# Variable-Precision Approximate Floating-Point Multiplier for Efficient Deep Learning Computation

Hao Zhang, *Member, IEEE*, and Seok-Bum Ko, *Senior Member, IEEE*

***Abstract*—In this brief, a variable-precision approximate floating-point multiplier is proposed for energy efficient deep learning computation. The proposed architecture supports approximate multiplication with BFloat16 format. As the input and output activations of deep learning models usually follow normal distribution, inspired by the posit format, for numbers with different values, different precisions can be applied to represent them. In the proposed architecture, posit encoding is used to change the level of approximation, and the precision of the computation is controlled by the value of product exponent. For large exponent, smaller precision multiplication is applied to mantissa and for small exponent, higher precision computation is applied. Truncation is used as approximate method in the proposed design while the number of bit positions to be truncated is controlled by the values of the product exponent. The proposed design can achieve 19% area reduction and 42% power reduction compared to the normal BFloat16 multiplier. When applying the proposed multiplier in deep learning computation, almost the same accuracy as that of normal BFloat16 multiplier can be achieved.**

***Index Terms*—Approximate multiplier, posit format, deep learning computation, variable precision.**

## I. Introduction

APPROXIMATE computing [1] is a computing paradigm for error-resilient applications that utilizes simplified logic circuits to trade off results accuracy for better performance and resource efficiency. Deep learning computation is error-resilient and approximate computing could be applied in deep learning computation to seek for reduced computational intensity and improved energy efficiency [2].

In the literature, some approximate multipliers are proposed for deep learning computation [3], [4]. However, these works are designed for fixed-point computations. In addition to fixed-point formats, floating-point based formats, such as the BFloat16 [5] and the DLFloat [6], are popularly used in deep learning training and inference. On one hand, the small bit-width floating-point format has comparable hardware metrics compared to fixed-point format [7]. On the other hand, using floating-point format can avoid the complex neural network model quantization process [8]. Therefore, designing an approximate floating-point multiplier for deep learning computation is necessary.

There are several approximate floating-point units [9]–[13] available in the literature. However, for each of these approximate designs, only one precision mode is supported and thus, all computations need to be performed with the same level of approximation. For deep learning, as the data and weight follow normal distribution, it is expected that the data intensive range should be more precise while the less intensive range can be less precise.

The recent proposed posit format [14] can provide such variety in precision. Unlike the floating-point format, the bit-width of components in posit format will change for different values. When the exponent is large, the regime part in the posit format will occupy more bit positions and thus mantissa will have fewer bit-width. This non-uniformed encoding fits well with the data distribution of deep learning. Due to the variable bit-width format, posit arithmetic unit is more hardware expensive than IEEE floating-point unit [15], [16]. However, we could utilize the posit encoding method in BFLoat16 design to seek for various levels of approximation for the mantissa.

In this brief, a variable-precision approximate floating-point multiplier is proposed. The proposed architecture is designed for BFloat16 operations for efficient deep learning computation. Truncation is used as the approximate method in the proposed architecture. Various approximation levels for the mantissa are provided in the proposed design. The approximate level is determined by the value of the exponent. Posit encoding is used to encode the product exponent so that the required mantissa bit-width can be determined and computation for those extra bits can be avoided. By using the proposed architecture, a 19% area reduction and a 42% power reduction can be achieved when compared to normal BFloat16 multiplication. In addition, a case study of applying the proposed multiplier in deep learning computation is performed and the results show that the proposed design can achieve almost the same accuracy as that of the exact multiplier.

The rest of this brief is organized as follows: Section II presents the posit encoding method. The proposed approximate multiplier architecture is presented in Section III. Section IV presents the implementation results and their analysis. Finally, Section V concludes the whole paper.

Manuscript received February 7, 2022; accepted March 16, 2022. Date of publication March 22, 2022; date of current version May 3, 2022. This brief was recommended by Associate Editor K. Chen. *(Corresponding author: Seok-Bum Ko.)*

Hao Zhang is with the Faculty of Information Science and Engineering, Ocean University of China, Qingdao 266100, China (e-mail: hao.zhang@ouc.edu.cn).

