e' + \tau - \lambda) \cdot \log(2) - \log(2^\tau \cdot r_i) + \log(1 + u).$$

- Since F cannot be evaluated directly, we approximate the function F by another function P:

  $$P(x) = (e' + \tau - \lambda) \cdot L_2 + \text{log\_tbl}(i) + a(u).$$

  Here L_2 approximates $\log(2)$ so as $|L_2 - \log(2)| \leq \theta_1$, and log_tbl($i$) returns $-\log(2^\tau \cdot r_i)$ with an error no greater than θ_2. And finally $a(u)$ is a polynomial approximant of the function $\log(1+u)$ over the interval $\mathcal{I}$ defined in (6) for ϵ, and with the approximation error defined as:

  $$\max_{u \in \mathcal{I}} |\log(1+u) - a(u)| \leq \theta_3.$$

- We finally evaluate P by a finite-precision evaluation program $\mathcal{P}$, that computes $r_h + r_\ell$ and returns r_h.

Using the triangular inequality, we have:

$$\begin{aligned} |r - \log(x)| &\leq |F(x) - P(x)| + |P(x) - \mathcal{P}(x)| \\ &\leq |e' + \tau - \lambda| \cdot \theta_1 + \theta_2 + \theta_3 + \theta_4, \end{aligned}$$

where θ_4 is a bound on the evaluation error of $\mathcal{P}$. Assuming the three cases of Section II-C, in order to ensure that the condition in (12) holds, we must build a program so that

$$|e' + \tau - \lambda| \cdot \theta_1 + \theta_2 + \theta_3 + \theta_4 \leq \theta. \tag{13}$$

This is detailed in Section IV-B.

B. How to handle the special input 1 in the general flow?

Let us now consider the case $x = 1$. Hence we have $x' = 1$, and $m' = 1$. And from (4), we know that

$$\texttt{rcp}(m') \in [1 - \kappa, 1 + \kappa].$$

Recall that the quantity r_i equals $\texttt{rcp}(m')$ plus one half-ulp on i bits, then truncated on i fraction bits. Therefore if the parameter i is chosen such that

$$2^{-i-2} \geq \kappa \quad \text{and} \quad 2^{-i-1} \leq 2^{-i} - 2^{1-p} - \kappa, \tag{14}$$

then, we can deduce that $r_i = 1$. In this case, the computation of $\log(x)$ is reduced to the evaluation of $\log(1+u)$. For our implementations, this appears for all the values i in Table I, but for $i = 10$ when $(p, \kappa) = (24, 1.5 \cdot 2^{-12})$ and $(p, \kappa) = (53, 1.5 \cdot 2^{-12} + 2^{-24} + 1.5 \cdot 2^{-36})$.

As a consequence, since u is the error entailed by the approximation of $1/m'$ by r_i, we deduce also that $u = 0$. It follows that to handle the special input 1 in the general flow, i must be chosen to satisfy (14), and the polynomial approximant $a(u)$ in Section III-A must be built so as the first coefficient (a_0) is zero, to ensure that $a(u) = 0$ in this case.

IV. Some implementation details

This section gives some details on our implementation process, for the example of the binary32 format. After some elements on the way used to write branchless programs, it presents how to certify the accuracy of the output programs.

A. Branchless program

For performance purpose, a key point consists in writing branchless programs. First τ is a boolean introduced to avoid the catastrophic cancellation that may occur while reconstructing the result. It is actually an integer $\in \{0, 1\}$ determined in a branch-free fashion using comparison instructions.

