Comparison of Floating-point Representations for the Efficient Implementation of Machine Learning Algorithms

Saras Mani Mishra*, Ankita Tiwari, Hanumant Singh Shekhawat
Prithwijit Guha, Gaurav Trivedi
Department of Electronics and Electrical Engineering,
Indian Institute of Technology Guwahati, Guwahati, India
Email: **m.saras@iitg.ac.in*

Pidanic Jan†, Zdenek Nemec
Department of Electrical Engineering,
University of Pardubice, Pardubice, Czech Republic
Email: †*jan.pidanic@upce.cz*

***Abstract*—Smart systems are enabled by artificial intelligence (AI), which is realized using machine learning (ML) techniques. ML algorithms are implemented in the hardware using fixed-point, integer, and floating-point representations. The performance of hardware implementation gets impacted due to very small or large values because of their limited word size. To overcome this limitation, various floating-point representations are employed, such as IEEE754, posit, bfloat16 etc. Moreover, for the efficient implementation of ML algorithms, one of the most intuitive solutions is to use a suitable number system. As we know, multiply and add (MAC), divider and square root units are the most common building blocks of various ML algorithms. Therefore, in this paper, we present a comparative study of hardware implementations of these units based on bfloat16 and posit number representations. It is observed that posit based implementations perform $1.50\times$ better in terms of accuracy, but consume $1.51\times$ more hardware resources as compared to bfloat16 based realizations. Thus, as per the trade-off between accuracy and resource utilization, it can be stated that the bfloat16 number representation may be preferred over other existing number representations in the hardware implementations of ML algorithms.**

***Index Terms*—Floating-point representations, Deep Learning, Posit, Training,**

I. Introduction

For decades, artificial intelligence (AI) has been integrated into almost every aspect of digital life. As artificial neural networks (ANN) is not new, re-emergence is transpiring to introduce upgrades and modifications in models. The model learning is a process, which takes a huge labeled dataset as an example to construct a classification network. However, there are a number of datasets to be employed to improve convergence of the algorithm. The common machine learning (ML) algorithms take time to converge even when they are executed on a high performance computing machine. As we know, training is the most compute intensive step in AI modeling. When the input dataset is in the form of images, while training a classification model, it needs to adjust millions of weights continuously. As the process requires data to be in a wide dynamic range, the floating-point representation presents an enormous range in fixed bit width, which is not possible by integer or fixed-point representation. Therefore, floating-point representations have become the primary choice in the realization of ML models.

The floating-point number is represented by significand, exponent and base as $significand \times base^{exponent}$. IEEE754 [1] is the most commonly used floating-point representation. It is expressed using varied bit-widths, such as 64, 32, 16, 8-bits etc., and delivers high numeric precision and throughput. However, the IEEE754 standard was introduced in 1985 exhibiting expensive processing and memory access. Nowadays, computation is cheap and accessing the memory is expensive, therefore it makes the existing IEEE754 representations inefficient. In 2014, Google Brain introduced a modified version and called it the brain floating-point representation (bfloat16) [2]. This representation performs well for ML application [3] [4], and leads the machine learning model towards optimization. Intel has also proposed its Cooper Lake Xeon processor [5], which employs bfloat16 representation, for accelerating the DNN model.

The few other works [6] [7] [8] depict that posit [9], a newer version of floating-point representation may replace the IEEE754 standard. Further, [8] describes that it provides better accuracy for DNN models. In this paper, we propose the study of various number representations for the efficient realization of ML algorithms. We have compared various floating-point number representations based on accuracy, area and power consumption of the hardware realization of a few selected most commonly used units, viz. MAC, divider and square root units in this paper. These units are the most commonly used modules in ML algorithms.

Since ML algorithms employ above mentioned arithmetic modules during their hardware implementation. The analysis of these arithmetic units is essential for the fair comparison of various floating-point number representations. It is observed that to achieve minimum resource utilization, single-precision compliant training needs additional parameters. When results are compared with posit based training of ML algorithms, the accuracy of posit based training seems to be higher as compared to all other representations [10]. Further, the analytical study illustrates that except for posit, all other representation based trainings showcase worse accuracy than bfloat16 based training. It is found that bfloat16 based implementations outperform posit based realizations during the experimental analysis.

