Correctly rounded vector implementation of the exponential function in binary64 arithmetic

Nicolas Brisebarre*, Tom Hubrecht†, Christoph Lauter‡, Jean-Michel Muller*, Kristalys Ruiz-Rohena‡
*CNRS, ENS de Lyon, Inria Pascaline, Univ. C. Bernard Lyon 1, LIP, UMR 5668, Lyon, France
†ENS de Lyon, CNRS, Inria Pascaline, Univ. C. Bernard Lyon 1, LIP, UMR 5668, Lyon, France
‡University of Texas at El Paso, El Paso, TX, USA

***Abstract*—We present a vectorized implementation of the exponential function that is correctly rounded in rounded-to-nearest binary64 floating-point arithmetic, demonstrating the feasibility of vectorized correctly rounded implementations. The vectorization is done automatically by the compiler.**

***Index Terms*—Floating-Point Arithmetic, Exponential Function, Correct Rounding, SIMD Vectorization, binary64 Arithmetic, Vectorized Functions.**

I. Introduction

As advocated in [5], requiring correct rounding for a well chosen set of mathematical functions (exp, log, sin, ...) would have several advantages: clear and unambiguous specification of the functions, better portability of numerical programs and better reproducibility of calculations. A function is *correctly rounded* if for all floating-point inputs, the returned result is the same as the one we would obtain if we could first compute the exact result, and then apply the chosen rounding function to that result. Since 1991, a few libraries have provided correctly rounded elementary functions: examples are MathLib/LibUltim [39], CRLibm[1] [10], and more recently RLIBM [32], and CORE-MATH[2] [22], [36], which provides efficient scalar functions. Indeed, several papers have been dedicated to the *scalar* implementation of correctly rounded functions (e.g., [9], [11], [13], [15], [17], [22], [29], [30], [32], [33], [39]). As explained in [5], the main idea for implementing scalar correctly rounded functions in binary, precision-p, FP arithmetic is, after a possible initial filtering of special cases, to proceed as follows:

- first, a *fast path* delivers an approximation to the function accurate to, say, $p+\ell$ bits. A classical heuristic [5], [6] tells us that from that approximation we can deduce the correctly rounded result with probability $\approx 1-2^{-\ell}$;
- then a *rounding test* checks if the result of the fast path suffices to deduce the correctly rounded result;
- and, if needed, an *accurate path* is called, that must be accurate enough to obtain correct rounding for all inputs.

If ℓ is large enough, the average delay of the calculation is only slightly more than the delay of the fast path. A typical value of ℓ for *scalar* functions is around 10.

The aim of this paper is to present the design of correctly rounded *vector* functions, assuming round-to-nearest, ties-to-even as the desired rounding function. Several vector function libraries have been presented in the literature [1], [28], [34], [35], [37], but we believe we are the first to present *correctly rounded* vector functions. Similar to the scalar case, vector correctly rounded functions will be based on two paths, with three differences. First, as all elements of a vector follow the same stream of calculations, if the accurate path is needed for one element of a vector, it is applied to all elements of that vector. This is an implementation choice; other approaches "fix" the slots for which correct rounding could not be obtained with the fast path. The probability of having to call the accurate path is approximately equal to the size of the vector times the probability of that path being needed for one element. When the vectors grow larger (for instance of 8 or more elements), in order to maintain a very small probability for the accurate path to be called, we therefore need a significantly larger value of ℓ. Second, as the accurate path is less suited for vector computing (it may require tables, several tests, etc.), a reasonable solution is not to implement a specific *vector* accurate path, but to call a *scalar* accurate path instead. This is an additional incentive to reduce the probability of having to use the accurate path. We target a value of ℓ around 20. Finally, the rounding test must be adapted for vectorization, as it is difficult to make separate tests on all elements of a vector.

On Intel i7-1260P and AMD Zen4 platforms, this implementation increases the throughput of parallel calls of a correctly rounded exponential compared to a state of the art scalar library (CORE-MATH). Our library is written in portable C, without intrinsics and hence also demonstrates how the auto-vectorization offered by modern compilers can be leveraged to produce fast functions that take advantage of the available SIMD instructions. This follows the approach of [28].

A. Assumptions and notation

We assume a Floating-Point (FP) arithmetic compliant with the IEEE 754-2019 Standard [23], with support for the FMA instruction. Binary64 is the target FP format. RN is the *round-to-nearest, ties-to-even,* rounding function, and RZ is the *round-towards-zero* function. $\Omega = 2^{1024} - 2^{1024-53}$ is the largest finite FP number, and $\eta = 2^{-1022}$ is the smallest normal number. The *normal range* is constituted by the FP numbers whose absolute value is between η and Ω. The *unit round-off* is $u = 2^{-53}$. It bounds the relative error committed when performing an arithmetic operation whose result is in

[1] https://github.com/taschini/crlibm
[2] https://core-math.gitlabpages.inria.fr/

the normal range. Following the definition given in [26], a *double-word*[3] (DW) number x is the unevaluated sum $x_h + x_\ell$ of two FP numbers x_h and x_ℓ such that $x_h = \mathrm{RN}(x)$. We often note $x = (x_h, x_\ell)$. We will at times use the TwoSum and FastTwoSum algorithms [2]: from an input pair (a, b) of FP numbers, they return a double-word (s, r) such that $s = \mathrm{RN}(a+b)$ and $s+r = a+b$. Accurately representing real numbers by such unevaluated sums of two (or three) floating-point numbers is often useful, and various authors have introduced and/or analyzed algorithms that perform the arithmetic operations in *double-word* [26], [27], [31], *triple-word* [16] (TW) or even *quad-word* [21] arithmetics. These somehow "generic" algorithms can be used for implementing elementary functions. However, we will also need *specific* algorithms that manipulate pairs or triplets of FP numbers that are not necessarily double- or triple-word numbers : the most useful of them are presented in the appendix (Section A). When we manipulate unevaluated pairs (x_h, x_ℓ) of FP numbers such that $|x_h|$ is significantly larger than $|x_\ell|$ but without necessarily satisfying the condition $x_h = \mathrm{RN}(x_h + x_\ell)$ we will follow Lange and Rump's notation [27], and call that *pair arithmetic*.

B. Organization of this article

Section II deals with the vector implementation of the exponential function. Much care is taken in the design of the fast path (Section II-A), because overall performance will mainly depend on that path. To make reading easier, most of the technical details are included in appendix: algorithms for DW and TW arithmetic are in Section A, and scripts for the FP proof generator Gappa can be downloaded.[4]

II. Vector exponential function

To allow for efficient vectorization, the fast path is free of elementwise tests, which would lead to divergence, and does not use large tables, which would require the use of *gather* instructions. Then a special "vector" rounding test, working horizontally on the whole vector, derived from Ziv's rounding test [14], [39] is used. Finally, as it will be seldom called, the accurate path remains scalar.

A. Fast path

In the fast path, performance is paramount. All complicated cases that require tests are left to the accurate path. Hence here we only consider input arguments whose exponential lies in the normal domain: x is assumed between $\mathrm{RZ}(\log(\eta)) = -6231120794008786 \times 2^{43} \approx -708.3964$ and $\mathrm{RZ}(\log(\Omega)) = 6243314768165359 \times 2^{43} \approx 709.7827$. We target an initial range reduction to the domain $[-\log(2)/2, \log(2)/2]$. Due to the inexactness of the calculations, the actual domain of the reduced argument will be slightly larger. This needs to be taken into account when building the polynomial approximation.

[3]The term *double-double* is often used. We tend to avoid it, first because the underlying arithmetic is not necessarily always double precision, and because what authors call *double-double* may differ from one article to another.

[4]At https://github.com/Vectorized-CR/vectorized-exp.

