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. 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]. Another extreme example is IBM's 8-bit floating point which halves the necessary bit-width further [9]. 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.

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 number systems is provided. This primarily involves the construction of a multiply-and-accumulate unit at the Register-Transfer Level (RTL). The area and energy implications of the number representations ae then compared by working these MAC units into a systolic array, a regular grid of processing elements commonly-used to accelerate generalized matrix multiplication (GEMM) operations found in neural network computations. Behavioral models of these number systems running a simple neural network that classifies the MNIST dataset are also used to investigate the potential accuracy implications of these number systems in running matrix multiplications associated with a semi-realistic neural network. In Section 4, these results of the investigation is presented and in Section 5, a set of criteria for number systems is outlined for future work. Some concluding remarks will be found in Section 6.

2 Relevant Works

In this section, different number formats, including the IEEE 754 floating point standard, Google's BFLOAT16 and Gustafson's Posits, 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. 8-bit formats such as IBM's 8-bit floating point is not featured here due to the different bit-width. 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

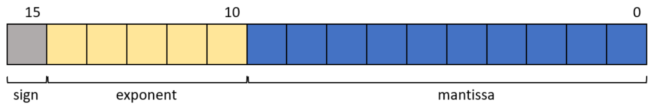

Fig. 1. IEEE 754 FP16 format

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 reduced accuracy 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 to produce sufficiently accurate weight updates. Research from Baidu and NVIDIA [10] found that 5% of weight gradients are smaller than the smallest representable value in FP16 and thus saturated to zero. 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 $2^{-126} = 1.18 \times 10^{-38}$.

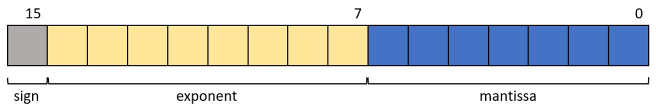

Fig. 2. Google BFLOAT16 format

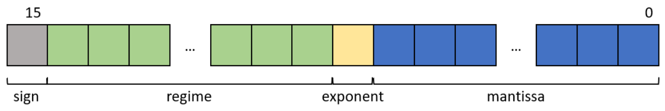

Fig. 3. (16, 1) Posit format

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.

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.

3 Methodology

In this study, two different versions of multiply-and-accumulate (MAC) units were 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 128-bit quire accumulation.

3.1 RTL Design

To facilitate parametrization of the bit-widths and maximize composition and reuse of different module blocks, the design was done in Chisel [11]. Chisel is a hardware generation language that facilitates circuit design using abstract software concepts, through Scala, that improves reusability and parametrization of hardware blocks.

Taking inspiration from the HardFloat library from UC Berkeley [12], a unified unpacked format for both floating point and Posits were designed, with signals indicating the *zero*, *NaN*, *sign*, *exponent* and *fraction* bits. Next, 16-bit floating point or Posit inputs are converted to the unified format. Table 1 shows the bit-widths of the unpacked format for both FP16 and (16, 1) Posits. This unified interface facilitates subsequent multiplier and multiply-and-accumulate design. Both FP16 and (16, 1) Posit formats require leading-zero detection, for subnormal normalization in FP16, and for exponent decoding in (16, 1) Posits. This increases the complexity of the hardware, requiring variable shifting. However, since subnormal numbers are crucial to allow the FP16 format to represent significantly smaller numbers which appear in weight gradients of neural networks, they were chosen to be supported. The leading '1' bit for the fractional bits is also added for both FP16 and (16, 1) Posits during the conversion to facilitate the subsequent multiplication.

TABLE 1
Bit-widths of the signals in the internal encoding of FP16 and (16, 1) Posit formats

	FP16	(16, 1) Posit
Zero	1	1
NaN	1	1
Sign	1	1
Exponent	6	6
Fraction	11	13

A unified multiplier is subsequently designed to accept variable exponent and fraction bit-widths to accommodate the recoded representation. The multiplier is responsible for adding the exponents and multiplying the fractions, while taking into account the zero and NaN bits.

Using a technique known as Kulisch accumulation, the output of the multiplier is a wide fixed point value with a radix point that is, in theory, supposed to be able to accommodate the entire range of values of the result of the multiplication between two floating point numbers. A similar concept was developed by Gustafson for Posits known as quire accumulation. The rationale behind Kulisch accumulation is to enable exact-multiply-add, reducing the error associated with the alignment of exponents when two floating point numbers are added using typical floating point addition. In the case of FP16, this occurs when two normal floating point numbers with a ratio of greater than 2^10 are added. Using Kulisch accumulators defers any error in the computation to the final conversion from fixed point back to floating point. For FP16, the Kulisch accumulator needs to be 80 bits wide, with the radix point at the 48th bit, wheareas for (16, 1) Posits, the quire needs to be 128 bits wide, with the radix point at the 56th bit.