Rest of the paper is organized as follows. Section II introduces details of floating-point number representations available in the literature. Section III demonstrates hardware realization of different arithmetic modules based on these representations. Section V presents performance analysis based on resource utilization and power consumption of various arithmetic modules. The proposed work is concluded in section VI.

II. Related Work

In this section, we brief about the most popular floating-point number representations.

A. Posit

In 2017, another floating-point representation, named as posit [9], is proposed. It is claimed that it performs better than the existing IEEE754 based representations [9], and does not support overflow or underflow, which reduces complexity of exception handling. This number system is represented using Figure 1. The decimal value of this representation is calculated using equation 1, where k is the length of the regime field. The regime bits are realized by $useed$, where $useed$ is $2^{2^{ES}}$. Here, ES is the decimal representation of exponent bits considering it as an unsigned integer. Posit does not contain any bias to implement exponent term. The remaining bits are allocated to mantissa.

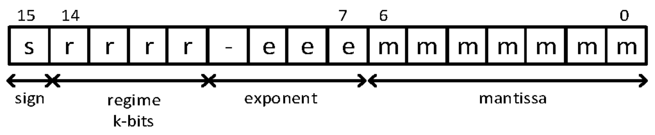

Fig. 1. Posit number format

$$Value = (-1)^S \times useed^k \times 2^E \times (1 + M) \qquad (1)$$

B. Brain floating-point representation (bfloat16)

Bfloat16 representation is proposed with a slight modification in single-precision floating-point (FP32) representation by Google brain. It is a $16-bit$ representation, which contains $8-bit$ exponent and $7-bit$ mantissa and a sign bit at MSB. The number system is shown in Figure 2 and equation 2 is used to calculate the decimal value of bfloat16 representation.

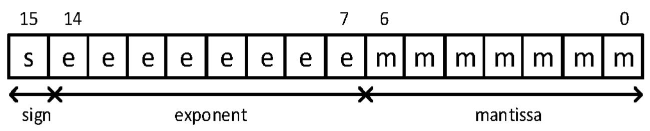

Fig. 2. Brain floating-point representation

$$FinalValue = (-1)^S \times 2^{E-127} \times (1.M) \qquad (2)$$

III. Implementation Details

The machine learning algorithms require ample computation during training and testing of the models. Implementing computation module in the hardware requires an efficient number representation. In a machine learning model, data may vary in a wide range, which creates the requirement of floating-point representation. There are a number of floating-point representations available in the literature, which are described in section II. In this section, we discuss basic computation elements (fused multiply and addition, division and square-root) based on *16-bit fixed point*, *single-precision (FP32)*, *posit*, and *bfloat16* representations.

To perform any arithmetic operation on floating-point representations, it is imperative to identify the representation first, which requires an extraction unit to find exponent, mantissa or fraction. However, fixed-point, single-precision, and bfloat16 representations can be directly employed in the computation modules, whereas, posit requires a setup to extract the above mentioned information. *Single-precision (FP32)*, and *bfloat16* are given directly in terms of exponent and mantissa. An extraction code[1] for posit based implementations is employed, which is realized using 28 OR, 18 AND, 8 NAND and 24 multiplexers. The area utilization and power consumption of the extraction unit are $4.095\mu m^2$ and $3.757mW$, respectively. As mentioned above, in order to find the best number representation for the realization of ML algorithms, basic computation elements, such as a fused multiply and addition, division and square-root are used. These basic computations elements are described below for completeness.

A. Fused Multiply and Addition

The fused multiply and add operations are performed using three inputs, where the product of the initial two inputs is summed with the third one. Algorithm 1 [11] is employed for each floating-point representation except the fixed-point representation because it contains different extraction methods. The fixed-point representation has used a traditional [12] algorithm for the said purpose.

