A Comparison of Number Representations for Hardware Multiply-and-Accumulate Units

Li Yang Kat, *UC Berkeley,* Jingyi Xu, *UC Berkeley*

Abstract—Arithmetic computations in training neural networks are largely performed using the IEEE 754 floating point standard currently. Taking into consideration that the IEEE 754 standard was established in 1985, long before the current era of deep learning, it is timely to consider alternative number representations that may be more efficient for arithmetic operations involved in deep learning, most prominently the multiply-and-accumulate (MAC) operation extensively used in linear algebra. The Posit number format, first conceptualized by Gustafson, is one such alternative, boasting larger dynamic range than floating point. Johnson combined Gustafson's Posit numbers with a logarithmic number system and showed comparable neural network performance and simpler hardware with floating point and integer number representations of similar or higher bit-widths. Another contender is Google's BFLOAT16 floating-point format which boasts higher dynamic range at the cost of precision. In this study, a comprehensive comparison of these alternatives is presented from a hardware perspective. Multiply-and-accumulate units for each of these alternatives are synthesized and compared for their area and energy efficiency.

1 Introduction

The present era of deep learning presents unique problems in the hardware space for area and energy efficient computations [1]. The enormous and ever-growing number of computations required to perform computations in deep neural networks consume significant amounts of energy and are generally inefficient when mapped onto general-purpose hardware. This has channelled significant research effort into developing neural network accelerators that are able to perform these computations more efficiently.

With this development of custom hardware designed specifically to accelerate neural networks, there is an opportunity to re-think how these computations are fundamentally performed in digital circuits [2]. One of these threads pursued by researchers is to investigate how number representations change the efficiency of hardware for neural network computations and how different characteristics of these number systems may make them more or less suitable to represent numerical values in neural networks. These alternative number representations often seek to enhance the IEEE 754 floating point representation by increasing the dynamic range of values at the cost of precision over the dynamic range [3] [4], the precision of values within the range that has the greatest impact on prediction accuracy at the cost of precision in other regions [5], and/or simplify hardware implementations for multiply-and-accumulate units [6] which are the predominant linear algebra operation in neural networks.

This interest in alternative number representations is not only an academic one, but also an area of active research in industry. Google's BFLOAT16 [4] is a pertinent example of huge industrial users of deep learning exploring this space. A number of researchers from Facebook have also explored the implications of BFLOAT16 on the training of deep neural networks [7], while others have presented alternatives based on a novel number system first introduced in academia by Gustafson, the Posit number system [8]. This paper describes a comparative review and analysis of several promising number representations that have been presented in different works, spanning both academic and industry efforts.

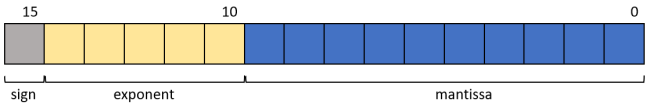

Fig. 1. IEEE 754 FP16 format

As part of this mid-term report, a detailed look into several floating point alternatives is presented in Section 2. In Section 3, a brief description of the basis of comparison across the various number systems is provided. This primarily involves the construction of a multiply-and-accumulate unit at the Register-Transfer Level (RTL) and a comparison of the area and energy estimates of the synthesized module. Next, a set of hypotheses outlining the expected outcomes of the subsequent RTL development and experiments on the various synthesized units is described in Section 4 and some concluding remarks will be found in Section 5.

2 Relevant Works

In this section, different number formats are presented. Here, a 16-bit bit-width is used across all the representations as a standard, since Google's BFLOAT16 is defined solely as a 16-bit format. The IEEE 754 16-bit floating point format is used as the benchmarks for these number representations, since it is still the most prevalent representation standard for high dynamic range computations with reasonably good precision.

2.1 IEEE 754 Floating Point

The IEEE 754 16-bit floating point, also known as the half-precision floating point format, follows the bit assignment shown in Figure 1.

The first most-significant bit is the sign bit which describes whether the represented value is positive or negative. Bit 14 to bit 10 describe the biased representation of the exponent with an offset of 15 and the 10 least-significant bits describe the mantissa, or the fractional part of the number with an implicit leading 1 bit ahead of the radix point before the 9th bit. Depending on the implementation, subnormal numbers can also be used where there is no leading 1 bit ahead of the 9th bit and the position of the varying radix point in the mantissa is after the first non-zero digit.