1) Range reduction: Define the following:

$$\begin{aligned}
&\mathrm{InvL2} = \mathrm{RN}\left(1/\log(2)\right) = 3248660424278399 \times 2^{-51},\\
&L_h = \mathrm{RN}(\log(2)) = 6243314768165359 \times 2^{-53}, \text{ and}\\
&L_\ell = \mathrm{RN}(\log(2) - L_h)) = 7525737178955839 \times 2^{-108}.
\end{aligned}$$

We have $|\mathrm{InvL2} - 1/\log(2)| \le 2.03553 \times 10^{-17}$, and $\Delta_L = |(L_h + L_\ell) - \log(2)| \le 5.70771 \times 10^{-34}$. If $t \in \mathbb{R}$, we note $\lceil t \rfloor$ the integer nearest to t (for instance with ties broken to even; this has little importance here). It is easily computed using the "1.5 trick" [20], [25]. We first compute $k = \lceil \mathrm{RN}(\mathrm{InvL2} \times x) \rfloor$. The bounds on x imply $|k| \le k_{\max} = 1024$. We have $|\mathrm{InvL2} \cdot x - k| \le \frac{1}{2} + u \cdot \mathrm{InvL2} \cdot x$, and therefore $\left|\frac{x}{\log(2)} - k\right| \le \frac{1}{2} + u \cdot \mathrm{InvL2} \cdot x + x \cdot \left|\mathrm{InvL2} - \frac{1}{\log(2)}\right|$, which gives,

$$|x - k\log(2)| \le \frac{\log(2)}{2} + 1.25132 \times 10^{-16} x, \qquad (1)$$

hence $|x - kL_h| \le \frac{\log(2)}{2} + 1.25132 \times 10^{-16} x + k \cdot |L_h - \log(2)|$. The first step of the range reduction consists in computing $r_1 = x - kL_h$. Note that r_1 is a FP number:

- this is obvious if $k = 0$, as in that case $r_1 = x$;
- if $|k| \ge 1$, then $|x \cdot \mathrm{InvL2}| \ge \frac{1}{2} - \frac{u}{4}$, hence $|x| \ge 0.346$. Hence x is an integer multiple of $\mathrm{ulp}(1/4)$. We deduce (as $L_h \ge 1/4$) that $x - kL_h$ is a multiple of $\mathrm{ulp}(1/4)$. As its absolute value is less than $1/2$ it is a FP number.

We therefore obtain $r_1 = \mathrm{RN}(x - kL_h)$. Now, define $r_2 = \mathrm{RN}(\hat{r}_2)$, with $\hat{r}_2 = -kL_\ell$. As $|\hat{r}_2| \le 1024 \times L_\ell < 2^{-45}$, $|\hat{r}_2 - r_2| \le \frac{1}{2}\mathrm{ulp}(2^{-46}) = 2^{-99}$. Therefore the pair (r_1, r_2) of FP numbers approximates $|x - k\log(2)|$ with error less than

$$\epsilon_y := k \cdot |L_h + L_\ell - \log(2)| + 2^{-99} \le 2.1622 \times 10^{-30}.$$

Using (1) we find that $|r_1 + r_2| \le \log(2)/2 + 1.25132 \times 10^{-16} x + \epsilon_y \le 0.3465736$. The reduced argument is obtained from (r_1, r_2) by a FastTwoSum operation. This yields to

Theorem II.1. *The double word (y_h, y_ℓ) obtained from x by the following operations*

$$\begin{array}{lcl}
t & = & \mathrm{RN}(\mathrm{InvL2} \times x), \quad s = \mathrm{RN}(t + \gamma),\\
k & = & \mathrm{RN}(s - \gamma), \text{\# 1.5 trick}\\
r_1 & = & \mathrm{RN}(x - kL_h), \quad r_2 = -\mathrm{RN}(kL_\ell),\\
(y_h, y_\ell) & = & \mathrm{FastTwoSum}(r_1, r_2),
\end{array}$$

(where $\gamma = 3 \times 2^{51}$ and InvL2, L_h and L_ℓ are defined above) satisfies $|y_h + y_\ell| \le 0.3465736$, and $|(y_h + y_\ell) - (x - k\log(2))| \le \epsilon_{\text{red}} = 2.1622 \times 10^{-30}$.

2) Polynomial approximation: For building polynomial approximations to functions in some interval, the traditional method would consist in first generating an "infinite-precision" polynomial, obtained either using the Remes algorithm or a projection on a basis of orthogonal polynomials. After this the coefficients of that polynomial would be rounded to the target FP format, to obtain a "floating-point polynomial". However, there is no reason for that final polynomial to be the best approximation to the function among the FP polynomials. More recent methods [3], [4] allow one to directly compute a good FP polynomial (or a polynomial with different constraints such

as a coefficient being exactly 1, or being a DW number). The algorithm in [3] is implemented in Sollya[5] [8], which also provides a safe bound on the approximation errors [7].

The reduced argument $y = y_h + y_\ell$ is a DW number that belongs to $I = [-0.3465736, +0.3465736]$. Further reduction can be done: as I is rather large, approximating e^y in I with the required accuracy needs a polynomial of large degree. One may, however, compute $z_1 = e^{y/2}$ (or $z_2 = e^{y/8}$) and retrieve e^y by squaring z_1 (or by squaring z_2 thrice) in pair arithmetic. The squarings involve additional operations and rounding errors but the evaluation of z_1 or z_2 uses smaller polynomials, and as the input argument ($y/2$ or $y/8$ instead of y) is smaller, the evaluation of the polynomial is more accurate. We compared these solutions, implemented the computation of z_2 followed by three squarings. The chosen polynomial has degree 10 and is of the form $Q(t) = 1 + t + \frac{1}{2}t^2 + q_3t^3 + \cdots + q_{10}t^{10}$, where $t = (t_h, t_\ell)$ is a pair, with $t_h = y_h/8$ and $t_\ell = y_\ell/8$, where q_3 is a DW number, and coefficients q_4 to q_{10} are FP numbers. Coefficients q_0, q_1 and q_2 are requested to be 1, 1, and 1/2 respectively. Q approximates e^t in $[-0.043335, 0.043335]$. The coefficients of Q, computed by Sollya, are:

$$\begin{aligned}
&q_3 = 6004799503160661 \times 2^{-55} + 46723697929451 \times 2^{-102},\\
&q_4 = 6004799503160661 \times 2^{-57},\ q_5 = 2401919801264309 \times 2^{-58},\\
&q_6 = 6405119470062833 \times 2^{-62},\ q_7 = 7320136529247375 \times 2^{-65},\\
&q_8 = 7320135365339369 \times 2^{-68},\ q_9 = 3253539763564979 \times 2^{-70},\\
&q_{10} = 5223286544655071 \times 2^{-74}.
\end{aligned}$$

The relative approximation error is bounded by $\epsilon_{\text{approx}} = 1.37025 \times 10^{-25}$. As $|t|$ is small and the terms q_i decrease quickly, for large i the influence of q_it^i on the final result is small. Hence, when evaluating Q using a Horner scheme, only the very last operations require the use of t_h and t_ℓ: in the first operations we can use t_h only. $Q(t)$ is computed as follows.

First, $q_5 + q_6t_h + q_7t_h^2 + \ldots + q_{10}t_h^5$ is computed using the Horner scheme with FMA instructions, in FP arithmetic. That is, we iteratively compute $\sigma_9 = \text{RN}(q_9 + q_{10}t_h)$, $\sigma_8 = \text{RN}(q_8 + \sigma_9t_h)$, …, $\sigma_5 = \text{RN}(q_5 + \sigma_6t_h)$.