With the use of Kulisch accumulators in mind, the result of the exponent addition is used to shift the fraction accordingly, based on the number of bits in the Kulisch accumulator and its radix point. Any shifts that underflows the accumulator sets the zero bit and any shifts that overflows the accumulator sets the NaN bit. The use of the zero and NaN bit alongside the Kulisch accumulator defers the handling of underflows and overflows to the final output of the systolic array. However, in theory, unless the Kulisch accumulator has been constrained in bit-width to reduce complexity, there should be no under- or overflows since the accumulator is meant to accommodate the entire range of possible outputs of the multiplication.

To compose the MAC unit, the fixed point output of the multiplier is added to an additional fixed point input, presumably from a previous MAC unit in the case of a systolic array. Underflows and overflows associated with the fixed point addition similarly set the zero and NaN bits respectively.

3.2 Systolic Array Simulation and Synthesis

To perform generalized matrix multiplication (GEMM) in hardware, the most common architecture employed is the systolic array. The systolic array is composed of a regular grid of processing elements that perform MAC operations. Depending on the dataflow adopted for the computation, most commonly either weight-stationary or output-stationary, the weights or outputs are stored locally in the processing elements to maximize reuse of these values and reduce memory accesses.

For this study, a 4-by-4 systolic array that employs the weight-stationary dataflow was designed for benchmarking. The implementation of the systolic array for this study takes inspiration from Gemmini [13], a systolic array generator designed in Chisel at UC Berkeley. Each processing element in the systolic array consists of two registers that can be pre-loaded with weights. When one of the register values is used for the MAC operation, the other can be loaded with values to overlap operation. In general, the weight-stationary systolic array designed for this study computes

$$C = A \times B + D$$

where B are the weights that are pre-loaded into the systolic array and A and D are fed as inputs into the systolic array, aligned in a manner that the output of each PE coincides with the inputs of the next to compute the vector dot-product. The output matrix C is then produced from the systolic array, re-aligned, and stored in an accumulator SRAM.

A SystemVerilog testbench was then developed to exercise the systolic array, for both RTL simulation and post-synthesis simulations for power estimation. The testbench tiles a large matrix of configurable size into 4-by-4 tiles to be fed into the systolic array.

To perform post-synthesis power estimation, the systolic array was synthesized using Cadence Genus in the ASAP7 predictive process design kit. The gate-level simulation was then performed using Synopsys VCS. For the simulation, a large 128-by-128 matrix of random values were fed into the systolic array. While it would be more appropriate to conduct the power estimation on the training and inference of a neural network to better simulate circuit activity, even the simplest multi-layer perceptron networks that classify the MNIST dataset require extremely large matrix multiplications up to the order of 1024-by-1024 elements that would be unrealistic under the constraints of this study. The circuit activity levels from the gate-level simulation are extracted from the simulation results and passed to Genus for power estimation.

The area is estimated from the synthesis results in Genus based on the cell sizes of the synthesized gates, and the single-cycle period of each design is estimated by attempting to constrain the clock period fed into Genus until a setup slack violation occurs.

TABLE 2
Percentage of weight and bias gradients smaller than the smallest representable number in each number representation from the 64-bit model

	FP16	(16, 1) Posit	Reduction
Hidden weights	2.0	1.0	2.0x
Hidden biases	3.6	1.4	2.6x
Output weights	0.0	0.0	-
Output biases	1.9	0.6	3.2x

3.3 Behavioral Simulation

To examine the effect of the number representation on the accuracy of neural network computations, a behavioral model for each number representation was designed in C++. C++ libraries available for half floating point [14] (that is, FP16) and Posits [15] were used to construct the neural network accordingly. The behavioral model classifies the MNIST dataset using a multi-layer perceptron model. The MNIST dataset comprises of handwritten digits with a training set of 60,000 examples and a test set of 10,000 testcases. The network model consists of 784 input nodes, 64 hidden nodes and 10 output nodes. The small number of hidden nodes reduces the complexity of computations while maximizing the contribution of each hidden node towards the output, ensuring that any inaccuracies from the computation is not hidden by the complexity of the network itself.

The network is trained using a batch size of 128 over 5000 training batches from randomly drawn samples of the training set and the accuracy of the model is evaluated from inferring the digit witten in all 10,000 testcases in the test set.

As a baseline measurement, the network achieves an average of 96.93% accuracy on the test set using 64-bit floating point precision to perform the computations. This validates that the network model can achieve relatively good accuracy despite the small number of hidden nodes, provided the number representation enables accurate computations.

