Approximating Floating-Point Addition Using The Geometric Mean

1st Theodor Lindberg
Dept. of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden
theodor.lindberg@liu.se

2nd Oscar Gustafsson
Dept. of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden
oscar.gustafsson@liu.se

***Abstract*—Previous work on approximate floating-point addition has focused on replacing standard components in exact designs with approximate counterparts. This paper presents a novel approach to approximating floating-point addition by using its geometric mean. The geometric mean can in turn be efficiently approximated using integer operations, making the approach practical for integer processors. Restricted to positive inputs, it can be suitable for applications involving sums of squares and exponentials. Maximum relative errors are derived. Synthesis results for standard-cell and FPGA technology demonstrate a favorable trade-off between resource usage, delay, and numerical performance compared to related work.**

***Index Terms*—Approximate arithmetic, floating-point, addition**

I. Introduction

A well-established approach to approximate floating-point operations is to compute in the logarithmic domain [1]–[3]. Using Mitchell's approximation [4], the base-two logarithm of floating-point numbers can be obtained as fixed-point numbers. Floating-point operations such as multiplication and division can, in that way, be reduced to integer addition and subtraction. This enables efficient hardware implementations and allows computations on integer processors without dedicated hardware. Platforms with floating-point units may also find it beneficial as idle integer resources can be utilized. However, as there is no similar logarithmic property for sums, addition has not been implemented using this approach.

Previous work on approximate floating-point addition has focused on replacing standard components in exact designs with approximate counterparts. Different variations are employed on three components: the adders used for subtracting exponents and adding mantissas, and the mantissa normalizer. In [5] approximate rounding was used, and [6], employed approximate addition of mantissas. An inexact mantissa adder and normalizer were used by [7]. Approximate addition of both the exponents and mantissas was presented in [8], where [9] also approximated the normalization. Runtime programmable precision tuning was proposed in [10] to reduce switching activity at the cost of increased area.

Moreover, extensive research exists on implementing exact floating-point arithmetic on integer processors [11]–[16], but, to the best of the authors' knowledge, there is no prior work on approximate soft floating-point addition. A soft implementation can be used alongside existing floating-point libraries such as FLIP [17], SEGGER's emFloat [18], and compilers' own libraries to accelerate sections of programs. Acceleration of software implementations has instead been limited to clever programming, excluding subnormal numbers and special values, and using alternative rounding modes.

In this paper, floating-point addition is approximated using its geometric mean. The geometric mean is, in turn, approximated using integer operations in the logarithmic domain, making the approach practical for integer machines. Restricted to positive inputs, it can be suitable when sums are known to be positive a priori, such as sums of squares and exponentials. Applications include vector norms and the softmax activation function. It is shown that the maximum absolute relative error is 0.2, regardless of format. It can, however, be improved by adjusting a constant. Synthesis results for standard-cell and FPGA technology show that the proposed design provides a favorable trade-off between resource usage and accuracy.

The rest of this paper is as follows: Section II introduces the approach and Section III presents the proposed approximation. Maximum relative errors are derived in both sections. In Section IV, numerical performance and hardware implementation are evaluated. The approach is discussed in Section V along with future work. Section VI concludes the paper.

II. Approximating Sums Using Geometric Means

The approximation

$$\log(x+y) \approx \log(2) + \log(\sqrt{xy}) \tag{1a}$$

$$= \log(2) + \frac{\log(x) + \log(y)}{2} \tag{1b}$$

for $x, y \geq 0$, was posted in a forum [19]. The intended use was for estimating the logarithm of larger numbers using smaller ones. For example, the logarithm of 11 can be estimated as

$$\log(11) = \log(5+6) \approx \log(2) + \frac{\log(5) + \log(6)}{2}. \tag{2}$$

It is the most accurate when $x \approx y$ as it suggests

$$x + y \approx 2\sqrt{xy}, \tag{3}$$

This work was supported by the ELLIIT strategic research environment through the project D3 ACRE—Approximate Computing Reducing Energy.

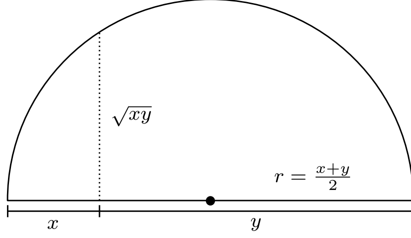

Fig. 1. The geometric mean visualized with a semicircle where the diameter is the sum of x and y, illustrating $x + y = 2r \approx 2\sqrt{xy}$. When x equals y, the geometric mean and the arithmetic mean coincide.

where the left-hand side and right-hand side are equal when $x = y$. One can interpret it as the sum being approximated using its geometric mean. A visualization of the geometric mean and how it approximates a sum of two variables is shown in Fig. 1. This idea is used as inspiration moving forward.

A. Proposed Approach

Unless the operands are of similar magnitude, the approximation will be poor. At some point, it is better to return the largest operand rather than the geometric mean:

$$x + y \approx \max(x,\, y,\, 2\sqrt{xy}). \tag{4}$$

The largest error will occur at the two tipping points since all terms under-approximate the exact result. To calculate the maximum error, assume, without loss for generality, that x is strictly greater than y, then the point of interest is

$$x = 2\sqrt{xy}. \tag{5}$$

A relationship between the variables is obtained as

$$x = 2\sqrt{xy} \Leftrightarrow \tag{6a}$$
$$x^2 = 4xy \Leftrightarrow \tag{6b}$$
$$x = 4y. \tag{6c}$$

Meaning that it is better to approximate the sum as x than $2\sqrt{xy}$ when x is four times greater than y, and vice versa. The maximum relative error will also occur at this point:

$$\frac{x}{x+y} = \frac{x}{x + x/4} = \frac{x}{5x/4} = \frac{4}{5} = 0.8, \tag{7}$$

giving a relative error of -0.2.

