Efficient, arbitrarily high precision hardware logarithmic arithmetic for linear algebra

Jeff Johnson
Facebook AI Research
Los Angeles, CA, United States
jhj@fb.com

***Abstract*—The logarithmic number system (LNS) is arguably not broadly used due to exponential circuit overheads for summation tables relative to arithmetic precision. Methods to reduce this overhead have been proposed, yet still yield designs with high chip area and power requirements. Use remains limited to lower precision or high multiply/add ratio cases, while much of linear algebra (near 1:1 multiply/add ratio) does not qualify.**

We present a dual-base approximate logarithmic arithmetic comparable to floating point in use, yet unlike LNS it is easily fully pipelined, extendable to arbitrary precision with $\mathcal{O}(n^2)$ overhead, and energy efficient at a 1:1 multiply/add ratio. Compared to float32 or float64 vector inner product with FMA, our design is respectively 2.3× and 4.6× more energy efficient in 7 nm CMOS. It depends on exp and log evaluation 5.4× and 3.2× more energy efficient, at 0.23× and 0.37× the chip area for equivalent accuracy versus standard hyperbolic CORDIC using shift-and-add and approximated ODE integration in the style of Revol and Yakoubsohn. This technique is a novel alternative for low power, high precision hardened linear algebra in computer vision, graphics and machine learning applications.

***Index Terms*—elementary function evaluation, approximate arithmetic, logarithmic arithmetic, hardware linear algebra**

I. Introduction

Energy efficiency is typically the most important challenge in advanced CMOS technology nodes. With the *dark silicon* problem [1], the vast majority of a large scale design is clock or power gated at any given point in time. Chip area becomes exponentially more available relative to power consumption, preferring "a new class of architectural techniques that 'spend' area to 'buy' energy efficiency" [2]. Memory architecture is often the most important concern, with 170-6400× greater DRAM access energy versus arithmetic at 45 nm [3]. This changes with the rise of machine learning, as heavily employed linear algebra primitives such as matrix/matrix product offer substantial local reuse of data by algorithmic tiling [4]: $\mathcal{O}(n^3)$ arithmetic operations versus $\mathcal{O}(n^2)$ DRAM accesses. This is a reason for the rise of dedicated neural network accelerators, as memory overheads can be substantially amortized over many arithmetic operations in a fixed function design, making arithmetic efficiency matter again.

Many hardware efforts for linear algebra and machine learning tend towards low precision implementations, but here we concern ourselves with the opposite: enabling (arbitrarily) high precision yet energy efficient substitutes for floating point or long word length fixed point arithmetic. There are a variety of ML, computer vision and other algorithms where accelerators cannot easily apply precision reduction, such as hyperbolic embedding generation [5] or structure from motion via matrix factorization [6], yet provide high local data reuse potential.

The logarithmic number system (LNS) [7] can provide energy efficiency by eliminating hardware multipliers and dividers, yet maintains significant computational overhead with *Gaussian logarithm* functions needed for addition and subtraction. While reduced precision cases can limit themselves to relatively small LUTs/ROMs, high precision LNS require massive ROMs, linear interpolators and substantial MUXes. Pipelining is difficult, requiring resource duplication or handling variable latency corner cases as seen in [8]. The ROMs are also exponential in LNS word size, so become impractical beyond a float32 equivalent. Chen et al. [9] provide an alternative fully pipelined LNS add/sub with ROM size a $\mathcal{O}(n^3)$ function of LNS word size, extended to float64 equivalent precision. However, in their words, "[our] design of [a] large word-length LNS processor becomes impractical since the hardware cost and the pipeline latency of the proposed LNS unit are much larger." Their float64 equivalent requires 471 Kbits ROM and at least 22,479 full adder (FA) cells, and 53.5 Kbits ROM and 5,550 FA cells for float32, versus a traditional LNS implementation they cite [10] with 91 Kbits of ROM and only 586 FA cells.

While there are energy benefits with LNS [11], we believe a better bargain can be had. Our main contribution is a trivially pipelined logarithmic arithmetic extendable to arbitrary precision, using no LUTs/ROMs, and a $\mathcal{O}(n^2)$ precision to FA cell dependency. Unlike LNS, it is substantially more energy efficient than floating point at a 1:1 multiply/add ratio for linear algebra use cases. It is approximate in ways that an accurately designed LNS is not, though with parameters for tuning accuracy to match LNS as needed. It is based on the *ELMA* technique [12], extended to arbitrary precision with an energy efficient implementation of exp/log using restoring shift-and-add [13] and an ordinary differential equation integration step from Revol and Yakoubsohn [14] but with approximate multipliers and dividers. It is tailored for vector inner product, a foundation of much of linear algebra, but remains a general purpose arithmetic. We will first describe our hardware exp/log implementations, then detail how they are used as a foundation for our arithmetic, and provide an accuracy analysis. Finally, hardware synthesis results are presented and compared with floating point.

II. Notes on hardware synthesis

All designs considered in this paper are on a commercially available 7 nm CMOS technology constrained to only SVT cells. They are generated using Mentor Catapult high level synthesis (HLS), biased towards min latency rather than min area, with ICG (clock gate) insertion where appropriate. Area is reported via Synopsys Design Compiler, and power/energy is from Synopsys PrimeTime PX from realistic switching activity. Energy accounts for combinational, register, clock tree and leakage power, normalized with respect to module throughput in cycles, so this is a per-operation energy. We consider pipelining acceptable for arithmetic problems in linear algebra with sufficient regularity such as matrix multiplication (Section VI-B), reducing the need for purely combinational latency reduction. Power analysis is at the TT@25C corner at nominal voltage. Design clocks from 250-750 MHz were considered, with 375 MHz chosen for reporting, being close to minimum energy for many of the designs. Changing frequency does change pipeline depth and required register/clock tree power, as well as choice of inferred adder or other designs needed to meet timing closure by synthesis.

III. exp/log evaluation

Our arithmetic requires efficient hardware implementation of exponential b^x and logarithm $\log_b(x)$ for a base b, which are useful in their own right. Typical algorithms are power series evaluation, polynomial approximation/table-based methods, and shift-and-add methods such as hyperbolic CORDIC [15] or the simpler method by De Lugish [16]. Hardware implementations have been considered for CORDIC [17], ROM/table-based implementations [18], approximation using shift-and-add [19] with the Mitchell logarithm approximation [20], and digit recurrence/shift-and-add [21]. CORDIC requires three state variables and additions per iteration, plus a final multiplication by a scaling factor. BKM [22] avoids the CORDIC scaling factor but introduces complexity in the iteration step.