After this, the product σ_5t_h is expressed exactly as a DW number (μ_4^h, μ_4^ℓ) (this is done using $\mu_4^h = \text{RN}(\sigma_5t_h)$ and $\mu_4^\ell = \text{RN}(\sigma_5t_h - \mu_4^h)$), and we add (μ_4^h, μ_4^ℓ) and the FP number q_4, to obtain a pair $\sigma_4 = q_4 + (\mu_4^h, \mu_4^\ell)$. Finally, the computation of $1 + t(1 + t(q_2 + t(q_3 + \sigma_4t)))$ is done in pair arithmetic, with t being represented by (t_h, t_ℓ): now t_ℓ is also used.

We need to bound the error committed when evaluating Q. The bound must be certain (otherwise the project of guaranteeing correct rounding is ruined), and as tight as possible to avoid undue costly calls to the accurate path. The Gappa[6] software [12] can compute such a bound. One can even build a formal proof of the error bound with tools that are reasonably easy to use [19]. Gappa automatically takes into account the errors of the usual FP operations. To help it to handle DW or TW operations, we feed it with the error bounds we have obtained for these special operations, described in the Appendix. Gappa returns the relative error bound $\epsilon_{\text{evalpol}} = 5.4363 \times 10^{-25}$.

We deduce that the pair $(\sigma_0^h, \sigma_0^\ell)$ obtained from the polynomial evaluation satisfies $\sigma_0^h + \sigma_0^\ell = e^t(1 + \epsilon_a)(1 + \epsilon_e)$, with $|\epsilon_a| \le \epsilon_{\text{approx}}$ and $|\epsilon_e| \le \epsilon_{\text{evalpol}}$. This gives

$$\sigma_0^h + \sigma_0^\ell = e^{y/8}(1 + \eta) \text{ with } |\eta| \le 6.807 \times 10^{-25}. \quad (2)$$

To obtain e^y from that approximation to $e^{y/8}$, we perform three consecutive squarings, using algorithm **d_square**, given in Section A1. Theorem A.1, tells us that the pair (z_h, z_ℓ) of FP numbers resulting from the three squarings approximates e^y with relative error better than $\epsilon_{\text{polynom}} = 5.446 \times 10^{-24}$.

3) Error bound for the fast path: Let $y^* = x - k\log(2)$. The analysis done in Sections II-A1 and II-A2 give

$$\begin{cases} |y - y^*| & \le \quad \epsilon_{\text{red}} = 2.1622 \times 10^{-30}, \\ |(z_h + z_\ell) - e^y| & \le \quad \epsilon_{\text{polynom}} \cdot e^y = 5.446 \times 10^{-24} \cdot e^y. \end{cases}$$

From this we deduce that $\left|e^y - e^{y^*}\right| \le \left|\frac{e^y}{e^{y^*}} - 1\right| \cdot e^{y^*} \le (e^{\epsilon_{\text{red}}} - 1) \cdot e^{y^*}$, and $|(z_h + z_\ell) - e^y| \le \epsilon_{\text{polynom}} \cdot e^{\epsilon_{\text{red}}} \cdot e^{y^*}$. Therefore $\left|(z_h + z_\ell) - e^{y^*}\right| \le (\epsilon_{\text{polynom}} \cdot e^{\epsilon_{\text{red}}} + e^{\epsilon_{\text{red}}} - 1) \cdot e^{y^*}$. The relative error of the fast path, say $\epsilon_{\text{fastpath}}$, thus satisfies

$$\epsilon_{\text{fastpath}} = \left|\frac{2^k(z_h + z_\ell) - e^x}{e^x}\right| \le \epsilon_{\text{polynom}} \cdot e^{\epsilon_{\text{red}}} + e^{\epsilon_{\text{red}}} - 1. \quad (3)$$

The error in ulps of the fast path is therefore bounded by $\epsilon_{\text{fastpath}} \cdot 2^{53} \le 4.907 \times 10^{-8}$ulp. This corresponds to $\ell \approx 23$.

4) Vector Rounding test: We start from a rounding test due to Ziv [39] and analyzed in [14]. Let (z_h, z_ℓ) be a DW (i.e., $z_h = \text{RN}(z_h + z_\ell)$), with z_h in the normal domain. Assume it represents $z \in \mathbb{R}$ with relative error bounded by ϵ, i.e., $|(z_h + z_\ell) - z| < \epsilon \cdot |z|$. Assume $\epsilon < 1/(2^{p+1} + 1)$, and define $e = \text{RU}\left(\frac{1}{1-\epsilon-2^{p+1}\epsilon}\right)$, where $\text{RU}(t)$ is the smallest FP number larger than or equal to t. Theorem 3.2 in [14] states that $z_h = \text{RN}(z_h + z_\ell e)$ implies $z_h = \text{RN}(z)$. With $\epsilon = \epsilon_{\text{fastpath}}$, this gives $e = 4503600069202551 \times 2^{-52} \approx 1.00000009811$.

In vector computing, checking whether for each component of a vector we have $z_h = \text{RN}(z_h + z_\ell e)$ is not an obvious task. We can however compute the sum of the elements of a vector of integers, and make a test on that sum. We therefore compute a boolean/integer $z_h == \text{RN}(z_h + z_\ell e)$ for each vector element, and compute the sum of all these terms. If the sum equals the vector length (which is the most likely), then z_h is the correctly rounded result for all terms of the vector. Otherwise, we call the accurate path, described in Section II-B. Our tests, presented in Section III-B, indicate that for a given input value, with the polynomial approximation and evaluation scheme given above, the probability of not having $z_h = \text{RN}(z_h + z_\ell e)$ is around 4×10^{-4}. For vectors of 8 elements, the probability of having to call the accurate path for a given vector is around 3×10^{-3}.

B. Accurate path

In case the fast path does not allow for correct rounding, we need to compute a more accurate approximation to e^x. As that case will be very unfrequent (as said above, the estimated probability is around 0.3%), it can be handled in a "scalar fashion". The accurate path also handles all special cases.

[5]https://www.sollya.org
[6]https://gappa.gitlabpages.inria.fr

1) Range reduction: To approximate e^x more accurately than in the fast path, one could use a polynomial approximation of higher degree and use DW (and even TW) arithmetic more often. However, this would significantly hinder performance. A more sensible choice is to reduce the input argument to a significantly smaller domain (which was hardly possible for the fast path without the use of tables). In the following, we try to reduce the argument to the interval $J = \left[-\frac{\log(2)}{512}, +\frac{\log(2)}{512}\right] \subset \left[-2^{-9.528}, +2^{-9.528}\right]$. Let x be the input number. Roughly speaking, the range reduction consists in first computing $k^* = \lceil 256 \cdot x/\log(2) \rfloor$, and obtaining the reduced argument $t^* = x - k^* \cdot \log(2)/256$. Let us define $E = \lfloor k^*/256 \rfloor$. We compute $e^x = e^{t^*} \cdot 2^E \cdot 2^{\frac{k^*}{256} - E}$, where e^{t^*} is obtained using a polynomial approximation described in Section II-B2, 2^E is exact and straightforwardly obtained, and $2^{\frac{k^*}{256} - E} = 2^{\frac{j}{256}}$, where $j = k^* - 256E \in \{0, 1, \ldots, 255\}$.

