A Two-Stage Operand Trimming Approximate Logarithmic Multiplier

Ratko Pilipovic®, Member, IEEE, Patricio Bulić®, Member, IEEE, and Uroš Lotrič®

Abstract—We present an approximate logarithmic multiplier with two-stage operand trimming, which prioritises area and energy consumption while retaining acceptable accuracy. The multiplier trims the least significant parts of input operands in the first stage and the mantissas of the obtained operands’ approximations in the second stage. We evaluated the multiplier’s efficiency in terms of error, energy, and area utilisation using NanGate 45nm. The experimental results show that the proposed multiplier exhibits smaller area utilisation and energy consumption than the state-of-the-art designs and that it behaves well in image processing and image classification with convolutional neural networks.

Index Terms—Logarithmic multiplier, approximate computing, arithmetic circuit design, Mitchell’s multiplier, energy-efficient processing.

I. INTRODUCTION

POWER consumption has become a considerable challenge and a serious obstacle when increasing computing performance. It is possible to obtain a significant performance gain through the shrinking of the CMOS transistor channel length. As the channel length is reduced, the power per switching event decreases, which allows for faster switching. At the same time, the number of transistors per chip increases, which in turn increases the total chip power. A possible way to overcome this obstacle is to design devices and algorithms that require fewer transistors, even at the price of accuracy.

Many applications, e.g., image processing, can produce an inaccurate output relying on the limited human perception. Some other applications, including adaptive filtering and machine learning, can refine their results iteratively, exhibiting inherent tolerance for a small computation error. Moreover, many signal processing and machine learning applications deal with input data distorted by the noise caused by sensors and quantization processes, setting a limit in the precision or accuracy. Pursuing meaningless precision in any of the above cases leads to excessive energy consumption. Therefore, we may introduce a small error in computation that would not impose noticeable degradation of the output results.

Hence, approximate computing has emerged as a new and promising paradigm for high-performance and energy-efficient systems. In approximate computing, a small and acceptable error may be induced in computing at all layers to achieve more energy efficient processing. For example, various approximate arithmetic circuits have been designed to save chip area and energy by generating inaccurate but acceptable results [1]–[6].

This paper focuses on approximate multipliers, as multiplication represents a ubiquitous arithmetic operation found in various applications. For example, approximate unsigned multipliers have appeared in some image processing applications, like sharpening and smoothing, and video compression [7]–[10], while approximate signed multipliers have performed well in deep-learning accelerators [11]–[16].

We propose an energy-efficient approximate logarithmic multiplier with two-stage operand trimming that exhibits smaller energy consumption and area utilisation than the state-of-the-art designs. We show that we can deploy the proposed approximate multiplier in image processing and image classification applications without noticeable degradation in performance.

In the remainder of this paper, we first review the related work in the field of approximate multipliers. In section III, we describe the architecture of the proposed multiplier, which we further analyse in terms of the error characteristics and the synthesis results in section IV and section V, respectively. Section VI shows the applicability of the proposed design in image smoothing and convolutional neural networks applications. Lastly, we conclude the paper with the main findings.

II. RELATED WORK

Multipliers are complex circuits. To approximate multiplication, we usually convert it into some simpler operations. By using algorithmic simplifications, we can significantly improve multipliers’ energy performance. Approximate multipliers aim to achieve the best possible trade-off between accuracy and design efficiency [1], [2], [17]–[19]. The work [2] provides a comprehensive evaluation of recently proposed approximate arithmetic circuits, which are compared under different design constraints and applied to image processing and deep learning applications. Most of these multipliers follow one of the two major approaches: approximate logarithmic and approximate non-logarithmic multiplication.
As the name suggests, approximate logarithmic multipliers use the logarithmic product approximation, which replaces the multiplication with the addition of operands’ logarithms [12], [13], [15], [20]–[26]. Approximate non-logarithmic multipliers focus on simplifications of partial product addition [27]–[29], and partial product generation [7], [9], [30]–[33]. Approximate logarithmic multipliers deliver a more straightforward design but exhibit significantly higher computational error. At the same time, approximate non-logarithmic multipliers have a lower computational error with a price of higher design complexity [13].

In this chapter, we briefly review the approximate logarithmic and the approximate non-logarithmic multipliers.

A. Approximate Logarithmic Multipliers

Mitchell was the first to introduce unsigned logarithmic product approximation [20]. Mitchell’s algorithm employs the approximation of a binary number logarithm. It replaces multiplication with addition in the logarithm domain and serves as the basis of all logarithmic multipliers. Mitchell’s multiplier always underestimates the actual product, so its main drawback is a high computational error. Mahalingam et al. [21] improved Mitchell’s multiplier accuracy through operand decomposition. The iterative logarithmic multiplier (ILM), developed by Babić et al. [22], achieves arbitrary accuracy through an iterative procedure but still underestimates the actual product.

Liu et al. [23] proposed the unsigned ALM-SOA multiplier, which uses a truncated binary-logarithm converter and a set-one-adder (SOA) for the addition of logarithms. The set-one-adder with \( k \) approximation bits (SOA\(^k\)) sets the \( k \) least significant bits to ‘1’. Hence, the actual sum of the logarithms is overestimated. As Mitchell’s multiplier always underestimates the actual product, SOA is employed to compensate the negative errors.