Much of the hardware elementary function literature is concerned with latency reduction rather than energy optimization. Variants of these algorithms such as high radix formulations [23] [24] [21] or parallel iterations [17] increase switching activity via additional active area, iteration complexity, or adding sizable MUXes in the radix case. In lieu of decreasing combinational delay via parallelism, pipelining is a worthwhile strategy to reduce energy-delay product [25], but only with high pipeline utilization and where register power increases are not substantial. Ripple-carry adders, the simplest and most energy efficient adders, remain useful in the pipelined regime, and variants like variable block adders improve latency for minimal additional energy [26]. Fully parallel adders like carry-save can improve on both latency and switching activity for elementary functions [21], but only where the redundant number system can be maintained with low computational overhead. For example, in shift-and-add style algorithms, adding a shifted version of a carry-save number to itself requires twice the number of adder cells as a simple ripple-carry adder (one to add each of the shifted components), resulting in near double the energy. Eliminating registers via combinational multicycle paths (MCPs) is another strategy, but as the design is no longer pipelined, throughput will suffer, requiring an introduction of more functional units or accepting the decrease in throughput. There is then a tradeoff between clock frequency, combinational latency reduction, pipelining for timing closure, MCP introduction, and functional unit duplication versus energy per operation.

IV. e^x shift-and-add with integration

We consider De Lugish-style restoring shift-and-add, which will provide ways to reduce power or recover precision with fewer iterations (Section IV-A and IV-B). The procedure for exponentials $y = b^x$ is described in Muller [13] as:

$$
\begin{aligned}
L_0 &= x \\
L_{n+1} &= L_n - \log_b(1 + d_n 2^{-n}) \\
E_0 &= 1 \\
E_{n+1} &= E_n(1 + d_n 2^{-n}) \\
d_n &= \begin{cases} 1 & \text{if } L_n \geq \log_b(1 + 2^{-n}) \\ 0 & \text{otherwise} \end{cases} \\
y &= E_I \text{ (at desired iteration } I)
\end{aligned}
$$

The acceptable range of x is $[0, \sum_{n=0}^{\infty} \log_b(1 + 2^{-n}))$, or $[0, 1.56\dots)$ for $b = e$ (Euler's number). Range reduction techniques considered in [13] can be used to reduce arbitrary x to this range. This paper will only consider $b = e$ and limiting x as fixed point, $x \in [0, \ln(2))$, restrictions discussed in Sections IV-A and VI-C.

We must consider rounding error and precision of x, L_n and E_n. Our range-limited x can be specified purely as a fixed point fraction with x_{bits} fractional bits. The iteration $n = 1$ is skipped as $x \in [0, \ln(2))$. All subsequent L_n are < 1 and can be similarly represented as a fixed point fraction. These L_n will use ℓ fractional bits ($\ell \geq x_{\text{bits}}$) with correctly rounded representations of $\ln(1+d_n 2^{-n})$. $E_n \in [1, 2)$ is the case in our restricted domain, which is maintained as a fixed point fraction with an implicit, omitted leading integer 1. Multiplication by 2^{-n} is a shift by n bits, so we append this leading 1 to the post-shifted E_n before addition. We use p fractional bits to represent E_n. At the final I-th iteration, E_I is rounded to $y_{\text{bits}} \leq p$ bits for the output y. Ignoring rounding error, the relative error of the algorithm is $|e^x - E_I|/e^x = 2^{-I+1}$ at iteration I, so for 23 fractional bits, $I = 24$ is desired.

All adders need not be of size ℓ or p, either. L_n reduces in magnitude at each step; with $y \in [1, 2)$, L_n only needs the $\ell - \max(0, n-2)$ LSB fractional bits. E_n has a related bit size progression $0, 0, 1, 3, 5, 9, 14, \dots$, as E_n is $\max(p, \text{size}(E_{n-1}) + n)$ fractional bits, except starting at $n \geq 3$ we are off by 1 (d_1, d_2, d_3 cannot all be 1, as $\sum_{i=1}^{3} \ln(1 + 2^{-i}) > \ln(2)$). While L_n successively reduces in precision from ℓ, we limit E_n to p fractional bits via truncation (bits shifted beyond position p are ignored). As

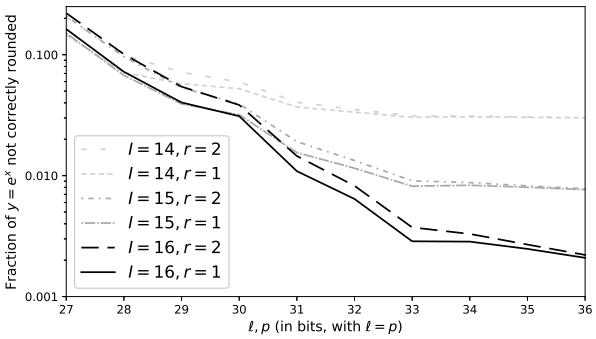

Fig. 1. Our e^x accuracy at $x_{\text{bits}} = y_{\text{bits}} = 23$ relative to I, ℓ, p, r. All configurations have ≤ 1 ulp error.

with [21], we can deal with truncation error by setting $p = y_{\text{bits}} + (\lceil \log_2(I) \rceil + 1)$, using extra bits as guard bits.

L_n requires an adder and MUX ($L_n = L_{n-1} - \ln(1 + 2^{-n+1})$ if $d_n = 1$, or $L_n = L_{n-1}$ if $d_n = 0$). The constants $\ln(1+2^{-n})$ are hard-wired into adders when iterations are fully unrolled (a separate adder for each iteration). The E_n do not use a full adder of size p in the general case; only shifted bits that overlap with previously available bits need full adder cells. The L_n can also be performed first, with d_n stored in flops (to reduce glitches) for data gating E_n additions, reducing switching activity at the expense of higher latency, as 25% of the d_i on average will remain zero across iterations.

One can use redundant number systems for L_n and E_n and avoid full evaluation of the L_n comparator [13], but E_n is problematic. In the non-redundant case, only a subset of the shifted E_n require a full adder, and the remainder only a half adder. With a carry-save representation for E_n, two full adders are required for the entire length of the word, one to add each portion of the shifted carry-save representation. While the carry-save addition is constant latency, it requires more full adder cells. In our evaluation carry-save for E_n prohibitively increases power over synthesis-inferred adder choice. At high clock frequencies (low latency) this tradeoff is acceptable, but low power designs will generally avoid this regime.