The approximation can be studied more easily by considering a fixed sum, as shown in Fig. 2, since the behavior will be the same for all sums.

Because (4) yields an under approximation, the maximum relative error can be improved by scaling with a constant:

$$c \times \max(x,\, y,\, 2\sqrt{xy}). \tag{8}$$

The optimal constant is found by solving the minimax problem

$$\min_c \max\left(|1 - 0.8c|,\, |1 - c|\right). \tag{9}$$

Since c will be greater than one, the optimum is

$$1 - 0.8c = -(1 - c) \Leftrightarrow c = \frac{10}{9}, \tag{10}$$

giving a maximum relative error of $\frac{1}{9}$.

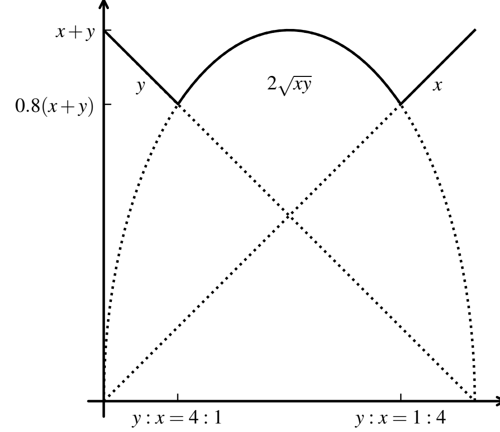

Fig. 2. Visualization of $\max(x,\, y,\, 2\sqrt{xy})$ when the sum of x and y is constant. The solid line shows the output, and the dotted lines indicate the inactive functions. As seen, the maximum error occurs when one operand is four times larger than the other.

III. Proposed Approximation

A positive normalized floating-point number x can be expressed as

$$x = 2^{e_x}(1 + m_x), \tag{11}$$

where $e_{\min} \leq e_x \leq e_{\max}$ and $0 \leq m_x < 1$, as defined by the IEEE 754 Standard [20]. Stored in hardware, the biased exponent $E_x = e_x + b$ is followed by the t-bit mantissa $m_x 2^t$.

Subnormal numbers are not considered since they are generally not used in approximate computing. Previous work [5]–[10] do not consider them either and some formats, such as bfloat16 [21] and DLFloat [22], exclude them.

A. Implementation in the Logarithmic Domain

An approximate base-two logarithm of a floating-point number can be obtained as a fixed-point number using Mitchell's approximation [4]. As explained in [2], [3], interpret the binary representation of a floating-point number as a fixed-point number with t fractional bits and subtract the bias:

$$\log_2(x) \approx X - B = \hat{X}, \tag{12}$$

where $B = b \ll t$. To convert back, interpret

$$\hat{X} + B \tag{13}$$

as a floating-point number.

Equation (3) can, based on this, be implemented using the approximate integer expressions presented in [2]. In particular, the following expression will be used for multiplication:

$$x \times y : X + Y - B, \tag{14}$$

and the expression

$$\sqrt{x} : (X + B) \gg 1, \tag{15}$$

for the square root. Multiplication by two can be done exactly by incrementing the exponent field by one:

$$2x : X + (1 \ll t). \tag{16}$$

For convenience, denote $D = 1 \ll t$. Combining all expressions, (3) can be implemented as

$$((X + Y - B) + B) \gg 1 + D = (X + Y + 2D) \gg 1. \quad (17)$$

The comparison needed for finding the maximum is cheap since floating-point numbers can be compared directly by their bit representations:

$$\max\left(X,\, Y,\, (X + Y + 2D) \gg 1\right). \quad (18)$$

Variable shifts are notably avoided.

B. Maximum Relative Error

Consider

$$(X + Y + 2D) \gg 1 = \left\lfloor \frac{X + Y + 2D}{2} \right\rfloor. \quad (19)$$

Similar to before, the tipping point

$$X = \left\lfloor \frac{X + Y + 2D}{2} \right\rfloor \quad (20)$$

is of interest since an under approximation is obtained. Flooring gives the inequality

$$X \leq \frac{X + Y + 2D}{2} < X + 1. \quad (21)$$

Concentrating at the upper bound, it follows that

$$\frac{X + Y + 2D}{2} < X + 1 \Leftrightarrow \quad (22a)$$
$$X + Y + 2D < 2X + 2 \Leftrightarrow \quad (22b)$$
$$Y < X - 2D + 2 \quad \Leftrightarrow \quad (22c)$$
$$Y \leq X - 2D + 1. \quad (22d)$$

Since the variables are integers, the largest error occurs at one of the two points:

$$X - 2D \leq Y \leq X - 2D + 1. \quad (23)$$

Two cases must be considered to evaluate the error since $X - 2D + 1$ may have a different exponent than $X - 2D$.