TABLE I
Number of gates, area, power and delay of each representation during fused multiply and add operation.

Datatype	**Gates**	**Area** (μm^2)	**Power**(mW)	**Delay**(ns)
Fixed-point	4308	266.905	211.912	88.113
Single-precision	6393	359.736	285.866	211.572
Posit	2247	115.721	106.711	109.971
Bfloat16	1998	113.038	97.452	108.979

It is assumed that the 16-bit fixed-point representation contains a 6-bit integer and a 10-bit fraction. The number of gates used by each representation in multiply and add operations are shown in Table I. Since multiplication is the costliest operation among basic arithmetic operations, it consumes more resources with the increase of bits. The fixed-point number and other

[1]source code obtained from https://github.com/manish-kj/Posit-HDL-Arithmetic

Algorithm 1 Fused Multiply-Addition

```
procedure FUSED MULTIPLY-ADD(Input1, Input2, Input3)
    Given: N = word length, E = exponent, Bias = (2**(E-1))-1
    Input: Input1, Input2, Input3
    Data Extraction:
        Input1: $Inf_1$, $Z_1$, $S_1$, $E_1$, $M_1$
        Input2: $Inf_2$, $Z_2$, $S_2$, $E_2$, $M_2$
        Input3: $Inf_3$, $Z_3$, $S_3$, $E_3$, $M_3$
    Exceptions: Zero $\leftarrow$ $Z_1$ AND $Z_2$
            Infinity $\leftarrow$ $Inf_1$ OR $Inf_2$
    Multiplication Operation:
        $S_p \leftarrow S_1$ XOR $S_2$
        Mantissa Multiplication:
        Multiplication of $M_1$ and $M_2$: $M_{mul} \leftarrow M_1 \times M_2$
        Mantissa Overflow: $f_{mov} \leftarrow M_{mul}[MSB]$: $M_{mul}$<<1
            $M_p \leftarrow M_{mul}$: $M_{mul}$<<1
        Exponent processing: $E_p \leftarrow E_1$ - $E_2$ + $f_{mov}$
    Add/Sub processing:
        Operation: Op $\leftarrow$ $S_p$ XOR $S_3$
        Greater: $I_{p\_}grt\_I_3 \leftarrow E_p$, $M_p >$ Input3[N-2:0] ? 1 : 0
        Large (L) values: $S_l$, $E_l$, $M_l$
        Small (S) values: $S_s$, $E_s$, $M_s$
        Mantissa Addition:
        Exponent difference: $Diff_e \leftarrow E_l - E_s$
            $M_{s_t} \leftarrow$ shifting mantissa by $Diff_e$
        Addition of $M_l$ and $M_{s_t}$: $M_{fma} \leftarrow$ Opr ? $M_l + M_{s_t}$ : $M_l - M_{s_t}$
        Mantissa Overflow: $f_{mov} \leftarrow M_{fma}[MSB]$
            $M_{fma} \leftarrow M_{fma}$: $M_{fma}$<<1
        Exponent processing: $E_{fma} \leftarrow Diff_e$ ? $E_l$ + $Diff_e$ : $E_l$
        Data Processing:
        Final Output: $O_{fma} \leftarrow S_l, E_{fma}, M_{fma}$
```

representations are divided into sections (such as fraction, exponent, and mantissa), which makes it less costly compared to other representations. It is observed from Table I, bfloat16 consumes a $1.60\times$ smaller number of gates compared to other representations. The area utilization of bfloat16 representation is $1.63\times$ less among all representations. For better analysis, we have used another example to compare the results in the next subsection.

B. Division & Square-root

Since division and square-root (DSQRT) is another most commonly employed arithmetic operation, it is implemented for analysis. DSQRT module operates in two modes, one is division and another is square root. When the module is operating in division mode, it will receive two inputs and produce an output. When working in square root mode, there is only one input that results in a square root. We employed the non-restoring algorithm 2 [13] for analysis purposes. The algorithm is not employed to fixed-point representation due to a different extraction pattern of these representations.