The IEEE 754 FP16 standard is capable of representing numbers from $-(2-2^{-10}) \times 10^{15} = -65504$ to $(2-2^{-10}) \times 10^{15} = 65504$. Numbers are represented with a higher precision closer to zero with a maximum precision of $2^{-10} \times 2^{-14} = 5.96 \times 10^{-8}$ and a minimum precision of $2^{-10} \times 10^{15} = 32$. With the inclusion of subnormal numbers, the smallest positive representable value is $2^{-24} = 5.96 \times 10^{-8}$. Without the inclusion of subnormal numbers, the smallest positive repreentable value is $2^{-14} = 6.10 \times 10^{-5}$.

2.2 Google's Brain Floating Point

Google introduced the brain floating point, commonly known as BFLOAT16 [4], as an alternative to the IEEE 754 floating point standard. There were several reasons for Google's investment into this new floating point representation. Firstly, the use of the IEEE 754 FP16 format was found to cause non-convergence in deep learning models. With essentially 10 bits used for precision and 5 bits used for dynamic range, the FP16 representation was found to have insufficient dynamic range for convergence. Furthermore, the bit assignment in BFLOAT16 enabled extremely convenient conversion between the FP32 format, which is still commonly used for training, and the BFLOAT16. This essentially enables training deep learning models in FP32 on large-scale GPU farms, while deploying them in the more efficient BFLOAT16 format in edge servers and devices. Figure 2 shows the bit assignment in the BFLOAT16 format.

Similar to FP16 format, there is a sign bit in the MSB. However, the exponent resembles the FP32 representation, with 8 bits instead of 5 for the exponent, and an exponent offset of 127. Because of the increased bit-width for the exponent, the number of bits reserved for the mantissa is reduced, effectively decreasing the precision across the entire dynamic range for each exponent value when compared to FP16.

The BFLOAT16 format can represent numbers ranging from $-(2-2^{-7}) \times 10^{127} = -3.39 \times 10^{38}$ to $(2-2^{-7}) \times 10^{127} = 3.39 \times 10^{38}$. The maximum precision is $2^{-7} \times 2^{-126} = 9.18 \times 10^{-41}$ and the mnimum precision is $2^{-10} \times 2^{127} = 1.66 \times 10^{35}$. With the inclusion of subnormal numbers, the smallest positive representable value is $2^{-133} = 9.18 \times 10^{-41}$ and the smallest positive representable value without the inclusion of subnormal numbers is $2^{-126} = 1.18 \times 10^{-38}$.

BFLOAT16 trades precision between consecutive exponent values, where the highest precision of the mantissa is $2^{-7} = 7.8125 \times 10^{-3}$ compared to $2^{-10} = 9.77 \times 10^{-4}$ in FP16, for higher dynamic range and the ability to represent numbers much closer to zero. This may assist with computations in training deep neural networks, where diminishing gradients during backpropagation may involve increasingly small numbers. The ability to perform computations on these small numbers enable the models to converge.

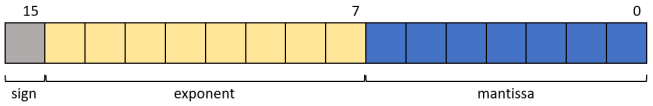

Fig. 2. Google BFLOAT16 format

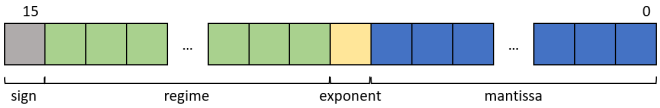

Fig. 3. (16, 1) Posit format

2.3 Posits

Posits were first introduced in 2016 by Gustafson as a replacement for the IEEE 754 floating point standard [5]. It is a subset of the universal numbers, or unums, where Posits are also known as type III unums. In his work introducing the novel number system, Gustafson suggests that Posits have a larger dynamic range, higher accuracy and takes less circuitry than the IEEE 754 floating point.

What is perhaps the greatest difference between Posits and FP16 is the regime in the Posit format. The regime is a set of bits with variable run-length that confers greater dynamic range. While a floating point system is described by the numbers of bits used for the exponent and mantissa ((5, 10) describes FP16 and (8, 7) describes BFLOAT16), the Posit number system is described by the total number of bits and the number of bits assigned to the exponent, or (N, es). The recommended format for 16-bit Posits is (16, 1), where 16 is the total length of each Posit number in the format, N, and 1 is the number of bits describing the exponent, es. The format of a (16, 1) Posit is shown in Figure 3.

