Computing floating-point logarithms with fixed-point operations

Julien Le Maire
Univ Lyon, ENS Lyon
julien.le_maire@ens-lyon.fr

Nicolas Brunie
Kalray
nicolas.brunie@kalray.eu

Florent de Dinechin
Univ Lyon, INSA Lyon, Inria, CITI
florent.de-dinechin@insa-lyon.fr

Jean-Michel Muller
Univ Lyon, CNRS, Inria, LIP
jean-michel.muller@ens-lyon.fr

***Abstract*—Elementary functions from the mathematical library input and output floating-point numbers. However it is possible to implement them purely using integer/fixed-point arithmetic. This option was not attractive between 1985 and 2005, because mainstream processor hardware supported 64-bit floating-point, but only 32-bit integers. This has changed in recent years, in particular with the generalization of native 64-bit integer support. The purpose of this article is therefore to reevaluate the relevance of computing floating-point functions in fixed-point. For this, several variants of the double-precision logarithm function are implemented and evaluated.**

Formulating the problem as a fixed-point one is easy after the range has been (classically) reduced. Then, 64-bit integers provide slightly more accuracy than 53-bit mantissa, which helps speed up the evaluation. Finally, multi-word arithmetic, critical for accurate implementations, is much faster in fixed-point, and natively supported by recent compilers.

Thanks to all this, a purely integer implementation of the correctly rounded double-precision logarithm outperforms the previous state of the art, with the worst-case execution time reduced by a factor 5. This work also introduces variants of the logarithm that input a floating-point number and output the result in fixed-point. These are shown to be both more accurate and more efficient than the traditional floating-point functions for some applications.

***Keywords*-floating-point, fixed-point, elementary function, logarithm, correct rounding**

I. Introduction

Current processors essentially support two families of native data-types: integer, and floating-point. In each family, several variants are supported. Integer sizes are typically 8 and 16 bits in low-power controllers such as MSP430, and 8, 16 and 32 bits for middle-range embedded processors such as the ARM family. Workstation processors and high-end embedded ones have recently added native 64-bit integer support, essentially to address memories larger than 2^{32} bytes (4Gb). Floating-point was standardized in 1985 with two sizes, 32 and 64 bits. A 80-bit format is supported in the x86 and IA64 families. The 2008 revision of the IEEE-754 standard [1] has added a 128-bit format, currently unsupported in hardware.

To compute on reals, floating-point numbers are the preferred format. The main reason for this is their ease of use, but performance is another compelling reason: until recently, native support for floating-point was more accurate (64-bit versus 32). Therefore, to compute on precisions larger than 32 bits, using floating-point was faster than using integers (Figure 1). Indeed, the floating-point units and memory subsytems of workstations processors are designed to achieve maximum floating-point performance on essential numerical kernels, such as the vector product around which all linear algebra can be built. For such kernels, at the time of writing this article, it remains true that a high-end processor can achieve more floating-point operations than integer operations per seconds [2]. This is mainly due to the wide vector units (such as Intel's AVX extensions) not fully supporting 64-bit integer multiplication.

However, integer computations are inherently simpler than floating point. For instance, integer addition takes one cycle (actually it more or less defines the processor cycle). Conversely, floating-point addition involves exponent processing, shifts and leading-zero counts: it typically requires 3 to 6 cycles.

Besides, integers may provide more accuracy in situations where the range of the data is well framed: in such cases the exponent is known, and its bits in a floating-point representation are therefore wasted.

One such situation is the the implementation of an elementary function from the standard mathematical library (libm). There, the programmer has a very good understanding of the ranges of all the intermediate data. Indeed, integer-only implementations have been studied for embedded processors lacking hardware floating-point, such as the Intel/ARM XScale processor [3] or the STMicroelectronics ST200 [4].

Still, for high-end processors, mainstream elementary function implementations use floating-point for the internal computations, probably because of the performance gap shown in Figure 1. The objective of the present work is to re-evaluate this design choice.

The logarithm was chosen for two reasons. The first is that it is one of the easiest functions to implement, with a large body of literature and open-source implementations to build upon, and a reasonably simple range reduction. The second

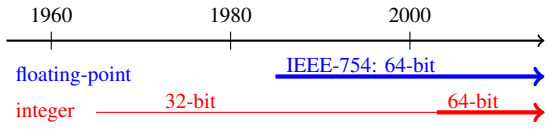

Figure 1: Arithmetic capabilities of mainstream processors

was to study a floating-point-in, fixed-point-out variant that will be motivated and detailed in Section VI.

A remarkable conclusion is that the implementation effort is comparable or lower than for an equivalent floating-point implementation. Once the proper tools are set up (reviewed in Section II), reasoning on integers is actually easier than reasoning on floating-point. Proving code is even easier.