A. Euler method integration

This algorithm is simple but has high latency from the sequential dependency of many adders, high ℓ, p and iterations I for accurate results. For significant latency and energy reduction, Revol and Yakoubsohn [14] show that about half the iterations can be omitted by treating the problem as an ordinary differential equation with a single numerical integration step. e^x satisfies the ODE $y' = f(x, y) = y$ where $y(0) = 1$. They consider in software both an explicit Euler method and 4th order Runge-Kutta (RK4). RK4 involves several multipliers and is not a good energy tradeoff to avoid more iterations. The explicit Euler method step has a single multiplication:

$$\begin{aligned} y &= E_I && \text{no integration step} \Rightarrow \\ y &= E_I + L_I E_I && \text{explicit Euler method step} \end{aligned}$$

at the I-th terminal iteration, with residual L_I used as the step size. They give a formula for I at a desired accuracy of ϵ of $L_I \leq \sqrt{2\epsilon/e^{1.56}}$, ignoring truncation error. Note that $L_{n+1} < 2\ln(1 + 2^{-n}) \approx 2^{-n+1}$. Thus, for single-precision $\epsilon = 2^{-24}$, we need $L_I \leq 2^{-12.63\ldots}$, or $I \geq 14$. Double-precision $\epsilon = 2^{-53}$ has $I \geq 29$, and quad precision $\epsilon = 2^{-113}$ has $I \geq 59$. Implementation of 2^x from this requires pre-multiplication of x by a fixed point rounding of $\ln(2)$, a significant energy overhead.

TABLE I
FULLY PIPELINED EXP/LOG, $x_{\text{BITS}} = y_{\text{BITS}} = 23$ SYNTHESIS RESULTS

e^x	(0.5, 1] ulp err	Cycles	Area μm^2	Energy pJ
CORDIC	9.98%	4	1738	2.749
Ours	**9.90%**	**2**	**407.2** (0.23×)	**0.512** (0.19×)
$\ln(x)$				
CORDIC	14.4%	4	2084	3.573
Ours	**14.8%**	**4**	**769.4** (0.37×)	**1.107** (0.31×)

B. Integration via approximate multiplication

We have $L_I \in [0, 1)$, $E_I \in [1, 2)$ when $x \in [0, \ln(2))$. The Euler method step multiplication $L_I E_I$ would be a massive $\ell \times (1 + p)$ bits, with the 1+ for the leading integer 1 bit of E_I. Let L_I^f and E_I^f denote fractional portions of L_I and E_I. The step can then be expressed as:

$$y = E_I + (0 + L_I^f)(1 + E_I^f) = E_I + L_I^f + L_I^f E_I^f$$

$L_I^f E_I^f$ now solely involves fractional bits, of which we only care about y_{bits} to p MSBs produced. L_I^f has $\max(I - 2, 0)$ zero MSBs, so there are $\max(I - 2, 0)$ ignorable zero MSBs in the resulting product, yielding a $p \times (\ell - \max(I - 2, 0))$ multiplier, still an exact step calculation. Assuming $I > 2$ and given these zero MSBs, we only need $(p - I + 2)$ MSBs of the result, so we truncate both L_I^f and E_I^f to limit the result to this size (truncation ignores carries from the multiplication of the truncated LSBs). We do this symmetrically, and since usually $p > y_{\text{bits}}$, we take $\ell - (I - 2) - r$ fractional MSBs from E_I^f, with an option to remove another r bits, $0 \leq r \leq 4$. This may not produce enough bits to align properly with p, so we append zeros to the LSBs as needed to match the size of p. For example, at $x_{\text{bits}} = 23$, $y_{\text{bits}} = 24$, $\ell = p = 28$, $I = 14$, $r = 2$, we have a 14×14 multiplier, of which we only need 16 MSBs (based on alignment with E_I^f), and the ultimate carry from the 12 LSBs. One can consider other approximate multipliers [27], but truncation seems to work well and provides a significant reduction in energy.

C. Error analysis and synthesis results

The table maker's dilemma is unavoidable for transcendental functions [28]. For $x_{\text{bits}} = y_{\text{bits}} = 23$, we need to evaluate e^x to at least 42 bits to provide correctly rounded results for fixed point $x \in [0, \ln(2))$. In lieu of exact evaluation, we demand function monotonicity, ≤ 1 ulp error, and consider the occurrence of incorrectly rounded results (> 0.5 ulp error). Figure 1 considers error in this regime with a sweep of I, ℓ, p and r, with $\ell = p$. $\ell, p < 27$ has maximum > 1 ulp error.

Table I shows fully-pipelined (iterations unrolled), near iso-accuracy synthesis results for our method ($I = 14$, $\ell = p = 28$, $r = 2$) and standard hyperbolic CORDIC (28 iterations and

29 fractional bit variables). All implementations have ≤ 1 ulp error, with the fraction at $(0.5, 1]$ error shown shown. We are 5.4× more energy efficient, 0.23× the area, and half the latency in cycles; as discussed earlier, most CORDIC modifications reduce latency at the expense of increased energy.

V. $\ln(x)$ SHIFT-AND-ADD WITH INTEGRATION

$y = \ln(x)$ is similar to e^x with roles of E_n and L_n reversed, with division for the integration [14]:

$$
\begin{aligned}
E_0 &= 1 \\
E_{n+1} &= E_n(1 + d_n 2^{-n}) \\
L_0 &= 0 \\
L_{n+1} &= L_n + d_n \ln(1 + 2^{-n}) \\
d_n &= \begin{cases} 1 & \text{if } E_n(1 + 2^{-n}) \leq x \\ 0 & \text{otherwise} \end{cases} \\
y &= L_I + (x - E_I)/E_I \text{ (Euler method step)}
\end{aligned}
$$

We restrict ourselves to $x \in [1, 2)$. The error of $E_I \approx \ln(x)$ is $\leq 2^{-I+1}$, with the target number of iterations I (ignoring truncation error) for error ϵ given when $(x - E_I) \leq \sqrt{2\epsilon}$. For single precision $\epsilon = 2^{-24}$, $I = 13$, and double precision $\epsilon = 2^{-53}$, $I = 27$, and $\epsilon = 2^{-113}$ is $I = 57$. Prior discussion concerning the E_n and L_n sequences and data gating with d_n carry over to this algorithm. It is also the case that the running sum L_n is not needed until the very end, so a carry-save adder postponing full evaluation of carries is appropriate. It is possible to use a redundant number system for E_I and avoid full evaluation of the comparison [13], but the required shift with add increases switching activity significantly.