Algorithm 2 Division & Square-root

```
procedure DIVISION & SQUARE-ROOT(Input1, Input2, DSQRT)
    Given: N = word size, E = exponent, Bias = (2**(E-1))-1
    Input: Input1, Input2
    Operation: Op $\leftarrow$ DSQRT? DIV : SQRT
    Division:
        Data Extraction:
            Input1: $Inf_1$, $Z_1$, $S_1$, $E_1$, $M_1$
            Input2: $Inf_2$, $Z_2$, $S_2$, $E_2$, $M_2$
        Exceptions: Zero $\leftarrow$ $Z_1$ AND $Z_2$
                Infinity $\leftarrow$ $Inf_1$ OR $Inf_2$
        Multiplication process:
            $S_{div} \leftarrow S_1$ XOR $S_2$
            Mantissa Division:
            Division of $M_1$ and $M_2$: $M_{div} \leftarrow M_1 \div M_2$
            Mantissa Overflow: $f_{mov} \leftarrow M_{div}[MSB]$: $M_{div}$<<1
                $M_{div} \leftarrow M_{div}$: $M_{div}$<<1
            Exponent processing: $E_{div} \leftarrow E_1$ - $E_2$ + $f_{mov}$
            Data Processing:
                Final Output: $O_{div} \leftarrow S_{div}, E_{div}, M_{div}$
    Square-root:
        Data Extraction:
            Input1: $Inf$, $Z$, $S$, $E$, $M$
        Exceptions: Zero $\leftarrow$ $Z$
                Infinity $\leftarrow$ $Inf$
        $S_{sqrt} \leftarrow S$
        Exponent processing: $E_{sqrt} \leftarrow E$ + 1
        Mantissa Square-root:
            if $E_{sqrt}$ is odd
                $M_{sqrt} \leftarrow M_{sqrt} << 1$
        Data Processing:
            Final Output: $O_{sqrt} \leftarrow S_{sqrt}, E_{sqrt}, M_{sqrt}$
```

TABLE II
NUMBER OF GATES, AREA, POWER AND DELAY OF EACH REPRESENTATION DURING DIVISION AND SQUARE-ROOT OPERATION.

Datatype	**Gates**	**Area** (μm^2)	**Power**(μW)	**Delay**(ns)
Fixed-point	273	17.126	13.227	50.725
Single-precision	647	52.201	37.087	240.142
Posit	449	24.733	22.132	126.998
Bfloat16	370	21.392	19.230	126.110

The fixed-point representations have used a traditional division and square-root [14] algorithm for the operation. The number of gates used by each representation in the operation is shown in Table I. Since fixed-point representations have limitations of less range, they are not a better choice for machine learning applications. While comparing among single-precision, posit and bfloat16, it is observed that bfloat16 representation consumes $1.79\times$, $1.53\times$ and $1.45\times$ less area, power, and delay, respectively.

On the basis of the above analysis, it is worth saying that among floating-point representations, for arithmetic computation, bfloat16 consumes minimum resources. The reason for this decreased resource utilization is its fewer mantissa bits. However, the bfloat16 representation shows the values in the same range as single-precision, which makes it more compatible for applying in machine learning computation models.

Mean Relative Error Distance (MRED) is used to resemble the potential of single-precision, posit, and bfloat16 floating-point number systems. A single-event upset (SEU) as mentioned in [15], is employed in the analysis. The study is performed by flipping the mantissa, exponent, and regime bit one by one. As long as a 16-bit bfloat16 representation cannot be compared with a 32-bit representation, bfloat16 is compared with a 16-bit posit. The study shown in [15], depicted the comparison between posit (32,2) and single-precision and it is proved that posit is more error resilient than single-precision.