Kim et al. [12], [25] truncated the logarithm representation of operands to deliver more efficient logic and proposed an iterative error correction procedure. The authors used the one’s complement as an approximation of the two’s complement to handle negative numbers, as was previously proposed in [11]. Their signed Mitchell–trunc\( k \)-C1 multiplier keeps only \( k \) upper bits of mantissa in the logarithmic representation of input operands. Due to mantissa truncation, Kim et al. achieved efficient logarithm and antilogarithm conversions, but at the same time, mantissa truncation causes a large error. Their two-stage design uses two truncated logarithmic multipliers for error correction.

Ansari et al. [15], [24] proposed two improved unsigned logarithmic multipliers (ILM), with the exact (ILM-EA) and approximate adder (ILM-AA) in the antilogarithm step. They introduced two novelties. Firstly, they use a near-one-detector (NOD) to round both operands to their nearest powers of two. As the output of the NOD uses a one-hot representation and some entries in the truth table of a conventional adder cannot occur, the authors proposed a compact adder for the reduced truth table. Secondly, they used the modified SOA\(^k\) adder. In the modified SOA\(^k\), instead of setting all of the \( k \) least significant bits to ‘1’, these bits are set alternately to ‘1’ and ‘0’, which leads to a double-sided error distribution – one of its major benefits. The proposed ILM-AA is more accurate and has the smallest error values compared to other unsigned logarithmic designs in the literature.

Yin et al. [26] proposed unsigned and signed designs of a dynamic range approximate logarithmic multiplier (DR-ALM). Mitchell’s multiplier product is always smaller than its exact counterpart, the DR-ALM dynamically truncates input operands and sets the least significant bit of the truncated operand to ‘1’ to compensate for negative errors. DR-ALM uses smaller bit-width logarithmic converters, adder, and antilogarithmic converter to generate the product due to the previous truncation of operands.

Pilipović et al. [13] proposed the LOBO approximate multiplier that uses the radix-4 Booth encoding to compute the higher part of the product and the logarithmic approximation to generate the least significant part of the product.

B. Approximate Non-Logarithmic Multipliers

The Booth algorithm is commonly used to design approximate multipliers where various approaches have been proposed to simplify the partial product generation stage.

Jiang et al. [30] proposed an approximate radix-8 Booth encoding multiplier (ABM) using the approximate recoding adder with and without the truncation of several less significant bits in the partial product. The approximate recoding adder is used to calculate the triples of multiplicands in a Wallace tree. It adds seven upper bits exactly, whereas the nine lower bits are obtained approximately with 3-input XOR gates. Among all ABM multipliers, the one with 15-bit truncation achieves the best overall performance in terms of hardware and accuracy.

Liu et al. [7] designed approximate Booth multipliers based on approximate radix-4 modified Booth encoding (R4ABM-\( k \)) algorithms and a regular partial product array that employs an approximate Wallace tree. The main idea is to generate lower bits of partial products with approximate radix-4 encoding, while the upper bits are generated exactly. Radix-4 Booth approximation is performed through a modified Karnaugh logic table for exact radix-4 Booth encoding. Two approximate radix-4 Booth encoders are proposed to speed-up the partial product generation to generate \( k \) least significant partial product bits. By changing the value of \( k \), the R4ABM-\( k \) multiplier can achieve different tradeoffs between accuracy and hardware efficiency.

Approximate hybrid high radix encoding multipliers (RAD2\(^ k \)) are presented in [9]. The proposed approach aims to overcome the limitations of high radix Booth encoding through the omission of hard multiplies. The multiplier employs a hybrid encoding technique, where the \( n \)-bit input operand is divided into two groups: the upper part of \( n - k \) bits and the lower part of \( k \) bits. The configuration parameter, \( k \geq 4 \), is an even number. The upper part is exactly encoded using the radix-4 encoding, while the lower part is approximately encoded with the radix-2\(^ k \) encoding. The approximations are performed by rounding the radix-2\(^ k \) values to their nearest power of two. Although this design is highly accurate, it exhibits a large error for small numbers. The authors showed
that RAD1024 delivers the best compromise between energy reduction and accuracy.

A hybrid low radix encoding-based approximate Booth multiplier (HLR-BM), proposed in [31], addresses the issue of generating odd multiples (i.e., multiples of the ±3 multiplicand) in the radix-8 Booth encoding. HLR-BM approximates the ±3 multiplicands to their nearest power of two, such that the errors complement each other. Similar to RAD2 \(^k\) [9], the authors employed hybrid encoding in which the least significant bits of multiplicand are encoded with the approximate radix-8 encoding. Due to smaller radix approximate encoding, HLR-BM achieves higher accuracy than RAD2 \(^k\), but offers smaller energy and area gains.

In [32], the authors proposed hybrid partial product-based building blocks by considering the probability distribution of the input operands. While [31] concentrated on simplification of Booth encoding, here, the authors focused on decomposing an 8-bit multiplier into 4-bit approximate multipliers. An efficient hardware implementation of approximate 4-bit multipliers uses the high-performance approximate NOR-based half adder and full adder cells. The authors used the proposed recursive partitioning to implement 8-bit multipliers (Ax8). Among the three different strategies (Ax8-1/2/3), Ax8-3 exhibits the smallest error and energy.

III. THE PROPOSED MULTIPLIER

In fixed-point computation, we must carefully select the numbers’ bit-width, as it directly determines the range of values we can represent. We can obtain a very rough estimate of a number’s value from its leading-one bit. The more bits we consider after the leading-one bit, the more accurate is the estimated number’s value and the more complex becomes the arithmetic circuitry. Many applications may deliver an acceptable result even if we use only the leading-one bit and a few bits that follow.