Seok-Bum Ko is with the Department of Electrical and Computer Engineering, University of Saskatchewan, Saskatoon, SK S7N 5A9, Canada (e-mail: seokbum.ko@usask.ca).

Digital Object Identifier 10.1109/TCSII.2022.3161005

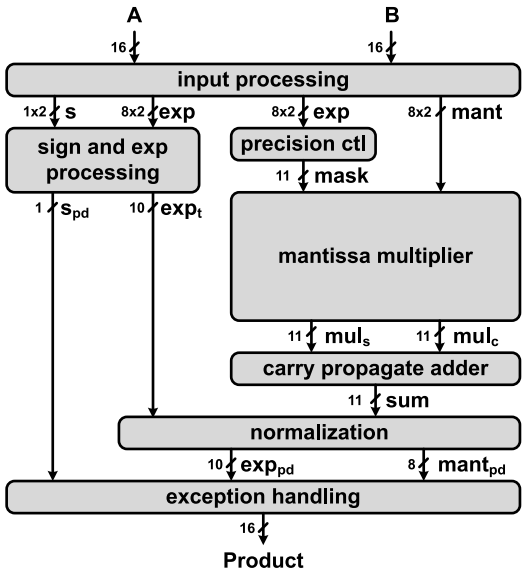



Fig. 1. Architecture of the proposed approximate BFloat16 multiplier.

## II. Background

Posit format is denoted as Posit(*nb*, *es*), where *nb* represents the total bit-width and *es* represent the exponent bit-width. A posit format is composed of sign (*s*), regime (*r*), exponent (*e*), and mantissa (*frac*). Except the sign bit, the bit-width of all other components is not a constant. The regime part always exists and its bit-width is determined by its value. The exponent and mantissa only appear when there is available bit positions for them.

Regime is encoded as a series of zeros (ones) ended by 1-bit of one (zeros). If it is *m*-bit zeros followed 1-bit one, then the regime value is $-m$. Otherwise, if it is *m*-bit ones followed by 1-bit zero, then the regime value is $m-1$.

The value of the posit number can be calculated with:

$$value = (-1)^s \times (2^{2^{es}})^{rg} \times 2^{exp} \times (1 + frac) \tag{1}$$

Therefore, the effective exponent can be represented as $rg \times 2^{es} + exp$. This can be used to convert formats between posit format and floating-point format. In the proposed design, this effective exponent will be used to convert the BFloat16 exponent to posit exponent and generate the regime format. So that the bit-width of regime and thus the bit-width of mantissa will be generated. This mantissa bit-width will be used to control the level of approximation performed in the mantissa multiplier.

## III. The Proposed Approximate BFloat16 Multiplier

The architecture of the proposed approximate BFloat16 multiplier (BFLP16-Prop) is shown in Fig. 1. It follows a standard floating-point multiplier datapath with the following changes:

- Add a precision control module in the datapath to evaluate the exponent for the product and thus generate a mask signal to control the level of approximation in mantissa multiplier.
- Implement an approximate mantissa multiplier with truncation method. Therefore, the bit-width of the mantissa multiplier also changes.
- To further save area and power, rounding unit is removed from the datapath. This is the same as in [10].
- Simplify exception handling module. Since the proposed design is approximate unit and no rounding is performed, the detection of inexact result is removed. As a results, logic circuits to generate floating-point flags are also simplified.

In the following text, the design of the precision control unit and the design of the approximate mantissa multiplier will be discussed in detail.

### A. Precision Control Unit

The precision control unit is responsible for calculating the exponent of the product, converting it to posit encoding format, and then generate a mask signal to control the level of approximation in the mantissa multiplier. The exponent of the product is calculated by adding two operand exponents and then subtracting $2 \times bias$, where $bias = 127$ for BFloat16.

Before doing the conversion to posit exponent, the posit format needs to be determined. The representation range of posit format should be no less than the BFloat16 format, so that all the BFloat16 values can be accommodated in the target posit format.