A. Integration via approximate division

We approximate the integration division by truncating the dividend and divisor. The dividend $(x - E_I) \in [0, 1)$ has at least $\max(0, I-3)$ zero fractional MSBs, and the divisor $E_I \in [1, 2)$, so the result is a fraction that we must align with the ℓ bits in L_I for the sum. We skip known zero MSBs, and some number r of the LSBs of the dividend. For the divisor E_I, we need not use the entire fractional portion but choose only some number of fractional bits s. We then have a $(p - \max(0, I - 3) - r)$ by $1+s$ fixed point divider (1+ is for the leading integer 1 of E_I). $r = 3, s = 9$ is reasonable in our experiments. This is higher area and latency than the truncated multiplier (we only evaluated truncated division with digit recurrence), but the increase in resources of log versus exp is acceptable for linear algebra use cases (Section VIII).

B. Error analysis and synthesis results

As before, we only consider monotonic implementations with ≤ 1 ulp error, and consider the frequency of incorrectly rounded results. Figure 2 shows such error occurrence versus a sweep of I, ℓ, p, s, with $\ell = p$. s has a larger accuracy effect than r, and $r = 3$ yields reasonable results, so all are constrained to $r = 3$. Table I shows near iso-accuracy synthesis results for our method ($I = 15$, $\ell = p = 28$, $r = 3$, $s = 9$) and standard hyperbolic CORDIC (28 iterations and 30 fractional bit variables). Our implementation is 3.2× more energy efficient at 0.37× area versus CORDIC, with much of the latency and energy coming from the truncated divider. The higher resource consumption of log over exp CORDIC is from the initialization of the X and Y CORDIC variables to $x+1$, $x-1$ rather than 1, with propagation of inferred required adder length by HLS throughout when synthesizing the design.

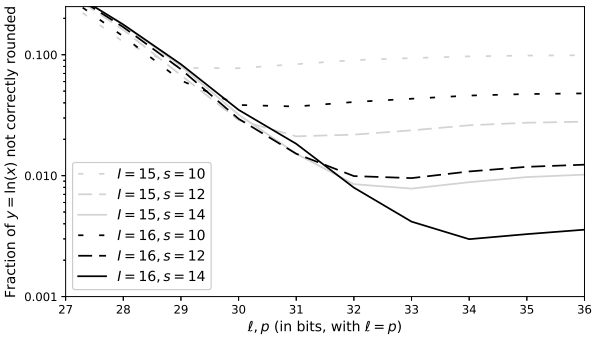

Fig. 2. Our $\ln(x)$ accuracy at $x_{\text{bits}} = y_{\text{bits}} = 23$ relative to I, ℓ, p, s with $r = 3$. All configurations have ≤ 1 ulp error.

VI. APPROXIMATE LOGARITHMIC ARITHMETIC

We show how the preceding designs are used to build an arbitrarily high precision logarithmic arithmetic with some (tunably) approximate aspects.

A. LNS arithmetic

The sign/magnitude logarithmic number system (LNS) [7] represents values $x \in \mathbb{R}$ as a rounded fixed point representation to some number of integer and fractional bits of $\log_b(|x|)$, plus a sign and zero flag. The base b is typically 2. We refer to $x' = \{\pm \log_b(|x|) \text{ or } 0\}$ as a representation of x in the *log domain*. We refer to rounding and encoding x as integer, fixed or floating point as a *linear domain* representation, though note that floating point itself is a combination of log and linear representations for the exponent and significand.

The benefit of LNS is simplifying multiplication, division and power/root. For log domain x', y', multiplication or division of the corresponding linear domain x and y is $x' \pm y'$, n-th power of x is nx' and n-th root of x is x'/n, with sign, zero (and infinity/NaN flags if desired) handled in the obvious manner. Addition and subtraction, on the other hand, require *Gaussian logarithm* computation. For linear domain x, y, log domain add/sub of the corresponding x', y' is:

$$
\begin{aligned}
\log_b(|x| + |y|) &= x' + \log_b(1 + b^r) \\
\log_b(|x| - |y|) &= x' + \log_b(|1 - b^r|)
\end{aligned}
$$

where $r = y' - x'$. Without loss of generality, we restrict $y' \leq x'$, so we only consider $r \leq 0$. These functions are usually implemented with ROM/LUT tables (possibly with interpolation) rather than direct function evaluation, ideally realized to ≤ 0.5 log domain ulp relative error. The subtraction function has a singularity at $r = 0$, corresponding to exact cancellation $y - x = 0$, with the region $r \in [-\epsilon, 0)$ very

near the singularity corresponding to near-exact cancellation. Realizing this critical region to 0.5 log ulp error without massive ROMs (241 Kbits in [29]) is a motivation for *subtraction co-transformation* to avoid the singularity, which can reduce the requirement to at least 65 Kbits [11]. Some designs are proposed as being ROM-less [30], but in practice the switching power and leakage of the tables' combinational cells would still be huge. Interpolation with reduced table sizes can also be used, but the formulation in [31] only considers log addition without the singularity. An ultimate limit on the technique not far above float32 equivalent is still faced, as accurate versions of these tables scale exponentially with word precision [9].

Pipelined LNS add/sub is another concern. As mentioned in Section I, Chen et al. [9] have an impractical fully pipelined implementation. Coleman et al. [8] have add/sub taking 3 cycles to complete, but chose to duplicate rather than pipeline the unit, and mention that the latency is dominated by memory (ROM) access. Arnold [32] provides a fully pipelined add/sub unit, but with a "quick" instruction version that allows the instruction to complete in either 4 or 6 cycles if it avoids the subtraction critical region. On the other hand, uniformity may increase latency, as different pipe stages are restricted to different ROM segments.

When combining an efficient LNS multiply with the penalty of addition for linear algebra, recent work by Popoff et al. [11] show an energy penalty of 1.84× over IEEE 754 float32 (using naive sum of add and mul energies), a 4.5× area penalty for the entire LNS ALU, and mention 25% reduced performance for linear algebra kernels such as GEMM. Good LNS use cases likely remain workloads with high multiply-to-add ratios.