The two-stage operand trimming logarithmic multiplier (TL) exploits this fact and splits the operands into two parts. The multiplier then uses only one part of an operand in the following way: if the upper part contains at least one non-zero bit, then the upper part enters the multiplier, otherwise, the lower part enters the multiplier. Thus, after splitting the operands, the multiplier operates on the reduced number of bits, leading to a smaller design. However, this approach requires additional logic at the multiplier’s input to select the operands’ important parts and at the multiplier’s output to form the product correctly.

A. Background

The decimal value of a signed \(z\)-bit number \(Z\) in two’s complement is

\[
Z = -b_{Z,-1}2^{-1} + \sum_{i=0}^{z-2} b_{Z,i}2^i,
\]

where \(b_{Z,i}\) represents the \(i\)-th bit of \(Z\). To convert the binary number into its logarithm, we separate the sign \(s_Z = \text{sign}(Z)\) and the absolute value

\[
|Z| = 2^{k_Z} + \sum_{i=0}^{k_Z-1} b_{|Z|i}2^i = 2^{k_Z} (1 + m_Z),
\]

where the exponent \(k_Z = [\log_2 |Z|]\) denotes the position of the leading one bit and

\[
m_Z = 2^{-k_Z} \sum_{i=0}^{k_Z-1} b_{|Z|i}2^i,
\]

represents the mantissa. According to [20], we approximate the logarithm of \(|Z|\) as

\[
\log_2 |Z| = k_Z + \log_2 (1 + m_Z) \approx k_Z + m_Z.
\]

B. Logarithm Conversion of Operands

To get the final approximation \(P_0\) of the exact product \(P_0 = X_0 \cdot Y_0\), we start by computing the approximate logarithms of the \(n\)-bit signed operands \(X_0\) and \(Y_0\). Conversion of both operands follows the same steps, so we limit the detailed description to operand \(X_0\) only. Fig. 1 illustrates the circuitry used to obtain operand’s sign from the most significant bit and approximate its absolute value \(|X_0|\) by one’s complement. In the schemes, a thick line with designated bit-width represents a bus, while a thin line represents a single wire. To get \(|X_0|\), we first take the most significant bit, i.e., the sign bit \(s_{X_0} = b_{X_0,n-1}\), from the bus \(X_0\). Then we drive \(n\) instances of the sign bit along with \(X_0\) into XOR. In the first operand trimming stage, we split the absolute value \(|X_0|\) into two parts: the \(v\)-bit upper part and the \((n-v)\)-bit lower part, where \(v \geq n/2\). We use the OR-reduction

\[
u_{X_0} = 1 + \sum_{i=0}^{n-1} b_{|X_0|,i} = \begin{cases} 
1, & |X_0| \geq 2^{n-v} \\
0, & \text{otherwise}
\end{cases}
\]

to detect if there is any non-zero bit in the upper part. If at least one non-zero bit exists, we approximate the absolute value with its \(v\)-bit upper part, otherwise, we approximate it with its \((n-v)\)-bit lower part. In the latter case, all upper-part bits are zero, so we can drive the \(v\) lower bits to the multiplexer. As the most significant bit of the absolute value is always ‘0’ due to the previous XOR with the same bit, we use wire routing to shift left the upper part by one place and set its least significant bit, thus getting the mean value approximation of the neglected bits. We illustrate this operation by the fork and the join patterns: from the \(v\) upper bits, we take out the most significant bit \(b_{|X_0|,n-1}\) at the fork, and drive \(v-1\) remaining bits to the join, where we attach ‘1’ at the least
bit-wise AND between the shifted leading-one bit and \( w - i \) of the trimmed mantissa leading-one bit. The \( w \)-th bit then enters the \( w \)-bit priority encoder to obtain the \( \lfloor \log_2 v \rfloor \)-bit exponent \( k_X \). The right side of the circuitry extracts the mantissa \( m^i_X \).

In the second operand trimming stage, we propose a new mantissa extractor. To further reduce the overall complexity of the binary-to-logarithm conversion circuitry, we propose to keep only \( w < v \) leftmost bits in the mantissa. In the following text, we refer to the obtained mantissa as trimmed mantissa \( m^i_X \). As \( w \) is a constant value, we can replace the costly barrel shifter with a simple AND-OR net. Fig. 3 illustrates the extraction of the \( i \)-th bit of the mantissa. In the first step, we shift the leading-one bit \( w - i \) places right. For the \( i \)-th bit, the value \( w - i \) is constant; therefore, we can implement the shift as a simple wire routing. In the second step, we perform bit-wise AND between the shifted leading-one bit and \( X \). The output of the barrel shifter is a \( (2w - i) \)-bit number. As the approximations \( m^i_X \) and \( X \) are mutually shifted by \( 2^k \), which we drive to the AND, we are mutually shifted by \( w - i \).

Employment of mantissa extractor reduces delay in the logarithm conversion circuitry. The critical path (i.e. the path with the maximal delay) passes either through mantissa extractor or priority encoder, while in previous designs (e.g. [12], [23]) the critical path includes the priority encoder and the barrel shifter.

**C. Summation and Antilogarithm Conversion**

The approximate product’s logarithm is the sum of approximate logarithms of both input operands

\[
k_p + m_p = k_X + m^i_X + k_Y + m^i_Y,
\]

where \([1 + \log_2 v] \) most significant bits represent the integer exponent \( k_p \), and the \( w \) least significant bits form the fractional mantissa \( m_p \).