Second, we must compute $x' = x \cdot 2^\lambda$, defined in (2). Once λ is known, this could be performed with a floating-point multiplication. However, as observed in [17], multiplying two floating-point numbers may lead to a huge overhead, when one of both operands is a subnormal number which may be the case. An alternative is to work on the bit string encoding x using the standard binary interchange format encoding [3, § 3.4]. Let the following bit string encode a binary32 subnormal floating-point number x:

$$0\ \ 00000000\ \ \underbrace{000000000}_{\lambda=9}101010000000000.$$

Computing $x \cdot 2^\lambda$ just consists in shifting this bit string by $\lambda + 1$ bits to the left. With $\lambda = 9$, we obtain:

$$0\ \ 00000001\ \ 01010000000000\underbrace{0000000000}_{\lambda+1=10}$$

which encodes $x' \geq 2^{e_{\min}}$. This sequence of operations relies on arithmetic and logical instructions, but it requires a routine returning λ if x is a subnormal number, and 0 otherwise. The instruction set we target does not embrace any instruction enabling to count the number of leading zeros in a bit string. Various techniques can be implemented in software for computing this, but either they are costly or they use branches [18]. In our implementation, we use the following piece of code, presented here for the binary32 arithmetic, but that can be adapted to any other formats. It works on an integer X, that represents the bit string of a floating-point x.

```
typedef union { uint32_t i; float f; } cfloat32;

uint32_t lambda_or_zero(uint32_t X) {
  cfloat32 Z;
  Z.i = X | 0x3f800000; // exponent of the
                        // floating-point number 1
  Z.f -= 1.f;
  uint32_t mask = 0xffffffff + ((X >> 23) != 0);
  uint32_t value = ((Z.i >> 23) - 127) & mask;
  return -value;
}
```

Table II
IMPLEMENTATION PARAMETERS FOR VARIOUS VALUES i AND $p = 24$, AND TABLE AND POLYNOMIAL COEFFICIENT SIZES (# BYTES).

i	d	θ_2	θ_3	table	coefficient
3	9	$2^{-52.69}$	$2^{-52.01}$	64	68
4	7	$2^{-52.29}$	$2^{-49.62}$	128	52
5	6	$2^{-52.10}$	$2^{-49.82}$	256	44
6	5	$2^{-51.36}$	$2^{-48.88}$	512	36
7	5	$2^{-51.20}$	$2^{-54.50}$	1024	36
8	4	$2^{-51.04}$	$2^{-48.98}$	2048	28
9	4	$2^{-51.03}$	$2^{-52.93}$	4096	28

1) If x is a normal number, its exponent field (`X >> 23`) is non-zero, and `mask` = 0. Hence the routine returns 0.
2) Otherwise if x is a subnormal number:

$$x = 2^{-126} \times 0.\underbrace{000\cdots000}_{\lambda}1\cdots.$$

Then at Line 5, we have:

$$Z = 2^{0} \times 1.\underbrace{000\cdots000}_{\lambda}1\cdots.$$

At Line 7, the subtraction is exact due to Sterbenz lemma. Thus as long as $1 - p \geq e_{\min}$, we have $Z = 2^{-\lambda} \times 1.\cdots$.The exponent field of x being zero, we deduce that `mask` = $2^{32} - 1 = 1111\cdots1111$, and `value` = $-\lambda$. Hence the routine returns λ.

This is a well-known technique, but to our knowledge, this has never been published.

B. Certified evaluation program

Recall that in this section, we target the binary32 arithmetic, that is, with $p = 24$. For accuracy purpose, we must build an evaluation program, so that the bound in (13) is satisfied. And as mentioned in Section II-C, we must distinguish three cases. Indeed, the error bound $\theta = 2^{-48}$, which is the tightest in (12), could not have been proven on the whole input interval.

From now we start by determining the polynomial degree and coefficients. To avoid branches for deciding in which cases the inputs fall, we choose to implement the $\log(x)$ function with a single degree-d polynomial. Since $\theta_1, \theta_2, \theta_3 \geq 0$, we know from (12) and (13) that L_2 and $a(u)$ must be built so as $\theta_1, \theta_3 \leq 2^{-48}$. To reach such θ_1 bound, L_2 must be represented as the unevaluated sum of two binary32 floating-point numbers. Remark also that the lookup table index size i influences the approximation interval, and thus the degree d: once the couple (i, d) is chosen, the polynomial approximant together with the certified error bounds θ_1, θ_2 and θ_3 are computed using the Sollya tool.[8] This way, we obtain $\theta_1 \approx 2^{-53.33}$. In addition Table II shows the polynomial degree d, the error bounds θ_2 and θ_3, as well as the memory occupancy in bytes for lookup table and polynomial coefficients, for different table index sizes i. In the rest of the example, we choose to implement $\log(x)$ using a table indexed by 8 bits and a degree-4 polynomial, which represents the best trade-off in terms of performance as shown in Section VI.