In practice, things are slightly more complex: first, k^* may not be exact if $\frac{256 \cdot x}{\log(2)}$ is extremely close to half an odd integer, and $\log(2)$ can only be approximated. Let $\mu = \mathrm{RN}(256/\log(2)) = 3248660424278399 \times 2^{-43}$. We first compute $k = \lceil \mathrm{RN}(\mu \cdot x) \rfloor$. Note that the bound on $|x|$ implies $|k| \le k_{\text{max}} = 262144$. We easily obtain

$$\begin{aligned} \left| x - k\frac{\log(2)}{256} \right| &\le \frac{\log(2)}{512} + x \left(u\mu \frac{\log(2)}{256} + \left| 1 - \mu \frac{\log(2)}{256} \right| \right) \\ &\le \frac{\log(2)}{512} + 8.88162 \times 10^{-14}. \end{aligned} \quad (4)$$

Then $\log(2)/256$ is approximated by the unevaluated sum of five FP numbers λ_0, λ_1, λ_2, λ_3 and λ_4 that fit in 33 bits (so that multiplying them by k is an exact operation). This gives

$$\begin{aligned} \lambda_0 &= 372130559 \times 2^{-37}, & \lambda_3 &= 265018045 \times 2^{-138}, \\ \lambda_1 &= 3099728209 \times 2^{-74}, & \lambda_4 &= 8162528485 \times 2^{-177}. \\ \lambda_2 &= 20739699 \times 2^{-101}, \end{aligned}$$

We have $\left| \frac{\log(2)}{256} - \sum_{i=0}^{4} \lambda_i \right| \le 1.00782 \times 10^{-54}$. Without error, we compute $\pi_i = k \cdot \lambda_i$, for $i = 0, \cdots, 4$. The number $x - \sum \pi_i$ approximates the exact reduced argument with error less than $\xi^0_{\text{red}} = k_{\text{max}} \times 1.00782 \times 10^{-54}$. As x and π_0 are very close, the calculation of $\tau_0 = x - \pi_0$ is exact thanks to Sterbenz Theorem [2]. We then compute:

$$\begin{aligned} (\tau^h_1, \tau^\ell_1) &= \mathrm{TwoSum}(\tau_0, -\pi_1), \\ (\tau^h_2, \tau^\ell_2) &= \mathrm{TwoSum}(\tau^\ell_1, -\pi_2), \\ (\tau^h_3, \tau^\ell_3) &= \mathrm{TwoSum}(\tau^\ell_2, -\pi_3). \end{aligned}$$

This gives

$$\tau^h_1 + \tau^h_2 + \tau^h_3 + \tau^\ell_3 = x - \pi_0 - \pi_1 - \pi_2 - \pi_3. \quad (5)$$

The number $|\tau_0 - \pi_1| = |x - k(\lambda_0 + \lambda_1)|$ is less than or equal to $\left| x - k\frac{\log(2)}{256} \right| + k_{\text{max}} \left| \frac{\log(2)}{256} - \lambda_0 - \lambda_1 \right|$, from which we deduce

$$\begin{aligned} |\tau^h_1| &\le \mathrm{RN}\left(\left| x - k\frac{\log(2)}{256} \right| + k_{\text{max}} \cdot \left| \frac{\log(2)}{256} - \lambda_0 - \lambda_1 \right| \right) \\ &\le 6243314768574961 \times 2^{-62} \end{aligned}$$

and $|\tau^\ell_1| \le \frac{1}{2}\mathrm{ulp}(\tau^h_1) = 2^{-63}$. From $|\tau^\ell_1 - \pi_2| \le 2^{-63} + k_{\text{max}} \cdot |\lambda_2|$, we obtain $|\tau^h_2| \le 21788275 \times 2^{-83}$, and $|\tau^\ell_2| \le \frac{1}{2}\mathrm{ulp}(\tau^h_2) = 2^{-112}$. Very similarly, we find: $|\tau^h_3| \le 265018301 \times 2^{-120}$, $|\tau^\ell_3| \le 2^{-146}$.

The following calculation is $\tau_4 = \mathrm{RN}(\tau^\ell_3 - \pi_4)$.

As $|\tau^\ell_3 - \pi_4| \le 2^{-146} + k_{\text{max}} \cdot |\lambda_4|$, we find $|\tau_4| \le 8162536677 \times 2^{-159}$, and $\xi^1_{\text{red}} = \tau_4 - (\tau^\ell_3 - \pi_4)$ satisfies $|\xi^1_{\text{red}}| \le 2^{-180}$. We obtain

$$\tau^h_1 + \tau^h_2 + \tau^h_3 + \tau_4 = x - \pi_0 - \pi_1 - \pi_2 - \pi_3 - \pi_4 + \xi^1_{\text{red}}. \quad (6)$$

The following operations are errorless:

$$(r^h_{\text{temp}}, v_1) = \mathrm{TwoSum}(\tau^h_1, \tau^h_2), \quad (r^m_{\text{temp}}, v_2) = \mathrm{TwoSum}(v_1, \tau^h_3).$$

After these operations, we have $r^h_{\text{temp}} + r^m_{\text{temp}} + v_2 = \tau^h_1 + \tau^h_2 + \tau^h_3$. We then perform $r^\ell_{\text{temp}} = \mathrm{RN}(v_2 + \tau_4)$.

As $|v_1| \le \frac{1}{2}\mathrm{ulp}(\tau^h_1 + \tau^h_2) \le 2^{-63}$ and therefore $|v_2| \le \frac{1}{2}\mathrm{ulp}(v_1 + \tau^h_3) \le 2^{-116}$, we deduce $|r^\ell_{\text{temp}}| \le 8804255558885 \times 2^{-159}$, and $\xi^2_{\text{red}} := r^\ell_{\text{temp}} - (v_2 + \tau_4)$ satisfies $\left|\xi^2_{\text{red}}\right| \le 2^{-169}$.

Finally, we *renormalize* the reduced argument:

$$(r_h, r_m, r_\ell) \leftarrow \text{scalar_renormalize_3}(r^h_{\text{temp}}, r^m_{\text{temp}}, r^\ell_{\text{temp}}),$$

where scalar_renormalize_3 is an errorless transformation. The total absolute error of the range reduction is bounded by

$$\xi_{\text{red-acc}} = \left|\xi^0_{\text{red}}\right| + \left|\xi^1_{\text{red}}\right| + \left|\xi^2_{\text{red}}\right| \le 2.369043 \times 10^{-51}. \quad (7)$$

2) Polynomial approximation: The polynomial Q is generated with the Sollya command line

```
Q = fpminimax(exp(x), 12, [|1, 1, D, TD, DD, DD, DD, DD,
D...|],  [-2^(-9.528); 2^(-9.528)]);
```

Q is of the form $Q(t) = 1 + t + q_2 t^2 + q_3 t^3 + \cdots + q_{12} t^{12}$, where $t = t_h + t_\ell$ is a DW number, produced by the range reduction step, q_2 is a FP number, q_3 is a TW number, q_4, q_5, q_6 and q_7 are DW numbers, and q_8, q_9, q_{10}, q_{11} and q_{12} are FP numbers. The coefficients provided by Sollya are:

$$\begin{aligned} q_2 &= 1/2, \\ q_3 &= 6004799503160661 \times 2^{-55} + 6004799503160661 \times 2^{-109} \\ &\quad + 6004801702402763 \times 2^{-163}, \\ q_4 &= 6004799503160661 \times 2^{-57} + 6004799503160661 \times 2^{-111}, \\ q_5 &= 4803839602528529 \times 2^{-59} + 1200959900632117 \times 2^{-113}, \\ q_6 &= 6405119470038039 \times 2^{-62} - 4403519618402317 \times 2^{-116}, \\ q_7 &= 3660068268593165 \times 2^{-64} + 114379210652599 \times 2^{-119} \\ q_8 &= 3660068268593165 \times 2^{-67}, \ q_9 = 6506788033054509 \times 2^{-71}, \\ q_{10} &= 2602715212381443 \times 2^{-73}, \ q_{11} = 7571535498912681 \times 2^{-78}, \\ q_{12} &= 5126783099304275 \times 2^{-81}. \end{aligned}$$

Using the `supnorm` function of Sollya [7], we obtain:

$$\epsilon_{\text{approx-acc}} = \sup_{t \in J} \left| \frac{Q(t) - e^t}{e^t} \right| = 9.869434 \times 10^{-48}. \quad (8)$$

To evaluate Q, we first compute $\sigma_{11} = \mathrm{RN}(q_{11} + q_{12} t_h)$, $\sigma_{10} = \mathrm{RN}(q_{10} + \sigma_{11} t_h)$, and $\sigma_9 = \mathrm{RN}(q_9 + \sigma_{10} t_h)$. After this the intermediate results are stored using pairs for increased precision. The number $\sigma_8 := (\mu^h_8, \mu^\ell_8)$ is the exact sum $\mathrm{RN}(\sigma_9 t_h) + q_8$, obtained with a TwoSum. The following operations are done using pair arithmetic: $\sigma_7 \approx q_7 + \sigma_8(t_h + t_\ell)$, $\sigma_6 \approx q_6 + \sigma_7(t_h + t_\ell)$, $\sigma_5 \approx q_5 + \sigma_6(t_h + t_\ell)$.