To obtain the approximate intermediate product

\[
\tilde{P} = (1 + m_p) \cdot 2^{k_p},
\]

we use the circuitry depicted in Fig. 5. Due to the trimming, the adder and the barrel shifter have reduced bit-width manifesting in small size and delay. We first form operands’ approximate logarithms by concatenating ‘0’ with its exponent and mantissa. The approximate logarithms enter the \([1 + \log_2 v + w] \)-bit adder to get the approximate logarithm of the intermediate product with the \([1 + \log_2 v] \)-bit exponent \( k_p \) and the \( w \)-bit mantissa \( m_p \). A number, formed by concatenating ‘1’ and the mantissa \( m_p \), enters the barrel shifter that shifts it left by \( k_p \) bits. The output of the barrel shifter is a \( (2w + v) \)-bit number. As the approximations \( X \) and \( Y \) are \( v \)-bit numbers, the output’s upper \( 2w \) bits represent the approximate intermediate product \( \tilde{P} \).

Finally, we form the final approximation

\[
\tilde{P}_0 = s_{X_0} s_{Y_0} 2^{(n - u - 1)(u_{X_0} + u_{Y_0})} \cdot \tilde{P},
\]

where we shift left the approximate intermediate product \( \tilde{P} \) to compensate for the shift of an operand \( n - u - 1 \) places right in the first trimming phase (6). Recall that \( u_{X_0} \) and \( u_{Y_0} \) (5) are bits that identify whether the upper part of input operands is used in further processing. The circuitry in Fig. 6 uses a multiplexer to select the appropriately shifted \( \tilde{P} \) based on the bits \( u_{X_0} \) and \( u_{Y_0} \). If we take both lower parts of the input operands, then \( \tilde{P} \) is already the absolute value of the
remove the sign conversion circuitry and drive the upper part and one upper part of the input operands, we shift the approximate intermediate product \((n - v - 1)\) bits to the left (multiplexer input 01). And finally, if we take both upper parts of the input operands, we shift the approximate intermediate product \(2(n - v - 1)\) bits to the left (multiplexer input 10). Note that these shifts are just simple wire routings. As the final approximate product is \(2^n\), we also append the required number of leading zeros. We calculate the sign of the final product approximation as an XOR operation between the operands’ sign bits. Again, we approximate the sign conversion using one’s complement. In the rest of the paper, we denote the two-stage operand trimming logarithmic multiplier as TL\(n - v/w\).

With minor modifications of the proposed design, we could also implement an unsigned multiplier. To do so, we have to remove the sign conversion circuitry and drive the upper \(v\)-bits of \(|X_0|\) directly to the multiplexer in the input stage (Fig. 1), and also remove the sign conversion circuitry from the output stage (Fig. 6).

IV. ERROR ANALYSIS

The error analysis of the TL\(n - v/w\) multiplier includes an error study and an empirical assessment of the error characteristics.

A. Error Study

The TL\(n - v/w\) multiplier introduces errors in the logarithm and antilogarithm conversion steps. Fig. 7 shows that the highest error in the logarithm conversion emerges when the input operand is \(2^n\), from where it exponentially decreases. Mantissa trimming introduces small errors from \(2^w\) onwards.

Firstly, we limit our analysis to the approximate products of unsigned operands \(X_0 = 2^kX\) and \(Y_0 = 2^kY\), where \(kX, kY \in [0, n - 1]\). These values are multiples of \(2^n\), hence having the largest relative approximation errors (Fig. 7). The operand approximation of \(X_0 = 2^kX\),

\[
\bar{X}_0 = 2^kX (1 + r_{kX} + 2^{-w}),
\]

with

\[
r_{kX} = \begin{cases} 
-2^{-kX+n-v-1}, & n-v \leq kX < n-v+w-1 \\
0, & \text{otherwise,}
\end{cases}
\]

comes with two exponential terms, originating from the mean value approximation of the trimmed bits. The antilogarithm conversion approximates the product \(P_0(X_0, Y_0) = 2^{kX+kY}\) as

\[
\tilde{P}_0(X_0, Y_0) = \begin{cases} 
2^{kX+kY}(2^{2^{-w}}), & kX = kY = n-v \\
2^{kX+kY}(1+rkX+rkY+2^{1-w}), & \text{otherwise.}
\end{cases}
\]

The error distance \(|P_0(X_0, Y_0) - \tilde{P}_0(X_0, Y_0)|\) increases with \(kX\) and \(kY\). Fig. 8 shows that the relative error distance \(\text{RED}(X_0, Y_0) = |P_0(X_0, Y_0) - \tilde{P}_0(X_0, Y_0)|/|P_0(X_0, Y_0)|\) is large for operands with non-zero \(rk_x\) or \(rk_y\).

Assuming that operands with non-zero \(rk_x\) or \(rk_y\) lead to products with a large relative error, we can estimate the pessimistic theoretical lower bound of the portion of the products with the relative error distance below \(2^{1-w}\),

\[
P_{\text{RED}<2^{1-w}} > 1 - \frac{N_r}{2^{n-1}}, \quad N_r = (2^{w-1})2^{n-w},
\]

(13)
where the nominator represents products with non-zero \( r_kx \) or \( r_ky \), and the denominator represents the number of all possible products.