In a posit number format, the MSB is also the sign bit. A sign bit of 1 represents a negative number while a sign bit of 0 represents a positive number. If the sign bit is 1, the rest of the Posit number is encoded in two's complement. The next component of the Posit number following the sign bit is the regime. The value of the regime is encoded using run-length encoding, where a string of '1's terminated by a '0' represents a regime greater than or equal to zero, while a string of '0's terminated by a '1' represents a regime less than zero.

$$k = \begin{cases} -m, & \text{with } m \text{ '0's terminated by '1'} \\ m-1, & \text{with } m \text{ '1's terminated by '0'} \end{cases}$$

With the value k, the value of the regime is then given by u^k, where the useed, $u = 2^{2^{es}}$. For a (16, 1) posit, $u = 4$.

After the regime, the exponent is encoded, if there are sufficient bits left. For instance, if the regime consists of 14 '0's terminated by a '1', there would be no bits left to encode the exponent, and the Posit would not include the exponent. The value of the exponent is similar to that in FP16 and BFLOAT16, given by 2^e.

The rest of the bits, if there are any left, is the mantissa or fraction. There are no subnormal numbers in Posits, since the regime can simply be extended to represent smaller numbers. There is always a leading '1' before the fractional bits.

With these four different components in the Posit number, the value of the Posit number is given by

$$x = (-1)^s \times u^k \times 2^s \times f$$

The (16, 1) Posit can represent numbers from $(-1)^1 \times 4^{14} \times 2^0 \times 1 = -268,435,456$ to $(-1)^0 \times 4^{14} \times 2^0 \times 1 = 268,435,456$. The maximum precision is $4^{-14} = 3.73 \times 10^{-9}$ and the minimum precision is $4^{14} - 4^{13} = 201,326,592$. The smallest representable positive number is $4^{-14} = 3.73 \times 10^{-9}$.

Compared to FP16, Posits have a larger dynamic range, and higher precision at values closer to zero. The complexity in decoding the regime bits is offset by the simplicity with the lack of subnormal numbers. Other advantages include a single representation of the value zero and NaN (not a number), compared to FP16 which has both positive and negative values for zero and NaN.

2.4 Logarithmic Number Systems

Rather than a completely separate number representation, logarithmic number systems represent numbers by the logarithm of their absolute value instead. The resultant logarithms can be represented in any of the earlier formats, IEEE 754 standard floating point, alternative floating point, and the Posit number system.

The key advantage of a logarithmic number system lies in the simplicity of implementing multiplication operations, since

$$log_a(x \times y) = log_a(x) + log_a(y)$$

In this manner, a typically hardware-intensive multiplication operation is transformed to a relatively simple addition operation.

However, the simplification of the multiplication (and by extension, division) operations comes at the cost of complicating the addition (and similarly, subtraction) operations. The addition and subtraction operations involve computing an additional factor that is a function of the difference between the two numbers, that is,

$$log_a(x \pm y) = log_a(x) + \sigma_{\pm}(log_a(y) - log_a(x))$$

The $\sigma_{\pm}$ function can either be stored as a look-up table or computed from a piece-wise linear approximation.

One alternative, suggested by Johnson from Facebook AI Research [8], is to accumulate the output of the multiplier in what is known as a Kulisch accumulator [9]. A Kulisch accumulator is a wide, fixed-point accumulator that is able to accommodate the full range of values of the result of a multiplication between two floating point numbers. As an example, an FP16 Kulisch requires 80 bits, given that the range of positive value in FP16 runs from 2^{-24} to 2^{15} with 1 additional bit for sign, for 40 bits, multiplied by 2 for the product of 2 fixed point values. Gustafson calls a similar idea the quire in the case of Posits, where a (16, 1) posit requires a 128-bit quire for exact multiplication.

In FP16 or (16, 1) Posits, these Kulisch accumulators or quires for Posits, enable the computation of exact multiplication between two numbers in their respective representation. Without these accumulators, the product of two floating point numbers or two Posits would be scaled for normalization and hence lose precision. However, these accumulators come at a cost of additional circuitry to convert a floating point or posit number to a fixed point representation, and deep adder chains to perform fixed point addition.