The analysis is performed by injecting a fault at the i^{th} bit position of a number represented in posit and bfloat16 formats. An error is calculated with respect to its actual value using equation 3 in this analysis. The range of half-precision representation is substantially less and not sufficient for a large value, which is why it is not including in the analysis.

$$MRED = \frac{1}{n}\sum_{i=0}^{n-1}\frac{|Val_i - Val_i^*|}{Val_i} \tag{3}$$

where, Val_i and Val_i^* are the actual and modified values generated before bit-flip and after bit-flip, respectively. Figure 3 presents comparison between various numbers represented in bfloat16 and posit formats. It can be observed that posit(16,1) is more error resilient than bfloat16.

Table I and II show that bfloat16 based modules utilizes $1.69\times$ and $1.33\times$ better for fused multiply and addition and division and square-root operations. The comparison is made between posit and bfloat16 because they are the best two performing floating-point representations.

TABLE III
ERROR ANALYSIS OF POSIT (16,1) AND BFLOAT16 BASED FUSED MULTIPLY AND ADDITION.

Representation	**Parameter**	**Iterations**					**Average**
		1st	2nd	3rd	4th	5th	
Posit	MSE	7.44E-06	7.47E-06	7.66E-06	7.41E-06	7.55E-06	7.51E-06
Bfloat16	MSE	5.05E-06	5.04E-06	5.01E-06	4.88E-06	4.98E-06	4.99E-06
Posit	Area (μm^2)	115.721					
	Power (mW)	106.711					
Bfloat16	Area (μm^2)	113.038					
	Power (mW)	97.452					
Posit ($Area \times Power \times MSE$)		9.19E-02	9.22E-02	9.46E-02	9.15E-02	9.32E-02	9.27E-02
Bfloat16 ($Area \times Power \times MSE$)		5.56E-02	5.55E-02	5.52E-02	5.38E-02	5.49E-02	5.50E-02
Improvement							1.69×

TABLE IV
ERROR ANALYSIS OF POSIT (16,1) AND BFLOAT16 BASED FLOATING-POINT DIVISION AND SQUARE-ROOT UNIT.

Representation	**Parameter**	**Iterations**					**Average**
		1st	2nd	3rd	4th	5th	
Posit	MSE	1.87E-06	9.50E-07	9.48E-07	9.42E-07	9.40E-07	1.13E-06
Bfloat16	MSE	1.88E-06	9.44E-07	9.42E-07	9.36E-07	9.34E-07	1.13E-06
Posit	Area (μm^2)	24.733					
	Power (mW)	22.132					
Bfloat16	Area (μm^2)	21.392					
	Power (mW)	19.230					
Posit ($Area \times Power \times MSE$)		1.02E-04	5.20E-04	5.19E-04	5.16E-04	5.15E-04	6.19E-04
Bfloat16 ($Area \times Power \times MSE$)		7.73E-04	3.88E-04	3.88E-04	3.85E-04	3.84E-04	4.64E-04
Improvement							1.33×

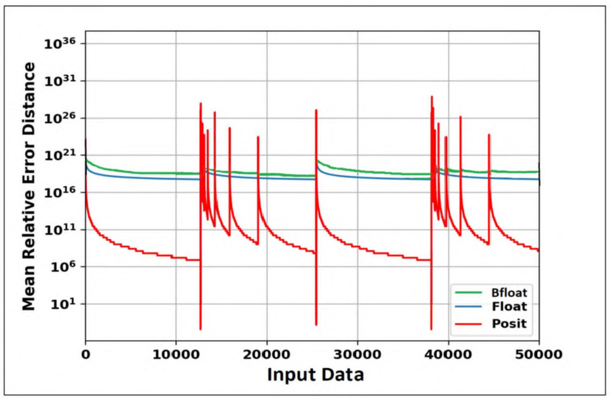

Fig. 3. MRED comparison between bfloat16, posit and IEEE754 based floating-point representation under single event upset

IV. APPLICATIONS

In this section, we demonstrated the training of a machine learning model using bfloat16 representation. The literature work published in [17] [18] [19] [20], proclaimed that 16-bit precision is adequate for the training of an artificial neural network. In this paper, it is proved that low precision formats perform better in terms of area and power. Generally, single-precision (FP32) floating-point representation is employed for training the neural network. A few pieces of research [17] [18] employed half-precision (FP16) floating-point representation for training. However, the short dynamic range of the FP16 representation makes it less competent to represent an error gradient value while performing back-propagation. To overcome this limitation, loss techniques [21] are employed during training. These techniques express error gradients in the given dynamic range to support FP16 based training. Although it is easier to implement for feed-forward networks by a constant scaling factor in terms of multiplication loss, still the network consumes more resources and time for optimization.