Table 2 shows the percentage of weight and bias updates which are smaller than the smallest representable number in each of the number representations. The output weight updates are all larger than the smallest representable numbers. The data shows that in terms of representing these weights and updates, Posits have a clear advantage, reducing the number of weight and updates that would be rounded to zero by at least 2 times. However, this comes at the cost of representing this numbers precisely, and the effects of this reduction in precision could lead to poor overall network performance.

TABLE 3
Total cell area for 4-by-4 systolic array and constituent modules composed using both FP16 and (16, 1) Posit number representations in μm^2

	FP16	(16, 1) Posit	Increase
Systolic array	75440	105817	40.2%
Systolic array PE	4548	6322	39.0%
MAC unit	3392	4820	42.1%
Multiplier	2662	3653	37.2%
Decoder	183	389	112.5%

4 Results

In this section, comparative results between the systolic array implemented using FP16 and (16, 1) Posit are presented, according to four metrics - physical area, power consumption, maximum achievable frequency for a single-cycle PE, and accuracy/precision of computation as measured by a pseudo-workload comprising the neural network described in Section 3.

4.1 Area

Table 3 shows the area estimates obtained from the synthesis results. The sub-modules in the systolic array employing (16, 1) Posits as the number representation occupy approximately 40% more area than the sub-modules in the systolic array employing the FP16 format. Furthermore, the decoding from a Posit to the unified sign, exponent and fraction representation passed as inputs to the multiplier also demands more complex circuitry than the decoding from an FP16 format even if subnormals are supported.

In the systolic arrays designed for this study, the decoding of the numbers is only performed once, prior to the entry of the values into the systolic array. This helps to amortize the overhead of decoding the input subnormal FP16 or (16, 1) Posits over the size of the systolic array. The decoded values are then passed on to subsequent PEs as is. Thus, although the overhead of decoding Posits may be much higher than that for FP16 values, the increase in area of the decoder by 112.5% may be a less significant contribution to the increase in overall area. Nevertheless, the increase in area required to implement the multiplier for Posits is significant at approximately 40%, which contributes to the increase in area of the overall systolic array by 40.2%. This can be attributed in part to the 60% wider accumulator width necessary for (16, 1) Posit accumulation.

4.2 Power Consumption

In terms of power consumption, the systolic array using (16, 1) Posits also consume 33.6% more power than the systolic array using FP16. The largest contributor to this increase comes from leakage power that increased by 41.1%. This can be attributed to two factors. Firstly, the larger number of gates required to implement a Posit multiplier compared to a FP16 multiplier increases the leakage, internal and switching powers. Secondly, the increase in logic depth required to implement Posit logic forces the synthesis tool to implement the logic path using more Low Voltage Threshold (LVT) and Super Low Voltage Threshold (SLVT) gates that consume more power than Regular Voltage Threshold (RVT) gates.

TABLE 4
Total power consumption of a 4-by-4 systolic array using both FP16 and (16, 1) Posit number representations in W

	FP16	(16, 1) Posit	Increase
Leakage	1.90×10^{-3}	2.68×10^{-3}	41.1%
Internal	4.11×10^{-3}	5.53×10^{-3}	34.5%
Switching	1.54×10^{-3}	1.91×10^{-3}	24.0%
Total	7.56×10^{-3}	1.01×10^{-2}	33.6%

TABLE 5
Highest achievable clock frequency for a 4-by-4 systolic array with a single-cycle data path in each PE for the different number formats

	FP16	(16, 1) Posit
Highest achievable clock frequency	1.25 GHz	1.11 GHz
Time taken for a single matmul	14.4 ns	16.2 ns

TABLE 6
Neural network accuracy trained using different number representations

	FP64	FP16	(16, 1) Posit
Run 1	96.88%	90.01%	90.45%
Run 2	96.89%	90.06%	89.7%
Run 3	96.95%	90.57%	90.32%
Run 4	96.97%	90.17%	89.7%
Run 5	96.97%	91.06%	89.72%
Average	96.93%	90.37%	89.98%

4.3 Achievable Clock Frequency

To measure the highest achievable clock frequency, the 4-by-4 systolic array was repeatedly synthesized while decreasing the clock period constraint to Genus.

Table 5 shows the highest achievable clock frequency for each 4-by-4 systolic array, under the assumption that a single-cycle datapath is adopted within each PE of the systolic array. Each 4-by-4 isolated matrix multiplication takes 18 cycles to compute, excluding the time taken to preload the weights in the weight-stationary dataflow, which results in 14.4 ns taken for each 4-by-4 matrix multiplication in the FP16 format and 16.2 ns in the (16, 1) Posit format.

4.4 Neural Network Accuracy

Using the behavioral C++ model to perform the neural network computations as outlined in Section 3, the following results were obtained.