A second remarkable conclusion is that the performance is well on par with floating-point implementations for 64-bit computations. Besides, as soon as intermediate computations require double-word arithmetic, integers offer large speed-ups over floating-point. This is the case for correctly-rounded functions [5]: the present work demonstrates a worst-case performance improvements by a factor 5 over the previous state of the art. This worst-case time is even better than the *average* time of the default GNU/Linux `log` function.

The remainder of this article is organized as follows. Section II presents in more details fixed-point arithmetic, the art of representing reals using integers. Section III introduces the variants of the double-precision logarithm that are studied in this work. Section IV presents the algorithm used, with a focus on new techniques in the integer context. Section V presents and analyses performance results. Section VI advocates, with an application case study, a new interface to the 64-bit logarithm, inputting a floating-point number but outputting a fixed-point result. Section VII concludes.

II. Floating-point versus fixed-point arithmetic

A. IEEE-754 floating-point

Floating-point numbers are commonly used in computer science as an easy-to-use and accurate approximation for real numbers. Binary floating-point computing was first introduced in the Zuse Z2 electro-mechanical computer in the 1940s. A sign bit s, an integer exponent e and a fraction $f \in \pm[1,2[$ are used to represent the number $(-1)^s \cdot 2^e \cdot f$.

The IEEE-754 Standard for Floating-Point Arithmetic, last revised in 2008, defines several floating-point formats, the main ones being a 32-bit format and a 64-bit format respectively called binary32 and binary64 in the following. More importantly, it also precisely defines the behaviour of the basic operations on these formats. The standard mandates that an operation must return floating-point number uniquely defined as the exact result, rounded according to a well-specified rounding mode. The 2008 revision of the standard also includes specifications of the most used elementary functions, such as sin, tan, exp, log, $\log_{10}$... However, correct rounding is only recommended for them, due to its perceived high performance overhead. A contribution of this article is to reduce this overhead.

B. Generalities on fixed-point arithmetic

In a fixed-point program, each variable is an integer scaled by a constant exponent. This exponent is implicit: it is not stored in the number itself, contrary to the exponent of a floating-point number. Therefore, it has to be managed explicitely. For instance, to add two variables that have different implicit exponents, an explicit shift must be inserted in the program before the integer addition in order to align both fixed points (see Figure 2). Thus, implementing an algorithm in fixed point is more complex and error-prone than implementing the same algorithm in floating point:

- It requires a good understanding of the orders of magnitude of all the variables, in order to decide where their fixed point should be.
- There is no representation of infinities as in floating-point. Possible overflows have to be predicted and explicitely managed.
- Integer numbers may be signed (using two's complement), or not.
- The cheapest rounding mode available is truncation (which can be achieved by shifting right or or masking). This corresponds to a rounding toward $-\infty$, both for signed and unsigned arithmetic. Rounding to the nearest is achieved by $\circ(x) = \lfloor x+1/2 \rfloor$ at the cost of an addition.
- When multiplying two integers of the same size, the size of the result is twice the size of the inputs. If the inputs were fractional fixed-point numbers, the high part of the product (which holds the most significant bits) is needed, as illustrated by Figure 3.

```
  0x0.00001234
+ 0x0.00567800
--------------
= 0x0.00568A34
```

(a) Math. view

```
uint32_t a = 0x1234;
  // exponent: -32
uint32_t b = 0x5678;
  // exponent: -24
uint32_r r = a + (b << 8);
  // exponent: -32
```

(b) Source code in C

Figure 2: Addition of two fixed-point numbers.

```
uint32_t a = 0x12345678;  // 0x0.12345678
uint32_t b = 0x9ABCDEF0;  // 0x0.9ABCDEF0
uint64_t r = (uint64_t)a * b;
                // = 0x0.0B00EA4E242D2080
uint32_t s = (r >> 32); // ~ 0x0.0B00EA4E
```

Figure 3: Multiplication of two fixed-point numbers.

C. Support of fixed-point in current hardware and compilers

A limited set of primitives are needed to efficiently implement floating-point elementary functions in fixed-point: addition and multiplication, boolean operations (to implement bit masks, etc), but also leading zero count and shifts (to convert between fixed-point and floating-point). Processor instructions exist to perform most of these tasks. On the x84-64 architecture [6], [7], they are available on 64 bits natively,

and can be chained efficiently to operate on 128-bit data. For instance,

- a 128-bit addition can be computed by two 64-bit instructions: an `add`, then an `adc` (add with carry);
- the full multiplication (of two 64-bit values into a 128-bit one) is done with the `mulq` assembler instruction;
- The `lzcnt` instruction performs leading zero count.

What is more, the integer datatypes `__int128_t` and `__uint128_t`, while not belonging to standard C, are supported by most mainstream compilers (GCC, ICC, Clang). The addition in C of two `__int128_t` will generate the sequence of `add` and `adc` previously mentioned. A full multiplication of two 64-bit integers can be achieved by casting operands to the types `__uint128_t` and `__uint128_t`, similarly to Figure 3. The generated code will only consist of one `mulq` instruction. We define the macro `fullmul` for this (it inputs two int64 and returns an int128), and we also define a macro `highmul` (that returns the high 64 bits only).