The following product is computed in pair arithmetic and the addition gives a TW: $\sigma_4 := (\mu^h_4, \mu^m_4, \mu^\ell_4) = DW((t_h + t_\ell)\sigma_5) + q_4$. Finally, the rest of the operations are done using

TW arithmetic: $q_0+t(q_1+t(q_2+t(q_3+t\sigma_4)))$. Using Gappa and error bounds on individual arithmetic operations[7], we find that the relative error of the polynomial evaluation is bounded by $\epsilon_{\text{evalpol-acc}} = 2^{-154.18}$. Combining this with the approximation error given in Eq. (8), we deduce that the final TW result approximates e^t with relative error

$$\epsilon_{\text{pol-acc}} = (1+\epsilon_{\text{approx-acc}})(1+\epsilon_{\text{evalpol-acc}})-1 \leq 4.852351\times 10^{-47}. \quad (9)$$

In Section II-B3, we take into account the error of the range reduction, combined with Eq. (9), to explain with which accuracy the obtained TW number approximates the exponential of x. We now need to round these three FP numbers to one FP number. This is done as follows. From $\sigma_0 = (\mu_0^h, \mu_0^m, \mu_0^\ell)$, we first perform some kind of "normalization", by computing

$$(t_1^h, t_1^\ell) = \text{FastTwoSum}(\mu_0^h, \mu_0^m),$$
$$(t_2^h, t_2^\ell) = \text{FastTwoSum}(t_1^\ell, \mu_0^\ell),$$
$$(t_3^h, t_3^m) = \text{FastTwoSum}(t_1^h, t_2^h).$$

Define $t_3^\ell := t_2^\ell$. The triplet (t_3^h, t_3^m, t_3^ℓ) satisfies $t_3^h+t_3^m+t_3^\ell = \mu_0^h+\mu_0^m+\mu_0^\ell$, $|t_3^m| \leq \frac{1}{2}\text{ulp}(t_3^h)$, and t_3^m is a multiple of $\text{ulp}(t_2^h)$ while $|t_3^\ell| \leq \frac{1}{2}\text{ulp}(t_2 h)$. All this implies that

- if $|t_3^m| < \frac{1}{2}\text{ulp}(t_3^h)$ then $|t_3^m| \leq \frac{1}{2}\text{ulp}(t_3^h) - \text{ulp}(t_2^h)$ so that $\text{RN}(t_3^h+t_3^m+t_3^\ell) = t_3^h$ whatever the value of t_3^ℓ;
- if $|t_3^m| = \frac{1}{2}\text{ulp}(t_3^h)$ then $\text{RN}(t_3^h+t_3^m+t_3^\ell)$ is determined uniquely by the sign of t_3^ℓ.

In all cases, we can replace the computation of $\text{RN}(t_3^h+t_3^m+t_3^\ell)$ by the computation of $\text{RN}(t_3^h+q)$, where

$$q = t_3^m + s\cdot \text{ulp}(t_3^m),$$
$$s = \begin{cases} 0 & \text{if } t_3^\ell = 0, \\ \text{sign}(t_3^\ell) & \text{if } t_3^m \text{ and } t_3^\ell \text{ have the same sign,} \\ -\text{sign}(t_3^\ell) & \text{otherwise.} \end{cases}$$

3) Error bound for the accurate path: As we did for the fast path (Eq. (3)), we obtain a relative error bound for the accurate path by combining the bounds of Eq. (7) (range reduction) and Eq. (9) (polynomial approximation and evaluation):

$$\epsilon_{\text{acc-path}} \leq \epsilon_{\text{pol-acc}} \cdot e^{\xi_{\text{red-acc}}} + e^{\xi_{\text{red-acc}}} - 1 \leq 4.8526\times 10^{-47}, \quad (10)$$

i.e., the approximation is within $2^{53}\times 4.8526\times 10^{-47} \leq 2^{-100.851}$ulp from the exact result. The list of hardest-to-round cases for e^x,[8] gives all binary64 numbers whose exponential is within 2^{-40}ulp from the exact middle of two consecutive FP numbers. Eq. (10) implies that if the accurate path does not provide a correctly-rounded value of e^x then x is in the list of hardest-to-round cases. This allows us to easily check that our algorithm always returns a correctly rounded result, except for one value, given in Section III. That specific value will be handled with a test.

[7]Given at https://github.com/Vectorized-CR/vectorized-exp
[8]Available at https://core-math.gitlabpages.inria.fr/worst.html and https://www.vinc17.net/research/testlibm/

III. Evaluation and Results

To evaluate both the *fast* and *accurate* paths, we developed a wrapper function for large vectors of any size, which need to be broken into chunks of `VECTOR_LENGTH` that our core SIMD implementation can handle. The wrapper handles memory alignment, segmentation into chunks and padding.

A. Worst cases tests

Using the worst case database discussed above, we tested the implementation for correctly rounded results. As a reference, we used the values returned by GNU MPFR [18]. We took care of handling subnormal rounding with MPFR correctly. Of the list of worst cases, only one input failed the test, x = `0x1.aca7ae8da5a7bp+0`. Three approaches were considered to address this issue: using a higher-degree polynomial, making the evaluation more accurate with more DW or TW arithmetic, or checking for that specific case. Since the check is true in one case among $\approx 2^{64}$, a test does not significantly impact the other cases. Henceforth, the issue was addressed using the third approach.

B. Accurate Path probability

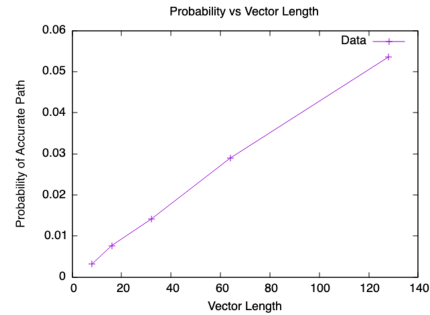

Fig. 1. Plot of probability vs. VECTOR_LENGTH

To estimate the *fast* vs. *accurate* path ratio, we tested for 10^9 inputs in the range `[-0x1.74910d52d3052p9, 0x1.62e42fefa39fp9]` and processed them in segments of size `VECTOR_LENGTH`. For size 8, this means $10^9/8$ vector evaluations. Out of those, 0.31629% went through the accurate path. Figure 1 shows that ratio for different vector lengths.