Before calculating the relative error, recall the exact addition of two positive floating-point numbers:

$$x + y = 2^{e_x}(1 + m_x) + 2^{e_y}(1 + m_y) \quad (24a)$$
$$= 2^{e_x}\left(1 + m_x + 2^{e_y - e_x}(1 + m_y)\right). \quad (24b)$$

Let $Y = X - 2D + 1$ and $u = 2^{-t}$, and consider when $m_x + u < 1$:

$$\frac{x}{x + y} = \frac{2^{e_x}(1 + m_x)}{2^{e_x}\left(1 + m_x + 2^{-2}(1 + m_x + u)\right)} \quad (25a)$$
$$= \frac{1 + m_x}{1 + m_x + 2^{-2}(1 + m_x + u)} \quad (25b)$$
$$= \frac{1 + m_x}{\frac{5}{4}(1 + m_x) + \frac{u}{4}}. \quad (25c)$$

The largest error occur when the quotient is at its smallest, which is $m_x = 0$:

$$\frac{1 + 0}{\frac{5}{4}(1 + 0) + \frac{u}{4}} = \frac{1}{\frac{5}{4} + \frac{u}{4}} = \frac{4}{5 + u}. \quad (26)$$

This gives the relative error

$$\frac{4}{5 + u} - 1 = \frac{4}{5 + u} - \frac{5 + u}{5 + u} = -\frac{1 + u}{5 + u}. \quad (27)$$

The lower bound yields the same calculation but without u, resulting in a slightly lower error of exactly -0.2.

When $m_x + u = 1$, the quotient is

$$\frac{x}{x + y} = \frac{2^{e_x}(1 + m_x)}{2^{e_x}\left(1 + m_x + 2^{-1}(1 + 0)\right)} \quad (28a)$$
$$= \frac{1 + m_x}{1 + m_x + \frac{1}{2}} \quad (28b)$$
$$= \frac{2 - u}{2 - u + \frac{1}{2}}, \quad (28c)$$

and the relative error is

$$\frac{2 - u}{2 - u + \frac{1}{2}} - 1 = \frac{-\frac{1}{2}}{2 - u + \frac{1}{2}} = -\frac{1}{5 - u}. \quad (29)$$

Compared to the previous case, this error is smaller since

$$\frac{1 + u}{5 + u} > \frac{1}{5 - u} \quad (30)$$

for all $0 < u < 1$.

In conclusion, the largest error occurs when x is an exact power of two and y is the floating-point number next up from $x/4$, or vice versa, giving the maximum relative error of

$$-\frac{1 + u}{5 + u} \quad (31)$$

as a closed-form expression. The error tends to -0.2 as the number of mantissa bits increase, i.e., the error obtained with the mathematical model.

C. Further Error Analysis

To gain a better understanding of the approximation, a general error analysis is given. Again consider

$$(X + Y + 2D) \gg 1 = \left\lfloor \frac{X + Y + 2D}{2} \right\rfloor. \quad (32)$$

Interpret the variables as fixed-point numbers with t fractional bits and let $\lfloor m \rfloor_t$ denote m truncated to t fractional bits:

$$\left\lfloor \frac{(E_x + m_x) + (E_y + m_y) + 2}{2} \right\rfloor_t, \quad (33a)$$
$$= \left\lfloor \frac{E_x + E_y}{2} + 1 + \frac{m_x + m_y}{2} \right\rfloor_t. \quad (33b)$$

The least significant bit of the exponent will be shifted into the mantissa field, adding $1/2$ to the mantissa when $E_x + E_y$ is odd. While the addition $(m_x + m_y)/2$ cannot spill over to the exponent, it may when an exponent bit is shifted down: $1/2 + (m_x + m_y)/2$. There are therefore three cases:

1) $E_x + E_y \equiv 0 \mod 2$,
2) $E_x + E_y \equiv 1 \mod 2$
   a) $1/2 + \frac{m_x + m_y}{2} \geq 1$,
   b) $1/2 + \frac{m_x + m_y}{2} < 1$,

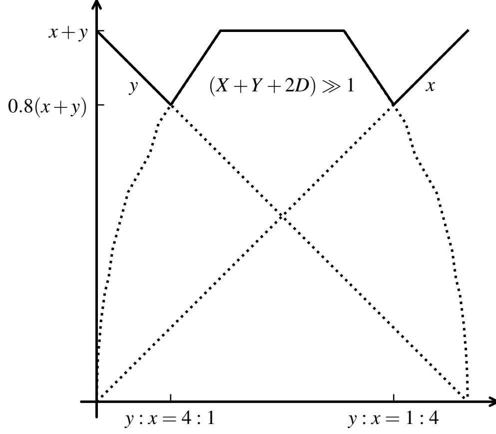

Fig. 3. The behavior when the sum of two variables is exactly between two binades. Several combinations of inputs will have the same exponent, and thus, a correctly rounded result is obtained more often.

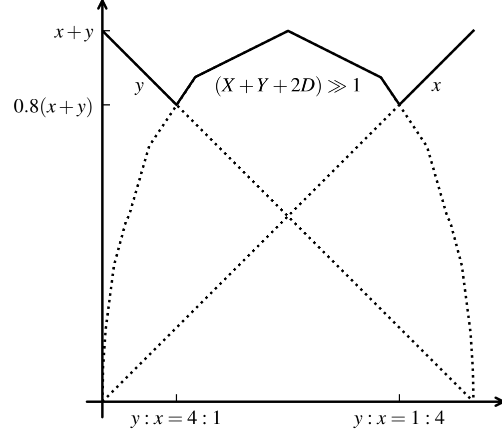

Fig. 4. The behavior when the sum of two variables is an exact power of two. The inputs share the same exponent in only one instance.