### B. An Empirical Assessment of Error Characteristic

To empirically assess the error, we examine the influence of parameters \( v \) and \( w \) for all pairs of operands’ values in the interval \([-2^{n-1}, 2^{n-1} - 1]\) in terms of

- NMED, the mean error distance, normalised to the maximal product [7], [23],
- MRED, the mean relative error distance,
- \( P_{\text{RED}} < 11\% \), the portion of products with relative error distance smaller than 11%, as it represents the maximal error of the Mitchell logarithm product approximation [20].

Table I shows error metrics for different instances of TL multipliers. In the 16-bit multipliers, the parameter \( v \) has a significant influence on MRED and \( P_{\text{RED}} < 11\% \) but does not affect NMED, which is mainly affected by the products of large operands. The instances with wider mantissa provide more accurate product approximations. A large drop in NMED from \( w = 3 \) to \( w = 4 \) indicates that we should prefer only multipliers with \( w \geq 4 \). The smaller 8-bit multipliers have significantly larger errors NMED and MRED and lower \( P_{\text{RED}} < 11\% \).

![Fig. 8. The relative error distance of products \( 2^kx \) and \( 2^ky \) in the TL16-8/4 multiplier.](image)

![Fig. 9. The portion of products with the relative error distance smaller than \( p \) for the TL16-8/4 and TL8-4/4 multipliers.](image)

![Fig. 10. Error measures NMED and MRED as a function of the mantissa width \( w \) with \( n = 32, v = 16 \) (left) and as a function of the bit-width \( n \) with \( v = n/2, w = 4 \) (right).](image)

**Table I**

| Multiplier   | NMED [10^{-3}] | MRED [%] | \( P_{\text{RED}} < 11\% \) [%] |
|--------------|----------------|----------|-------------------------------|
| TL16-8/3     | 17.14          | 7.22     | 77.8                          |
| TL16-8/4     | 11.84          | 5.20     | 91.7                          |
| TL16-8/5     | 10.04          | 4.49     | 95.6                          |
| TL16-9/3     | 17.12          | 7.07     | 78.4                          |
| TL16-9/4     | 11.82          | 5.02     | 92.7                          |
| TL16-9/5     | 10.03          | 4.32     | 96.7                          |
| TL8-4/3      | 22.96          | 13.43    | 62.3                          |
| TL8-4/4      | 22.99          | 12.79    | 62.8                          |

V. SYNTHESIS RESULTS

We analyse the hardware performance of the proposed TL multipliers [34] in terms of power, area, delay, and
power-delay-product (PDP) and compare them with several state-of-the-art approximate multipliers. In addition to hardware metrics, we compare NMED for all evaluated multipliers. The selected multipliers were implemented in Verilog and synthesised to 45nm Nangate Open Cell Library. The timing constraints, used for all evaluated designs, specify clock-related parameters, which affect synthesis and timing analysis. We set a clock signal with a period of 5ns, hence not violating a critical path. To evaluate the power, we used timing with a 10MHz virtual clock, a 5% signal toggle rate and output load capacitance equal to 10fF.

### A. 16-Bit Multipliers

Table II shows the synthesis results for 16-bit multipliers. The multipliers in [15], [20], [23] are all unsigned multipliers, while the proposed multiplier, and the multipliers in [7], [9], [12], [13], [26], [31] are signed. To compare all multipliers fairly, we have extended the unsigned multipliers to support signed operands in the same way as in the proposed design by adding the logic for operand absolute value computation and the logic for assigning a sign to the approximate product.

| Multiplier                  | Delay [ns] | Power [μW] | Area [μm²] | PDP [fJ] | NMED [10⁻³] |
|----------------------------|------------|------------|------------|----------|-------------|
| Exact radix-4              | 1.74       | 69.20      | 1,578.58   | 120.41   | 0           |
| TL16-8/3                   | 1.15       | 23.40      | 641.59     | 26.91    | 17.14       |
| TL16-8/4                   | 1.13       | 25.40      | 702.24     | 28.70    | 11.84       |
| TL16-8/5                   | 1.33       | 27.70      | 744.53     | 36.84    | 10.04       |
| TL16-9/4                   | 1.25       | 28.20      | 782.84     | 35.25    | 11.82       |
| TL16-9/5                   | 1.46       | 31.10      | 798.27     | 45.41    | 10.03       |
| Mitchell* [20]             | 1.63       | 35.40      | 916.64     | 57.70    | 9.27        |
| Mitchell-trunc8-C1 [12]    | 1.44       | 32.30      | 840.83     | 46.51    | 14.43       |
| Mitchell-trunc8-C1 [12]    | 1.43       | 35.00      | 910.25     | 50.05    | 10.56       |
| Mitchell-trunc8-6/1U [12]  | 1.41       | 32.80      | 841.36     | 46.25    | 6.93        |
| Mitchell-trunc8-6/1U [12]  | 1.49       | 37.70      | 969.84     | 56.17    | 6.85        |
| ALM-SOA10* [23]            | 1.58       | 37.40      | 954.67     | 59.09    | 8.35        |
| ALM-SOA11* [23]            | 1.47       | 36.70      | 952.01     | 53.95    | 8.06        |
| ALM-SOA12* [23]            | 1.48       | 33.50      | 874.08     | 49.58    | 8.73        |
| ALM-SOA13* [23]            | 1.33       | 30.30      | 813.69     | 40.30    | 13.18       |
| DR-ALM [26]                | 1.24       | 30.20      | 779.92     | 35.30    | 12.43       |
| DR-ALM [26]                | 1.47       | 37.70      | 831.78     | 42.80    | 5.27        |
| ILM-AA* [15]               | 1.51       | 27.80      | 780.18     | 41.98    | 7.20        |
| LOBO12-12/8 [13]           | 1.71       | 36.10      | 904.93     | 61.73    | 1.85        |

* converted to signed

### B. Multipliers of Other Bit-Widths

Table III shows the synthesis results for 8-bit multipliers. Again, we have extended the unsigned multipliers [12], [20], [23] to support signed operands. Although the TL multipliers TL8-4/2 and TL8-4/3 are the best in terms of hardware measures, all other multipliers fairly outperform them in terms of the error measure NMED. Considering the error and hardware measures, we selected the TL8-4/3 multiplier for the application studies.