The remaining part is to compute the program evaluation error bound θ_4, and to check if the bound (13) holds. This error corresponds mainly to the error due to the evaluation of the polynomial approximant $a(u)$ in finite precision arithmetic, combined with the error entailed by the reconstruction. Note that the way used to evaluate $a(u)$ may be determined by using the software tool CGPE,[9] which enables to build several schemes to evaluate a given polynomial. Then bounding the evaluation error is carried out using the Gappa tool, that uses interval arithmetic and rewriting rules in order to bound error entailed by the evaluation of a program in finite precision.[10] Notice that our implementation works with double-single arithmetic, that is, with numbers represented as the unevaluated sum of two binary32 floating-point numbers. Thus we first need to adapt the error bounds computed in [22, § 4] for double-double addition and multiplication to our context, and to pass them to Gappa: we found a relative error bound of 2^{-44} and 2^{-45} for double-single addition and multiplication, respectively. Using them, we thus find that the absolute evaluation error of the polynomial is not greater than $\approx 2^{-51.75}$, which is actually less than 2^{-48}. Then according to the case, we obtain:

$$\theta_4 \leq \begin{cases} 2^{-50.75} & \text{in Case 1,} \\ 2^{-45.49} & \text{in Case 2,} \\ 2^{-36.21} & \text{in Case 3,} \end{cases}$$

and in all cases, the condition in (13) holds.[11]

V. TOWARD AUTOMATED IMPLEMENTATION

This section presents some insights on how to automate the implementation of this function. It starts with a description of the MetaLibm framework, before some details on its extension to our context.

A. The MetaLibm framework

The MetaLibm framework enables to describe the implementation of a function using a meta-language in a Python syntax. More particularly it consists in describing a mathematical function implementation as a set of high-level basic blocks. Then the MetaLibm backend translates each of these basic blocks in a specific instruction set, thus automatically outputting programs optimized for various floating-point formats and micro-architectures [1], [2]. For example, the following piece of code describes the multiplication $x \cdot x$ in this meta-language, where x is the input variable of our program.

```
y = Multiplication(x, x, tag="y",
                   precision = ML_Binary32)
```

Then, if scalar architectures are targeted, MetaLibm simply generates the sequence `y = x * x`, while if x86 architectures with AVX2 are targeted, it provides the following piece of C code, where `vec_x` is the vector equivalent of x.

[8] See http://sollya.gforge.inria.fr/ and [19].

[9] See http://cgpe.gforge.inria.fr/ and [20].

[10] See http://gappa.gforge.inria.fr/ and [21].

[11] The Sollya and Gappa scripts are available upon request.

```
carg = GET_VEC_FIELD_ADDR(vec_x);
tmp = _mm256_load_ps(carg);
y = _mm256_mul_ps(tmp, tmp);
```

As other examples, when `fma` or `rcp` instructions are not available, MetaLibm may replace them by a call to `fma`/`fmaf` libm functions or a loookup table, respectively.

So far this framework has been actively developed as a fast and efficient code generation tool for two kinds of final users. The first kind is experienced software developers who aim at implementing elementary mathematical functions libraries optimized for different architectures. The second kind is non-expert numerical software users, interested in customized mathematical code generation for their applications. Among the main features, it enables:

- To develop multi-architecture-specific meta-codes, and to support transparently different accuracies (improved, normal, degraded or customized), producing automatically different code versions,
- To factor common code across software and hardware architectures, for various IO precisions and micro-architectures, reducing thus the development time,
- To vectorize meta-code according to different micro-architectures, to fit at best the underlying architecture.

The latter mainly consists in removing code branches, and inserting tests and data selections, instead. This technique implies speculative execution, and thus increases the code size, and eventually the evaluation latency. However for throughput purpose, this may be quite efficient as shown on Cephes library [23]. Our programs are already written in a branchless fashion: thus this vectorization step will not have any impact on the produced optimized code. But this remark reinforces our choice to write branchless codes, including the handling of subnormal inputs in the general flow.