giving the integer patterns

$$\begin{cases} \dfrac{E_x+E_y}{2}+1.\left\lfloor \dfrac{m_x+m_y}{2} \right\rfloor_t, & \text{case 1} \\ \left\lfloor \dfrac{E_x+E_y}{2} \right\rfloor+2.\left\lfloor \dfrac{m_x+m_y-1}{2} \right\rfloor_t, & \text{case 2a} \\ \left\lfloor \dfrac{E_x+E_y}{2} \right\rfloor+1.\left\lfloor \dfrac{1+m_x+m_y}{2} \right\rfloor_t, & \text{case 2b,} \end{cases} \tag{34}$$

which in turn correspond to the floating-point values

$$\begin{cases} 2^{1+(e_x+e_y)/2}\left(1+\left\lfloor \dfrac{m_x+m_y}{2} \right\rfloor_t\right), & \text{case 1} \\ 2^{2+\lfloor (e_x+e_y)/2 \rfloor}\left(1+\left\lfloor \dfrac{m_x+m_y-1}{2} \right\rfloor_t\right), & \text{case 2a} \\ 2^{1+\lfloor (e_x+e_y)/2 \rfloor}\left(1+\left\lfloor \dfrac{1+m_x+m_y}{2} \right\rfloor_t\right), & \text{case 2b.} \end{cases} \tag{35}$$

Remarkably, correct rounding is obtained when x and y share the same exponent. By comparing case 1 with the exact addition of two floating-point numbers of the same exponent,

$$x+y = 2^{e_x}(1+m_x)+2^{e_y}(1+m_y) \tag{36a}$$
$$= 2^{e_x}\left(2+m_x+m_y\right)) \tag{36b}$$
$$= 2^{1+e_x}\left(1+\frac{m_x+m_y}{2}\right), \tag{36c}$$

one can see that that either the exact result is obtained or the result rounded is down from a tie break.

Consequently, the number of inputs that produce a correctly rounded result for a given sum varies. The most favorable case occurs when the sum is exactly between two binades, as illustrated in Fig. 3, whereas the least favorable case occurs when the sum is an exact power of two, shown in Fig. 4.

Figures 5 and 6 show the relative error when sweeping over six binades. A slight shade of gray is present in the diagonal squares due to rounding on tie breaks.

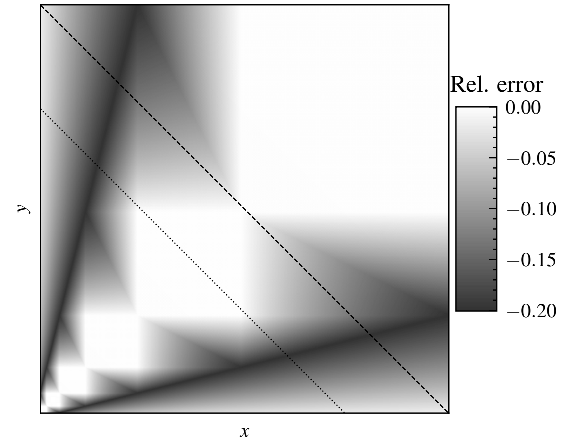

Fig. 5. Relative error when sweeping six binades. The white squares show that correctly rounded results are obtained when the inputs have the same exponent. The dark edges show the crossover of the max function. The dotted line and the dashed line indicate the fixed sums in Figs. 3 and 4, respectively.

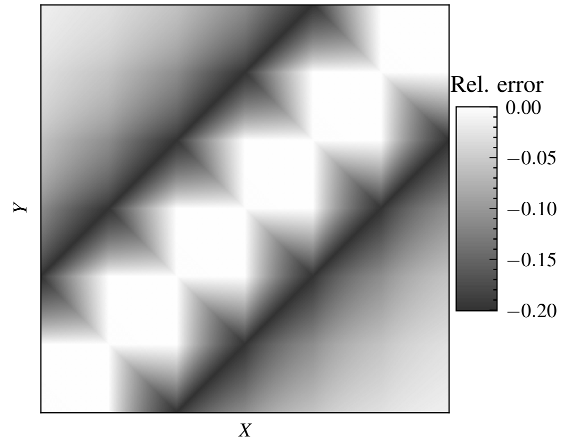

Fig. 6. Relative error when sweeping six binades. Like Fig. 5 but with bit patterns on the axes.

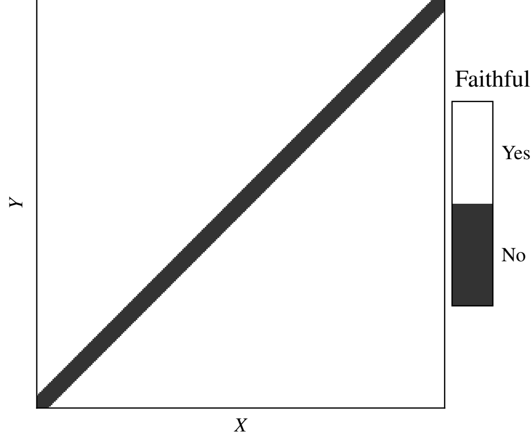

Fig. 7. Instances where $\max(X, Y)$ yields a faithfully rounded result for bfloat16, swept from zero to NaN.

When the difference in exponents is greater than the number of mantissa bits, returning the largest operand yields a correctly rounded result. Depending on the format, this can account for a significant portion of the function space. Figure 7 shows when faithful rounding is obtained from $\max(X, Y)$ for bfloat16, sweeping all bit patterns from zero to NaN. When simply returning the largest operand, the maximum relative error is -0.5. The proposed approximation improves this error and the dark diagonal band visible in Fig. 7.