For BFloat16 format, the exponent range is from $-126$ to 127. For posit format, the effective exponent is $rg \times 2^{es} + exp$. When the regime takes the maximum value, it will occupy all the bit-width and thus no exponent appear in the format. For 16-bit posit, the maximum regime is 15-bit which is equal to 14. Therefore, to ensure the posit format has larger range than BFloat16, we need to solve $14 \times 2^{es} \geq 127$. As a result, *es* should be at least 4-bit. When using larger exponent bit-width, as the exponent part can take larger value, the maximum bit-width of the regime will become smaller. The overall precision for mantissa will not change. In the proposed design, we choose to use $es = 4$-bit.

When $es = 4$, the effective exponent in posit format is $rg \times 2^4 + exp$. When doing the conversion, the least significant 4-bit of the BFloat16 exponent will be directly pass to the *exp* in posit format. The most significant 4-bit of the BFloat16 exponent will be used to encode the regime. The relationship between the regime bits and the mask is represented in Table I.

Take the fourth entry of Table I as an example. When the most significant 4-bit of BFloat16 exponent is 0011 (rg), the regime value (rg_val) will be 3. According to the posit encoding method discussed in Section II, the regime string (rg_str) will be 11110 which is a 5-bit vector. In Posit(16,4) format, when regime is 5-bit, there is still space for exponent, and thus the exponent will take another 4-bit. As a result, the remaining bit position for mantissa (bw_mant) is 6-bit. Considering the implicit bit, we need a 7-bit product. However, as in the direct product of mantissa multiplier, there are 2-bit to the left of the radix point (before normalization), therefore, a 8-bit product (bw_pd) is required to be generated from the mantissa multiplier. Therefore, the mask contains 8-bit ones followed by 3-bit zeros.

TABLE I
TABLE TO GENERATE MASK FROM REGIME

| rg | rg_val | rg_str | bw_mant | bw_pd | mask |
|---|---|---|---|---|---|
| 0000 | 0 | 10 | 9-bit | 11-bit | 11111111111 |
| 0001 | 1 | 110 | 8-bit | 10-bit | 11111111110 |
| 0010 | 2 | 1110 | 7-bit | 9-bit | 11111111100 |
| 0011 | 3 | 11110 | 6-bit | 8-bit | 11111111000 |
| 0100 | 4 | 111110 | 5-bit | 7-bit | 11111110000 |
| 0101 | 5 | 1111110 | 4-bit | 6-bit | 11111100000 |
| 0110 | 6 | 11111110 | 3-bit | 5-bit | 11111000000 |
| 0111 | 7 | 111111110 | 2-bit | 4-bit | 11110000000 |
| 1000 | -8 | 000000001 | 2-bit | 4-bit | 11110000000 |
| 1001 | -7 | 00000001 | 3-bit | 5-bit | 11111000000 |
| 1010 | -6 | 0000001 | 4-bit | 6-bit | 11111100000 |
| 1011 | -5 | 000001 | 5-bit | 7-bit | 11111110000 |
| 1100 | -4 | 00001 | 6-bit | 8-bit | 11111111000 |
| 1101 | -3 | 0001 | 7-bit | 9-bit | 11111111100 |
| 1110 | -2 | 001 | 8-bit | 10-bit | 11111111110 |
| 1111 | -1 | 01 | 9-bit | 11-bit | 11111111111 |