B. ELMA/FLMA logarithmic arithmetic

The ELMA (*exact log-linear multiply-add*) technique [12] is a logarithmic arithmetic that avoids Gaussian logarithms. It was shown that an 8-bit ELMA implementation with extended dynamic range from posit-type encodings [33] is more energy efficient in 28 nm CMOS than 8/32-bit integer multiply-add (as used in neural network accelerators). It achieved similar accuracy as integer quantization on ResNet-50 CNN [34] inference on the ImageNet validation set [35], simply with float32 parameters converted via round-to-nearest only and all arithmetic in the ELMA form. Significant energy efficiency gains over IEEE 754 float16 multiply-add were also shown, though much higher precision was then impractical.

We describe ELMA and its extension to FLMA (*floating point log-linear multiply-add*). In ELMA, mul/div/root/power is in log domain, while add/sub is in linear domain with fixed point arithmetic. Let $p(x')$ convert log domain x' (with E integer and F fractional log bits) to linear domain, and $q(y)$ convert linear domain y to log domain. $p(x')$ and $q(y)$ are both approximate conversions (LNS values are irrational). $p(x')$ produces fixed point (ELMA) or floating point (FLMA); in base-2 FLMA, we obtain $p(x') = \{\pm 2^{\lfloor x' \rfloor} 2^{(x' - \lfloor x' \rfloor)}$ or $0\}$, yielding a linear domain floating point exponent and significand. $p(x')$ can increase precision by α bits, with the exponential evaluated to $y_{\text{bits}} = f + \alpha$ fractional bits. Unique conversion for base-2 requires $\alpha \geq 1$, as the minimum derivative of 2^x, $x \in [0, 1)$ is less than 1.

FLMA approximates the linear domain sum $\sum_i x_i$ on the log domain x'_i as $q(\sum_i p(x'_i))$. $q(\cdot)$ uses the floating point exponent as the log domain integer portion, and evaluates $\log_2$ on the significand, back to the required F log domain fractional bits. The fixed or floating point accumulator can use a different fractional precision A ($A \geq F + \alpha$) than $p(\cdot)$, in which case $q(\cdot)$ can consider $F + \beta$ linear domain MSB fractional bits of A with rounding for the reverse conversion. $q(\cdot)$ is similarly unique only when $\beta \geq 1$. Typically we have $\alpha = \beta \geq 1$, and $A = F + \alpha$. As α, β increase, we converge to exact LNS add/sub. As with LNS, if add/sub is the only operation, ELMA/FLMA does not make sense. It is tailored for linear algebra sums-of-products; conversion errors are likely to be uncorrelated in use cases of interest (Sections VII-B and VII-C), but is substantially efficient over floating point at a 1:1 multiply-to-add ratio (Section VIII).

Unlike LNS, a ELMA design (and FLMA, depending upon floating point adder latency) can be easily pipelined and accept a new summand every cycle for accumulation without resource duplication (*e.g.*, LNS ROMs). Furthermore, accumulator precision A can be (much) greater than log domain F; in LNS this requires increasing Gaussian logarithm precision to the accumulator width. These properties make ELMA/FLMA excellent for inner product, where many sums of differing magnitudes may be accumulated. FLMA is related to [36], except that architecture is oriented around a linear domain floating point representation such that all mul/div/root is done with a log conversion to LNS, the log domain operation, and an exp conversion back to linear domain. Their log/exp conversions were further approximated with linear interpolation. Every mul/div/root operation thus included the error introduced by both conversions.

C. Dual-base logarithmic arithmetic

ELMA/FLMA requires accurate calculation of the fractional portions of $p(x')$ and $q(y)$. Section IV shows calculation of e^x and $\ln(x)$ more accurately for the same resources versus 2^x and $\log_2(x)$. While Gaussian logarithms can be computed irrespective of base, FLMA requires an accessible base-2 exponent to carry over as a floating point exponent. A base-e representation does not easily yield this.

An alternative is a variation of multiple base arithmetic by Dimitrov et al. [37], allowing for more than one base b_i (one of which is usually 2 and the others are any positive real number), with exponents x_i as small integers, producing a representation $\pm \prod_i b_i^{x_i}$ (or zero). We instead use a representation $\pm 2^a e^b$ (or zero) with $a \in \mathbb{Z}$ (encoded in E bits), $b \in [0, \ln(2))$ (encoded as an F bit fixed point fraction). e^b when evaluated yields a FLMA floating point significand in the range $[1, 2)$, which we will refer to as the *Euler significand*. The product of any two of these values $2^a e^b \times 2^c e^d = 2^{a+c} e^{b+d}$ has $e^{b+d} \in [1, 4)$ and $(b + d) \in [0, 2\ln(2))$. For division, $e^{b-d} \in (0.5, 2), (b - d) \in (-\ln(2), \ln(2))$. We no longer have a unique representation

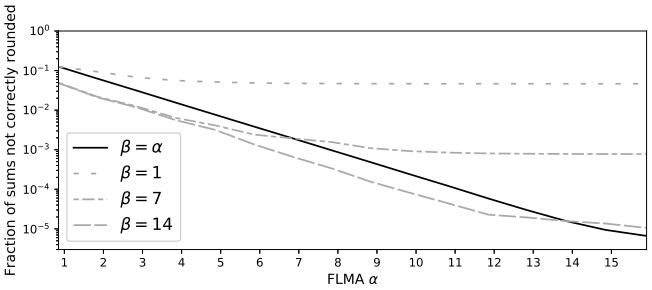

Fig. 3. FLMA sum error of log domain $x+y$ (with $x, y \in [1, 2)$) as a function of α, β. All configurations have ≤ 1 log ulp error, except for $\alpha = \beta = 1$ with ≤ 2 log ulp error.

when we do not limit the base-e exponent to $[0, \ln(2))$; for example, $2^1 e^{0.4} \times 2^{-1} e^{0.3} = 2^0 e^{0.7} = 2^1 e^{(0.7-\ln(2))}$.

We call a base-e exponent in the range $[0, \ln(2))$ a *normalized Euler significand*. Normalization subtracts (or adds) $\ln(2)$ from the base-e exponent and increments (or decrements) the base-2 exponent as necessary to obtain a normalized significand. There are two immediate downsides to this. First, we do not use the full encoding range; our base-e exponent is encoded as a fixed point fraction, but we only use $\approx 69.3\%$ of the values. Encoding a precision/dynamic range tradeoff with the unused portion as in [33] could be considered. The second downside is considered in Section VII-A.