D. Reducing the Error

While the error can be reduced by scaling the whole expression, as in (8), a more interesting case is to modify the constant $2D$, as the latter does not add overhead. Finding an optimal constant naturally depends on what error measure is considered. However, by using the same approach as in Section III-B, it can be proven that by adding $2D + 1$ instead of $2D$, the maximum relative error will be -0.2 when x is exactly four times greater than y, for all formats. Correct rounding is still obtained when the operands have the same exponent, but tie breaks are now rounded up. The proposed approximation moving forward is therefore

$$\max\left(X,\, Y,\, (X + Y + C) \gg 1\right), \tag{37}$$

where $C = 2D + 1$.

E. Handling Infinity and Not-a-Number

When infinity and Not-a-Number (NaN) are encoded as per the IEEE 754 Standard, both will be propagated correctly by the $\max$ function when $X + Y + C$ does not overflow. This is because the bit patterns are ordered.

Support for overflow handling can be added by comparing with the bit pattern for infinity:

$$\max\left(X,\, Y,\, \min\left((X + Y + C) \gg 1,\, \texttt{FP_INF}\right)\right). \tag{38}$$

Infinities and NaNs are still propagated correctly.

IV. Evaluation

The approach is evaluated in terms of numerical performance and resource usage for both ASIC and FPGA. Five floating-point formats are considered: E5M2 and E4M3 from the OFP8 standard [23], bfloat16 (BF16) [21], as well as binary16 (FP16) and binary32 (FP32) from the IEEE 754 standard. Results for DLFloat [22] and TensorFloat-32 (TF32) [24] are excluded for brevity. They are also not reported by related work. To put the results into perspective, comparisons are made against both approximate designs [9], [10] and against exact designs generated using FloPoCo [25].

The work of [9] proposed using OR gates for the lower parts of integer adders and an approximate leading-zero detector for mantissa normalization. Since only positive inputs are considered in this work, the approximate normalization step is disregarded. Two designs were presented in [9], although the method allows several variations. The first design, referred to as revised lower-OR adder (RLOA) I, replaces the entire mantissa adder with OR gates. The second design, RLOA II, uses an exact mantissa adder with an inexact exponent subtractor with an OR gate for the least significant bit.

In [10], two run-time configurable floating-point adders were proposed, one for FP16 and one for FP32. These designs will be referred to as approximate configurable floating-point adders (ACFPA). Using control signals, the precision is tuned in two ways to reduce switching activity. First, a threshold is set so that when the exponent difference is too large, the largest operand is returned, and the signals going to the mantissa adder are forced to zero. Second, the control signals specify the number of mantissa bits to add by zeroing the lower bits. An increase in area is expected due to the runtime configurability. The designs presented support eight configurations, of which the configuration with the best accuracy (Acc=7) and the worst accuracy (Acc=0) will be used for comparison.

A. Numerical Performance

The error metrics used by previous work will be adopted in this paper. The absolute relative error is calculated as

$$\left|\frac{\widetilde{x} - x}{x}\right|, \tag{39}$$

with double-precision floating-point used as the reference. The metrics that will be reported are the maximum absolute relative error distance (Max. RED), the mean absolute relative error (MRED), the normalized mean relative error (NMED), and the error rate (ER). NMED is calculated by dividing the error distance by the largest finite floating-point number. Error rate is calculated as the percentage of times a bit-exact result was not obtained. The last is a debatable metric since a specific rounding mode, often unspecified, is used as a reference. While IEEE 754's default mode can be assumed, one can argue that it is better to define error rate as the percentage of times a non-faithful result is obtained. However, to follow the conventions of previous work, the default mode (roundTiesToEven) will be used as a reference. Conversion

TABLE I
ERRORS OF (37) WHEN USING UNIFORM DISTRIBUTIONS, FP32

Variable	Max. RED	MRED	NMED	ER
Bit pattern	2.00×10^{-1}	3.63×10^{-3}	1.28×10^{-5}	18.1%
Float	2.00×10^{-1}	6.30×10^{-2}	2.68×10^{-2}	75.1%

TABLE II
ERRORS OF APPROXIMATE FLOATING-POINT ADDERS

Format	Design	Max. RED	MRED	NMED	ER
E5M2	Proposed (37)	2.00×10^{-1}	2.85×10^{-2}	1.05×10^{-3}	13.2%
	RLOA I [9]	5.00×10^{-1}	4.81×10^{-2}	2.08×10^{-3}	18.7%
	RLOA II [9]	3.85×10^{-1}	3.14×10^{-2}	1.33×10^{-3}	13.2%
E4M3	Proposed (37)	2.00×10^{-1}	5.42×10^{-2}	4.96×10^{-3}	43.9%
	RLOA I [9]	5.00×10^{-1}	8.98×10^{-2}	9.35×10^{-3}	48.3%
	RLOA II [9]	3.60×10^{-1}	4.39×10^{-2}	4.75×10^{-3}	30.6%
BF16	Proposed (37)	2.00×10^{-1}	3.62×10^{-3}	1.30×10^{-5}	6.3%
	RLOA I [9]	5.00×10^{-1}	4.60×10^{-3}	2.18×10^{-5}	5.9%
	RLOA II [9]	3.35×10^{-1}	1.49×10^{-3}	8.28×10^{-6}	3.9%
FP16	Proposed (37)	2.00×10^{-1}	2.88×10^{-2}	9.91×10^{-4}	61.0%
	RLOA I [9]	5.00×10^{-1}	3.78×10^{-2}	1.64×10^{-3}	58.7%
	RLOA II [9]	3.34×10^{-1}	1.16×10^{-2}	6.01×10^{-4}	39.6%
	ACFPA Acc=0 [10]	2.00×10^{-1}	1.49×10^{-2}	4.77×10^{-4}	62.5%
	ACFPA Acc=7 [10]	7.75×10^{-3}	5.19×10^{-4}	1.86×10^{-5}	37.8%
FP32	Proposed (37)	2.00×10^{-1}	3.63×10^{-3}	1.28×10^{-5}	18.1%
	RLOA I [9]	5.00×10^{-1}	4.51×10^{-3}	2.24×10^{-5}	17.7%
	RLOA II [9]	3.33×10^{-1}	1.35×10^{-3}	7.62×10^{-6}	11.3%
	ACFPA Acc=0 [10]	4.88×10^{-4}	4.95×10^{-6}	1.53×10^{-8}	18.4%
	ACFPA Acc=7 [10]	4.77×10^{-7}	1.33×10^{-8}	4.12×10^{-11}	11.8%