Finally, instructions that are not exposed in the C syntax (such as add-with-carry or leading-zero-count) can be accessed thanks to intrinsics which are also de-facto standard. For example, in GCC, ICC and CLang, 128-bit leading zero count can be achieved with the builtin function `__builtin_clzll`.

To sum up, writing efficient code manipulating 64-bit and 128-bit fixed-precision numbers is possible in a de-facto standard way.

III. Variants of the log function

The logarithm function, along with the exponential, it is one of the most useful elementary functions. It is continuous and monotonically increasing, and also has the following mathematical property:

$$\log(x \cdot y) \quad = \quad \log(x) + \log(y) \quad .$$

Its output range is much smaller than the input range: for any binary64 x, $|\log(x)| < 2^{10}$). Therefore, it is easy to define fixed-point formats that capture the whole output range of a logarithm function. Such formats have 11 bits to the left of the fixed-point (including a sign bit).

Therefore, the core of our algorithm will first compute an approximation of $\log(x)$ on such a format (with the fraction size depending on the variant studied). It will then round this approximation to the nearest floating-point number.

However, an option is also to skip this final rounding. Section VI shows an application case study where this improves both performance and accuracy.

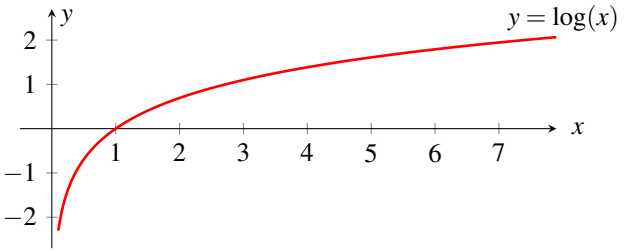

Figure 4: The natural logarithm function

A. Correctly rounded logarithm

Modern implementation of elementary functions are at least *faithful*, i.e. they return one of the two floating-point numbers surrounding the exact value $\log(x)$ [8], [9], [10]. To achieve this, it is enough to compute an approximation with a confidence interval smaller than the difference between two consecutive floating-point numbers (Figure 5). This requires to use an internal format of slightly higher precision than the destination floating-point format.

Correct rounding is more difficult to achieve. In the cases when the confidence interval includes a rounding boundary, the correctly rounded value can not be decided (Figure 5, bottom). To ensure correct rounding, one has to make sure that the exact result $y = f(x)$, which can be anywhere in the confidence interval, is on the same side of the midpoint as the computed approximation.

Ziv's algorithm [11] recomputes in such cases the function with increasing accuracy, reducing the confidence interval until it is completely on one side of the rounding boundary. As such recomputations are rarely needed, this algorithm is fast in average. However, it may take a lot of time when the actual solution is very close to a mid-point between two consecutive floating-point numbers (Figure 5).

To control this, the worst-case situation among all possible inputs can be computed beforehand [12]. Thus, Ziv's iteration can be limited to two phases [5], and this is the approach taken in the present work.

The first *(quick)* phase computes (in a 128-bit integer) an approximation with an absolute accuracy of 2^{-63}. This is enough to return correct rounding in most cases. Otherwise, the second phase is lauched.

The second *(accurate)* phase computes to an absolute accuracy of 2^{-118} which, according to [12, p. 452] is enough to ensure correct rounding for all floating-point inputs.

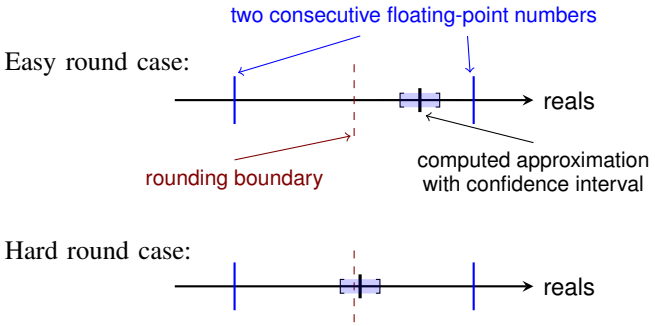

Figure 5: Rounding to the nearest an accurate approximation.

B. Variants of the binary64 logarithm function

From this two-phase implementations, several variants can be derived.

- A single-phase correctly-rounded, IEEE-754-2008 compliant variant, essentially consisting of the accurate phase only.
- A floating-point in, fixed-point out log, outputting an `int64_t` with an absolute accuracy of 2^{-52}, essentially consisting of the first phase without the final conversion to floating-point.
- A floating-point in, fixed-point out log outputting an `int128_t` with an absolute accuracy of 2^{-116}, essentially consisting of the second phase only without final conversion to floating-point.

The last two functions will be justified by an application case study in Section VI.