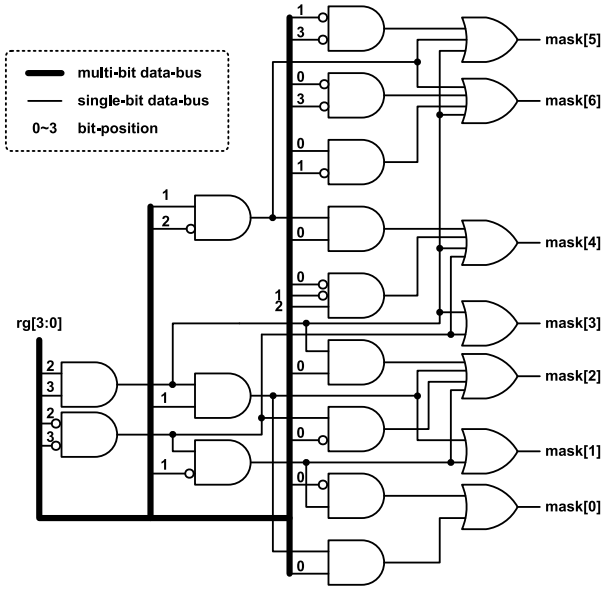


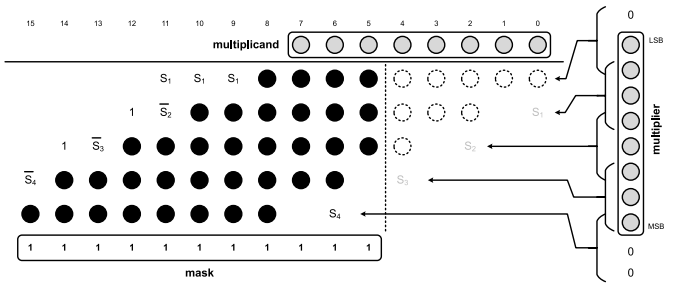

Fig. 2. Logic circuit to generate mask from most significant 4-bit of exponent.

Considering all 16 cases, the circuit to generate mask from the most significant 4-bit of the BFloat16 exponent is shown in Fig. 2. Note that the smallest bit-width for product is 4-bit, therefore, the most significant 4-bit of mask, mask[10:7], is always logic 1.

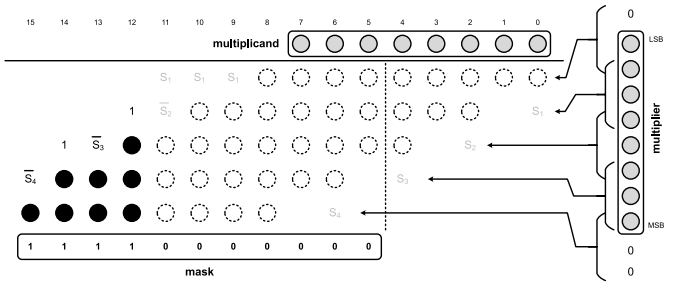
### B. Approximate Mantissa Multiplier

The mantissa multipliers in both the benchmark design (BFLP16-Norm) and the proposed design (BFLP16-Prop) are designed with radix-4 modified Booth algorithm [17]. The partial product array of the proposed mantissa multiplier is shown in Fig. 3. The dotted circles or the grey texts represent the corresponding bits does not have computation.

In the proposed design, as the maximum bit-width of the required mantissa product is 11-bit, therefore, the lower 5-bit are always truncated. The mask generated with the circuit shown in Fig. 2 is applied to the higher 11-bit position. If




(a) mask=11111111111




(b) mask=11110000000

Fig. 3. The proposed approximate mantissa multiplier.

the mask bit is 1, then the corresponding position will use compressor to accumulate the partial products, otherwise, the computation in that bit position will be disabled. When the mask is 11-bit ones and 4-bit ones, the partial product arrays are shown in Fig. 3(a) and Fig. 3(b), respectively.

### C. Other Modules

The input processing module is to extract each components from the BFloat16 format operands. The sign and tentative exponent of the product are generated in the sign and exp processing module.

The carry-save format products are added to generate the final mantissa product in the carry propagate adder module. Since the bit-width of the adder operands is small, carry-lookahead adder is used for this adder. The normalization module is a 1-bit right shifter to make the mantissa product back to $[1, 2)$ range. Exponent is also adjusted when a right shift of the mantissa product is performed. The exception handling module is used to generate flag for underflow, overflow, and invalid results. Since rounding is not performed, the flag for inexact result is not detected.