between relative error and error in units in last place (ulp) is possible, but information is lost in both directions [26].

Special care is required when calculating average errors in situations where exhaustive testing is not feasible, in this case for FP32. Monte Carlo simulations were used in [10] to obtain results within certain confidence intervals, but it was not specified how the random inputs were generated. The average error depends on whether the distribution is for floating-point values or bit patterns. Because sweeping over all possible inputs yields a uniform distribution of bit patterns, the latter is to be used. Table I shows the impact of the two. Clearly, there is a large difference. The data were generated as two vectors of 2^{15} random elements. All possible combinations of values were tested, yielding 2^{30} cases. Subnormal numbers were not generated, and the largest input value was set to half of the largest finite floating-point value to avoid overflow.

The numerical results for the designs are presented in Table II. All positive bit patterns were swept from the smallest normal floating-point number to the largest finite floating-point value divided by two, again avoiding subnormal numbers and the risk of overflow. For FP32, uniformly distributed bit patterns were tested using the method described above. Slight deviations are present for ACFPA when compared to the results reported in [10] since only positive inputs are evaluated. The proposed approximation yields better results than RLOA I but, in general, slightly worse results than RLOA II. The best results are obtained with ACFPA, but, as seen later, it requires the most hardware resources.

The average absolute relative error depends on the number of exponent bits and mantissa bits, as shown in Fig. 8. A non-monotonic pattern is observed when sweeping the number of mantissa bits, since the constant C affects the mantissa. When sweeping the number of exponent bits, it is not surprising that the average error benefits from a large dynamic range.

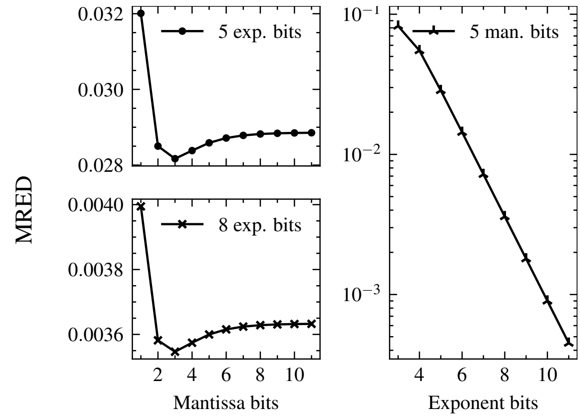

Fig. 8. Effect on the average absolute relative error when the number of exponent bits and mantissa bits change independently, plotted in logarithmic scale.

B. Synthesis Results

For the proposed method, (37), the maximum of X and Y is computed in parallel to $(X + Y + C) \gg 1$.

FloPoCo [25] was used to generate a correctly rounded floating-point adder for reference, optimized for positive-only inputs. It does not support subnormal numbers by default, but

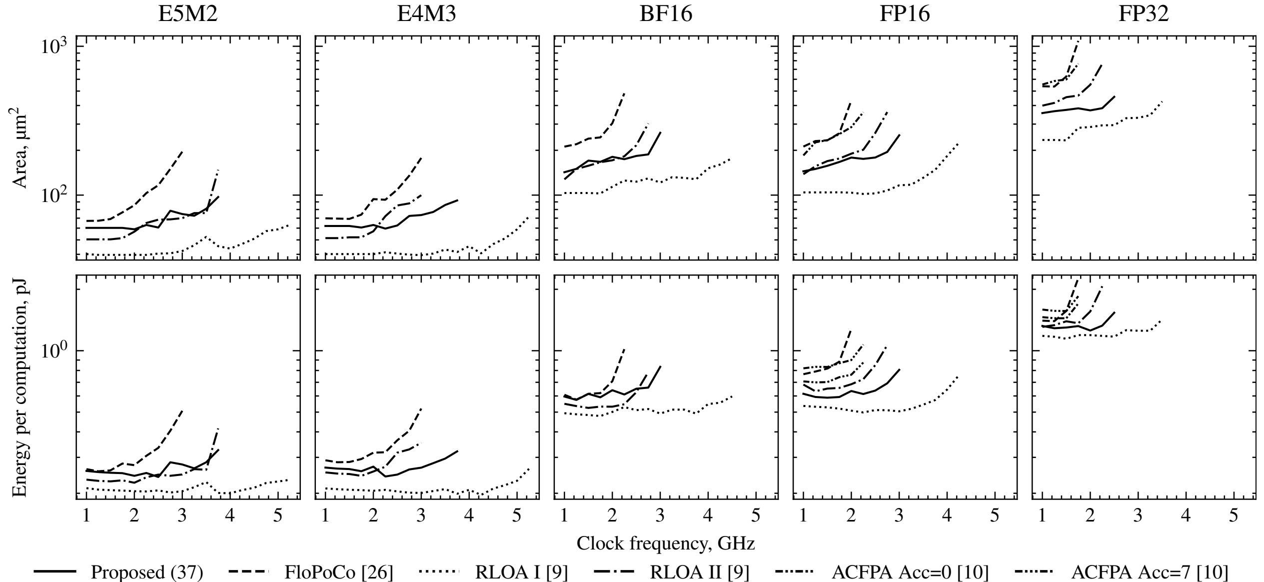

Fig. 9. Area and energy consumption of different floating-point adders for different formats, plotted in logarithmic scale. The energy consumption of the ACFPA designs was measured twice: once configured with Acc=0 and once with Acc=7.