These variants are simplifications of the two-phase correctly rounded implementation, whose algorithm is presented now.

IV. Algorithm and implementation

The algorithm can be decomposed into the following steps:

A. Filtering the special cases

The algorithm begins with a copy of the binary representation of x into an `int64_t` variable. From this, normal input can be decomposed as $2^E \times x$ where x is a normalized fraction $x = 1 + 2^{-52} f_{52} \in [1;2)$.

Before that, however, the following special cases must be handled [1]: input is negative or zero; input is $\pm\infty$; input is *NaN*; input is a subnormal number. One test (predicated false) is enough to detect all thoses cases [13]. It captures the cases when the sign bit is set, or the exponent is either the min or max value. This test is performed on the binary representation of the input.

On all these special cases, the function returns a special value, except for subnormals: these are normalized, the input exponent is updated accordingly, and the control flow gets back to the normal case.

B. Argument range reductions

The first range reduction is straightforward:

$$\log(2^E \times x) = E \times \log(2) + \log(x)$$

When computing $E \times \log(2) + \log(x)$, there is a possible catastrophic cancellation when x is close to 2 and E =-1, i.e. when the input is very close to (but smaller than) 1. To avoid it, most floating-point implementations [8], [5] first re-center the fraction around 1, using a variant of the following test: if x>1.4 then x=$x/2$ and E=E+1. We currently choose to skip this classical recentering. This saves the cost of this test itself, but also a few additional operations in the subsequent range reductions, where round to nearest would be needed instead of truncation. The trade-off is that the accurate phase will be launched in case of a massive cancellation. This design choice, initially motivated by simplicity, is *a posteriori* justified by the low overhead of the accurate phase in the present implementation. However its relevance will be discussed in the conclusion.

The constant $\log(2)$ can be stored to arbitrary accuracy as several 64-bit chunks. This, as well as the computation of $E \times \log(2)$, will be detailed in Section IV-D.

Let us now address the computation of $\log(x)$ with $x \in [1,2)$. This interval can be further reduced by an algorithm due to Tang [14]. The main idea is to read, from a table addressed by the k most significant bits of x, two values: $inv_x \approx \frac{1}{x}$ and $\log(inv_x)$. Then if we compute $y = inv_x \cdot x$, the logarithm can be computed as $\log(x) = \log(y) - \log(inv_x)$.

The closer inv_x is to $\frac{1}{x}$, the closer y will be to 1. Indeed, if $|inv_x - \frac{1}{x}| \leq \varepsilon$ then:

$$|y-1| \;\leq\; |inv_x \cdot x - 1| \;\leq\; |inv_x - \frac{1}{x}| \cdot x \;\leq\; \varepsilon \cdot x \;\leq\; 2\varepsilon$$

Since the range of y is much smaller than the range of x, computing $\log(y)$ will be easier than computing $\log(x)$.

In detail, let x_1 the value of x taking only in consideration its k leading bits. The value stored is $inv_x = \lceil 1/x_1 \rceil$, in order to ensure $y \geq 1$. This ensures that we can use only unsigned arithmetic in the sequel.

Then $y = inv_x \cdot x$ can be computed exactly, because x is a binary64 mantissa (53 bits) and we work with 64-bit integers. Therefore, with a value of inv_x on 11 bits, the product can be computed exactly in the low part of a 64-bit product. Besides, the k leading bits of y are known, as illustrated by Figure 6, so their overflow in the high part of the product is harmless: for $k = 7$, it is actually possible to approximate inv_x on 18 bits and still compute $y - 1$ exactly in a 64-bit integer.

Thanks to this, it is possible to do this argument range reduction twice. First, the 7 leading bits of x (excluding the constant 1) are used as an index in a table that provides inv_x and $\log(inv_x)$. Then, $y - 1$ has its 6 most significants bits being zeros (more precisely $y \in [0, 2^{-6.41504}]$). Its 7 next bits are used as an index in a table that provides inv_y as well as $\log(inv_z)$. Now $z = inv_y \times y$ is even closer to 1 than y: $z \in [1, 1 + 2^{-12.6747}]$.

This argument reduction is depicted on Figure 7. With a total of 2^8 table entries only, it achieves a similar reduction as the single-step approach would with a 2^{13}-entry table. The

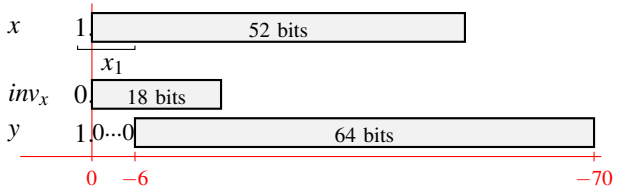

Figure 6: Tang's argument reduction, here for $k = 7$.
The gray boxes are the values represented in the program.