## IV. RESULTS AND ANALYSIS

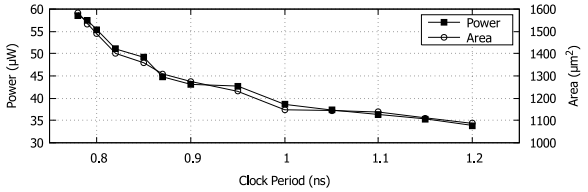
### A. Hardware Metrics

The proposed approximate BFloat16 multiplier is modeled with Verilog HDL. The Verilog design is verified with extensive testing vectors. Then, the Verilog design is synthesized using Synopsys Design Compiler with TSMC-65nm library with typical case parameters. The synthesized netlist is simulated again and the signal activity file generated during post-synthesis simulation is used in Synopsys PrimeTime PX for power estimation.

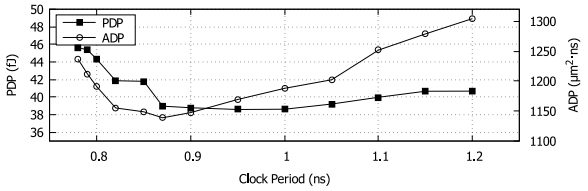
The comparison of the BFLP16-Prop and the BFLP16-Norm is shown in Table II. The two designs have similar

TABLE II
COMPARISON OF THE BFLP16-PROP WITH BFLP16-NORM

| Design | Delay ($ns$) | Area ($\mu m^2$) | Power ($\mu W$) | ADP ($\mu m^2 \cdot ns$) | PDP ($fJ$) |
|---|---|---|---|---|---|
| BFLP16_Norm | 0.75 | 1956 | 101.4 | 1467 | 76 |
| BFLP16_Prop | 0.78 | 1585 | 58.5 | 1236 | 45 |




(a) Delay-Area-Power Curve




(b) Delay-ADP-PDP Curve

Fig. 4. Hardware metrics under various timing constraints.

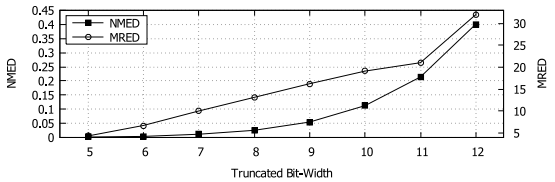
TABLE III
COMPARISON OF EACH COMPONENT OF BFLP16-PROP WITH BFLP16-NORM

| Module | Area ($\mu m^2$) | | | Power ($\mu W$) | | |
|---|---|---|---|---|---|---|
| | norm | prop | ratio | norm | prop | ratio |
| input process | 238 | 321 | +34.87% | 12.0 | 12.8 | +6.67% |
| precision control | n/a | 86 | n/a | n/a | 2.46 | n/a |
| mantissa multiplier | 905 | 622 | -31.27% | 56.7 | 25.2 | -55.56% |
| final adder | 257 | 222 | -13.61% | 14.6 | 7.67 | -47.46% |
| normalization | 183 | 168 | -8.19% | 8.19 | 7.03 | -14.16% |
| rounding | 195 | n/a | n/a | 5.05 | n/a | n/a |
| exception handle | 178 | 166 | -6.74% | 4.85 | 3.32 | -31.54% |
| total | 1956 | 1585 | -18.96% | 101.4 | 58.5 | -42.32% |

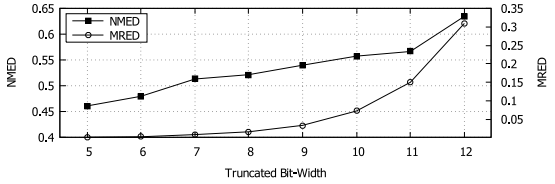
delay. The BFLP16-Prop removes rounding modules, however, the proposed precision control model is in the critical path and thus the total delay is slightly larger. In terms of area and power, due to the approximated multiplier, the smaller final adder, and simplified normalization circuit, the proposed design can achieve 19% and 42% improvement, respectively. The proposed design can achieve smaller area delay product (ADP) and power delay product (PDP).