VII. FLMA analysis

We investigate dual-base $\pm 2^a e^b$ arithmetic with FLMA parameters $E = 8$, $F = 23$ (roughly IEEE 754 binary32 equivalent), exp $p = \ell = 27 + \alpha, r = 2, I = 13 + \alpha$, log $p = \ell = 27 + \beta, I = 14 + \beta, r = 3, s = 9$, accumulator $A = F + \alpha$ for choice of α, β. We call this log32 FLMA. For relative error, units in the last place in a fractional log domain representation we call *log ulp*. For instance, 5 (base-e) log ulps are between $2^1 e^{\mathtt{b0.0001}}$ and $2^1 e^{\mathtt{b0.0110}}$, where `b0.0110` is the binary fixed point fraction 0.375.

A. Multiply/divide accuracy

LNS and single base FLMA have 0 log ulp mul/div error, but dual-base FLMA can produce a non-normalized significand, requiring add/sub of a rounding of $\ln(2)$, introducing slight error (≈ 0.016 log ulp for $F = 23$). The extended exp algorithm can avoid this for multiply-add with an additional iteration and integer bits for L_n and E_n, as $2\ln(2) < 1.56$. We would still require additional normalization of $e^{b+d} \in [1, 4)$ to a floating point significand in the range $[1, 2)$. The dropped bit is kept by enlarging the accumulator, or is rounded away. Normalization is still required if more than two successive mul/div operations are performed.

B. Add/subtract accuracy

Given $q(p(x') + p(y'))$ where both x' and y' are the same sign (*i.e.*, not strict subtraction), the error is bounded by twice maximum $p(\cdot)$ error, plus maximum floating point addition error and maximum $q(\cdot)$ error. In practice the worst case error is hard to determine without exhaustive search. Limiting ourselves to values in a limited range of $[1, 2)$, we evaluate log domain FLMA addition of $x + y$ for all $x \in [1, 2)$ and a choice of 64 random $y \in [1, 2)$ versus α, β in Figure 3 (2^{29} unique pairs per configuration). All $\alpha > 1, \beta \geq 1$ have max log ulp error ≤ 1, and $\alpha = \beta \leq 1$ has max log ulp error ≤ 2. With increased α, β there are exponentially fewer incorrectly rounded sums but the table maker's dilemma is a limiting factor. At $\alpha = \beta = 16$, about 0.0005% of these sums remain incorrectly rounded to max 0.5 log ulp.

For subtraction, catastrophic cancellation (a motivation for LNS co-transformation) still realizes itself. As with LNS, there is also a means of correction. While the issue appears with pairs of values very close in magnitude, consider linear domain $x = 1, y = 1 - \epsilon$, and evaluate $x - y$ with FLMA subtraction:

$$x' = +2^0 \times e^{\mathtt{b0.00000000000000000000000}}$$
$$y' = +2^{-1} \times e^{\mathtt{b0.10110001011100100001011}}$$

The base-e exponent of y' here is 1 ulp below $\ln(2)$ rounded to $F = 23$, and is thus our next lowest representable value from 1. With FLMA subtraction at $\alpha = 1$:

$$p(x') = +2^0 \times \mathtt{b1.00000000000000000000000}$$
$$p(y') = +2^{-1} \times \mathtt{b1.11111111111111111111111}$$
$$p(x') - p(y') = +2^{-23} \times \mathtt{b1.00000000000000000000000}$$
$$\approx 1.1920929 \times 10^{-7}$$

Then back to log domain at $\beta = 1$:

$$q(p(x') - p(y')) = +2^{-23} \times e^{\mathtt{b0.00000000000000000000000}}$$
$$\approx 1.1920929 \times 10^{-7}$$

If the calculation were done to ≤ 0.5 log ulp error, we get:

$$q(p'(x') - p'(y')) = +2^{-24} \times e^{\mathtt{b0.10101101010100101000101}}$$
$$\approx 1.1730463 \times 10^{-7}$$

or an absolute error between the two of $\approx 1.905 \times 10^{-9}$, but off by 135,111 log ulp (distance from the $F = 23$ bit rounding of $\ln(2)$). In floating point, the rounded result would have error ≤ 0.5 ulp. However, as (almost) all of our log domain values have a linear domain infinite fractional expansion, in near cancellation with a limited number of bits, FLMA misses the extended expansion of the subtraction residual.

If reducing relative error is a concern, we can increase α for the log-to-linear conversion. This will provide more

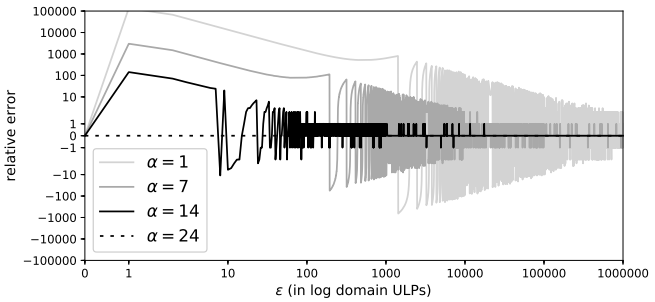

Fig. 4. log32 FLMA catastrophic cancellation: relative (log ulp) error of $1 - (1 - \epsilon)$ in log domain, as a function of α ($\beta = 1$ throughout).

of the linear domain infinite fractional expansion, reducing relative error to ≤ 0.5 log ulp almost everywhere if necessary (Figure 4). Absolute error remains bounded throughout the cancellation regime, from $< 10^{-8}$ at $\alpha = 1$ to $< 10^{-10}$ at $\alpha = 14$. We are not increasing the log precision F, but increasing the distinction in the linear domain between 1 and $1 - \epsilon$. $q(y)$ can maintain a reduced β, with any remainder $A - (F + \beta)$ accumulator bits rounded off.

C. Accuracy test via least squares QR solution

For a quick end-to-end accuracy test (encompassing sums, sums of products, multiplication, division and square root), we consider a least squares solution of x in $Ax = b$ given various reference vectors b and ill-conditioned reference matrices A. We control the condition number $\kappa(A)$ by generating random symmetric 64 $\times$ 64 matrices M with entries $\sim U(-2, 2)$, taking a SVD decomposition $M = U\Sigma V$, scaling the largest to smallest singular value ratio by κ to produce Σ_κ, then producing a reference $A = U\Sigma_\kappa V$ and b ($b_i \sim U(0, 1)$) in float64 arithmetic. A, b are rounded to A', b' and QR factorization $A' = Q'R'$ is performed via the Householder algorithm in the arithmetic under analysis, and x is recovered by backsolving $R'x = {Q'}^T b'$ [38]. A reference x_r from float64 A, b is similarly calculated via MATLAB VPA to 64 decimal digits, and we present the error $\|x - x_r\|_2$. Figure 5 shows this error across a sweep of condition number $\kappa(A)$ by powers of 10, with 5 trials for each condition number. Note that log32 FLMA remains roughly even against float32 (sometimes superior, sometimes inferior) despite the approximate nature of the design, even at high condition number $\kappa(A) = 10^{10}$.