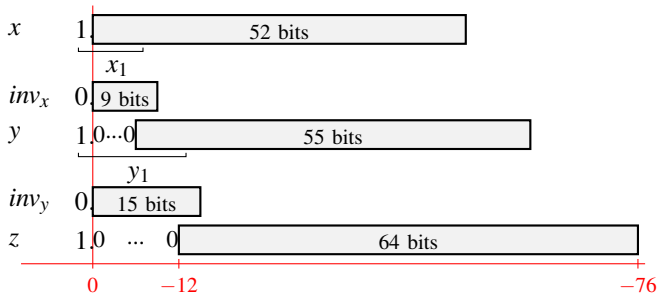

Figure 7: Two iterations of Tang's argument reduction.

Table size (bytes)	degree 1st	degree 2nd
39,936	3	5
12,288	3	6
4,032	**4**	**7**
2,240	4	8
2,016	4	9
900	5	10
594	6	12
298	7	14

Table I: A few Pareto points of the design space

delay of the second table read is more than compensated by the reduction in polynomial degree (from 7 to 4 in the quick phase and from 13 to 7 in the accurate phase). Table I shows that other trade-off between table size and polynomial degree would be possible.

This is an old idea in a hardware context [15], [16], [17] and it has been used in software in [3]. Note that it cannot be used efficiently to implement a binary64 log using binary64 arithmetic, because the latter cannot hold z (nor $z-1$) exactly.

The proof of these three steps of argument reduction was developed using Gappa [18]. We note $t = z-1$, and we now address the computation of $\log(1+t)$.

C. Polynomial approximation and evaluation

In each phase, the function $\log(1+t)$ may be approximated by a polynomial $P(t)$ of small degree (4 in the quick phase, 7 in the accurate phase). Thanks to the Taylor formula $\log(1+t) \approx t - t^2/2 \cdots$, its constant coefficient can be forced to 0: $P(t) = tQ(t)$. This polynomials is computed using the `fpminimax` command of the Sollya tool [19]. At runtime, it is evaluated using Horner's method.

This Horner evaluation can be implemented with no runtime shift, as shown on Figure 8. Instead, the approximation polynomial was constrained to include all the shifts within the coefficients themselves. More precisely, Sollya's `fpminimax` can be constrained to compute the optimal polynomial among those which will avoid any shift. This lead to coefficients with leading zeroes, which may seem a waste of coefficient bits. However, we observed that this provided almost the same accuracy as a polynomial where each coefficient is stored on exactly 64 bits, in which case shifts are needed during the evaluation. In other words, removing the shifts is for free in terms of approximation accuracy.

```
uint64_t a4 = UINT64_C(0x0000000003ffc147);
uint64_t a3 = UINT64_C(0x0000005555553dc6);
uint64_t a2 = UINT64_C(0x0007fffffffffd57);
uint64_t a1 = UINT64_C(0xfffffffffffffffa);
uint64_t q =
   a1-highmul(t,a2-highmul(t,a3-highmul(t,a4)));
uint128_t p = fullmul(t, q);
```

Figure 8: Shift-less polynomial evaluation

In the quick phase, the evaluation of $Q(t)$ is performed completely in 64-bit arithmetic (using the `highmul` primitive). Only the final multiplication by t needs a full multiplication. As t itself is exact, this last multiplication is exact, and the 128-bit result will therefore always hold almost 64 significant bits after the leading non-zero bit.

In the accurate phase, all the coefficients a_i of Q (except a_7) must be stored in 128-bit integers. However the evaluation remains quite cheap, because t remains a 64-bit integer. Specifically, Horner step i computes $q_i = a_i + t \times q_{i-1}$ where only the q_i and a_i are 128-bit. This rectangular multiplication is implemented as

```
q = ai - (fullmul(t,HI(q)) + highmul(t,LO(q)));
```

Again, there is no shift needed here.

D. Result reconstruction

Finally, the log will be evaluated as

$$\log(input) = E\log(2) - \log(inv_x) - \log(inv_y) + P(t) \ .$$

There, $-\log(inv_x)$ and $-\log(inv_y)$ are precomputed values. Thus, we can choose to decompose and align them the way we want. The best is probably to align them to the format we want for the result, so they can be added without shift (Figure 9).

The value of $E\log(2)$ is computed by multiplying E with slices of the binary writing of $\log(2)$. These slices are chosen in such a way that the product will be aligned to the format of the result: $\log(2)$ is decomposed into $\log(2)_{-1,-53}$, $\log(2)_{-54,-118}$, $\log(2)_{-119,+\infty}$.

The product of $\log(2)_{-1,-53}$ with E (a 11-bit number) fits on a 64-bit integer: it requires only the low part of a 64-bit multiplication, and will be aligned to the high word of the result format (see figure 9).

The product of $\log(2)_{-54,-118}$ requires a `fullmul`, but the obtained 128-bit result is also aligned. These two first terms are used in the quick phase.