In this work, an effort has been made to enhance the back-end of MetaLibm, with all the needed instructions unavailable so far, but also to provide support to build efficient polynomial evaluation schemes, as detailed in Section V-B below.

B. The case of polynomial evaluation schemes

As shown in Section II-A, our implementations of $\log(x)$ function rely on the evaluation of a polynomial, whose degree may vary from 4 up to 9 (Table II). Various schemes may be used to evaluate a given degree-d polynomial. In order to achieve high throughputs, we chose to use CGPE (Code Generation for Polynomial Evaluation) to generate an efficient polynomial evaluation scheme on the targeted architecture. Given the polynomial coefficients and a bound on the evaluation error, it enables to automatically write and certify programs to evaluate this polynomial. At first developed for VLIW integer processors to provide fixed-point computation abilities to this kind of hardware, it has recently been extended to handle floating-point computations. But above all, one of its main features is its capability in computing polynomial evaluation schemes, exposing more or less instruction-level parallelism. To do this, it is based on exhaustive and heuristic algorithms that look for low latency polynomial evaluation schemes on abstract customizable architectures.

CGPE is written in C++ and is available as a command-line tool. Since MetaLibm is written in Python, a first step was to develop Python bindings for CGPE (called PythonCGPE). This package provides a non-exhaustive interface to CGPE features in Python that gives means to automatically generate efficient polynomial schemes given adder and multiplier latencies. It uses heuristics for polynomials of degree greater than 6 to avoid blocking MetaLibm whenever the search space grows too large. Note that even with these heuristics, the search space may get huge when the degree increases. Hence as soon as $d \geq 12$, we choose to skip the schemes computation step, and to rely only on the classic Estrin's rule. This is a well-known limit of the CGPE software tool, and solving this issue is out of the scope of this article.

VI. Numerical results

We have written various versions of $\log(x)$, and evaluate their performance. This section compares our implementations to existing solutions, and it studies the impact of three design parameters: table size, `fma`/`rcp` use, and subnormal handling.

A. Experimental protocol

We measured the reciprocal throughput of handmade and generated routines, and evaluate them against the Libmvec routines. All benchmarks were run on an Intel® Xeon® Processor E5-2650v4, which features the AVX2 instruction set extensions. The operating system was CentOS Linux release 7.4.1708, running a Linux kernel version 3.10.0-693.2.2.el7.x86_64. All source code was compiled using GCC 7.2.0, linked against glibc 2.26, both compiled from source. Shared compiler flags included `-O3 -mtune=native`. For Libmvec benchmarks, we also had to enable `-ftree-loop-vectorize -ffast-math` and link against GNU libm. The former option is used to vectorize loops, while the latter enables mathematical simplifications, flushes subnormal arguments and results to zero, among other optimizations, and is required to activate the Libmvec. For all benchmarks, to enable AVX2 code generation we simply used the `-march=core-avx2` flag, while we used `-march=corei7` to restrict GCC to emitting SSE4 code.

To evaluate these performances, we used either automated micro-benchmarks provided by MetaLibm or custom micro-benchmarks for the handmade version and Libmvec routines. These custom micro-benchmarks tried to reproduce plausible workloads by using pseudo-random inputs, but also rare workloads by using only subnormal inputs. The benchmarks were designed to warm up the L1 cache before the main measurement, so that cache-miss penalties could be kept to a minimum. Also, we chose to take the minimum measured reciprocal throughput for each micro-benchmark. We claim that this represents the best the CPU can *actually* compute when there is the least noise perturbing the measures. This can be further justified by the fact that we benchmarked separately pseudo-random input vectors versus subnormals-only

Table III
MEASURED PERFORMANCES OF OUR IMPLEMENTATIONS COMPARED TO THE GNU LIBMVEC, IN CYCLES PER ELEMENT (CPE).