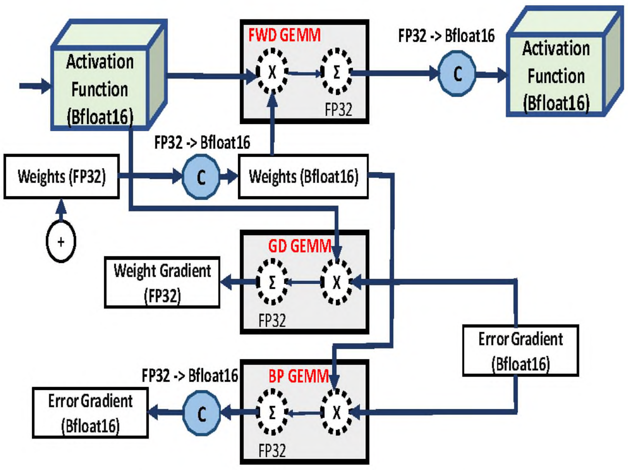

Fig. 4. Training machine learning models using mixed precision data flow with bfloat16 representation

The development of software tools, such as automatic mixed precision [30], are focused to ease the burden of data scientists.

However, these softwares also require modification and are not providing assurity of sufficient performance gains.

TABLE V
DYNAMIC RANGE COMPARISON AMONG BFLOAT16, IEEE754 FP32 AND FP16 REPRESENTATIONS.

Floating-point representation	Bits (s,e,m)	Min normalization	Max normalization
FP32	1,8,23	$1.17e-38$	$3.40e38$
FP16	1,5,10	$6.10e-5$	$6.55e4$
BF16	1,8,7	$1.17e-38$	$3.38e38$

On the other hand, bfloat16 representation is the truncated version of FP32 representation. Table V shows the dynamic range comparison of FP32, FP16 and bfloat16 representations. It shows that bfloat16 and FP32 are representing values in the same range, however FP16 has a noticeably short dynamic range. The extended dynamic range of bfloat16 makes it suitable to represent smaller gradient values and it does not need to apply loss scaling methods. As shown in the experimental analysis, the core computing units of bfloat16 such as fused multiply and addition operation are built by employing 8-bit multipliers, which reduces the area and power and maintains the same dynamic range as FP32.

The value in bfloat16 representation is easy to convert into FP32 by employing the zeros at the LSB of mantissa, which makes it a suitable choice for mixed-precision applications. The mixed precision dataflow employed for training the machine learning models using bfloat16 format as shown in Figure 4. A general matrix multiply (GEMM) computation unit receives the input in bfloat16 representation and accumulates the output in FP32, which covers the drawback of bfloat16 which leads towards less precision and consumes less area and power of bfloat16 computation unit. A converter (converts among bfloat16 and FP32), shown as C in Figure 4, alters the output value to bfloat16 before passing it to the next layer. The converter is also employed to alter the weight representation to pass into the next layer. The values of error gradients are given in bfloat16 representation. Except for GEMM computation, other computations such as activation functions and their derivatives are also computed in bfloat16 format. The bias value is same as FP32 and the optimization unit is always employing FP32 representation.

V. RESULT AND DISCUSSION

In this section, we present the comparison of various number systems to conclude the best performing floating-point representation for various machine learning applications. Table III and IV exhibit comparison of a mean square error, area and power analysis of fused MAC unit, and divider and square root unit, respectively. The analysis depicts that bfloat16 based units performs $1.51\times$ better as compared to posit based units. The fixed-point representation is excluded from the analysis due to its less range of number representations, due to which it is less preferred for the training of machine learning models.