### C. Discussion

Approximating the logarithm from the trimmed operand instead of the whole operand simplifies the logarithmic conversion circuitry in Fig. 2. When extracting the most significant bit’s position, the proposed multiplier uses a smaller v-bit leading-one detector and priority encoder circuits instead of smallest NMED but have a large energy consumption. On the other hand, the TL multipliers belong to the approximate logarithmic multipliers, prioritising efficient design over more accurate product approximation.
TABLE III
THE SYNTHESIS RESULTS AND NMED FOR 8-BIT MULTIPLIERS

| Multiplier          | Delay [ns] | Power [µW] | Area [µm²] | PDP [fJ] | NMED [10^-3] |
|---------------------|------------|------------|------------|----------|---------------|
| Exact radix-4       | 1.03       | 22.10      | 525.62     | 22.76    | 0             |
| TL8–4/2             | 0.88       | 11.80      | 325.05     | 10.03    | 32.3          |
| TL8–4/3             | 0.88       | 12.10      | 313.08     | 10.65    | 23.0          |
| TL8–4/4             | 0.91       | 13.50      | 359.37     | 12.29    | 23.0          |
| Mitchell* [20]      | 1.04       | 16.40      | 450.87     | 17.06    | 9.3           |
| Mitchell–trunc5–C1  | 1.04       | 14.00      | 360.70     | 14.28    | 17.8          |
| Mitchell–trunc6–C1  | 1.09       | 15.10      | 399.80     | 16.46    | 12.7          |
| Mitchell–trunc5–C1UB| 1.03       | 14.40      | 378.25     | 14.83    | 10.4          |
| Mitchell–trunc6–C1UB| 1.06       | 15.20      | 391.55     | 16.11    | 7.4           |
| ALM–SOA4* [23]      | 1.06       | 15.00      | 407.51     | 15.90    | 8.5           |
| ALM–SOA5* [23]      | 1.03       | 14.50      | 372.67     | 14.94    | 13.1          |
| ALM–SOA6* [23]      | 0.93       | 12.30      | 326.38     | 11.35    | 26.1          |
| DR–ALM3 [26]        | 0.91       | 12.00      | 317.07     | 10.92    | 20.7          |
| DR–ALM4 [26]        | 0.96       | 12.20      | 343.14     | 11.71    | 9.2           |
| Ax8_3* [32]         | 1.18       | 18.60      | 542.64     | 21.95    | 12.1          |

* converted to signed

Fig. 12. Hardware measures, PDP and area, as a function of the mantissa width \( w \) with \( n = 16, v = 8 \) (left) and as a function of the bit-width \( n \) with \( v = n/2, w = 4 \) (right).

larger \( n \)-bit circuits. Similarly, it extracts the mantissa from a \( v \)-bit trimmed operand, resulting in a simpler mantissa extractor. Considerable hardware savings are also in logarithm summation and antilogarithm conversion in Fig. 5. The proposed multiplier employs a simpler adder and a simpler barrel shifter due to a shorter mantissa. All these improvements considerably reduce the overall area and delay. However, to benefit from them, we need the operand trimming stage in Fig. 1 and the output stage in Fig. 6 that increase the multiplier’s overall complexity. As these stages mainly consist of simple multiplexers and wire routing, we can still profit from the design by adequately selecting the design parameters \( v \) and \( w \).

The superiority of the proposed multiplier’s hardware metrics emerges from its input stage. The operand’s logarithm approximation is determined based on two fixed ranges of the operand’s value, contrary to the dynamic range approach used in the DR-ALM multiplier [26]. Hence, the proposed method is more straightforward, but, with coarse-grain approximation, introduces larger errors.

VI. APPLICATION CASE STUDIES

We show the applicability of the TL16-8/4 multiplier and the TL8-4/3 multiplier in the image smoothing and the image classification with convolutional neural networks. In both applications, we compare the TL16-8/4 multiplier with the state-of-the-art approximate multipliers Mitchell–trunc6-C1, Mitchell–trunc8-C1, Mitchell–trunc6-C1UB, Mitchell–trunc8-C1UB [12], ILM-AA [15], ALM–SOA11 [23], DR–ALM4 [8], and RAD1024 [9] and the TL8-4/3 multiplier with the Mitchell–trunc5-C1, Mitchell–trunc5-C1UB [12], ALM–SOA4, ALM–SOA6 [23], DR–ALM3, DR–ALM4 [8] and Ax8_3 [32] multipliers.