Version	ISA	`-ffast-math` enabled	Accuracy	Workload	Reciprocal throughput (CPE) in binary32	Reciprocal throughput (CPE) in binary64
Libmvec 128 bits	SSE4	yes	4 ulp	Pseudo-random	1.2	5.9
Libmvec 256 bits	AVX2	yes	4 ulp	Pseudo-random	0.3	1.3
Libmvec 128 bits	SSE4	yes	4 ulp	Subnormals	6.7	40.5
Libmvec 256 bits	AVX2	yes	4 ulp	Subnormals	2.6	15.2
Handmade 128 bits	AVX2	no	1 ulp	Any	18.8	80.2
Handmade 256 bits	AVX2	no	1 ulp	Any	10.5	42.2
ML-generated 128 bits	AVX2	no	1 ulp	Any	19.0	n/a
ML-generated 256 bits	AVX2	no	1 ulp	Any	11.0	n/a

vectors, which may yield lower performance. We also benchmarked constants-only vectors, which might have yielded better throughputs for vectorized memory gather.

B. Performance of the $\log(x)$ function

In this section, we measured the reciprocal throughput of the generated routines, and compared them to SSE4 and AVX2 reference implementations coded by hand and to SSE4 and AVX2 versions of Libmvec `logf/log` routines. In this experiment, our implementations use `fma`, `rcp`, and tables indexed by $i = 8$ bits as in Section IV-B, and we use MetaLibm and custom micro-benchmarks, for generated and handmade codes, respectively. To gain in confidence, the binary32 versions have been exhaustively verified and compared with the values returned by the MPFR library.[12] Performance results are given in Table III, in cycle per element (CPE).

We observe that using our approach, we achieve a binary32 implementation with a reciprocal throughput of ≈ 18.8 CPE for vector size = 4 and ≈ 10.5 CPE for vector size = 8. In binary64, the measured performances are ≈ 80.2 CPE for vector size = 2 and ≈ 42.2 CPE for vector size = 4, that is, by a factor of ≈ 4 compared to binary32 versions. We observe similar performances for binary32 routines automatically generated with MetaLibm. Obviously, we are far away from the Libmvec implementations, whose reciprocal throughputs vary from 0.3 up to 6.7 in binary32 according to the vector sizes and the input ranges. This might be due to the fact that binary32 Libmvec implementation relies on the evaluation of a small degree polynomial, done using single precision arithmetic only, while in our cases polynomials are evaluated using double-single arithmetic. A direct consequence is the accuracy of the output: indeed our implementations are correct to 1 ulp, while Libmvec provides functions correct to at worse 4 ulp. Furthermore Libmvec has a scalar fallback for special inputs such as NaNs, infinities or zero. Since subnormal numbers are flushed to zero with `-ffast-math`, the Libmvec routines are much less efficient when dealing with a subnormals-only vector, by a factor of ≈ 5 in binary32 and ranging from 6.8 up to 11.7 in binary64. Conversely as our scheme unifies normal and subnormal handling, performance is not affected by a subnormals-only vector. (The same remarks hold for the binary64 format.)

[12] See http://www.mpfr.org/ and [24].

Table IV
MEASURED PERFORMANCES (CPE) ON AVX2 OF OUR GENERATED IMPLEMENTATIONS, IN BINARY32 ARITHMETIC AND FOR VECTORS OF SIZE 4 (V4) AND 8 (V8), WITH OR WITHOUT FMA/RCP.

i	3	4	5	6	7	8	9
v4 / original	57.7	49.6	43.4	20.3	20.4	19.0	19.0
v8 / original	30.3	25.5	13.5	12.2	12.2	11.0	11.0
v4 / no-fma	74.8	66.6	56.1	51.9	51.9	49.6	49.6
v8 / no-fma	39.8	35.6	30.4	27.7	27.7	26.5	26.5
v4 / no-rcp	98.0	70.3	65.3	55.0	49.5	46.9	46.9
v8 / no-rcp	55.3	42.0	34.2	29.3	25.7	15.3	15.3

An interesting observation is that the multiple-indices vectorized accesses to tabulated values were not penalizing, although data were accessed at non-contiguous locations. This may be explained by the fact that all tables are small — typically less than 2 kB for binary32 precision and less than 4 kB for binary64 — therefore fitting easily in the 32-kB L1d cache of the tested processor.