Only 11 more bits of $E\log(2)$ are needed for the accurate phase, but the corresponding term needs to be shifted to place. This will be done only in the accurate phase.

The approximation $P(t)$ of $\log(z)$ is finally added. Its implicit exponent is fixed by the fact that we want t exact and fitting on one int64: we therefore have to shift $tQ(t)$ before adding it.

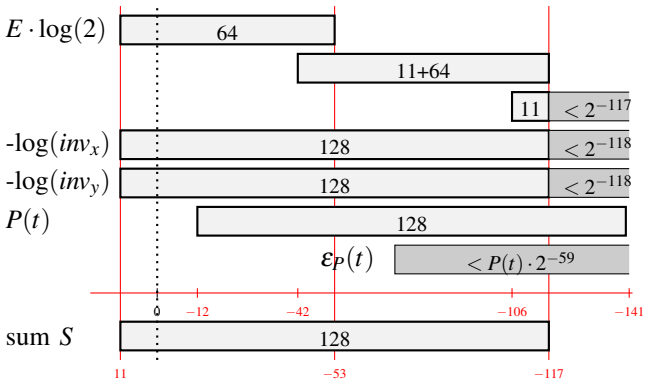

Figure 9: Solution reconstruction (error terms in darker gray)

One nice thing about fixed-point arithmetic is that this whole summation process is exact, therefore associative. The order in which it is written is not important, and this is a big difference compared to floating-point, where one has to think carefully about the parenthesing of such sums. In principle the compiler also has more freedom to schedule it than in floating-point, but it is not clear that this is exploited.

E. Rounding test

Floating-point-based correctly-rounded functions use a multiplication-based test due to Ziv, but documented much later [20]. Considering the previous summation, it is possible here to use a much simpler test. Upper bounds of the errors of each summand are added together to form an upperbound ε of the distance between our final sum S and the actual value of the logarithm. Then, at runtime, it is enough to check if $S-\varepsilon$ and $S+\varepsilon$ are rounded to two different floating-point numbers or not, as seen in the figure 5.

Figure 9 show in grey the error terms we have to sum to get ε. Most of this error summation is performed statically, only the error on $P(t)$ uses a dynamic shift, to benefit from the fact that $P(t)$ was evaluated with good relative error.

Since only sharp error bounds are added together, the value ε is also sharp, and so is the test.

F. Accurate phase

For the last step, the previous sum (before the addition of the polynomial) can be fully reused. The $\log(inv_x)$ and $\log(inv_y)$ terms were already accurate to 2^{-118} The accurate phase therefore only consists of adding the third part of $E\log(2)$, and recomputing a more accurate $tP(t)$ of degree 7, as already explained in Section IV-C.

V. Results and Analysis

A. Experimental protocol

The proposed correctly rounded implementation has been proven in Gappa, tested against the hard to round cases distributed with CRLibm, and against large numbers of random values. It passes all these tests.

The performance of this function is reported on two architectures: an Intel(R) Core(TM) i7-4600U CPU, and a Kalray Bostan core. The latter is a 64-bit, 4-way VLIW core with an FMA-based double-precision floating-point unit, several arithmetic and logic units, and a single 32-bits multiplier (no hardware 64-bits multiplier). A recent GCC is used in both cases (version 4.9). Experiments with ICC and CLang show no significant difference.

To mesure the performance of our function, we use standard procedures based on hardware counters (`rdtsc` on the Core i7, properly serialized). The functions are compiled with `gcc -O3` optimization. The test program is compiled with `-O0` to avoid unwanted optimizations. It consists of three similar loops that apply the function on an array of random data. the first calls an (externally linked) empty function, to calibrate the loop and function call overhead: all the timings are reported without this overhead. The second loop times the average case execution time of each function. The third calls the function on an array of hard-to-round cases, thus timing the worst-case execution time of the function.

The same test was run with three others implementations: **glibc** (resp **newlib**) is the function of the GNU C standard library distributed with the Linux system on the Core i7 and Bostan respectively. The glibc uses Ziv' algorithm, and is correctly rounded; **cr-td** is the function is the CRLibm variant that only uses binary64; **cr-de** is a variant of CRLibm that uses the double extended format (80 bits, with a fraction on 64 bits) available on the x86 platform. This comparison hints to what depends on having 64 bits instead of 53, and what depends on having faster arithmetic.

Table II presents some parameters of these implementations. It is worth to mention that the proposed implementation has the smallest memory consumption. The **glibc** is not a two-phase algorithm: it also uses a degree 3 polynomial for the case $|x-1| < 0.03$ (see file `u_log.c` in the **glibc** sources).