A. Image Smoothing

Image smoothing reduces noise and details in the image [35] and is implemented by convolving the image with a smoothing kernel

\[
\frac{1}{256} \begin{bmatrix} 1 & 31 & 21 \\ 31 & 48 & 31 \\ 21 & 31 & 21 \end{bmatrix}
\]

(14)

To evaluate the influence of product approximation, we use the mean structural similarity index (MSSIM) [36] and the peak-signal-to-noise ratio (PSNR) between the image smoothed with the exact multiplier and the image smoothed with an approximate multiplier. We perform the tests on five 16-bit grayscale images from TESTIMAGES database [37]: building, cards, flowers, snail, and wood game. The pixels in an input image are uniformly shifted from \( [0, 2^{16}) \) to \( [-2^{15}, 2^{15}) \) to adapt for signed 16-bit multipliers. For signed 8-bit multipliers analysis, we additionally scale the images to 8-bit range.

Table IV shows the results for image smoothing. The TL16-8/4 multiplier delivers MSSIM and PSNR similar to the Mitchell–trunc8-C1 [12] multiplier, and better than the Mitchell–trunc6-C1 [12] multiplier and the DR–ALM4 [26] multiplier. However, multipliers ALM–SOA11 [23] and RAD1024 [9] with very low NMED outperform the proposed TL16-8/4 design. Similarly, the TL8-4/3 multiplier outperforms the DR–ALM4 [26] multiplier and the Ax8_3 [32] multiplier in terms of PSNR, but lags behind the Mitchell–trunc5-C1 [12] multiplier and the ALM–SOA11 [23] multiplier. Nevertheless, high MSSIM and acceptable PSNR indicate that the TL multipliers can replace the exact multiplier without a significant image quality decrease.

Fig. 13 visualizes Wood game image smoothing with 16-bit multipliers. The most obvious difference is in a continuous gradation of grey tone in the image background – more pronounced posterization (banding) can be observed in the images with lower MSSIM and PSNR.

B. Image Classification With Convolutional Neural Networks

Convolutional neural networks (CNNs) are a class of deep neural networks, mainly used in image classification and analysis [38]. The neural network processing involves a vast number of multiplications. We anticipate that the adaptable
nature of CNNs makes them resilient to errors introduced by approximate multiplication.

To assess the influence of approximate multiplication on the inference phase, we deploy approximate multipliers in a CNN for image classification. For experiments, we utilise the Caffe framework [39] and replace the exact multiplication with the approximate one in the inference phase. Notably, we replace the calls to the cuBLAS multiplication routines with our C/C++ routines that implement various approximate multiplier designs.

As a test case, we select the ResNet-20 [40] network and the CIFAR10 dataset [41]. We repeat the training ten times using fixed-point arithmetic, random weight initialisation, and the predetermined split to training and test set [41]. To boost the training, we preprocess the images by subtracting their mean value.

We initially train the network in floating-point arithmetic for 64,000 iterations using stochastic gradient descent with the learning rate decay. To adapt the network to approximate multiplication, we perform additional 15,000 iterations of training, employing the approximate fixed-point multiplication in the inference phase and the exact floating-point multiplication in the learning phase. We quantify the floating-point weights and inputs to the signed fixed-point representation with \( q \) fractional bits as \( Z \cdot 2^{-q} \), where \( Z \) is a floating-point value. We set \( q = 12 \) for the 16-bit multipliers and \( q = 6 \) for the 8-bit multipliers, which gives the smallest accuracy degradation for the exact radix-4 multiplier.

Table V presents the classification accuracy on a test set in terms of top-1 and top-3 scores for the ResNet-20 network.
predicted classes. For the top-1 score, the TL16-8/4 multiplier achieves lower classification accuracy than multipliers Mitchell–trunc6-C1 [12], ALM–SOA11 [23], and RAD1024 [9] but comparable accuracy to the DR–ALM4 multiplier [26]. In terms of a more relaxed top-3 score, the proposed TL16-8/4 multiplier is in line with the more accurate multipliers and outperforms the DR–ALM4 multiplier [26].

In general, the 8-bit multipliers perform worse than the 16-bit multipliers. The 8-bit multipliers are less accurate than the 16-bit multipliers regarding the top-1 score, while the difference drops in terms of the top-3 score. Performance of the TL8-4/3 multiplier follows the error analysis results – higher values of error measures correlate with lower classification accuracy. Hence, among the tested 8-bit multipliers, the DR-ALM4 multiplier is probably a better choice than the TL8-4/3 multiplier, as it achieves better accuracy with slightly worse hardware measures.

VII. CONCLUSION

This paper presents an energy-efficient approximate logarithmic multiplier with two-stage operand trimming, which aims to deliver improvements in the area and energy consumption of approximate logarithmic multipliers.

The multiplier splits the input operands into the upper and lower parts and uses only the parts that give better approximations to the exact operands’ values. This way, the bit width is reduced, which results in simpler circuitry at the expense of a larger error for some products. Additionally, the proposed multiplier trims the mantissas of the shortened operands. For this operation, we propose a new mantissa extractor based solely on a simple AND-OR net to lower delay and energy consumption. Moreover, a shorter operand and mantissa lead to a smaller adder and barrel shifter in the antilogarithm conversion circuitry.

Using the error analysis, we derive a lower bound of a portion of acceptable errors and empirically show that the TL16-8/4 multiplier computes more than 90% of products with a relative error smaller than 12.5%. The TL16-8/4 multiplier exhibits smaller energy consumption and area utilisation than the state-of-the-art designs.