We have experimented with the realization of fused MAC, and divider and square root units based on four representations,

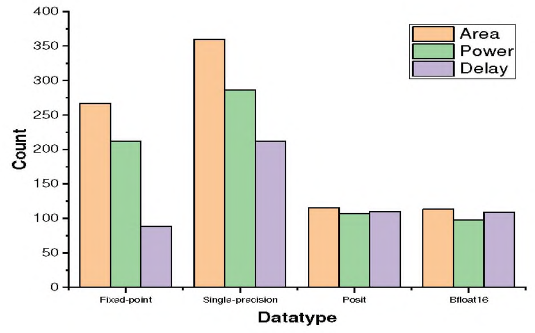

Fig. 5. Comparison of area, power, and delay for fused multiply and addition operation.

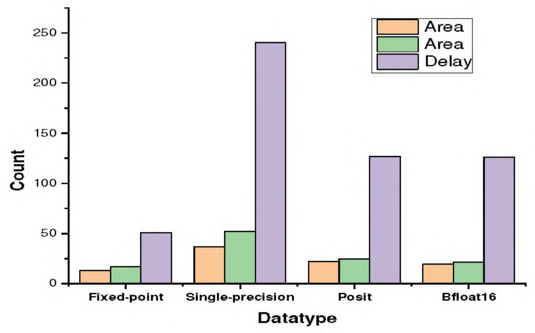

Fig. 6. Comparison of area, power, and delay for division and square-root operation.

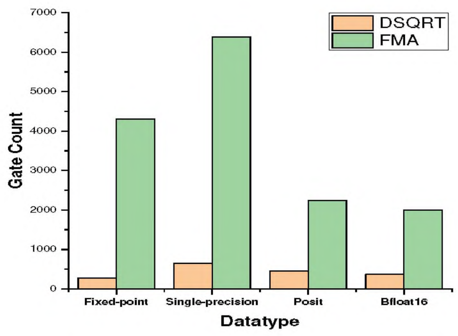

Fig. 7. Comparison of number of gates employed for fused multiply and add (Fused MAC) and division and square-root (DSQRT) operation.

fixed-point, *single-precision*, *posit* and *bfloat16*. In all the representations except fixed-point, the other representations perform worse in terms of area and power. It can be observed that the overall performance of bfloat16 is better among all the representations, and due to its larger exponent word size, its range is the same as single precision. The bar graph of the comparisons shown in Figures 5 and 6 illustrate that bfloat16 representation performs better on all the parameters, viz. area, power, delay and range. Posit performs better in terms of accuracy, but the figure of merit (product of area, power, and accuracy) of bfloat16 is better than posit.

Figure 7 exhibits the total number of gates employed in the realization of fused MAC, as well as divider and square root units employing posit and bfloat16 number representations. It shows that bfloat16 based implementations consume $1.35\times$ less gates as compared to posit based implementations. This states that bfloat16 based implementations can be utilized in the efficient realization of ML algorithms.

VI. Conclusion

The proposed research aims to present bfloat16 as the best choice for machine learning applications among existing floating-point representations. This is because of the same dynamic range of bfloat16 as single-precision, which is acceptable for the realization of ML algorithms. It is observed that bfloat16 based designs perform better in the resource utilization and power consumption than posit and other floating-point number representation based designs. However, posit based implementations perform better in terms of accuracy and precision as compared to other representation based implementations, but they consume an average of $\mathbf{1.51}\times$ more hardware resources. On the other hand, bfloat16 based designs consumes an average of $\mathbf{1.50}\times$ less power consumption than posit based designs. Further, bfloat16 based implementations exhibit a robust behavior while training a neural network, and the use of bfloat16 in the design eliminates hyper-parameter tuning in the block quantization. Due to all these observations, bfloat16 shows its relevance among other representations in terms of hardware implementation of various machine learning applications.