C. Impact of table size and architectural features

The range reduction presented in Section II-A heavily relies on some particular architectural features, like `fma` and `rcp` instructions. In this section, we modified our meta-implementation, to produce binary32 routines relying on multi-word arithmetic and table lookup, instead, respectively, to observe the impact on performance of using these features. Table IV reports the performance of these routines in CPE, for different table index sizes, from 3 up to 9 bits, and for different vector sizes (typically 4 and 8),

As expected, the performance of the routines increases when the index size and consequently the table size increase. Using this table we may conclude that, for performance purpose, a table indexed by 8 bits seems to be a good choice, with a thoughput of 19.0 CPE.

Note that `fma` and `rcp` instructions have a great impact on performance of our generated implementations. Indeed by replacing `fma` with multi-word arithmetic, we observe a slowdown factor between 1.29 and 2.61 for vector size = 4, and between 1.31 and 2.4 for vector size = 8. Regarding `rcp`, using table lookup, instead, leads to a slowdown of a factor between 1.41 and 2.7 for vector size = 4, and between 1.39 and 2.53 for vector size = 8. In the latter case, Property 1

Table V
MEASURED PERFORMANCES (CPE) ON AVX2 OF OUR GENERATED IMPLEMENTATIONS, IN BINARY32 ARITHMETIC AND FOR VECTORS OF SIZE 4 (V4) AND 8 (V8), WITH OR WITHOUT SUBNORMAL HANDLING.

i	3	4	5	6	7	8	9
v4 / no-sub	56.3	47.3	21.2	19.0	19.0	17.8	17.8
v8 / no-sub	28.7	24.7	12.9	11.5	11.5	16.6	16.7

cannot be verified: u must be represented as the unevaluated sum of two floating-point numbers, leading to huge overhead in the polynomial evaluation. This reinforces the interest and efficiency of our approach on recent micro-architectures, supporting both `fma` and `rcp` instructions.

D. Impact of subnormal handling

In this section, we measured the cost of supporting subnormals in the main flow. Thus we modified our meta-implementation, to generate binary32 routines with and without subnormal handling. To unplug subnormal handling, it suffices to remove λ from all the computations. Table V shows the performance of these routines in CPE, for a parameter i ranging from 3 up to 9, for different vector sizes (typically 4 and 8).

An interesting observation is that treating subnormal inputs in the main flow has no great impact on the performance of the function. Indeed, for vector size = 4, the overhead remains no greater than 2.3 CPE, expect for $i = 5$ (22.2 CPE), and it is always less than 2 CPE but for $i = 4$, which is acceptable. And for vector size = 8, this overhead is always less than 1 CPE, but for $i = 3$ (1.6 CPE). Remark that for $i \in \{8, 9\}$, this technique degrade routine performance, compared to the original. This reinforces our choice to avoid branches and fallback routines to treat subnormals separately.

VII. CONCLUSION AND PERSPECTIVES

This article addresses the implementation of $\log(x)$ functions on vector micro-architectures, with a particular focus on its automation through the MetaLibm framework. This enables to achieve high throughput implementations with 1-ulp accuracy, optimized for AVX/SSE and AVX2 micro-architectures, with a relatively small overhead due to handling subnormals in the main flow (less than 2 CPE).

In Table III, "n/a" indicates that MetaLibm were not able to generate these functions. This is actually due to the lack of certain meta-instructions in the framework backend. Ongoing work focuses on integrating these requirements. In a near future, this will enable generation of binary64 implementations, as well. More generally, the IEEE 754 standard defines two other formats, namely, the binary16 and binary128. These techniques being parametrized by the precision, as a consequence, using these to write implementations for these formats would be reachable and of interest. This would require efforts to eventually emulate underlying required missing arithmetic (SIMD support for binary128, for example), and to adapt the MetaLibm backend in consequence.

Research direction is twofold: First we could extend our efforts to the implementation of certain elementary functions. Following [13], a direct extension would be the optimized implementations of $\log_2(x)$ and $\log_{10}(x)$. In addition, we could concentrate on the design of efficient exponential or trigonometric functions, which are also widely used in HPC. Second we could adapt our work to produce programs with different accuracy levels, like in [5] where 8 ulp is guaranteed.