Although the proposed multiplier offers significant improvements in energy consumption and area utilisation, the relatively large normalised mean error distance may be a significant drawback for its deployment. Nevertheless, the proposed multiplier behaves well in image processing and image classification with convolutional neural networks. From the error analyses, we can see that the 8-bit design has a significantly larger error than the 16-bit design. While the 16-bit design behaves well in both application case studies, the 8-bit design is more appropriate for less demanding applications, like image smoothing in our case. The final choice of the appropriate multiplier merely depends on the application requirements prioritising energy over the accuracy or vice versa. The experimental results suggest that the proposed multiplier can be useful in error-resilient applications, in intensive computing, or mobile devices, as it delivers more computational power per chip area and per power-delay product than the state-of-the-art logarithmic multipliers.

We have successfully deployed the proposed multiplier in the inference phase on CNNs. The remaining challenge is the training of the neural networks. In our previous work [11], we showed that it is possible to train the fully connected layers with approximate multipliers efficiently. In future research into the topic, we should investigate the approximate multipliers influence in the training phase of much larger convolutional neural networks. We anticipate that with an improved training strategy, we can lower the multipliers’ accuracy requirements and provide even more area and energy-efficient designs.

REFERENCES

[1] W. Liu, F. Lombardi, and M. Schulte, “Approximate computing: From circuits to applications [Scanning the issue],” Proc. IEEE, vol. 108, no. 12, pp. 2103–2107, Dec. 2020.
[2] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, “Approximate arithmetic circuits: A survey, characterization, and recent applications,” Proc. IEEE, vol. 108, no. 12, pp. 2108–2135, Dec. 2020.
[3] A. Agrawal et al., “Approximate computing: Challenges and opportunities,” in Proc. IEEE Int. Conf. Rebooting Comput. (ICRC), San Diego, CA, USA, Oct. 2016, pp. 1–8.
[4] S. Mittal, “A survey of techniques for approximate computing,” ACM Comput. Surv., vol. 48, no. 4, p. 62, Mar. 2016.
[5] N. E. Jerger and J. S. Miguel, “Approximate computing,” IEEE Micro, vol. 38, no. 4, pp. 8–10, Jul. 2018.
[6] L. Eeckhout, “Approximate computing, intelligent computing,” IEEE Micro, vol. 38, no. 4, pp. 6–7, Jul. 2018.
[7] W. Liu, L. Qian, C. Wang, H. Jiang, J. Han, and F. Lombardi, “Design of approximate radix-4 booth multipliers for error-tolerant computing,” IEEE Trans. Comput., vol. 66, no. 8, pp. 1435–1441, Aug. 2017.
[8] R. Zendegani, M. Kamal, M. Bahadori, A. Afzali-Kusha, and M. Pedram, “RoBA multiplier: A rounding-based approximate multiplier for high-speed yet energy-efficient digital signal processing,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 25, no. 2, pp. 393–401, Feb. 2017.
[9] V. Leon, G. Zervakis, D. Soudris, and K. Pekmeszti, “Approximate hybrid high radix encoding for energy-efficient inexact multipliers,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 26, no. 3, pp. 421–430, Mar. 2018.
[10] H. Jiang, C. Liu, F. Lombardi, and J. Han, “Low-power approximate unsigned multipliers with configurable error recovery,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 66, no. 1, pp. 189–202, Jan. 2019.
[11] U. Lotrič and P. Bulić, “Applicability of approximate multipliers in hardware neural networks,” Neurocomputing, vol. 96, pp. 57–65, Nov. 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231212003311
[12] M. S. Kim, A. A. Del Barrio, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, “Efficient Mitchell’s approximate log multipliers for convolutional neural networks,” IEEE Trans. Comput., vol. 68, no. 5, pp. 660–675, May 2019.
[13] R. Pilipović and P. Bulić, “On the design of logarithmic multiplier using radix-4 booth encoding,” IEEE Access, vol. 8, pp. 64578–64590, Apr. 2020.
[14] M. S. Ansari, V. Mrazek, B. F. Cockburn, L. Sekanina, Z. Vasicek, and J. Han, “Improving the accuracy and hardware efficiency of neural networks using approximate multipliers,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 28, no. 2, pp. 317–328, Feb. 2020.
[15] M. S. Ansari, B. F. Cockburn, and J. Han, “An improved logarithmic multiplier for energy-efficient neural computing,” IEEE Trans. Comput., vol. 70, no. 4, pp. 614–625, Apr. 2020.
[16] M. S. Kim, A. A. Del Barrio Garcia, H. Kim, and N. Bagherzadeh, “The effects of approximate multiplication on convolutional neural networks,” IEEE Trans. Emerg. Topics Comput., early access, Jan. 12, 2021, doi: 10.1109/TETC.2021.3050989.
[17] V. Leon, G. Zervakis, S. Xydias, D. Soudris, and K. Pekmeszti, “Walking through the energy-error Pareto frontier of approximate multipliers,” IEEE Micro, vol. 38, no. 4, pp. 40–49, Jul. 2018.
[18] Z. Liu, A. Yazdanbakhsh, T. Park, H. Esmaeilzadeh, and N. S. Kim, “SiMul: An algorithm-driven approximate multiplier design for machine learning,” IEEE Micro, vol. 38, no. 4, pp. 50–59, Jul. 2018.
[19] W. Liu, F. Lombardi, and M. Shulte, “A retrospective and prospective view of approximate computing [Point of view],” Proc. IEEE, vol. 108, no. 3, pp. 394–399, Mar. 2020.