VIII. Arithmetic synthesis

We compare 7 nm area, latency and energy against IEEE 754 floating point without subnormal support. A throughput of T refers to a module accepting a new operation every T clock cycles ($T = 1$ is fully pipelined), while latency of L is cycles to first result or pipeline length. Table II shows basic arithmetic operations with FLMA parameters the same as Section VII with $\alpha = \beta = 1$. Note that the general LNS pattern of multiply energy being significantly lower but add/sub significantly higher still holds. Add/sub are two-operand, so this implementation includes two $p(\cdot)$ and one $q(\cdot)$ converters, and none will be actively gated in a fully utilized pipeline (they are all constantly switching). Naive sum of multiply with add energy lead to higher results as compared to floating point. However, as mentioned earlier, it is easier to efficiently pipeline FLMA add/sub compared to LNS add/sub.

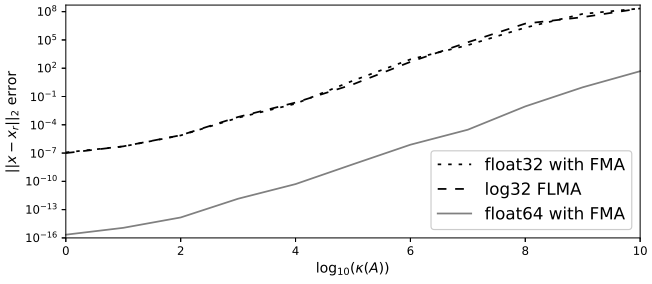

Fig. 5. Least squares $Ax = b$ solution average error via Householder QR, 64 $\times$ 64 matrices with condition number $\kappa(A)$ from 1 to 10^{10}.

TABLE II
Fully pipelined ($T = 1$) arithmetic synthesis

Type	**Latency**	**Area** μm^2	**Energy/op** pJ
float32 add/sub	1	138.4	0.274
log32 FLMA add/sub	**7**	**1577 (11.4×)**	**1.768 (6.45×)**
float32 mul	1	248.4	0.802
log32 FLMA mul	**1**	**40.2 (0.16×)**	**0.080 (0.10×)**
float32 FMA	1	481.2	1.443
log32 mul-add core $p(x' + y') + a$, no $q(\cdot)$	**3**	**706.5 (1.47×)**	**0.586 (0.41×)**

TABLE III
$N = 128$ inner product multiply-add synthesis results

Type	**Throughput**	**Area** μm^2	**Energy/op** pJ
float32 FMA	130	591.0	1.542
log32 FLMA $q(\sum_{i=1}^{128} p(x'_i + y'_i))$	**135**	**1271 (2.15×)**	**0.668 (0.43×)**
float64 FMA	131	1787.3	5.032
log64 FLMA $q(\sum_{i=1}^{128} p(x'_i + y'_i))$	**144**	**6651 (3.72×)**	**1.104 (0.22×)**

The situation changes when we consider a multiply-accumulate, perhaps the most important primitive for linear algebra. Table III shows FLMA modules for 128-dim vector inner product with a $T = 1$ inner loop, comparing against floating point FMA. The float64 comparison is against FLMA $E = 11$, $F = 52$, exp/log $\alpha = \beta = 1$, $p = \ell = 59$, $I = 29$, exp $r = 2$, log $r = 3, s = 9$, accumulator $A = 53$, called log64 FLMA. The benefit of the FLMA design can be seen in this case; log domain multiplication, $p(\cdot)$ conversion and floating point add is much lower energy than a floating point FMA. As with LNS or FLMA addition, a single multiply-add with a log domain result would be inefficient, but in running sum cases (multiply-accumulate), the $q(\cdot)$ overhead is deferred and amortized over all work, and this conversion (unlike the inner loop) need not be fully pipelined. Using a combinational MCP for this $q(\cdot)$ with data gating when inactive saves power and area, at the computational cost of 2 additional cycles for throughput. Increased accumulator precision (A independent of α, β) is also possible at minimal computational cost, as this only affects the floating point adder.

IX. Conclusion

Modern applications of computer vision, graphics (Figure 6) and machine learning often need energy efficient high precision arithmetic. We present an novel dual-base logarithmic arithmetic applicable for linear algebra kernels found in these applications. This is built on efficient implementations of e^x and $\ln(x)$, useful in their own right, leveraging numerical integration with truncated mul/div. While the arithmetic is approximate and without strong relative error guarantees unlike LNS or floating point arithmetic, it is extendible to arbitrary precision, easily pipelinable and retains moderate to low relative error and low absolute error. The area/power

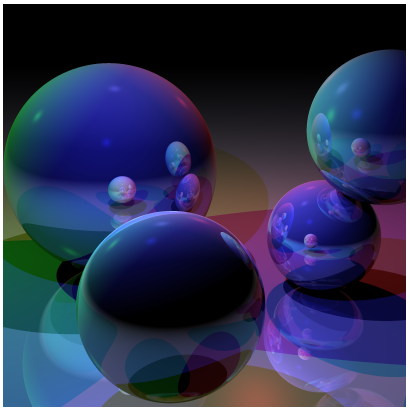

Fig. 6. 2048×2048 raytracing done entirely in dual base FLMA arithmetic (Section VII parameters with $\alpha = \beta = 1$). Pixel $\{\pm 2^a e^b$ or $0\}$ values clamped and rounded to nearest even integers for RGB output.

tradeoff is certainly not appropriate for many designs, but can provide a useful alternative to high precision floating or fixed point arithmetic when aggressive quantization is impractical.

ACKNOWLEDGMENTS We thank Synopsys for their permission to publish results on our research obtained by using their tools with a popular 7 nm semiconductor technology node.

REFERENCES

[1] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proc. of the Fifteenth Int. Conf. on Architectural Support for Prog. Languages and Operating Syst.*, ser. ASPLOS XV, 2010, pp. 205–218.

[2] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *Design Automation Conf. (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 2012, pp. 1131–1136.