B. Timings of correctly rounded implementations

Table III compares the speed of the available correctly rounded implementations on the Core i7 processor. For an absolute reference, it also shows what is probably the best available faithful scalar implementation of the logarithm for this processor, distributed in the Math Kernel Library (MKL) with Intel's own compiler `icc`. To obtain this measure, the same test was compiled with `icc -O0 -fp-model precise -lm`. This processor-specific implementation, written by Intel

	glibc	**crlibm-td**	**crlibm-de**	**cr-FixP**
degree pol. 1	3/8	6	7	4
degree pol. 2	20	12	14	7
tables size	13 Kb	8192 bytes	6144 bytes	4032 bytes
% accurate phase	N/A	1.5	0.4	4.4

Table II: Parameters of the correctly rounded variants.

cycles	*MKL*	glibc	crlibm	cr-de	cr-FixP
avg time	*25*	90	69	46	49
max time	*25*	11,554	642	410	79

Table III: Average and max runing time (in processor cycles) of correctly rounded implementations on Intel Core i7. Italic denotes lack of correct rounding.

cycles		Core i5	Bostan
System		(glibc) 90	(newlib) *105*
cr-FixP	quick phase alone	*42*	*94*
	accurate phase alone	74	181
	both phases (avg time)	49	121
	both phases (max time)	79	225

Table IV: Running times of each step

experts, probably exploits the hardware reciprocal approximation instead of a table of inverse [9] and the FMA available in recent AVX extensions [7].

The main results are the following.

- The average time is on par with the best state of the art: **cr-de**, that uses 80-bit floating-point and larger tables. It is within a factor two of the absolute reference, MKL, which is not correctly rounded.
- The worst-case time has been improved by a factor 5, and is now within a factor two of the average time.

Table IV compares these timings (broken down in their different steps) on two processors. What is interesting here is that the Bostan performance is respectable, although it only has 32-bit multiplication support.

In Table II, the proposed cr-FixP implementation is the worst on one metric: the percentage of calls to the accurate phase. This is essentially due to the cancellation problem for values slightly smaller than 1 – the random generator we use is actually biased towards this case. It would be a bad design choice to always lauch the accurate phase for such central values if the accurate phase was very slow. However, with an accurate phase within a factor two of the quick one, it becomes sensible. We initially felt that this choice allowed us to save a few cycles in the quick phase. This feeling can only be confirmed by the design and test of an implementation including the classical recentering and more signed arithmetic. What is already clear is that the polynomial evaluation schemes we already use will provide the required *relative* accuracy in this case. Therefore, when this implementation is available, running its first step only will provide a faithful result, which is not the case of the current implementation.

VI. Floating-point in, fixed-point out variant

Finally, Table V shows the performance of the variant that returns a 64-bit signed integer approximating $2^{52}\log(x)$. This variant performs the final summation completely in int64. Therefore it requires even fewer tables. Moreover,

output format	absolute accuracy	table size	Core i5 cycles	Bostan cycles
Fix64	2^{-52}	2304	24	66
Fix128	2^{-116}	4032	60	179
double (libm)	2^{-42}		90	105

Table V: Performance of floating-point in, fixed-point out functions. Fix64 is the code of the first step only, without the conversion to float. Fix128 is the code of the second step only, without the conversion to float.

a polynomial of degree 3 is enough to achieve absolute accuracy of 2^{-59} which is largely enough in this context. With all this, and with the removal of the final conversion to a binary64 floating-point number, this function offers performance on par with the MKL.

However, the main impact of offering this variant may be in applications that compute sums of logs. An archetype of this class of applications is a DNA sequence alignment algorithm known as TKF91 [21]. It is a dynamic programming algorithm that determines an alignment as a path within a 2D array. The borders of an array are initialized with log-likelihoods, then the array is filled using recurrence formulae that involve only max and + operations.

All current implementations of this algorithm use a floating-point array. With the proposed logFix64, an int64 array can be used instead. This should lead to an implementation of TKF91 that is both faster and more accurate:

- faster, not only because logFix64 is faster, but also because the core computation can now use int64 addition and max: both are 1-cycle, vectorizable operations;
- more accurate, because larger initialization logs impose an absolute error that can be up to 2^{-42}, whereas logFix64 will have a constant absolute error of 2^{-52}.
- more accurate, also because the computation of all the additions inside the array become exact operations: they still accumulate the errors due to the initialization logs, but they do not add rounding errors themselves, as floating-point additions do.

A quantitative evaluation of these benefits on TKF91 remains to be done. Of course, applications that compute exponentials of sums could similarly benefit from a fixed-point in, floating-point out exponential.

VII. Conclusion and future work

This work has demonstrated a correctly rounded logarithm using only 64-bit integer arithmetic that outperforms the previous state of the art, improving the worst-case execution time by a factor 5 while using smaller tables. This new worst case time is within a factor 4 of the best achievable faithful implementation (the MKL). Several new techniques have been developed, the more generally useful being the rounding test and the shift-less polynomial evaluation.