In the case of logarithmic number systems, these accumulators may simplify addition, due to the non-trivial $\sigma_{\pm}$ function that needs to be implemented otherwise. However, instead of converting from floating point to fixed point, Johnson's Kulisch accumulation involves the conversion of a logarithmic representation in Posits to a linear one in fixed point. Posits lend itself well to the representation of logarithmic numbers because represented numbers are tapered, which means that the number of bits allocated to fractional representation decreases as the absolute value increases. Here, the Kulisch accumulator serves to simplify the addition rather than calculate the exact multiplication. The bit-width of the accumulator is constrained to reduce the complexity of addition circuity, since logarithmic representations can lead to extremely fixed point large values.

The range of values representable by a logarithmic number system and the precision of the values depend on the underlying representation used to implement it.

3 Methodology

In this study, three different versions of multiply-and-accumulate (MAC) units are synthesized. Because the goal is to evaluate these MAC units in the context of neural network computations, the synthesized units are constructed with features insofar as it enables the effective computation of matrix multiplication in systolic arrays.

1) An IEEE 754 16-bit floating point MAC unit with Kulisch accumulation.
2) A (16, 1) Posit MAC unit with Kulisch accumulation.
3) A 16-bit Posit-based logarithmic MAC unit with constrained Kulisch accumulation.

These synthesized units are measured for their maximum achievable single cycle clock period, area and power consumption.

Next, matrix multiplication systolic arrays incorporating the different multipliers are synthesized and the area and power consumption are again compared. These systolic arrays are also simulated with input matrices of random values and the accuracy of the output is compared against a reference 64-bit floating point software matrix multiplication to evaluate whether the hardware tradeoffs in area and power consumption are worth the inaccuracies of the output.

4 Hypotheses

It is hypothesized that the IEEE 754 16-bit floating point implementation would occupy the least area and produce

consistent accuracy in matrix multiplications across the available range. This is because the IEEE 754 format has a constant exponent bit-width, which simplifies decoding logic. The number of bits used for the mantissa is also consistent for all values of the exponent, providing similar precision relative to the magnitude of the number across the range.

However, there are applications where a Posit or a logarithmic implementation would perform better. At values close to zero, Posits represent numbers with greater precision than their floating point representation. This results in greater accuracy and precision for matrix multiplications of values close to zero. This is particularly advantageous in the training of neural networks, where the convergence of the network depends on the precision to which changes in parameters can be represented.

Logarithmic number systems are advantageous for their range. For a smaller number of bits, a logarithmic number system can achieve a larger range. For representations with small bit-widths, multipliers and MAC units built using the logarithmic number system can lead to more efficient hardware, where look-up tables used to convert between the logarithmic and linear representations replace the complex decoding circuitry and fixed point multipliers required in linear representations. However, for larger bit-widths, these look-up tables get unreasonably large and it may be more appropriate to use piece-wise approximations to evaluate convert logarithmic numbers back to their linear representation. This provides a certain degree of freedom in the design-space exploration for logarithmic numbers, trading accuracy with hardware complexity.

5 Conclusion

In summary, this project investigated the impact of different number representations on the metrics of area and power efficiency for different multiply-and-accumulate units. The continued prevalence of deep learning as a computational tool motivates the search for efficient hardware architectures for implementing deep learning algorithms at different levels of the hardware abstraction hierarchy.

While the IEEE 754 floating point remains the most pervasive representation for high dynamic range and high precision mathematical computations, including matrix multiplications, this project suggests that there may be viable alternatives that are more compatible with the computations found in deep learning stacks. A more nuanced approach to define more application-specific numeric representations would indubitably benefit future research and platforms in the field of machine learning and artificial intelligence.

References

[1] V. Sze, Y. Chen, J. S. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," *CoRR*, vol. abs/1612.07625, 2016.

[2] Y. LeCun, "Deep learning hardware: Past, present, and future," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 12–19, 2019.

[3] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A transprecision floating-point platform for ultra-low power computing," *CoRR*, vol. abs/1711.10374, 2017.

[4] S. Wang and P. Kanwar, "Bfloat16: The secret to high performance on cloud tpus," Aug 2019.

[5] J. L. Gustafson, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, 2017.

[6] R. Matoušek, M. Tichý, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman, "Logarithmic number system and floating-point arithmetics on fpga," *Lecture Notes in Computer Science Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, p. 627–636, 2002.

[7] D. D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of BFLOAT16 for deep learning training," *CoRR*, vol. abs/1905.12322, 2019.

[8] J. Johnson, "Rethinking floating point for deep learning," *CoRR*, vol. abs/1811.01721, 2018.

[9] U. Kulisch, *Computer arithmetic and validity: theory, implementation, and applications*. De Gruyter, 2013.