[3] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *Solid-State Circuits Conf. Digest of Tech. Papers (ISSCC), 2014 IEEE Int.* IEEE, 2014, pp. 10–14.

[4] M. Wolfe, "More iteration space tiling," in *Proc. of the 1989 ACM/IEEE Conf. on Supercomputing*, ser. Supercomputing '89, 1989, pp. 655–664.

[5] M. Nickel and D. Kiela, "Poincaré embeddings for learning hierarchical representations," in *Advances in Neural Inf. Process. Syst.*, 2017, pp. 6338–6347.

[6] C. Tomasi and T. Kanade, "Shape and motion from image streams under orthography: A factorization method," *Int. J. Comput. Vision*, vol. 9, no. 2, pp. 137–154, Nov. 1992.

[7] E. E. Swartzlander and A. G. Alexopoulos, "The sign/logarithm number system," *IEEE Trans. Comput.*, vol. 100, no. 12, pp. 1238–1242, 1975.

[8] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Tichy, Z. Pohl, A. Hermanek, and N. F. Benschop, "The european logarithmic microprocesor," *IEEE Trans. Comput.*, vol. 57, no. 4, pp. 532–546, April 2008.

[9] Chichyang Chen, Rui-Lin Chen, and Chih-Huan Yang, "Pipelined computation of very large word-length lns addition/subtraction with polynomial hardware cost," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 716–726, July 2000.

[10] D. M. Lewis, "Interleaved memory function interpolators with application to an accurate lns arithmetic unit," *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 974–982, Aug 1994.

[11] Y. Popoff, F. Scheidegger, M. Schaffner, M. Gautschi, F. K. Gürkaynak, and L. Benini, "High-efficiency logarithmic number unit design based on an improved cotransformation scheme," in *Proc. of the 2016 Conf. on Design, Automation & Test in Europe*, ser. DATE '16, 2016, pp. 1387–1392.

[12] J. Johnson, "Rethinking floating point for deep learning," *NeurIPS Workshop on Syst. for ML*, vol. abs/1811.01721, 2018.

[13] J.-M. Muller, "Discrete basis and computation of elementary functions," *IEEE Trans. Comput.*, vol. 34, no. 9, pp. 857–862, Sep. 1985.

[14] N. Revol and J.-C. Yakoubsohn, "Accelerated shift-and-add algorithms," *Reliable Computing*, vol. 6, no. 2, pp. 193–205, May 2000.

[15] J. Volder, "The cordic computing technique," in *Papers Presented at the the March 3-5, 1959, Western Joint Comput. Conf.*, ser. IRE-AIEE-ACM '59 (Western), 1959, pp. 257–261.

[16] B. G. De Lugish, "A class of algorithms for automatic evaluation of certain elementary functions in a binary computer," Ph.D. dissertation, Champaign, IL, USA, 1970, aAI7105082.

[17] J. Duprat and J. M. Muller, "The cordic algorithm: New results for fast vlsi implementation," *IEEE Trans. Comput.*, vol. 42, no. 2, pp. 168–178, Feb. 1993.

[18] M. J. Schulte and E. E. Swartzlander, Jr., "Hardware designs for exactly rounded elementary functions," *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 964–973, Aug. 1994.

[19] K. H. Abed and R. E. Siferd, "Cmos vlsi implementation of a low-power logarithmic converter," *IEEE Trans. Comput.*, vol. 52, no. 11, pp. 1421–1433, Nov. 2003.

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

[21] J. A. Piñeiro, M. D. Ercegovac, and J. D. Bruguera, "High-radix logarithm with selection by rounding: Algorithm and implementation," *J. VLSI Signal Process. Syst.*, vol. 40, no. 1, pp. 109–123, May 2005.

[22] J.-C. Bajard, S. Kla, and J.-M. Muller, "Bkm: a new hardware algorithm for complex elementary functions," *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 955–963, Aug 1994.

[23] P. W. Baker, "Parallel multiplicative algorithms for some elementary functions," *IEEE Trans. Comput.*, vol. 24, no. 3, pp. 322–325, Mar. 1975.

[24] M. D. Ercegovac, T. Lang, and P. Montuschi, "Very high radix division with selection by rounding and prescaling," in *Proc. of IEEE 11th Symp. on Comput. Arithmetic*, June 1993, pp. 112–119.

[25] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *IEEE J. Solid-State Circuits*, vol. 31, no. 9, pp. 1277–1284, Sep. 1996.

[26] M. Vratonjic, B. R. Zeydel, and V. G. Oklobdzija, "Low- and ultra low-power arithmetic units: design and comparison," in *2005 Int. Conf. on Comput. Design*, Oct 2005, pp. 249–252.

[27] H. Jiang, C. Liu, N. Maheshwari, F. Lombardi, and J. Han, "A comparative evaluation of approximate multipliers," in *2016 IEEE/ACM Int. Symp. on Nanoscale Archit. (NANOARCH)*, July 2016, pp. 191–196.

[28] V. Lefevre, J.-M. Muller, and A. Tisserand, "Toward correctly rounded transcendentals," *IEEE Trans. Comput.*, vol. 47, no. 11, pp. 1235–1243, Nov 1998.

[29] J. N. Coleman and R. Che Ismail, "Lns with co-transformation competes with floating-point," *IEEE Trans. Comput.*, vol. 65, no. 1, pp. 136–146, Jan 2016.

[30] R. C. Ismail and J. N. Coleman, "Rom-less lns," in *Proc. of IEEE 20th Symp. on Comput. Arithmetic*, July 2011, pp. 43–51.

[31] F. Taylor, "An extended precision logarithmic number system," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 31, no. 1, pp. 232–234, February 1983.

[32] M. G. Arnold, "A vliw architecture for logarithmic arithmetic," in *Euromicro Symp. on Digit. Syst. Design, 2003. Proc..*, Sep. 2003, pp. 294–302.

[33] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of the IEEE Conf. on Comput. Vision and Pattern Recognition*, 2016, pp. 770–778.

[35] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *Int. J. of Comput. Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[36] F.-S. Lai and C.-F. E. Wu, "A hybrid number system processor with geometric and complex arithmetic capabilities," *IEEE Trans. Comput.*, vol. 40, no. 8, pp. 952–962, Aug 1991.

[37] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, "Theory and applications of the double-base number system," *IEEE Trans. Comput.*, vol. 48, no. 10, pp. 1098–1106, Oct 1999.

[38] G. H. Golub and C. F. van Loan, *Matrix Computations*, 4th ed. JHU Press.