Can this work be generalized? As soon as double-word arithmetic is heavily needed in the implementation of an elementary function (and this is always the case for correctly rounded versions), integer arithmetic probably becomes very relevant, essentially because 128-bit addition is a 2-cycle operation. Working in fixed point will not be a problem as soon as the arguments are reduced to a small domain, a common situation in elementary functions. For instance, the Payne and Hanek argument reduction for trigonometric functions is a fixed-point algorithm that provides a fixed-point reduced argument. Still, these claims require more experiments on other functions.

The main limitation of this work is probably that it can currently not be vectorized: vector units do not (yet?) offer the necessary multiplication operations. It is not unreasonable to expect that they could be added in the future. If this happens, running the accurate phase alone will provide a branchless implementation that we can expect to be within a factor 2 or 3 of the faithful vector version.

Implementing a function using floating point requires acute floating-point expertise, in particular a mastering of the range and accuracy of all intermediate variables. Considering this, we are also inclined to claim that an integer implementation is in many ways simpler, for instance when one has to add many terms.

A last claim of this article is that floating-point-in, fixed-point out variants of logarithms would make sense in some applications, both from a performance and from an accuracy point of view. It will be interesting to study if this idea may be generalized to other functions.

Some short-term questions remain, the main one being the possibility to achieve high relative accuracy around 1 without degrading the performance of the quick phase. Work has begun to integrate this code the Metalibm framework [22], which will make such experiments easier.

Another observation is that the published table-maker's dilemma worst-case data is not in an ideal format for a fixed-point implementation: it is currently only given in relative terms on a large range of exponents.

The best-of-class faithful implementation on Intel processors, MKL, still uses floating point. Maybe this hints that the optimal two-phase correctly rounded implementations should use floating-point in the quick phase, and integers in the accurate phase. This will add new challenges, such as sharing the tables between the phases.

ACKNOWLEDGEMENTS

This work was supported by the French National Research Agency through the INS program *MetaLibm*.

The authors are thankful to Alexandros Stamatakis for pointing to the TKF91 algorithm, to Sarah Lutteropp and Pierre Barbera for testing logFix64, and to Christoph Lauter for assistance with the test procedure.

REFERENCES

[1] "IEEE standard for floating-point arithmetic," IEEE 754-2008, also ISO/IEC/IEEE 60559:2011, Aug. 2008.

[2] C. Pernet, "Exact linear algebra algorithmic: Theory and practice," in *International Symposium on Symbolic and Algebraic Computation*. ACM, 2015, pp. 17–18.

[3] C. Iordache and P. T. P. Tang, "An overview of floating-point support and math library on the intel XScale architecture," in *16th Symposium on Computer Arithmetic*. IEEE, 2003.

[4] C.-P. Jeannerod and J. Jourdan-Lu, "Simultaneous floating-point sine and cosine for VLIW integer processors," *Application-Specific Systems, Architectures and Processors*, pp. 69–76, 2012.

[5] F. de Dinechin, C. Q. Lauter, and J.-M. Muller, "Fast and correctly rounded logarithms in double-precision," *Theoretical Informatics and Applications*, vol. 41, pp. 85–102, 2007.

[6] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Sep. 2015, no. 253669-056US.

[7] ——, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Sep. 2015, no. 248966-31.

[8] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*. Prentice Hall, 2000.

[9] M. Cornea, J. Harrison, and P. T. P. Tang, *Scientific Computing on Itanium®-based Systems*. Intel Press, 2002.

[10] J.-M. Muller, *Elementary Functions, Algorithms and Implementation*, 2nd ed. Birkhäuser, 2006.

[11] A. Ziv, "Fast evaluation of elementary mathematical functions with correctly rounded last bit," *ACM Transactions on Mathematical Software*, vol. 17, no. 3, pp. 410–423, 1991.

[12] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhauser, 2009.

[13] C. S. Anderson, S. Story, and N. Astafiev, "Accurate math functions on the Intel IA-32 architecture: A performance-driven design," in *Real Numbers and Computers*, 2006, pp. 93–105.

[14] 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.

[15] M. Ercegovac, "Radix-16 evaluation of certain elementary functions," *IEEE Transactions on Computers*, vol. C-22, no. 6, pp. 561–566, 1973.

[16] W. F. Wong and E. Goto, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 278–294, 1994.

[17] J. Detrey, F. de Dinechin, and X. Pujol, "Return of the hardware floating-point elementary function," in *18th Symposium on Computer Arithmetic*. IEEE, 2007, pp. 161–168.

[18] F. de Dinechin, C. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using Gappa," *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 242–253, Feb. 2011.

[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, vol. 6327. Springer, Sep. 2010.

[20] F. de Dinechin, C. Lauter, J.-M. Muller, and S. Torres, "On Ziv's rounding test," *ACM Transactions on Mathematical Software*, vol. 39, no. 4, 2013.

[21] J. Thorne, H. Kishino, and J. Felsenstein, "An evolutionary model for maximum likelihood alignment of DNA sequences," *Journal of Molecular Evolution*, vol. 33, no. 2, pp. 114–124, 1991.

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