The area-delay curve and power-delay curve are shown in Fig. 4(a). With larger timing constraint, both the area and delay of the proposed BFLP16-Prop can be reduced. The corresponding ADP and PDP under different timing constraint are shown in Fig. 4(b). To achieve a small ADP and PDP, the design should be run with a timing constraint of $0.9ns$.

The comparison of each component of BFLP16-Norm and BFLP16-Prop is shown in Table III. The sign and exponent processing module is included into the input process module. With the proposed approximate mantissa multiplier, the area and power of the mantissa multiplier is significantly reduced.




(a) Error of Whole Range




(b) Error of Operands with MSB=1

Fig. 5. Error metrics under various truncated bit-width.

TABLE IV
ERROR METRICS OF THE PROPOSED BFLP16-PROP

| | MED | NMED | MRED |
|---|---|---|---|
| BFLP16-Prop | $2.6 \times 10^{-3}$ | $1.03 \times 10^{-2}$ | $3.5 \times 10^{-3}$ |

In addition, the final adder in the proposed design is smaller and the logic in normalization and exception handling module is less complex which also leads to area and power reduction. For BFLP16-Norm, only the biased product exponent is required. However, in the proposed BFLP16-Prop, the unbiased product exponent value is also required to generate mask for mantissa multiplier, therefore, the area of the input process module is increased.

### B. Error Analysis

First, the error of the mantissa multiplier is analyzed. As the multiplier is an $8 \times 8$ multiplier, all possible input combinations are used to calculate errors of the approximate design. Although the approximate level is automatically controlled by the product exponent at run-time, we still split the analysis into each truncated bit-width. Here, the normalized mean error distance (NMED) and mean relative error distance (MRED), proposed in [18], are used. The NMED and MRED under different truncated bit are shown in Fig. 5(a). As the proposed approximate multiplier is used in a floating-point design, where the most significant bit of the operand is always 1 (the implicit bit, except the operand is 0). To analyze the error under this constraint, these operands are extracted, and the NMED and MRED is shown in Fig. 5(b).

The error of the proposed BFLP16-Prop is measured with mean error distance (MED), normalized MED (NMED), and mean relative error distance (MRED). The data range used here are summarized from the weights of AlexNet [19], VGG16 [20], ResNet18 [21], and MobileNetV2 [22], and activations when using the CIFAR-10 dataset [23]. The results are shown in Table IV.

### C. Deep Learning Accuracy

Four deep learning models, AlexNet [19], VGG16 [20], ResNet18 [21], and MobileNetV2 [22], are fine-tuned with

TABLE V
ACCURACY WHEN USING VARIOUS ARITHMETIC UNITS

| | AlexNet | VGG16 | ResNet18 | MobileNetV2 |
|---|---|---|---|---|
| IEEE-FLP32 | 91.63% | 94.12% | 95.38% | 95.58% |
| BFLP16-Norm | 90.72% | 94.08% | 95.26% | 94.65% |
| BFLP16-Prop | 90.70% | 94.05% | 95.25% | 94.60% |

the CIFAR-10 dataset [23]. Then, inference is performed with CIFAR-10 test set. The accuracy of models when running with various arithmetic units are summarized in Table V. Here, we use IEEE-FLP32 format to perform accumulation, according to [24]. As shown in Table V, although approximate computing is used in the proposed design, the accuracy of these models is almost not affected.

The truncation bit-width or the approximation level is generated with posit encoding method using the product exponent. Posit encoding can introduce an error of normal distribution. When the number of multiplication and accumulation is large, the overall effect is equivalent to adding the same amount of error to the accurate results. After adding the same amount of error, the comparison among those results is not changed, and thus the prediction of deep learning model is still correct. This result and analysis align with the discussion in [25].