C. Vectorization Performance

To evaluate the vectorization, we ran measurements using the tester from [28]. The first two executions were treated as warm-up runs. Measurements were taken onwards. All code was compiled with Clang 19 using `-std=c11 -O3 -march=native -fno-math-errno`, enabling auto-vectorization. All tested implementations were compiled into static libraries and linked against the tester. Input vectors were aligned and had lengths that were powers of 2 in the range $[2^8, 2^{14}]$. Each length had 2^{17} evaluations with newly generated inputs. To measure the fast-path and average latency, each vector was filled with one random value, so that the code path taken over the whole vector would be the

same. The random numbers used as inputs were generated as follows: the exponent is drawn uniformly in the range $[-57, 10]$, and the significant and sign are drawn uniformly. If the FP number assembled from those values is outside the finite domain for the exponential ($[-708.3, 709.7]$ here), then it is resampled. This gives a better generalization on the performance of the implemented functions, as the values are closer to what is usually found in computations. If the exponent was drawn uniformly over all its possible values, then it would disproportionately generate values close to 0.

Performance testing was done on the following platforms:

- i7-1260P: 12th Gen Intel® Core™ i7-1260P CPU with support for AVX2 instructions (x86-64);
- Zen4: AMD EPYC 9124 CPU, with support for AVX512 instructions (x86-64).

For performance analisis, we compared with the state of the art correctly-rounded scalar exponential implementation from CORE-MATH [36] and the vectorized implementation by Lauter [28] which has a maximum measured error of 2.6695 ulps. The cycle count histograms are given in Section B (Fig. 2 for the i7-1260P, and Fig. 3 for the Zen4). In Table I, [Avg] means the average count, which include the fast path count and the accurate path count weighted by the probability of taking it.

TABLE I
PERFORMANCE SUMMARY

Processor	Core-Math [Avg]	Lauter libm [28] [Avg]	Fast Path [min]	Full Path [Avg]
i7-1260P	10.7	5.5	9	9.8
Zen4	14.7	8.6	9	9.3

The code source for the implementation, the Gappa scripts and the performance tests are available.[9]

Conclusion

We have shown that vector implementation of a correctly rounded elementary function is feasible in a short amount of time (this is a 4-month academic project) with reasonable performance (better than state of the art scalar). We just use portable C, without intrinsics. Specializing our code to a particular architecture could certainly help to gain further performance. This could pave the way to inclusion of correctly rounded elementary functions in IEEE-754.

Acknowledgements

This material is partly based upon work supported by the NSF under Grant No. 2311708. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

[1] C. S. Anderson, J. Zhang, and M. Cornea. Enhanced vector math support on the Intel AVX-512 architecture. In *25th IEEE Symposium on Computer Arithmetic*, pages 120–124, 2018.

[2] S. Boldo, C.-P. Jeannerod, G. Melquiond, and J.-M. Muller. Floating-point arithmetic. *Acta Numer.*, 32:203–290, 2023.

[3] N. Brisebarre and S. Chevillard. Efficient polynomial L^∞-approximations. In *18th IEEE Symposium on Computer Arithmetic*, pages 169–176, 2007.

[4] N. Brisebarre and G. Hanrot. Floating-point L^2-approximations. In *18th IEEE Symposium on Computer Arithmetic*, pages 177–184, 2007.

[5] N. Brisebarre, G. Hanrot, J. Muller, and P. Zimmermann. Correctly-rounded evaluation of a function: Why, how, and at what cost? *ACM Comput. Surv.*, 58(1):27:1–27:34, 2026.

[6] N. Brisebarre, G. Hanrot, and O. Robert. Exponential sums and correctly-rounded functions. *IEEE Trans. Comput.*, 66(12):2044–2057, 2017.

[7] S. Chevillard, J. Harrison, M. Joldes, and C. Lauter. Efficient and accurate computation of upper bounds of approximation errors. *Theoretical Computer Science*, 412:1523–1543, 2011.

[8] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In *International Conference on Mathematical Software*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, September 2010. Springer.

[9] C. Daramy, D. Defour, F. de Dinechin, and J.-M. Muller. CR-LIBM, a correctly rounded elementary function library. In *SPIE 48th Annual Meeting Int. Symposium on Optical Science and Technology*, 2003.

[10] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Lauter, and J.-M. Muller. CR-LIBM A library of correctly rounded elementary functions in double-precision. Research report, LIP, Dec. 2006. https://ens-lyon.hal.science/ensl-01529804.

[11] F. de Dinechin, A. V. Ershov, and N. Gast. Towards the post-ultimate libm. In *17th IEEE Symposium on Computer Arithmetic*, pages 288–295, 2005.

[12] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Trans. Comput.*, 60(2):242–253, 2011.

[13] F. de Dinechin, C. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *Theor. Inform. Appl.*, 41(1):85–102, 2007.

[14] F. de Dinechin, C. Lauter, J.-M. Muller, and S. Torres. On Ziv's Rounding Test. *ACM Trans. Math. Software*, 39(4), 2013.

[15] D. Defour and J. Muller. Correctly rounded exponential function in double precision arithmetic. In *SPIE, 46th Annual Meeting International Symposium on Optical Science and Technology*, 2001.

[16] N. Fabiano, J.-M. Muller, and J. Picot. Algorithms for triple-word arithmetic. *IEEE Trans. Comput.*, 68(11):1573–1583, 2019.

[17] 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. Software*, 33(2), 2007.

[18] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13–es, jun 2007.

[19] P. Geneau de Lamarlière, G. Melquiond, and F. Faissole. Slimmer formal proofs for mathematical libraries. In *30th IEEE Symposium on Computer Arithmetic*, pages 32–35, 2023.

[20] C. Hecker. Let's get to the (floating) point. *Game Developer Magazine*, 2:19–24, 1996.

[21] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In *15th IEEE Symposium on Computer Arithmetic (ARITH-15)*, pages 155–162, June 2001.

[22] T. Hubrecht, C.-P. Jeannerod, and P. Zimmermann. Towards a correctly-rounded and fast power function in binary64 arithmetic. In *30th IEEE Symposium on Computer Arithmetic*, Portland, Oregon (USA), 2023.

[23] IEEE. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019)*. July 2019. https://ieeexplore.ieee.org/servlet/opac?punumber=8766227.

[24] C.-P. Jeannerod, M. Joldes, N. Louvet, and J.-M. Muller. Extended-precision fma under parameterized double-word overlap: Tight error bounds and examples. In *33rd IEEE Symposium on Computer Arithmetic*, 2026.

[25] C.-P. Jeannerod, J.-M. Muller, and P. Zimmermann. On various ways to split a floating-point number. In *25th IEEE Symposium on Computer Arithmetic*, pages 53–60, Amherst (MA, USA), June 2018.

[26] M. Joldeş, J.-M. Muller, and V. Popescu. Tight and rigorous error bounds for basic building blocks of double-word arithmetic. *ACM Trans. Math. Software*, 44(2):1–27, 2017.

[27] M. Lange and S. M. Rump. Faithfully rounded floating-point computations. *ACM Trans. Math. Software*, 46(3):1–20, July 2020.

[9] See https://github.com/Vectorized-CR/vectorized-exp.