overflow, infinity, and NaN handling were removed explicitly for direct comparison. All designs were implemented with the same capabilities and as combinatorial circuits. The proposed, however, does inherently propagate infinity and NaN when no overflow occurs, as mentioned in Section III-E.

1) ASIC: Synthesis was performed using Synopsys Design Compiler 2023.12, targeting 28-nm FD-SOI technology with a 1.0 V supply voltage, an ambient temperature of 125 °C, and a slow-slow process corner. Registers were added on the input and output, and are therefore included in the results. Energy measurements are based on the switching activity from 10^3 random input samples with uniformly distributed bit patterns.

Figure 9 shows the results for area and energy consumption. The most efficient design is RLOA I. This is expected since the mantissas are simply OR-ed together, and thus the exponent never has to be incremented. Otherwise, the proposed design compares better, or at least matches, the related work.

2) FPGA: The designs were synthesized for an AMD Artix-7, `xc7a12tcpg238-3`, using Vivado 2023.2 with default settings and out-of-context mode. Registers were added to the input and output.

Synthesis results are found in Table III. As seen, the proposed design uses the same number of LUTs for formats with the same bit lengths, which is not surprising. The proposed design also uses fewer LUTs and has a shorter delay than the exact adders from FloPoCo. Improvements over RLOA II are also observed for larger formats. ACFPA is again larger due to its configurability.

The addition $X + Y + C$ can be mapped efficiently using ternary adders on modern FPGAs [27]. On AMD platforms, where 6-input LUTs can be divided into two 5-input LUTs, ternary adders require only a single LUT6 per output bit [28].

TABLE III
FPGA Synthesis Results for Floating-Point Adders

Format	Design	Delay, ns	LUTs
E5M2	Proposed (37)	3.881	27
	RLOA I [9]	3.006	16
	RLOA II [9]	4.684	24
	FloPoCo [25]	5.405	32
E4M3	Proposed (37)	4.103	27
	RLOA I [9]	2.981	15
	RLOA II [9]	3.967	28
	FloPoCo [25]	4.652	37
BF16	Proposed (37)	4.521	55
	RLOA I [9]	4.396	46
	RLOA II [9]	6.556	56
	FloPoCo [25]	7.913	75
FP16	Proposed (37)	4.850	55
	RLOA I [9]	3.829	41
	RLOA II [9]	7.086	57
	ACFPA [10]	7.010	94
	FloPoCo [25]	7.209	79
FP32	Proposed (37)	4.987	111
	RLOA I [9]	5.450	117
	RLOA II [9]	8.415	139
	ACFPA [10]	9.816	233
	FloPoCo [25]	9.537	170

V. Discussion

The result selection in the proposed expression currently depends on an initial approximation. However, since the tipping points of the max function can be derived, it is possible to select the appropriate term based on the input. In particular, expression (37) is numerically equivalent to

$$\begin{cases} X, & \text{for } X - Y \geq 2D \\ Y, & \text{for } X - Y \leq -2D \\ (X + Y + C) \gg 1, & \text{otherwise.} \end{cases} \quad (40)$$

This provides more degrees of freedom in selecting terms, which can be useful for dedicated hardware implementations.

The proposed approximation can be implemented on many systems without requiring branches. It is not rare for general-purpose processors to support conditional assignments, and masking instructions are typically available on graphical processing units (GPUs). With either strategy, the pipeline does not need to be flushed, which is especially attractive for deeply pipelined architectures.

Future work includes finding optimal error-correcting constants and evaluating performance on processor platforms. Lastly, the idea $x + y \approx 2\sqrt{xy}$ can be extended to general sums,

$$\sum_{i=1}^{N} x_i \approx N \sqrt[N]{\prod_{i=1}^{N} x_i}, \tag{41}$$

but it will be sensitive. A $\max$ function must still be incorporated unless the summands have similar magnitude.

VI. Conclusions

A novel approach to approximating floating-point addition using the geometric mean was presented in this paper. The geometric mean is, in turn, approximated using integer operations in the logarithmic domain, making it practical for integer machines. Although restricted to positive inputs, it can be suitable for, e.g., sums of squares and exponentials. It is shown that the maximum absolute relative error is 0.2, independent of the format, but improvements are possible by adjusting a constant. Correct rounding is obtained when the operands share the same exponent, or when the exponent difference is greater than the number of mantissa bits. Infinity and NaN are propagated when no overflow occurs, unlike previous work. Synthesis results for standard-cell and FPGA technology demonstrate a favorable trade-off between resource usage, delay, and numerical performance. Compared to designs that reach higher clock frequencies on ASIC, the proposed design offers better accuracy. Compared to more accurate designs, the proposed design uses less area for ASIC and exhibits shorter delays for both ASIC and FPGA.