## V. CONCLUSION

In this brief, a variable-precision approximate BFloat16 multiplier architecture is proposed for energy efficient deep learning computation. Truncation is used as the approximate computing method in this brief. The amount of bits to be truncated is determined by the product exponent. In the proposed design, posit encoding is used to encode the product exponent to generate the required mantissa bit-width of the product. By using the proposed architecture, a 19% area reduction and a 42% power reduction can be achieved when compared to normal BFloat16 multiplication. The proposed multiplier is used in the computation of benchmark models. The accuracy generated by the proposed multiplier is almost the same as that generated by exact multiplier. With the proposed method, the benefit of posit encoding is utilized while avoiding the overhead of posit unit over floating-point unit.

In the future, the proposed approximate multiplier will be used and evaluated in deep learning training process and integrated into deep learning processors.

## ACKNOWLEDGMENT

The authors would like to thank Ocean University of China and the University of Saskatchewan for the financial support for this project.

## REFERENCES

[1] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Proc. 18th IEEE Eur. Test Symp. (ETS)*, May 2013, pp. 1–6.

[2] C. Chen, J. Choi, K. Gopalakrishnan, V. Srinivasan, and S. Venkataramani, "Exploiting approximate computing for deep learning acceleration," in *Proc. Design Autom. Test Europe Conf. Exhibition (DATE)*, Mar. 2018, pp. 821–826.

[3] 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.

[4] M. S. Kim, A. A. D. 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.

[5] S. Wang and P. Kanwar. "BFloat16: The Secret to High Performance on Cloud TPUs." Aug. 2019. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

[6] A. Agrawal *et al.*, "DLFloat: A 16-b floating point format designed for deep learning training and inference," in *Proc. IEEE 26th Symp. Comput. Arithmetic (ARITH)*, Jun. 2019, pp. 92–95.

[7] L. Lai, N. Suda, and V. Chandra, "Deep convolutional neural network inference with floating-point weights and fixed-point activations," Mar. 2017, *arXiv:1703.03073*.

[8] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

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

[10] P. Yin, C. Wang, W. Liu, and F. Lombardi, "Design and performance evaluation of approximate floating-point multipliers," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2016, pp. 296–301.

[11] M. Imani, R. Garcia, S. Gupta, and T. Rosing, "RMAC: Runtime configurable floating point multiplier for approximate computing," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, Jul. 2018, pp. 1–6.

[12] V. Camus, J. Schlachter, C. Enz, M. Gautschi, and F. K. Gurkaynak, "Approximate 32-bit floating-point unit design with 53% power-area product reduction," in *Proc. 42nd Eur. Solid-State Circuits Conf.*, Sep. 2016, pp. 465–468.

[13] M. Olivieri, F. Pappalardo, S. Smorfa, and G. Visalli, "Analysis and implementation of a novel leading zero anticipation algorithm for floating-point arithmetic units," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 54, no. 8, pp. 685–689, Aug. 2007.

[14] J. L. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov. Int. J.*, vol. 4, no. 2, pp. 71–86, Jun. 2017.

[15] Y. Uguen, L. Forget, and F. de Dinechin, "Evaluating the hardware cost of the posit number system," in *Proc. 29th Int. Conf. Field Programmable Logic Appl. (FPL)*, Sep. 2019, pp. 106–113.

[16] H. Zhang and S. Ko, "Design of power efficient posit multiplier," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 5, pp. 861–865, May 2020.

[17] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, no. 2, pp. 236–240, 1951.

[18] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *IEEE Trans. Comput.*, vol. 62, no. 9, pp. 1760–1771, Sep. 2013.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, vol. 1, Dec. 2012, pp. 1097–1105.

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, May 2015, pp. 1–14.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun 2016, pp. 770–778.

[22] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.

[23] A. Krizhevsky, "Learning multiple layers of features from tiny images," Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, Rep. TR-2009, Apr. 2009.

[24] G. Henry, P. T. P. Tang, and A. Heinecke, "Leveraging the BFloat16 artificial intelligence datatype for higher-precision computations," in *Proc. IEEE 26th Symp. Comput. Arithmetic (ARITH)*, Jun. 2019, pp. 69–76.

[25] 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.