[28] C. Lauter. A new open-source SIMD vector libm fully implemented with high-level scalar C. In *50th Asilomar Conference on Signals, Systems and Computers*, pages 407–411, 2016.
[29] C. Q. Lauter. *Arrondi Correct de Fonctions Mathématiques*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2008. https://www.christoph-lauter.org/these.pdf.
[30] V. Lefèvre, J.-M. Muller, and A. Tisserand. Toward correctly rounded transcendentals. *IEEE Trans. Comput.*, 47(11):1235–1243, 1998.
[31] X. S. Li et al. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, June 2002.
[32] J. P. Lim and S. Nagarakatte. High performance correctly rounded math libraries for 32-bit floating point representations. In S. N. Freund and E. Yahav, editors, *PLDI'21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event*, pages 359–374. ACM, 2021.
[33] J. P. Lim and S. Nagarakatte. One polynomial approximation to produce correctly rounded results of an elementary function for multiple representations and rounding modes. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022.
[34] J. Shen, B. Long, and C. Huang. A Quantitative Evaluation of Vector Transcendental Functions on ARMv8-Based Processors. *J. Comput. Sci. Technol.*, 38:686–701, 2023.
[35] N. Shibata and F. Petrogalli. SLEEF: A portable vectorized library of C standard mathematical functions. *IEEE Trans. Parallel Distrib. Syst.*, 31(6):1316–1327, 2020.
[36] A. Sibidanov, P. Zimmermann, and S. Glondu. The CORE-MATH project. In *29th IEEE Symposium on Computer Arithmetic*, pages 26–34, 2022.
[37] P. T. P. Tang. An Open-Source RISC-V Vector Math Library. In *31st IEEE Symposium on Computer Arithmetic*, pages 60–67, 2024.
[38] D. K. Zhang and A. Aiken. High-performance branch-free algorithms for extended-precision floating-point arithmetic. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 695–710. Association for Computing Machinery, 2025.
[39] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Trans. Math. Software*, 17(3):410–423, 1991.

APPENDIX

A. Specific double-double and triple-double algorithms

We present a few ad-hoc double double or triple double algorithms used in our programs. One may also consider more "generic" algorithms such as those presented in [38] or [24].

1) Square of a pair: Consider the following algorithm.

Algorithm 1 d_square

Input: A pair (σ_h, σ_ℓ) that satisfies the input condition of Theorem A.1.
Output: Computes a pair of FP numbers (z_h, z_ℓ) whose sum approximates $(\sigma_h + \sigma_\ell)^2$.
$b \leftarrow \sigma_\ell + \sigma_\ell$ // exact operation
$z_h \leftarrow \mathrm{RN}(\sigma_h^2)$
$\pi_\ell \leftarrow \mathrm{RN}(\sigma_h^2 - z_h)$ // exact operation
$z_\ell \leftarrow \mathrm{RN}(\sigma_h b + \pi_\ell)$
return (z_h, z_ℓ)

Theorem A.1. *If a pair of FP numbers (σ_h, σ_ℓ) is such that $|\sigma_\ell| \le \alpha u \cdot |\sigma_h|$ and $\sigma_h + \sigma_\ell$ represents a real number Σ with relative error no larger than βu^2 (i.e., $|\sigma_h + \sigma_\ell - \Sigma| \le \beta u^2 \Sigma$), then the pair (z_h, z_ℓ) of FP numbers returned by Algorithm 1 satisfies $|z_\ell| \le \alpha' u \cdot |z_h|$ and $z_h + z_\ell$ represents Σ^2 with relative error no larger than $\beta' u^2$, where $\alpha' = 1 + u + 2\alpha(1+u)^2$, and $\beta' = \frac{(1+\beta u^2)^2}{(1-\alpha u)^2}(1+\alpha)^2 + 2\beta + \beta^2 u^2$.*

Proof. Let $\sigma = \sigma_h + \sigma_\ell$. We have $\sigma = \Sigma(1+\epsilon_1)$ with $|\epsilon_1| \le \beta u^2$. We also have $|(z_h + z_\ell) - \sigma^2| \le \sigma_\ell^2 + |\mathrm{RN}(\sigma_h b + \pi_\ell) - (\sigma_h b + \pi_\ell)| \le \alpha^2 u^2 \sigma_h^2 + u|\sigma_h b + \pi_\ell|$. As $|\pi_\ell| \le u\sigma_h^2$, we can finally bound $|(z_h + z_\ell) - \sigma^2|$ by $u^2\sigma_h^2(1 + 2\alpha + \alpha^2) = u^2\sigma_h^2(1+\alpha)^2$. The bound given by the theorem is finally obtained by remarking that $|\sigma_h| \le \frac{|\sigma|}{1-\alpha u} \le \frac{1+\beta u^2}{1-\alpha u}\Sigma$, and $\left|(z_h + z_\ell) - \Sigma^2\right| \le \left|(z_h + z_\ell) - \sigma^2\right| + \left|\Sigma^2 - \sigma^2\right|$. ☐

2) Addition of triplets: Consider the following algorithm:

Algorithm 2 triplet_add

Input: Two triplets $a = (a_h, a_m, a_\ell)$ and $b = (b_h, b_m, b_\ell)$ that satisfy the input condition of Theorem A.2.
Output: Computes a triplet $(r_h, r_m, r_\ell) \approx a + b$.
$(t_{1_h}, t_{1_\ell}) \leftarrow \mathrm{TwoSum}(a_h, b_h)$
$(t_{2_h}, t_{2_\ell}) \leftarrow \mathrm{TwoSum}(a_m, b_m)$
$t_3 \leftarrow \mathrm{RN}(a_\ell + b_\ell)$, $(t_{4_h}, t_{4_\ell}) \leftarrow \mathrm{TwoSum}(t_{1_\ell}, t_{2_h})$
$t_5 \leftarrow \mathrm{RN}(t_{2_\ell} + t_3)$, $t_6 \leftarrow \mathrm{RN}(t_{4_\ell} + t_5)$
return $(r_h = t_{1_h}, r_m = t_{4_h}, r_\ell = t_6)$

Theorem A.2. *If (a_h, a_m, a_ℓ) and (b_h, b_m, b_ℓ) satisfy:*

$$\begin{array}{lll} |a_m| \le \mathrm{ulp}(|a_h|), & |a_\ell| \le \mathrm{ulp}(|a_m|), & |b_m| \le \mathrm{ulp}(|b_h|), \\ |b_\ell| \le \mathrm{ulp}(|b_m|), & |a_h + b_h| \ge \frac{1}{2}|a_h|, & |a_h + b_h| \ge \frac{1}{2}|b_h|, \end{array}$$

then, noting $a = a_h + a_m + a_\ell$, $b = b_h + b_m + b_\ell$ and $r = r_h + r_\ell + r_m$, the triplet (r_h, r_ℓ, r_m) returned by Algorithm 2 satisfies $|r - (a+b)| \le \frac{48u^3}{(1-2u-2u^2)} \cdot |a+b|$.

Proof. Define $M = \max(a_h, b_h)$. We easily find that $|t_{1_\ell}| \le \frac{1}{2}\mathrm{ulp}(t_{1_h}) \le \mathrm{ulp}(M)$, $|t_{2_h}| \le 2\mathrm{ulp}(M)$ and $|t_{2_\ell}| \le 2u\mathrm{ulp}(M)$. We also have $|a_\ell|, |b_\ell| \le \mathrm{ulp}(\mathrm{ulp}(M)) \le 2u\mathrm{ulp}(M)$ and therefore $|t_3| \le 4u\mathrm{ulp}(M)$ and

$$|t_3 - (a_\ell + b_\ell)| \le 4u^2\mathrm{ulp}(M). \tag{11}$$

From $|t_{1_\ell} + t_{2_h}| \le 3\mathrm{ulp}(M)$ we deduce that $|t_{4_h}| \le 3\mathrm{ulp}(M)$ and $|t_{4_\ell}| \le 2u\mathrm{ulp}(M)$. From $|t_{2_\ell} + t_3| \le 6u\mathrm{ulp}(M)$, we deduce that $|t_5| \le 6u\mathrm{ulp}(M)$ and

$$|t_5 - (t_{2_\ell} + t_3)| \le 4u^2\mathrm{ulp}(M). \tag{12}$$

From $|t_{4_\ell} + t_5| \le 8u\mathrm{ulp}(M)$, we deduce that

$$|t_6 - (t_{4_\ell} + t_5)| \le 4u^2\mathrm{ulp}(M). \tag{13}$$

The final absolute error bound $12u^2\mathrm{ulp}(M)$ is obtained by adding the individual errors bounds of the nonexact operations, given by Eq. (11), Eq. (12), and Eq. (13). Hence:

$$\left|\frac{r - (a+b)}{a_h + b_h}\right| \le \frac{12u^2\mathrm{ulp}(M)}{\left|\frac{1}{2}M\right|} \le 48u^3.$$