Acknowledgement

The work has been supported from the programme INTER-EXCELLENCE (LTAIN19100) funds of the Ministry of Education, Youth and Sports, Czech Republic, project LTAIN19100 Artificial Intelligence Enabled Smart Contactless Technology Development for Smart Fencing and Ministry of Science and Technology through project DST/INT/Czech/P-11/2019 entitled as "Smart Contactless Technology Development for Smart Fencing". The support provided for the projects by respective ministries in Czech Republic and India is gratefully acknowledged.

References

[1] Markstein, Peter. "The new IEEE-754 standard for floating point arithmetic." Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum fr Informatik, 2008.

[2] BFLOAT16 - hardware numerics definition. https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf. Accessed: 2019-03-22.

[3] Burgess, Neil, et al. "Bfloat16 processing for neural networks." 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH). IEEE, 2019.

[4] Defour, David, et al. "Shadow computation with BFloat16 to estimate the numerical accuracy of summations." 2021 IEEE 28th Symposium on Computer Arithmetic (ARITH). IEEE, 2021.

[5] Ozturk, Muhammed Emin, et al. "Distributed BERT Pre-Training & Fine-Tuning With Intel Optimized Tensorflow on Intel Xeon Scalable Processors." Science 11.5 (2017): 746-761.

[6] Chaurasiya, Rohit, et al. "Parameterized posit arithmetic hardware generator." 2018 IEEE 36th International Conference on Computer Design (ICCD). IEEE, 2018.

[7] Jaiswal, Manish Kumar, and Hayden K-H. So. "Pacogen: A hardware posit arithmetic core generator." Ieee access 7 (2019): 74586-74601.

[8] Jaiswal, Manish Kumar, and Hayden K-H. So. "Universal number posit arithmetic generator on FPGA." 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2018.

[9] Gustafson, John L., and Isaac T. Yonemoto. "Beating floating point at its own game: Posit arithmetic." Supercomputing frontiers and innovations 4.2 (2017): 71-86.

[10] Uguen, Yohann, Luc Forget, and Florent de Dinechin. "Evaluating the hardware cost of the posit number system." 2019 29th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2019.

[11] Dhanabal, R., Sarat Kumar Sahoo, and V. Bharathi. "Implementation of Low Power and Area Efficient Floating-Point Fused Multiply-Add Unit." Proceedings of the International Conference on Soft Computing Systems. Springer, New Delhi, 2016.

[12] Robison, Arch D. "N-bit unsigned division via n-bit multiply-add." 17th IEEE Symposium on Computer Arithmetic (ARITH'05). IEEE, 2005.

[13] Li, Yamin, and Wanming Chu. "Implementation of single precision floating point square root on FPGAs." Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186). IEEE, 1997.

[14] Li, Yamin, and Wanming Chu. "A new non-restoring square root algorithm and its VLSI implementations." Proceedings International Conference on Computer Design. VLSI in Computers and Processors. IEEE, 1996.

[15] I. Alouani, A. B. Khalifa, F. Merchant, and R. Leupers, An investigation on inherent robustness of posit data representation, in 2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID). IEEE, 2021, pp. 276281.

[16] A. Tiwari, G. Trivedi, and P. Guha, Design of a low power bfloat16 pipelined mac unit for deep neural network applications, in 2021 IEEE Region 10 Symposium (TENSYMP). IEEE, 2021, pp. 18.

[17] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In ICML, pages 17371746, 2015.

[18] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaev, Ganesh Venkatesh, et al. Mixed precision training. arXiv preprint arXiv:1710.03740, 2017.

[19] Urs Kster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. CoRR, abs/1711.02213, 2017.

[20] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj D. Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jess Corbal, Nikita Shustrov, Roman Dubtsov, Evarist Fomenko, and Vadim O. Pirogov. Mixed precision training of convolutional neural networks using integer operations. In ICLR, 2018.

[21] NVIDIA. Automatic mixed precision for deep learning. https://developer.nvidia.com/automatic-mixed-precision. Accessed: 2019-22-05.