From Figure 4, the neural networks trained using both the FP16 and (16, 1) Posit formats achieved similar training loss over the 5000 batches. Neither of the 16-bit formats were able to match the training loss of the 64-bit reference, which demonstrated a loss of approximately one order of magnitude lower.

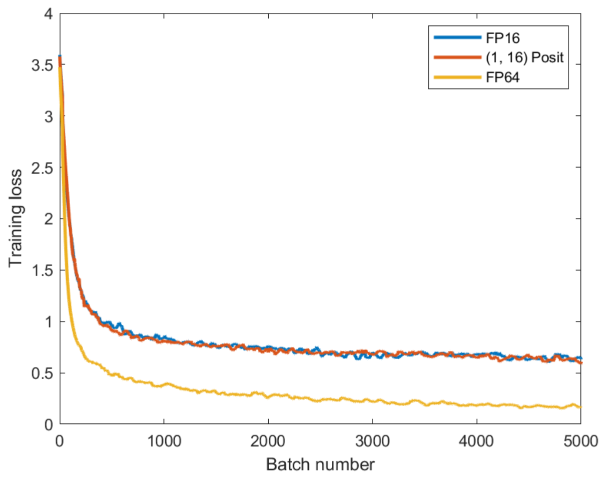

Fig. 4. Training loss of the MNIST classification neural network using different number representations

Each of the implementations were trained and tested five times, with the results shown in Table 6. The FP16 format performs marginally better than the (16, 1) format under the same parameters. The highest accuracy among the 16-bit format, at 91.06% is also achieved using the FP16 format. In terms of performing the computations in neural networks, which are predominantly GEMM operations, the FP16 format proves to be better than the (16, 1) Posit format.

5 Discussion

From the results, it is evident that the FP16 number format remains superior to the (16, 1) Posit format. The systolic array synthesized using the FP16 format requires less area, consumes less power, achieves a higher clock frequency and produces better accuracy in computations involved in neural networks.

However, this does not mean that there is no room for the Posit format. The (8, 0) and (8, 1) Posit formats remain strong contenders in the space of ultra-low bit-width computations. The (8, 1) Posit format is capable of representing numbers up to 4096, which can prove to be useful in non-critical, low-power computations that do not require significant levels of accuracy.

Another possibility, introduced by Johnson of Facebook [8], is the incorporation of Posits into a logarithmic number system. Logarithmic number systems represent numbers by the logarithm of their absolute value. As an example,

$$5 = 2^{2.322}$$

Thus, the number that is represented in a Posit logarithmic number format would be $01010011 = 2.375$. According to Johnson, using the logarithmic number format reduces the complexity of multiplication. The multiplication of two logarithmic numbers involves a simple addition operation. The subsequent conversion of the logarithmic number to a fixed point format compatible with a Kulisch accumulator also involves a relatively small lookup table. The bit-width and depth of this look-up table is configurable to different precisions as required by the computations and offers an additional dimension of flexibility in reducing hardware complexity.

TABLE 7
Preliminary synthesis results of a log Posit 4-by-4 systolic array

	(16, 1, 11, 11, 10) log Posit
Area	98290 μm^2
Total Power	7.64×10^{-3}

As a brief experiment in order to investigate the feasibility of this number format introduced by Johnson, the (16, 1) Posit format of this logarithmic number system was synthesized. Table 7 shows the results of the synthesis of the log Posit MAC unit used in a 4-by-4 systolic array.

The area used to implement the systolic array is still higher than that of the FP16 systolic array, but the power estimate is now much more comparable with that of the FP16 format and much lower than the (16, 1) Posit format. Furthermore, by sacrificing accuracy of the computations by reducing the width and depth of the look-up tables or, as suggested by Johnson adopting some form of compression for the look-up tables, the area and power consumption may be optimized further specific to different applications.

Future work would definitely involve a more thorough investigation into this log Posit format, with a behavioral software model to investigate the implications of these logarithmic computations on the accuracy of matrix multiplications and by extension neural network computations.

6 Conclusion

In summary, this project investigated the impact of different number representations on the metrics of area, power efficiency, cycle time and computation accuracy for the implementation of systolic arrays. 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. While the (16, 1) Posit was shown to be less feasible than initially thought, there are other possibilities such as the logarithmic Posit format suggested by Johnson. A more nuanced approach to define more application-specific numeric representations would also 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] N. Wang, J. Choi, D. Brand, C. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," *CoRR*, vol. abs/1812.08011, 2018.

[10] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *CoRR*, vol. abs/1710.03740, 2017.

[11] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, IEEE, 2012.

[12] J. Hauser, "Berkeley hardfloat," 2019.

[13] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, Y. S. Shao, B. Nikolic, I. Stoica, and K. Asanovic, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," 2019.

[14] C. Rau, "half," 2019.

[15] C. Leong, "Softposit."