The relative error bound given by the theorem derives from $|a+b| \ge |a_h + b_h|(1 - 2u - 2u^2)$. ☐

Algorithm 3 triplet_Plus_FP

Input: A FP number a, a triplet $b = (b_h, b_m, b_\ell)$ that satisfies the input condition of Theorem A.3.
Output: Computes a triplet $(t_h, t_m, t_\ell) \approx a + b$.
$(t_{1_h}, t_{1_\ell}) \leftarrow \text{TwoSum}(a, b_h)$
$(t_{2_h}, t_{2_\ell}) \leftarrow \text{TwoSum}(t_{1_\ell}, b_m)$, $t_3 \leftarrow \text{RN}(t_{2_\ell} + b_\ell)$
return $(t_h = t_{1_h}, t_m = t_{2_h}, t_\ell = t_3)$

3) Addition of a triplet and a FP number:

Theorem A.3. *Let a be a FP number. If the triplet (b_h, b_m, b_ℓ) of FP numbers satisfies $|b_m| \leq \text{ulp}(|b_h|)$ and $|b_\ell| \leq \text{ulp}(|b_m|)$, then, noting $b = b_h + b_m + b_\ell$ and $t = t_h + t_m + t_\ell$, the triplet returned by Algorithm 3 satisfies: $\left|t - (a + b)\right| \leq 2u^2 \cdot \text{ulp}\left(\max\{|a|, |b_h|\}\right)$.*

The proof is similar to the proof of Theorem A.2.

Algorithm 4 triplet_mul

Input: Two triplets (a_h, a_m, a_ℓ) and (b_h, b_m, b_ℓ) that satisfy the input condition of Theorem A.4.
Output: Computes a triplet (π_h, π_m, π_ℓ) that satisfies the output condition of Theorem A.4.
$(r_h, r_\ell) \leftarrow \text{TwoMult}(a_h, b_h)$, $(s_h, s_\ell) \leftarrow \text{TwoMult}(a_h, b_m)$
$(t_h, t_\ell) \leftarrow \text{TwoMult}(a_m, b_h)$, $(x_h, x_\ell) \leftarrow \text{TwoSum}(s_h, t_h)$
$(y_h, y_\ell) \leftarrow \text{TwoSum}(r_\ell, x_h)$
$c \leftarrow \text{RN}(a_h \cdot b_\ell)$, $d \leftarrow \text{RN}(a_m \cdot b_m)$, $e \leftarrow \text{RN}(a_\ell \cdot b_h)$
$t_3 \leftarrow \text{RN}(e + d)$, $t_4 \leftarrow \text{RN}(c + t_\ell)$, $t_5 \leftarrow \text{RN}(r_\ell + x_\ell)$
$t_6 \leftarrow \text{RN}(t_3 + t_4)$, $t_7 \leftarrow \text{RN}(t_5 + y_\ell)$, $t_8 \leftarrow \text{RN}(t_6 + t_7)$
return $(\pi_h = r_h, \pi_m = y_h, \pi_\ell = t_8)$

4) Multiplication of triplets:

Theorem A.4. *If the triplets of FP numbers (a_h, a_m, a_ℓ) and (b_h, b_m, b_ℓ) satisfy the following conditions:*

$$|a_m| \leq \text{ulp}(|a_h|), \quad |a_\ell| \leq \text{ulp}(|a_m|),$$
$$|b_m| \leq \text{ulp}(|b_h|), \quad |b_\ell| \leq \text{ulp}(|b_m|),$$

then, noting $a = a_h + a_m + a_\ell$, $b = b_h + b_m + b_\ell$ and $\pi = \pi_h + \pi_\ell + \pi_m$, the triplet (π_h, π_m, π_ℓ) returned by Algorithm 4 satisfies $|\pi - a \cdot b| \leq \frac{104u^3 + 16u^4}{(1-2u-4u^2)^2} \cdot |a \cdot b|$.

Proof. Let $M = \text{ulp}(a_h b_h)$. By elementary manipulation, we find $|s_h|, |t_h| \leq 2M$, $|s_\ell|, |t_\ell| \leq uM$, $|y_h| \leq 9M/2$, $|y_\ell| \leq 4uM$. We now consider the inexact operations:

- $|c|$, $|d|$, $|e| \leq 4uM$. The errors when computing them are bounded by $\epsilon_c = \epsilon_d = \epsilon_e = 2u^2 M$;
- $|t_3| \leq 8uM$. The error when computing it is bounded by $\epsilon_{t_3} = 4u^2 M$;
- similarly, $|t_4| \leq 5uM$, $\epsilon_{t_4} = 4u^2 M$, $|t_5| \leq 3uM$, $\epsilon_{t_5} = 2u^2 M$, $|t_6| \leq 13uM$, $\epsilon_{t_6} = 8u^2 M$, $|t_7| \leq 7uM$, $\epsilon_{t_7} = 4u^2 M$, and $|t_8| \leq 20uM$, $\epsilon_{t_5} = 16u^2 M$.

We obtain the final absolute error bound $52u^2 M + 8u^3 M$ by adding all the ϵ_i's plus bounds on the sum of the neglected terms $|a_m b_\ell| \leq 4u^2 M$, $|a_\ell b_m| \leq 4u^2 M$, and $|a_\ell b_\ell| \leq 8u^3 M$. The relative error bound follows from $|a| \geq |a_h|(1 - 2u - 4u^2)$, $|b| \geq |b_h|(1 - 2u - 4u^2)$, $M \leq 2u|a_h b_h|$. ☐

B. Performance measurements per tested architecture

The following histograms show the number of tested vectors that take the corresponding cycle count. The y-axis is in $\log_{10}$ scale and the x-axis is limited to 100 cycles.

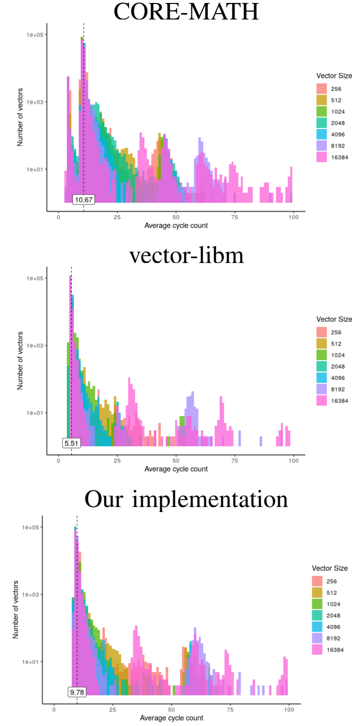

Fig. 2. Cycle count histograms on the i7-1260P

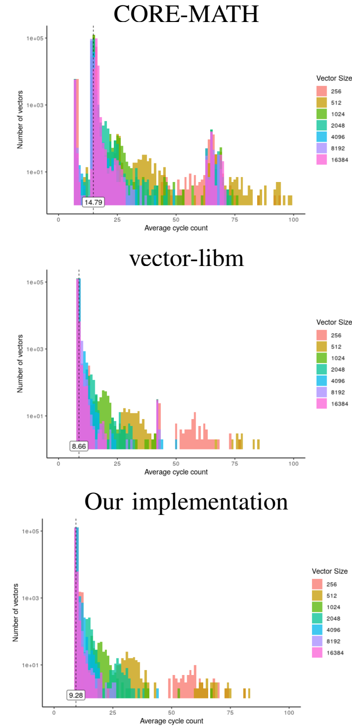

Fig. 3. Cycle count histograms on the Zen4