References

[1] J. F. Blinn, "Floating-point tricks," *IEEE Comput. Graph. Appl.*, vol. 17, no. 4, pp. 80–84, 1997.

[2] O. Gustafsson and N. Hellman, "Approximate floating-point operations with integer units by processing in the logarithmic domain," in *Proc. IEEE Symp. Comput. Arithmetic*, Jun. 2021, pp. 45–52.

[3] T. Lindberg and O. Gustafsson, "Exact eight-bit floating-point multiplication using integer arithmetic," in *Proc. Asilomar Conf. Signals Syst. Comput.*, 2025, pp. 315–320.

[4] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IRE Trans. Electron. Comput.*, vol. EC-11, no. 4, pp. 512–517, Aug. 1962.

[5] M. Kwak, S. Lee, and Y. Kim, "Design of approximate floating-point arithmetic units using hardware-efficient rounding schemes," *IEEE Embedded Syst. Lett.*, pp. 1–1, 2025.

[6] S. K. Beura, J. Samanta, B. P. Devi, and P. Saha, "Floating point adder using inexact mantissa adder," in *Proc. Int. Conf. Comput. Commun. Syst.*, 2023, pp. 1–5.

[7] M. Kwak, J. Lee, H. Seo, M. Sung, and Y. Kim, "Training and inference using approximate floating-point arithmetic for energy efficient spiking neural network processors," in *Proc. Int. Conf. Electron. Inf. Commun.*, 2021, pp. 1–2.

[8] R. Omidi and S. Sharifzadeh, "Design of low power approximate floating-point adders," *Int. J. Circuits Theory Appl.*, vol. 49, no. 1, pp. 185–195, 2021.

[9] W. Liu, L. Chen, C. Wang, M. O'Neill, and F. Lombardi, "Design and analysis of inexact floating-point adders," *IEEE Trans. Comput.*, vol. 65, no. 1, pp. 308–314, 2016.

[10] L. Tegazzini, G. Di Meo, A. Torino, F. Del Prete, D. de Caro, C. Parrella, and A. Giuseppe Maria Strollo, "Novel approximate floating-point adder with runtime tunable precision," *IEEE Access*, vol. 13, pp. 177 970–177 985, 2025.

[11] C. Bertin, N. Brisebarre, B. D. de Dinechin, C.-P. Jeannerod, C. Monat, J.-M. Muller, S.-K. Raina, and A. Tisserand, "A floating-point library for integer processors," in *Proc. Adv. Signal Process. Algorithms Archit. Implementations*, F. T. Luk, Ed., vol. 5559, International Society for Optics and Photonics. SPIE, 2004, pp. 101 – 111.

[12] C.-P. Jeannerod, C. Mouilleron, J.-M. Muller, G. Revy, C. Bertin, J. Jourdan-Lu, H. Knochel, and C. Monat, "Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors," in *Proc. ACM Int. Workshop Parallel Symbolic Comput.* New York, NY, USA: Association for Computing Machinery, 2010, p. 1–9.

[13] C.-P. Jeannerod, J. Jourdan-Lu, C. Monat, and G. Revy, "How to square floats accurately and efficiently on the ST231 integer processor," in *Proc. IEEE Symp. Comput. Arithmetic*, 2011, pp. 77–81.

[14] L. Gerlach, G. Payá-Vayá, and H. Blume, "Efficient emulation of floating-point arithmetic on fixed-point SIMD processors," in *Proc. IEEE Int. Workshop Signal Process. Syst.*, 2016, pp. 254–259.

[15] J. Le Maire, N. Brunie, F. De Dinechin, and J.-M. Muller, "Computing floating-point logarithms with fixed-point operations," in *Proc. IEEE Symp. Comput. Arithmetic*, 2016, pp. 156–163.

[16] J. Van Der Hoeven, "Multiple precision floating-point arithmetic on SIMD processors," in *Proc. IEEE Symp. Comput. Arithmetic*, 2017, pp. 2–9.

[17] "FLIP: Floating-point library for integer processors," Oct. 2021. [Online]. Available: https://flip.gitlabpages.inria.fr/www/

[18] "emFloat the floating-point library," https://www.segger.com/products/development-tools/runtime-library/technology/floating-point-library/.

[19] "I found a *useful* method to calculate log(a+b), check it out," Feb. 2016. [Online]. Available: https://www.physicsforums.com/threads/i-found-a-useful-method-to-calculate-log-a-b-check-it-out.859730/

[20] IEEE Computer Society, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019*, pp. 1–84, 2019.

[21] Intel. (2018) Bfloat16 - hardware numerics definition. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf

[22] A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan, "DLFloat: A 16-b floating point format designed for deep learning training and inference," in *Proc. IEEE Symp. Comput. Arithmetic*, 2019, pp. 92–95.

[23] "OCP 8-bit floating point specification (OFP8) revision 1.0," 2023. [Online]. Available: https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-12-01-pdf-1

[24] NVIDIA, "NVIDIA A100 tensor core GPU architecture," NVIDIA, Tech. Rep., 2020. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[25] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Des. Test. Comput.*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[26] 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*, 1st ed. Birkhäuser Basel, Nov. 2009.

[27] M. Kumm, O. Gustafsson, M. Garrido, and P. Zipf, "Optimal single constant multiplication using ternary adders," *IEEE Trans. Circuits Syst. II*, vol. 65, no. 7, pp. 928–932, 2018.

[28] J. M. Simkins and B. D. Philofsky, "Structures and methods for implementing ternary adders/subtractors in programmable logic devices," Sep. 25 2007, US Patent 7,274,211.