REFERENCES

[1] N. Brunie, "Contributions to computer arithmetic and applications to embedded systems," Ph.D. dissertation, 2014.

[2] N. Brunie, F. de Dinechin, O. Kupriianova, and C. Lauter, "Code generators for mathematical functions," in *22d IEEE Symposium on Computer Arithmetic*, Jun. 2015.

[3] "IEEE standard for floating-point arithmetic," IEEE Std. 754-2008, pages 1–58, 2008.

[4] D. Piparo, V. Innocente, and T. Hauth, "Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions," *Journal of Physics: Conference Series*, vol. 513, no. 5, p. 7, 2014.

[5] C. Lauter, "A new open-source SIMD vector libm fully implemented with high-level scalar C," in *2016 50th Asilomar Conference on Signals, Systems and Computers* , Pacific Grove, United States, Nov. 2016, pp. 407 – 411.

[6] P. T. P. Tang, "Table-driven implementation of the logarithm function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 16, no. 4, pp. 378 – 400, 1990.

[7] S. Gal, "Computing elementary functions: A new approach for achieving high accuracy and good performance," in *Proceedings of the Symposium on Accurate Scientific Computations*. London, UK, UK: Springer-Verlag, 1986, pp. 1–16.

[8] S. Gal and B. Bachelis, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Transactions on Mathematical Software*, vol. 17, no. 1, pp. 26–45, 1991.

[9] W. F. Wong and E. Goto, "Fast evaluation of the elementary functions in double precision," in *Twenty-Seventh Annual Hawaii International Conference on System Sciences*, 1994, pp. 349–358.

[10] ——, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers," pp. 278–294, 1994.

[11] ——, "Fast evaluation of the elementary functions in single precision," *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 453–457, 1995.

[12] M. Schulte and E. Swartzlander, "Exact rounding of certain elementary functions," in *Proc. of the 11th IEEE Symposium on Computer Arithmetic (ARITH'11)*, 1993, pp. 138–145.

[13] G. Revy, "Automated design of floating-point logarithm functions on integer processors," in *ARITH 23*, Jul. 2016.

[14] C. Lauter, "Arrondi correct de fonctions mathématiques - fonctions univariées et bivariées, certification et automatisation," Ph.D. dissertation, Université de Lyon - École Normale Supérieure de Lyon, 2008.

[15] F. de Dinechin, C. Lauter, and J.-M. Muller, "Fast and correctly rounded logarithms in double-precision," *RAIRO - Theoretical Informatics and Applications*, vol. 41, no. 1, pp. 85–102, 2007.

[16] J.-M. Muller, "On the definition of ulp(x)," 2005.

[17] H. de Lassus Saint-Geniès and G. Revy, "Performances de schémas d'évaluation polynomiale sur architectures vectorielles," in *ComPAS: Conférence en Parallélisme, Architecture et Système*, Jul. 2016.

[18] H. S. W. Jr., *Hacker's Delight*. Addison-Wesley, 2003.

[19] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *Mathematical Software - ICMS 2010*, ser. Lecture Notes in Computer Science, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., vol. 6327. Heidelberg, Germany: Springer, September 2010, pp. 28–31.

[20] C. Mouilleron and G. Revy, "Automatic Generation of Fast and Certified Code for Polynomial Evaluation," in *Proceedings of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*, E. Antelo, D. Hough, and P. Ienne, Eds. Tuebingen, Germany: IEEE Computer Society, July 2011, pp. 233–242.

[21] G. Melquiond, "De l'arithmétique d'intervalles à la certification de programmes," Ph.D. dissertation, ÉNS Lyon, France, 2006.

[22] C. Lauter, "Basic building blocks for a triple-double intermediate format," Tech. Rep. RR-5702, 2005.

[23] D. Piparo, "The VDT Mathematical Library," 2nd CERN Openlab/INTEL Workshop on Numerical Computing, 2012.

[24] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, 2007.